ABSTRACT

Title of dissertation: HEAP DATA ALLOCATION TO SCRATCH-PAD MEMORY IN EMBEDDED SYSTEMS

Angel Dominguez
Doctor of Philosophy, 2007

Dissertation directed by: Professor Rajeev K. Barua
Department of Electrical and Computer Engineering

This thesis presents the first-ever compile-time method for allocating a portion of a program's dynamic data to scratch-pad memory. A scratch-pad is a fast directly addressed compiler-managed SRAM memory that replaces the hardware-managed cache. It is motivated by its better real-time guarantees vs cache and by its significantly lower overheads in access time, energy consumption, area and overall runtime. Dynamic data refers to all objects allocated at run-time in a program, as opposed to static data objects which are allocated at compile-time. Existing compiler methods for allocating data to scratch-pad are able to place only code, global and stack data (static data) in scratch-pad memory; heap and recursive-function objects(dynamic data) are allocated entirely in DRAM, resulting in poor performance for these dynamic data types. Runtime methods based on software caching can place data in scratch-pad, but because of their high overheads from software address translation, they have not been successful, especially for dynamic data.

In this thesis we present a dynamic yet compiler-directed allocation method for

dynamic data that for the first time, (i) is able to place a portion of the dynamic data in scratch-pad; (ii) has no software-caching tags; (iii) requires no run-time per-access extra address translation; and (iv) is able to move heap data back and forth between scratch-pad and DRAM to better track the program's locality characteristics. With our method, code, global, stack and heap variables can share the same scratch-pad. When compared to placing all dynamic data variables in DRAM and only static data in scratch-pad, our results show that our method reduces the average runtime of our benchmarks by 22.3%, and the average power consumption by 26.7%, for the same size of scratch-pad fixed at 5% of total data size. Significant savings in runtime and energy were also observed when compared against cached memory organizations, showing our method's success with SPM placement of dynamic data under constrained memory sizes.

HEAP DATA ALLOCATION TO SCRATCH-PAD MEMORY IN EMBEDDED SYSTEMS

by

Angel Dominguez

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:
Professor Rajeev K. Barua, Chair/Advisor
Professor Manoj Franklin
Professor Shuvra S. Bhattacharrya
Professor Peter Petrov
Professor Chau-Wen Tseng

Table of Contents

List of Figures

Chapter 1

Introduction

The proposed research presents an entirely new approach to dynamic memory allocation for embedded systems with scratch-pad memory. In embedded systems, program data is usually stored in one of two kinds of write-able memories – SRAM or DRAM (Static or Dynamic Random-Access Memories). SRAM is fast but expensive while DRAM is slower (by a factor of 10 to 100) but less expensive (by a factor of 20 or more). To combine their advantages, often a large DRAM is used to build low-cost capacity, and then a small SRAM is added to reduce runtime by storing frequently used data. The gain from adding SRAM is likely to increase in the future since the speed of SRAM is increasing by 60% a year versus only 7% a year for DRAM [64].

In desktops, the usual approach to adding SRAM is to configure it as a hardware cache. The cache dynamically stores a subset of the frequently used data. Caches have been a success for desktops – a trend that is likely to continue in the future. One reason for their success is that code compiled for caches is portable to different sizes of cache; on the other hand, code compiled for scratch-pad is usually customized for one size of scratch-pad. Binary portability is valuable for desktops, where independently distributed binaries must work on any cache size. In embedded systems, however, the software is usually considered part of the co-design of

the system: it resides in ROM or another permanent storage medium, and cannot be easily changed. Thus, there is really no harm to the binaries being customized to one memory size, as required by scratch pad. Source code is still portable, however: re-compilation with a different memory size is automatically possible in our framework. This is not a problem, as it is already standard practice to re-compile for better customization when a platform is changed or upgraded.

For embedded systems, the serious overheads of caches are less defensible. Caches incur a significant penalty in area cost, energy, hit latency and real-time guarantees. All of these other than hit latency are more important for embedded systems than desktops. A detailed recent study [17] compares caches with scratch pad. Their results are definitive: a scratch pad has 34% smaller area and 40% lower power consumption than a cache of the same capacity. These savings are significant since the on-chip cache typically consumes 25-50% of the processor's area and energy consumption, a fraction that is increasing with time [17]. Even more surprising, the run-time cycle count they measured was 18% better with a scratch pad using a simple static knapsack-based [17] allocation algorithm, compared to a cache. Defying conventional wisdom, they found absolutely no advantage to using a cache, even in high-end embedded systems in which performance is important. With the superior dynamic allocation schemes proposed here, the run-time improvement will be larger. Given the power, cost, performance and real time advantages of scratch-pad, and no advantages of cache, it is not surprising that scratch-pads are the most common form of SRAM in embedded CPUs today (eg: [28, 4, 102, 135, 101]), ahead of caches. Trends in recent embedded designs indicate that the dominance of

scratch-pad will likely consolidate further in the future [120, 17], for regular as well as network processors.

Although many embedded processors with scratch-pad exist, compiling program data to effectively use the scratch-pad has been a challenge. The challenge is different for static data like code, global and stack variables, on one hand, and dynamic data like heap and recursive stack variables, on the other. The basis of this difference lies in the fundamental nature of the two data types and how program behavior affects their utilization. This is explained below.

Recent advances have made much progress in compiling *code, global and stack variables* into scratch-pad memory. Two classes of compiler methods for allocating these objects to scratch-pad exist. First, *static* allocation methods are those in which the allocation does not change at run-time; these include [16, 127, 66, 15, 126] and others not listed here. In such methods, the compiler places the most frequently used variables, as revealed by profiling, in scratch pad. Placing a portion of the stack variables in scratch-pad is not easy – [16] is the first method to solve this difficulty by partitioning the stack into two stacks, one for scratch-pad and one for DRAM. Second, recently proposed *dynamic* methods improve upon static methods by allowing variables to be moved at run-time [136, 132, 84, 140]. Being able to move variables enables tailoring the allocation to each region in the program rather than having a fixed allocation as in a static method. Dynamic methods aim to keep variables that are frequently accessed in a region in scratch-pad during the execution of that region. The methods in [136, 84] explicitly copy variables from DRAM into scratch-pad just before a region in which they are expected to the

frequently accessed. Other variables are evicted to DRAM by explicit copy out instructions to make space for incoming variables. Details concerning these and other existing methods relating to SPM allocation will be presented in Chapter 3.

Allocating dynamic data to scratch-pad has proven far more difficult. Indeed, as far as we know, no one has proposed a successful method to allocate a portion of a program's dynamic data to scratch-pad memory. To see why, it is useful to understand dynamic data and their available analysis techniques; an overview follows. We will focus on heap variables as the main focus of our methods although we have applied similar concepts for recursive stack objects (described later). Heap objects are allocated in programs by dynamic memory allocation routines, such as **malloc** in C and **new** in Java. They are often used to store dynamic data structures such as linked lists, trees and graphs in programs. Many compiler techniques for heap analysis group all heap objects allocated at a single site into a single heap "variable". Additional techniques such as shape analysis have aimed to identify logical heap structures, such as trees. Finally, in languages with pointers, pointer analysis [42, 129] is able to find all possible heap variables that a particular memory reference can access.

Having understood heap variables, let us consider why heap data is difficult to allocate to scratch-pad memory at compile-time. Two reasons for this difficulty are as follows. First, heap variables are usually of *unknown size* at compile-time. For example, linked lists, trees and graphs allocated on the heap typically have a data-dependent number of elements, and thus a compile-time-unknowable size. Thus it is difficult to guarantee at compile-time that the heap variable will fit in scratch-pad.

Such a guarantee is needed for a compiler to place that heap variable in scratch-pad. Second, moving data at run-time, as is required for any dynamic allocation method to scratch-pad, usually leads to the *invalid pointer problem* if the moved data is a heap object. To see why, consider that heap data often contains pointers to other heap data, such as the child pointers in a tree node. When a heap object is moved between scratch-pad and DRAM, all the pointers into it become invalid. Updating all these pointers at run-time is prohibitively expensive since it involves scanning through entire, possibly large, heap structures at each move. Static methods avoid this problem, but lack the better per-region customization of dynamic methods.

Lacking compile-time methods for heap allocation to scratch-pad, people have investigated run-time methods, *i.e.,* methods that decide what to place in scratch-pad only at run-time; however largely they have not been successful. Primary among run-time methods is *software caching* [100, 60]. This class of methods emulate the behavior of a hardware cache in software on the scratch-pad. Since caches decide their contents at run-time, software caching decides the subset of heap data to store in scratch-pad at run-time. Software caching is implemented as follows. A tag consisting of the high-order bits of the address is stored for each cache line in software. Before each load/store, additional instructions are compiler-inserted to mask out the high-order bits of the address, access the tag, compare the tag with the high-order bits and then branch conditionally to hit or miss code. Some methods are able to reduce the number of such inserted overhead instructions [100], but much of it remains, especially for non-scientific programs and for heap data. This implementation points to the primary drawbacks of software caching: the inserted

code before each load/store adds significant overhead, including (i) additional run-time; (ii) higher code size and dollar cost; (iii) higher data size and cost from tags; and (iv) higher power consumption. These overheads, especially for heap data, can easily exceed the gains from locality.

In conclusion, lacking compile-time methods and successful run-time methods for heap allocation to scratch-pad, *heap data is usually not allocated to scratch-pad at all* in modern embedded systems; instead it is placed entirely in DRAM.

Heap allocation method This paper proposes a new dynamic method for allocating a portion of the heap to scratch-pad. The method is outlined in the following three steps. First, it partitions the program into *regions* such that the start and end of every procedure and every loop is the beginning of a new region, which continues until the next region begins. This is not the only possible choice of regions; the reasons for this choice are in section 5.3. Second, straightforward analysis is done to determine the time-order between the regions by finding the set of possible predecessors and successors of each region. Third, copying code is inserted by the compiler at the beginnings of regions to copy in portions of heap variables into the scratch-pad; these portions are called *bins*. A cost-model driven heuristic method is used to determine which variables to copy in and what size their bins should be.

At first glance, the above method is similar in flavor to our compile-time dynamic method for code, global and stack data [132] in that it copies in data when the compiler expects that it will be frequently used in the next region. However its real novelty is seen in how it solves the unknown size problem and the invalid data problem mentioned earlier. How these problems are solved result in virtually every

aspect of the algorithm being different from our earlier method. The solutions to the unknown size problem and the invalid data problem are described in the next two paragraphs.

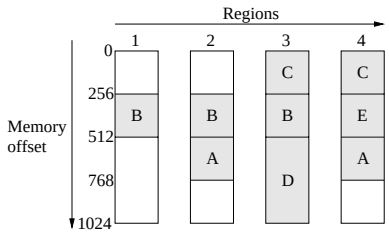
First, our heap method solves the problem of unknown-size heap variables by **not** storing all the elements of a heap allocation site in its SRAM bin, but only a fixed-size *subset*. (From here on "site" is used to mean the objects allocated at a site). This fixed-size portion for each site in scratch-pad is called the *bin* for that site. Fixed-size bins make possible compile-time guarantees that they will fit in scratch-pad. For example consider a linked list having nodes of size 16 bytes and an **unknown** number of nodes. Here, the compiler may allocate a bin of size 192 bytes for the allocation site of the list – this will hold only $192/16 = 12$ nodes from the list. The total number of nodes may be larger, but only twelve are allocated to the bin and the rest to DRAM. A bin is copied into SRAM just before every region where it is accessed (unless it is already in SRAM) and is subsequently evicted before a region where it is not [1] . When a bin is evicted it is maintained as a contiguous block in DRAM; it is copied back later to SPM contiguously if needed. This ensures that the offset of a particular data object inside its bin is not changed during its lifetime, regardless of whether the bin is in SRAM or DRAM.

It is important to understand that objects may be allocated or freed from either memory – *separate free lists are maintained for each bin, and there is a unified free list for heap data that are not in bins.* The bins are moved between

[1]This is the default behavior but it is selectively changed for some regions by the optimizations in section 6.5.

Site	Bin size (bytes)	Regions accessed
A	256	2,4
B	256	1,2,3
C	256	3,4
D	512	3
E	256	4

(a)



(b)

Figure 1.1: Example of heap allocation using our method showing (a) Heap Allocation sites for a program; (b) Memory layout of heap bins after allocation with our method.

SRAM and DRAM, but non-bin data is always in DRAM. New objects from a site are allocated to its bin if space is available, and to DRAM otherwise. Sites having a higher data re-use factor are assigned larger bins to increase the total run time gain from using bins. Figure 1.1(a) is an example showing the five allocation sites for a hypothetical program and bin size and regions-of-access for each site. Four regions 1-4 are assumed in the program, numbered in order of their timestamps (defined in section 5.3).

Second, our heap method solves the problem of invalid pointers by never changing the bin offset or size for any site in the regions it is accessed. For example, figure 1.1(b) shows the bin layout in scratch-pad for the sites in figure 1.1(a), for each of the four regions in the program. It shows that the offset of each bin is always the same when it is present. For example, site A is allocated at the same offset 512 in both regions 2 & 4 it is accessed. An entire bin may be evicted to DRAM in a region

it is not accessed (as revealed by pointer analysis). For example, site A is copied to DRAM in region 3. Moving a bin to DRAM temporarily results in invalid pointers that point to objects in the bin, but those invalid pointers are never dereferenced as they occur only during regions that pointer analysis has proven to not access the site.

Our heap method effectively improves run-time for three reasons. First, like a cache it allocates more frequently used data to SRAM. This is achieved by assigning larger bins to sites with high frequency-per-byte of access. Heap area is traded off with global and stack data as well – the frequency-per-byte of variables of all types (from profile data) are compared to determine which ones are actually copied to SRAM [2] . Any variable is placed in scratch-pad only if the cost model estimates that the benefits of locality exceed the cost of copying. Second, like caching our heap method is able to change the contents of the SRAM at runtime to match the requirements of different phases of the program. The allocation is dynamic, but is decided at compile-time. Third, unlike software caching, our method has no tags and no per-memory-access overhead.

Recursive Functions Recursion in computer programming defines a function in terms of itself. Recursion is deeply embedded in the theory of computation, with the theoretical equivalence of mu-recursive functions and Turing machines at the foundation of ideas about the universality of the modern computer. A good example application of recursion is in parsers for programming languages. The

[2]The use of frequency-per-byte itself is not new. It has been used earlier for allocating global and stack variables to SPM [126, 111]. The novelty in this paper is in the solution to the unknown size and invalid pointer problems; this allows heap data to be placed in SPM.

great advantage of recursion is that an infinite set of possible sentences, designs or other data can be defined, parsed or produced by a finite computer program.

Unfortunately, even the best compiler analysis tools are unable to place a bound (except in trivial cases) on the total size of stack memory allocated by a recursive function at run-time, as it strictly depends on the inputs applied. This presents a serious problem for existing SPM allocation schemes which only handle static program data. Using concepts obtained from our methods for heap data allocation, we have developed the first methods able to allocate recursive stack functions to SPM at run-time. By treating individual function invocations like individual heap objects, we are able to make minor modifications to our framework to support recursive stack optimization. We will later present results showing significant improvements for applications making heavy use recursive functions.

Comparison with caches The primary measure of success of our heap method is **not** its performance vs. hardware caches, but vs. all-DRAM heap allocation, the only existing method for scratch-pad. (Software caching has not been a success). There are a great many chips that have scratch-pad memory (SPM) and DRAM but no data cache; examples include low-end CPUs [105, 6, 115], mid-grade CPUs [7, 14, 12, 67, 72] and high-end CPUs [8, 68, 104]. We found at least 80 such embedded processors with SPM and DRAM but no D-cache in our search but have listed only the above eleven for lack of space. Thus our method delivers its full promised benefits for a great variety of chips. It is nevertheless interesting to see a quantitative comparison of our method for SPM against a cache. Section 9.4 presents such a comparison. It shows that our compile-time method is comparable

to or out-performs a cache of the same area in both run-time and energy usage for our benchmark suite.

1.1 Organization of Thesis

This thesis is organized in the following manner. The first four chapters constitute the background and related material for understanding the contributions of this thesis. Chapter 2 presents background material on embedded systems with a focus on typical hardware and software approaches for memory. The concept of static and dynamic data for compiler-based code optimization is also presented in this chapter. Chapter 3 presents a thorough review of recent research concerned with SPM allocation as well as related optimization concepts. Chapter 5 presents the best existing SPM allocation method for code, global and stack data, which is used in conjunction with our dynamic data method for a comprehensive program optimization approach. While the material in this chapter is not a new contribution from this thesis, it is presented as essential reading for a full understanding of our allocation methods.

The main contributions of this thesis are presented in Chapters 5– 7. We have decided to first present our core method for optimizing typical heap data before expanding on this for our handling of other types of dynamic data. We present our core method for analyzing and understanding heap data in Chapter 5. Chapter 6 presents a step-by-step explanation of our algorithm for SPM allocation. Once the core method for heap data has been presented, Chapter 7 completes our presentation

with discussion of the extensions we have developed for all other program objects currently not handled by existing SPM allocation schemes.

The final four chapters of this thesis present our supporting material. Chapter 8 discusses the development and simulation methodology employed to properly host and evaluate our compiler methods. The results obtained from a wide range of experiments are presented in Chapter 9 with focus on interesting scenarios. Chapter 10 concludes the thesis by summarizing our findings. Finally, Chapter 10 is an appendix containing brief results of interest not explicitly discussed in Chapter 9.

Chapter 2

Embedded Systems and Software Development

This chapter will primarily present a brief review of those concepts on which we base our method for dynamic memory allocation of SPM for embedded systems. The chapter begins with a review of what exactly constitutes an embedded system, with emphasis placed on typical hardware configurations for these systems. This is followed by some background on the C programming language[43], which is by far the dominant language for embedded systems development. We will discuss both language and compiler specific material as they apply to optimizing memory allocation for compiled applications.

To perform optimal memory allocation for a program requires both knowledge of the program and information on the target machine that will execute the application. This in turn requires advanced compiler techniques involving many areas from both the hardware engineering and computer science disciplines. For example, in order for compilers to make the best decisions when creating intermediate implementations of high-level language programs, they require complete knowledge of the language being compiled and its associated memory and semantic details. Further, the final back-end requires lower-level information on the hardware and instructions available for a target platform in order to generate optimized assembly programs. This is a daunting task for modern compilers and developers to handle

in its entirety without some basic concepts in place. The following overview should help the reader understand the fundamentals behind our compiler directed memory allocation approach.

2.1 Embedded Systems

At its simplest, an embedded system can be defined as any processing system that has been embedded inside of a larger product. For most engineers, embedded systems are more strictly defined as dependable, efficient, low-cost processing modules which are used as small components for a larger and more complicated technology. Regardless of the definition, embedded systems have become pervasive on modern society as advancements in technology have fostered an era of ubiquitous computing. Embedded devices have become a part of everyday life for most people and compose critical components in products such as automobiles, aircraft, mobile phones, media players and medical devices, among many more everyday objects. This proliferation in embedded systems has come about due to the advances in computer microprocessor technologies which have provided the boosts in size, complexity and efficiency needed.

As technology advanced, the exact definition of what constitutes an embedded system versus a traditional computer has become murky and difficult to pinpoint in some applications. General computing is generally divided among super-computers at its high-end and powerful servers and mainframes at its middle-level of performance. At the low-end of the traditional computer field lies the personal computer,

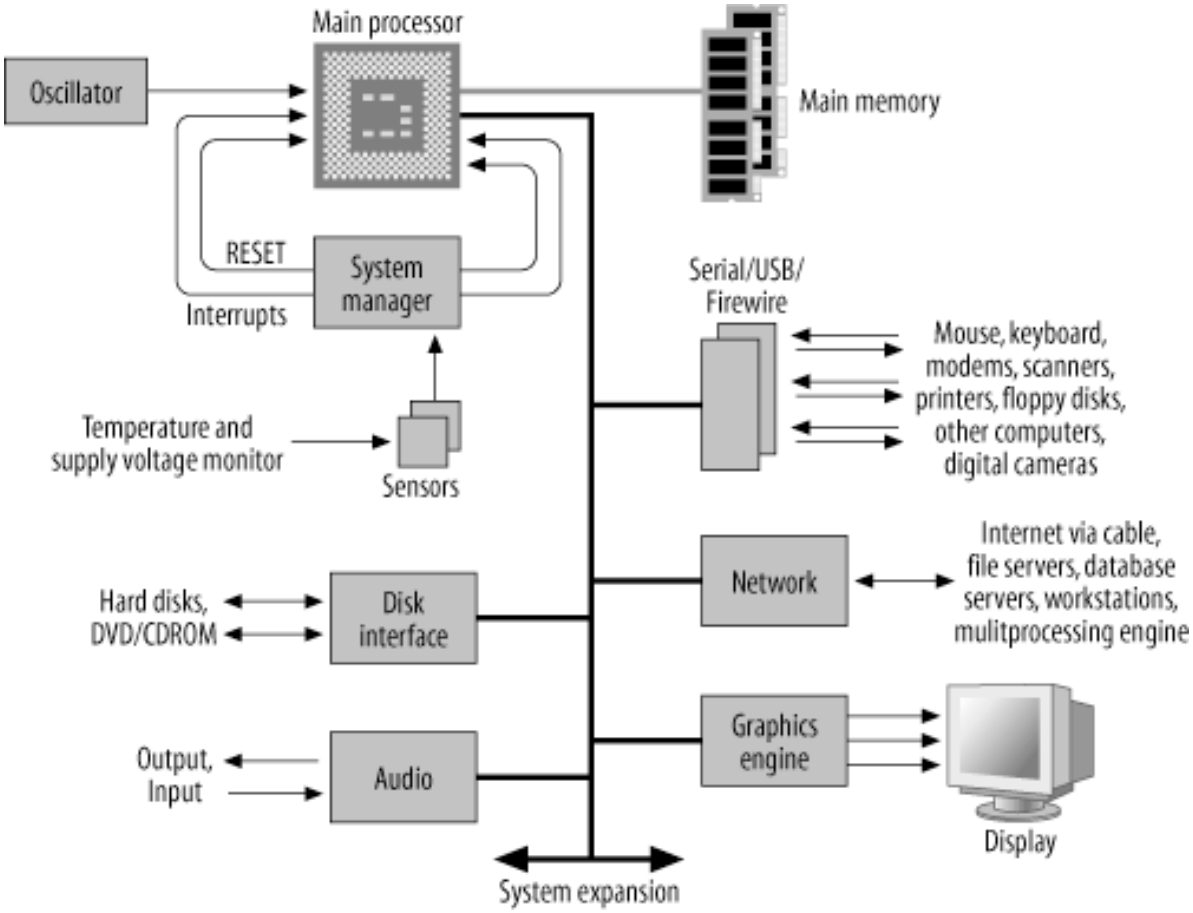


Figure 2.1: A block diagram view of a typical consumer computer.

most commonly found in the form of a desktop or laptop machine. A diagram of the components making up a typical consumer computer is shown in 2.1.

From this figure, a typical PC has a large main memory to hold the operating system, applications, and data, and an interface to mass storage devices (disks and DVD/CD-ROMs). It has a variety of I/O devices for user input (keyboard, mouse, and audio), user output (display interface and audio), and connectivity (networking and peripherals). The fast processor requires a system manager (BIOS) to monitor its core temperature and supply voltages, and to generate a system reset.

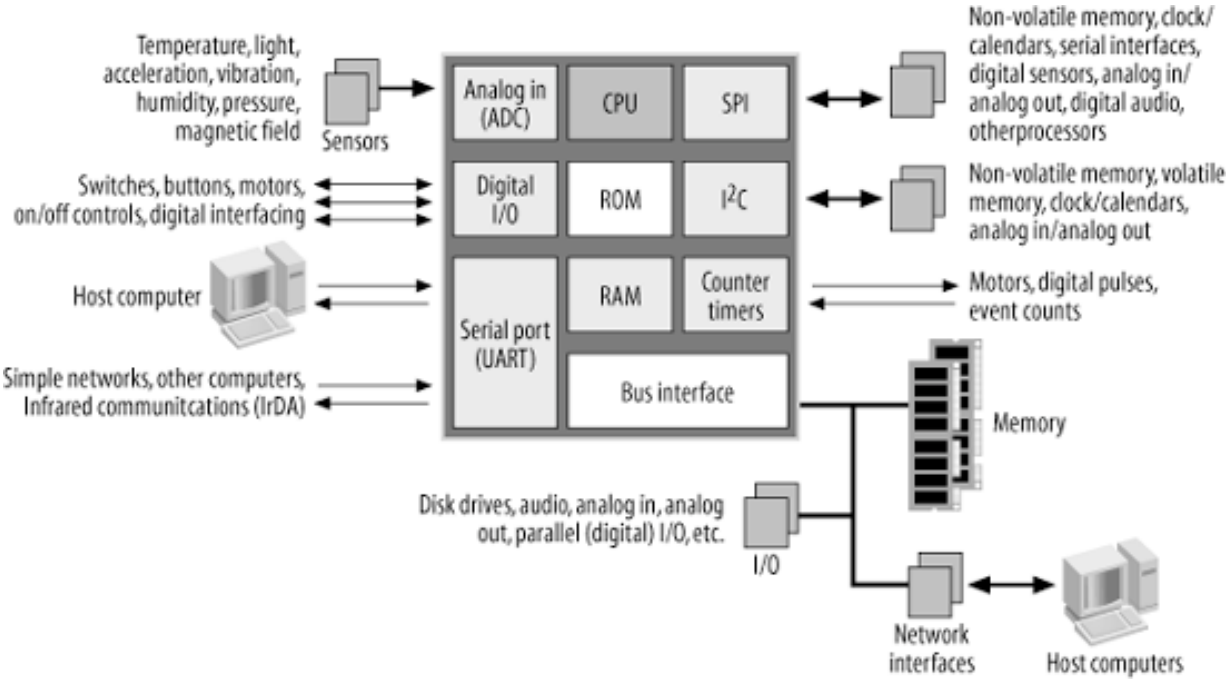


Figure 2.2: A block diagram view of a typical embedded computer.

Large-scale embedded computers may also take the same form. For example, they may act as a network router or gateway, and so will require one or more network interfaces, large memory, and fast operation. They may also require some form of user interface as part of their embedded application and, in many ways, may

simply be a conventional computer dedicated to a specific task. Thus, in terms of hardware, many high-performance embedded systems are not that much different from a conventional desktop machine.

Smaller embedded systems use microcontrollers as their processor, with the advantage that this processor will incorporate much of the computer's functionality on a single chip. An arbitrary embedded system, based on a generic microcontroller, is shown in Figure 2.2. The microcontroller has, at a minimum, a CPU, a small amount of internal memory (ROM and RAM), and some form of I/O, which is implemented within a microcontroller as subsystem blocks. These subsystems provide the additional functionality for the processor and are common across many processors.

Many types of memory devices are available for use in modern computer systems. Most software developers think of memory as being either random-access (RAM) or read-only (ROM). Not only are there several distinct subtypes of each but the past decade has seen an upsurge of a third class of hybrid memories. In a RAM device, the data stored at each memory location can be read or written as desired. In a ROM device, the data stored at each memory location can be read at will, but never written. In some cases, it is possible to overwrite the data in a ROM-like device. Such devices are called hybrid memories because they exhibit some of the characteristics of both RAM and ROM. Figure 2.1 provides a classification system for the memory devices that are commonly found in embedded systems.

Types of RAM There are two important memory devices in the RAM family: SRAM and DRAM. The main difference between them is the lifetime of the data

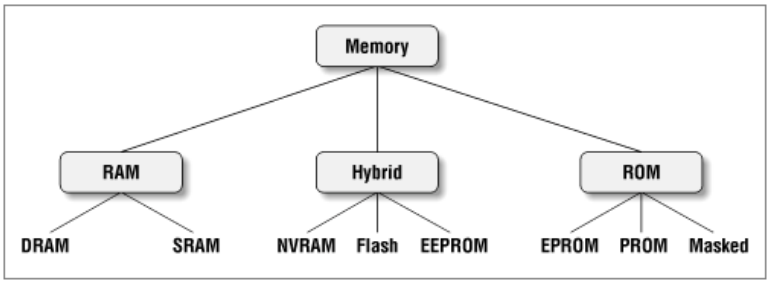


Figure 2.3: Memory types commonly used in embedded systems.

stored. SRAM (static RAM) retains its contents as long as electrical power is applied to the chip. However, if the power is turned off or lost temporarily then its contents will be lost forever. DRAM (dynamic RAM), on the other hand, has an extremely short data lifetime-usually less than a quarter of a second. This is true even when power is applied constantly, which is typically the case for DRAM memory organizations. DRAM thus tends to incur a much higher power as well as access time due to its design.

When deciding which type of RAM to use, a system designer must also consider access time and cost. SRAM devices offer extremely fast access times (approximately four times faster than DRAM) but are much more expensive to produce. Generally, SRAM is used only where access speed is extremely important. A lower cost per byte makes DRAM attractive whenever large amounts of RAM are required. Many embedded systems include both types: a small block of SRAM (a few hundred kilobytes) along a critical data path and a much larger block of DRAM (in the megabytes) for everything else.

Types of ROM Memories in the ROM family are distinguished by the methods used to write new data to them (usually called programming) and the number

of times they can be rewritten. This classification reflects the evolution of ROM devices from hardwired to one-time programmable to erasable-and-programmable. A common feature across all these devices is their ability to retain data and programs physically and not electronically, saving information even when power is not applied.

The very first ROMs were hardwired devices that contained a preprogrammed set of data or instructions. The contents of the ROM had to be specified before chip production, so the actual data could be used to arrange the transistors inside the chip. Hardwired memories are still used, though they are now called "masked ROMs" to distinguish them from other types of ROM. The main advantage of a masked ROM is its low production cost. Unfortunately, the cost is low only when hundreds of thousands of copies of the same ROM are required and used to store data that will never be modifiable.

One step up from the masked ROM is the PROM (programmable ROM), which is purchased in an unprogrammed state. The process of writing data to a PROM involves the use of specialized device programming hardware. The programmer attaches to the PROM device and writes data to the device one word at a time by applying electrical charges to the input pins of the chip. Once a PROM has been programmed this way, its contents can never be changed as the electrical charges fuse the internal transistor logic gates open or closed. If the code or data stored in the PROM must be changed, the current module must be discarded and replaced with a new memory module. As a result PROMs are also known as One-Time Programmable (OTP) devices.

An EPROM (Erasable-and-Programmable ROM) is a memory type that is

programmed in the same manner as a PROM, but that can be erased and reprogrammed repeatedly. Depending on the silicon production process involved, these memory modules are created using different structures able to store data bits in the chip wafer that can also be reset using external stimuli in the form of radiation. Like PROMs, EPROMs must be erased completely and programming occurs on the entire contents of memory each time. Though more expensive than PROMs, their ability to be reprogrammed makes EPROMs an essential part of the software development and testing process as well as for firmware which will be upgraded occasionally.

Hybrid Types As memory technology has matured in recent years, the line between RAM and ROM devices has blurred. There are now several types of memory that combine the best features of both. These devices do not belong to either group and can be collectively referred to as hybrid memory devices. Hybrid memories can be read and written as desired, like RAM, but maintain their contents without electrical power, just like ROM. Two of the hybrid devices, EEPROM and Flash, are descendants of ROM devices; the third, NVRAM, is a modified version of SRAM.

EEPROMs are electrically-erasable-and-programmable. Internally, they are similar to EPROMs, but the erase operation is accomplished electrically, rather than by exposure to more cumbersome methods like ultraviolet light. Any byte within an EEPROM can be erased and rewritten individually instead of requiring the entire module to be formatted. Once written, the new data will remain in the device forever-or at least until it is electrically erased. The trade-off for this improved functionality is mainly its higher cost. Write cycles are also significantly longer than writes to a RAM, rendering EEPROM a poor choice for main system

memory.

Flash memory is the most recent advancement in memory technology. It combines all the best features of the memory devices described thus far. Flash memory devices are high density, low cost, nonvolatile, fast (to read, but not to write), and electrically reprogrammable. These advantages are overwhelming and the use of Flash memory has increased dramatically in embedded systems as a direct result. From a software viewpoint, Flash and EEPROM technologies are very similar. The major difference is that Flash devices can be erased only one sector at a time, not byte by byte. Typical sector sizes are in the range of 256 bytes to 16 kilobytes. Despite this disadvantage, Flash is much more popular than EEPROM and is rapidly displacing many of the ROM devices as well.

The third member of the hybrid memory class is NVRAM (nonvolatile RAM). Non-volatility is also a characteristic of the ROM and hybrid memories discussed earlier. However, an NVRAM is physically very different from those devices. An NVRAM is usually just an SRAM with a battery backup. When the power is turned on, the NVRAM operates just like any other SRAM. But when the power is turned off, the NVRAM draws just enough electrical power from the battery to retain its current contents. NVRAM is fairly common in embedded systems. However, it is very expensive-even more expensive than SRAM-so its applications are typically limited to the storage of only a few hundred bytes of system-critical information that cannot be stored in any better way.

We summarize our review of common embedded memory technologies with a table comparing their distinguishing features in Figure 2.1.

Memory Type	Volatile?	Writeable?	Erase Size	Erase Cycles	Relative Cost	Relative Speed
SRAM	yes	yes	byte	unlimited	expensive	fast
DRAM	yes	yes	byte	unlimited	moderate	moderate
Masked ROM	no	no	n/a	n/a	inexpensive	fast
PROM	no	once, with programmer	n/a	n/a	moderate	fast
EPROM	no	yes, with programmer	entire chip	limited (see specs)	moderate	fast
EEPROM	no	yes	byte	limited (see specs)	expensive	fast to read, slow to write
Flash	no	yes	sector	limited (see specs)	moderate	fast to read, slow to write
NVRAM	no	yes	byte	none	expensive	fast

Figure 2.4: Comparison of the salient features for memory types common to embedded systems.

2.2 Intel StrongARM Microprocessor

Having overviewed memory designs in the previous section, this section will now discuss the hardware organization of a typical embedded processor. As a case study, an example processor from the Intel StrongARM family will be highlighted in this section and used throughout the rest of the thesis as our reference platform. The Intel StrongARM Microprocessor (SA-1110) is a highly integrated communications microcontroller that incorporates a 32-bit StrongARM RISC processor core, system support logic, multiple communication channels, an LCD controller, a memory and PCMCIA controller, and general-purpose I/O ports. The SA-1110 provides power efficiency, low cost, and high performance in an embedded package with flexible memory options. Typical embedded deployments will contain a StrongARM cpu along with a variety of memory configurations, such as a cache(SRAM) with DRAM as main memory, or an SPM-only system with only an SPM(SRAM) module as main memory (cache modules disabled). Figure 2.5 shows a block diagram of the component modules for the SA-1110.

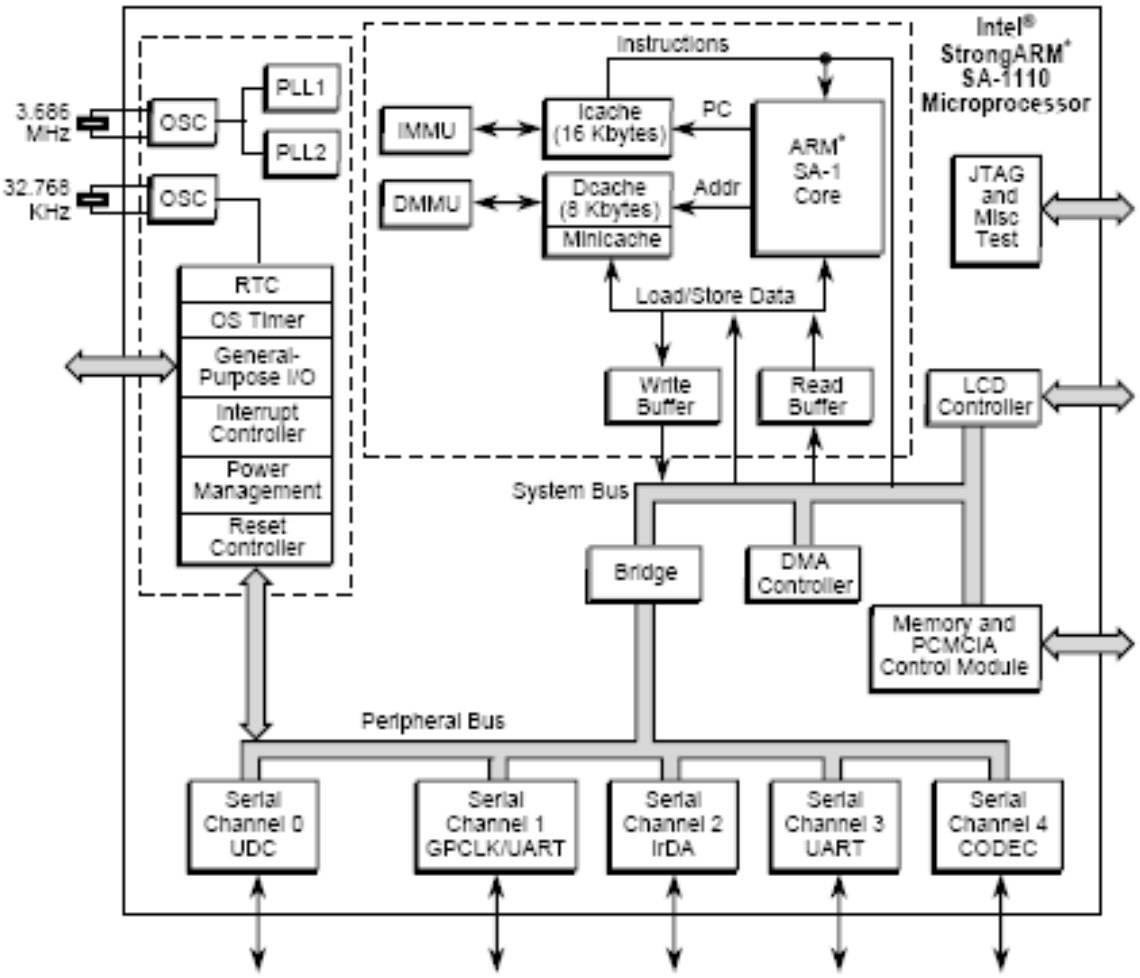


Figure 2.5: Block diagram for an Intel StrongARM within a complete embedded system.

The SA-1110 is a general-purpose, 32-bit RISC microprocessor with a clock speed adjustable up to 206 MHz. The embedded processor has a configurable instruction and data cache, memory-management unit (MMU), and read/write buffers for efficient data processing. The memory bus interfaces to many device types including SRAM(SPM), synchronous DRAM (SDRAM) and Flash(EEPROM). The StrongARM is software compatible back to the ARM V4 architecture processor family and can be used with ARM coprocessor chips such as I/O, memory, and video components. The ARM instruction set is a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors that need sophisticated compiler technology to manage complicated instruction interdependencies. The SA-1110 has been designed to run at a reduced voltage to minimize its power requirements. This makes it a good choice for portable applications where both of these features are essential.

Deployments using the StrongARM processor can choose from memory modules such as DRAM or SRAM to use as main memory, as well as FLASH for code storage. All memory modules are cacheable in the StrongARM using its on-board cache hardware. The SA-1110 contains a 16 Kb instruction cache as well as an 8Kb data cache. Each module can be enabled or disabled via a control register, with fine-grain management available using the MMU to control which addresses are cacheable. Deployments concerned with power and realtime guarantees often choose to disable the cache modules and operate the device using only SRAM and ROM modules.

Embedded Design Trends The StrongARM was designed with the same goals as all other embedded processors, from low-end 8-bit and 16-bit processors that may cost less than a dollar,to high-end embedded processors (that can execute a billion instructions per second and cost hundreds of dollars) for the newest portable video game system. All embedded systems are ultimately designed to be cheap, fast and power-efficient. Although the range of computing power in the embedded computing market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal is often meeting the performance need at a minimum price, rather than achieving higher performance at a higher price.

Two other key characteristics exist in many embedded applications: the need to minimize memory and the need to minimize power. In many embedded applications, the memory can be substantial portion of the system cost, and memory size is important to optimize in such cases. Sometimes the application is expected to fit totally in the memory on the processor chip; other times the applications needs to fit totally in a small off-chip memory. In any event, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application. Larger memories also mean more power, and optimizing power is often critical in embedded applications. When hardware methods alone are insufficient, designers can still turn to software-only and hybrid methods to squeeze the most performance from embedded hardware to reduce cost and power while maximizing performance.

2.3 Embedded Software Development

One of the few constants across almost all embedded systems is the use of the C programming language. More than any other, C has become the language of embedded programmers. This has not always been the case, and it will not continue to be so forever. However, at this time, C is the closest thing there is to a standard in the embedded world. Because successful software development is so frequently about selecting the best language for a given project, it is surprising to find that one language has proven itself appropriate for both 8-bit and 64-bit processors; in systems with bytes, kilobytes, and megabytes of memory; and for development teams that consist of from one to a dozen or more people. Yet this is precisely the range of projects in which C has thrived.

Of course, C is not without advantages. It is small and fairly simple to learn, compilers are available for almost every processor in use today, and there is a very large body of experienced C programmers. In addition, C has the benefit of processor-independence, which allows programmers to concentrate on algorithms and applications rather than on the details of a particular processor architecture. However, many of these advantages apply equally to other high-level languages, which has led many researchers in embedded systems to wonder why has C succeeded where so many other languages have mostly failed.

Perhaps the greatest strength of C,and what sets it apart from languages like Java, C++, Pascal and FORTRAN, is that C is a very "low-level" high-level language. As we shall see throughout the book, C gives embedded programmers

an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages. The "low-level" nature of C was a clear intention of the language's creators. In fact, Kernighan and Ritchie included the following comment in the opening pages of their book The C Programming Language :

> *C is a relatively "low level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.*

Few popular high-level languages can compete with C in the production of compact, efficient code for almost all processors. And, of these, only C allows programmers to interact with the underlying hardware so easily. The two other popular languages in use today for desktop and high-end embedded systems are C++ and Java. Both are object-oriented programming (OOP) languages which are more sophisticated than the C language in some ways, but that very complexity tends to be problematic for embedded systems development. Java and many other OOP languages enforce the use of garbage collectors to manage dynamically allocated memory. Garbage collectors greatly decrease the real-time guarantees for that system, an important consideration for many embedded designers. One of the other strengths of OOP languages lies in their abstraction of programs in terms of objects, which in turn requires more layers of translation during compilation to produce machine code. While allowing for more powerful and concise programming, these layers cause a disconnect when programming an embedded system that requires low-level

control over addressing to correctly use memory mapped devices, data allocation that is known and completely controllable by the programmer, as well as a myriad of low-level software interactions designers must employ when working with limited hardware platforms.

C-Language Allocation Basics Having understood some of the reasons for C's popularity in embedded development, a discussion of how C handles data allocation of program variables is also beneficial to understanding how compilers translate a C program into final machine code for a target processor. Traditional programming languages such as C and C++ maintain a list of attributes for each variable declared that minimally consists of its name, type, size, value, storage class, scope and linkage. Fundamental variable types and sizes are specified by that language's implementation, with names and values specified through user assigned identifiers in a valid program. Variable types in a language define its size, use and operator definitions for correct statement construction in a programming language. After a programmer declares a variable instance to possess a certain name and type, the instance declaration will also require a storage class specifier which dictates how its processed by the compiler for data allocation in the final machine code.

A variable's storage class specifier determines its storage class, scope and linkage in a compiler. The storage class also determines the lifetime of a variable during execution since variables can exist briefly, be repeatedly created and destroyed or exist and live throughout the entirety of program execution. The automatic storage class has automatic storage duration, and will create a declared variable when its declaration block is entered, exist while its active and destroy the variable once

the block has been exited. Local variables declared inside functions have automatic storage duration by default. The static storage class includes those variables and functions which exist from the point at which the program begins execution. For variables, storage is allocated and initialized upon beginning program execution and for functions the name of the function exists from program inception in a global symbol table. Although static duration variables and functions are created and exist throughout the entire program, their particular scope in the C-language will determine whether they may actually be referenced at any given point during program execution and the machine code generated will also reflect this.

Having looked at how the C-language dictates requirements for the compiler generation of program memory objects, we can now give a higher-level programmer view of C language memory objects which most people will be familiar with. Global variables are created by placing variable declarations outside any function definition by the compiler, and they retain their values throughout execution of the program. Global variables and functions can be referenced by any function that follows their declarations or definitions in the file. This is one of the main reasons for specifying function prototypes in included definition files such as 'stdlib.h', which allows the programmer to use a function interface properly even if that function body is present in another source file. This becomes of extreme importance when multiple source files are to be compiled together to produce an executable, as even a single source file program will likely use functions implemented in a system library file provided for low-level I/O support. A compiler must be able to differentiate between local and included source instance identifiers for program elements, and know how to

link separately compiled bundles of code together properly to produce a working executable. For any serious high-level implementation, this becomes of paramount importance and proper use of 'extern' and 'static' identifiers allow programmer's control over individual components as they relate to each other across multiple source files. By declaring a variable instance as 'extern', this tells the compiler that the variable exists in another file that will be included in the final program by the linker. Similarly, external function prototypes allow the linker to find the proper function implementation in another compiled object file.

The other static duration identifier is appropriately named 'static', and is most commonly used for local variable declarations which need to retain their values and storage throughout program execution. Local variables declared static are still bound to their local function scope, but retain their value when the function is exited and still available the next time it is entered. Local variables that are not declared static will be created upon entering a block with an initial value if specified and destroyed upon leaving that block. Static takes on further meaning when applied in a multi-file program being compiled. Normally global variables and functions have external linkage and can be accesses from other files if those files contain proper declarations and function prototypes. By adding the 'static' keyword to such declarations, this restricts the scope of such instances to the file in which if is defined, preventing its use by any function that is not defined in the same file. This allows a programmer to enforce the principle of least privilege in a C program, which consequently helps restrict compiler analysis to proper scopes and lifetimes as specified.

An important component of a declared data type concerns variable scope, which determines from where in the program that variable's identifier can be referenced correctly. For example, when a local variables is declared in a block, it can be referenced only in that block or in blocked nested within that program block. C provides function, file, block and function-prototype scope identifiers for a programmer to make use of. Only code labels have function scope and can be referenced anywhere inside the function they appear, but not outside of it. These generally appear in the form of case switches and 'goto' statements. Any variables declared outside a function have file scope, and their instances can be referenced from the point at which they are declared until the end of the file. Global variables, function declarations and function prototypes placed outside functions, such as include files, have file scope. Any variables declared inside a code block have block scope, denoted in C by enclosing the block inside of braces, eg: "{ int a; }". Local variables declared at the beginning of a function have block scope including the parameters passed to the function and any block can contain variable declarations. Any nested blocks that contain variables with the same instance names as those present in outer blocks will only reference the instances in the local block of scope. Even local variables that are declared static will only have block scope even though they exist from the time the program begins execution, illustrating that storage duration does not necessarily determine the scope of a variable. Finally, function-prototype scope only applies to the declarations inside the parameter list of a function prototype, for which the names are ignored but the types must match and function implementations in the file being compiled.

The largest and longest lived scope is that of global variables. Truly globally defined variables in C will have their label and associated memory requirement present in all files for a program being compiled, while those with only file-level scope will only be considered in scope for that file containing its declaration. The presence of procedures in the form of function blocks requires a more limited scope to be declared for stack variables while they are live. Locally scoped variables are those variables created for use by a function by occupying memory on the system's stack space. Creation of local stack variables occurs upon entering a procedure and are removed from the stack once it has been returned from or exited.

The nature of imperative languages such as C allow functions to call other functions, including themselves, and as execution progresses, a varying amount and number of locally scoped variables may live on the stack at any given time. These are not nearly as easy to predict and analyze for compiler optimization purposes as compared to global variables, since they may depend heavily on the control flow execution path dynamically taken through a program at run-time. Furthermore, compiler translation and optimization passes will affect how local variables are automatically created upon reaching their declaration block, and passes such as register allocation will decide final memory storage among registers and stack memory.On the other hand, global variables may be considered static in size and allocation terms, just as stack variables can be considered to be static in size and dynamic in allocation terms depending on the program path visited during execution.

The third and final type of variables in C are heap variables. When program's require a dynamic size and allocation method to efficiently handle inputs,

most platforms provide access to dynamic memory management systems through OS-level application calls. Dynamic memory managers maintain information on available free and used memory dedicated to heap memory. Indeed, the flexibility this buys programmers is in allowing size-efficient data structure creation as well as complete control over lifetimes of such structures that does not have to correlate with particular whole function lifetimes. This flexibility comes with a cost, and memory pointers must be used to access heap variables in addition to the memory management that must be performed explicitly by an operating system or another user program. Embedded systems usually require very fine-grain control over machine code running on limited hardware, so the loss of flexibility has minimal impact for most developers.

2.4 C Language Compilers

Figure 2.4 illustrates how a typical compiler platform can take various input files and generate appropriate output files to ultimately be used in building an executable image for a target processor. The developer writes the program in the C/C++ source files and header files. Some parts of the program can be written directly in assembly language when fine-grain control is needed, and are produced in the corresponding assembly source files. The developer creates a "Makefile" for use with the "make" utility to facilitate an environment that can easily track the file modifications and invoke the compiler and the assembler to rebuild the source files when necessary. From these source files, the compiler and the assembler produce

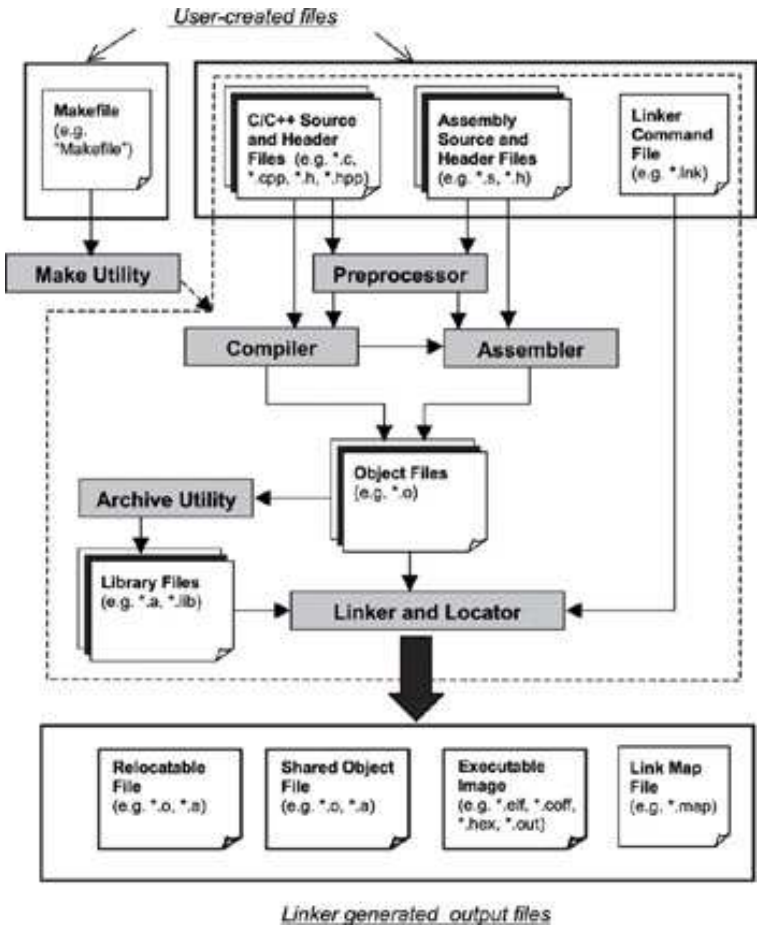


Figure 2.6: The typical compilation process for building an embedded executable from source files.

object files that contain both machine binary code and program data. The archive utility concatenates a collection of object files to form a library. The linker takes these object files as input and produces either an executable image or an object file that can be used for additional linking with other object files. The linker command file instructs the linker on how to combine the object files and where to place the binary code and data in the target embedded system.

The main function of the linker is to combine multiple object files into a larger relocatable object file, a shared object file, or a final executable image. In a typical program, a section of code in one source file can reference variables defined in another source file. A function in one source file can call a function in another source file. The global variables and non-static functions are commonly referred to as global symbols. In source files, these symbols have various names, for example, a global variable called "foobar" or a global function called "functionA". In the final executable binary image, a symbol refers to an address location in memory. The content of this memory location is either data for variables or executable code for functions.

Of course, each processor has its own unique ISA, so it is important to choose a compiler that is capable of producing programs for that specific processor. In the embedded systems case, this compiler almost always runs on a host computer. It simply does not make sense to execute the compiler on the embedded system itself. A compiler such as this-that runs on one computer platform and produces code for another-is called a cross-compiler. The use of a cross-compiler is one of the defining features of embedded software development. The GCC compiler and assembler can

be configured as either native compilers or cross-compilers. As cross-compilers these tools support an impressive set of host-target combinations including state of the art support for the ARM architecture.

The job of a compiler is mainly to translate programs written in some human-readable language into an equivalent set of binary code for a particular processor. In that sense, an assembler is also a compiler but one that performs a much simpler one-to-one translation from one line of human-readable mnemonics to the equivalent binary opcode. There are good reasons for using a high-level language, yet programmers often write directly in assembly language. Assembly and machine code, because they are "hand-written," can be finely tuned to get optimum performance out of the processor and computer hardware. This can be particularly important when dealing with time-critical operations with I/O devices. Furthermore, coding directly in assembly can sometimes (but not always) result in a smaller code space. If a programmer is trying to cram complex software into a small amount of memory and needs that software to execute quickly and efficiently, assembly language may be their best (and only) choice. The drawback, of course, is that the software is harder to maintain and has zero portability to other processors. A good software engineer can create more efficient code than the average optimizing C compiler; however, a good optimizing compiler will probably produce tighter code than a mediocre assembly-language software engineer.

Compiler Optimization

Code optimization refers to the techniques used by a compiler to improve the execution efficiency of the generated object code. It involves a complex analysis of

the intermediate code and the performance of various transformations; but every optimizing transformation must also preserve the semantics of the program. A compiler should not attempt any optimization that would lead to a change in the program's semantics.

Optimization can be machine-independent or machine-dependent. Machine-independent optimizations can be performed independently of the target machine for which the compiler is generating code; the optimizations are not tied to the target machine's specific platform or language. Examples of machine-independent optimizations are elimination of loop invariant computation, induction variable elimination, and elimination of common subexpressions. Machine-dependent optimization requires knowledge of the target machine. An attempt to generate object code that will utilize the target machine's registers more efficiently is an example of machine-dependent code optimization. The process of code optimization is somewhat of a misnomer; even after performing various optimizing transformations, there is no guarantee that the generated object code will be optimal. Hence, a compiler actually performs code improvement. When attempting any optimizing transformation, the following criteria should be applied. First, the optimization should capture most of the potential improvements without an unreasonable amount of effort. Second, the optimization should be such that the meaning of the source program is preserved. Finally, the optimization should, on average, reduce the time and space expended by the object code.

Code generation is the last phase in the compilation process. Being a machine-dependent phase, it is not possible to generate good code without considering the

details of the particular machine for which the compiler is expected to generate code. Even so, a carefully selected code-generation algorithm can produce code that is twice as fast as code generated by an ill-considered code-generation algorithm. Code generated by using simple statement-by-statement conversion strategies contain redundant instructions and suboptimal constructs. Therefore, to improve the quality of the target code, optimization is required. Peephole optimization is an effective technique for locally improving the target code. Short sequences of target code instructions are examined and replaced by faster sequences wherever possible. Other optimizations can account for larger regions of program code to perform transformations affecting different program areas, such as function inlining and outlining.

One of the important tasks that a compiler must perform is to allocate the resources of the target machine to represent the data objects that are being manipulated by the source program. That is, a compiler must decide the run-time representation of the data objects in the source program. Source program run-time representations of the data objects, such as integers and real variables, usually take the form of equivalent data objects at the machine level; whereas data structures, such as arrays and strings, are represented by several words of machine memory.

The strategies that can be used to allocate storage to the data objects are determined by the rules defining the scope and duration of the names in the programming language. The simplest strategy is static allocation, which is used in languages like FORTRAN. With static allocation, it is possible to determine the run-time size and relative position of each data object during compilation. A more-

complex strategy for dynamic memory allocation that involves stacks is required for languages that support recursion: an entry to a new block or procedure causes the allocation of space on a stack, which is freed on exit from the block or procedure. An even more-complex strategy is required for languages, which allows the allocation and freeing of memory for some data in a non-nested fashion. This storage space can be allocated and freed arbitrarily from an area called a "heap". Therefore, implementation of languages like PASCAL and C allow data to be allocated under program control. The run-time organization of memory as viewed by many compilers will be as shown in Figure 2.7. The run-time storage has been subdivided to hold the generated target code and the data objects, which are allocated statically for the stack and heap. The sizes of the stack and heap can change as the program executes.

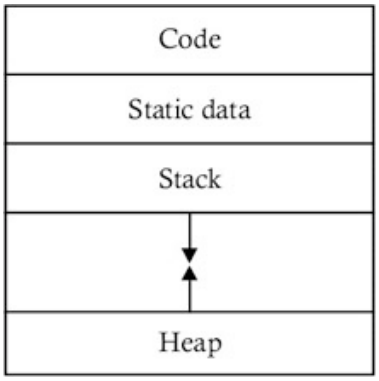


Figure 2.7: The typical compiler view of a program memory layout.

2.5 Heap Data Allocation

Figure 2.8 shows that the program code, program data, and system stack occupying physical memory after typical program initialization completes. Either

the RTOS or the kernel typically uses the remaining physical memory for dynamic memory allocation at run-time. This memory area is called the heap . Memory management in the context of this chapter refers to the management of a contiguous block of physical memory, although the concepts introduced in this section apply to the management of non-contiguous memory blocks as well. These concepts also apply to the management of various types of physical memory. In general, a memory management facility maintains internal information for a heap in a reserved memory area called the control block. Typical internal information includes the starting address of the physical memory block used for dynamic memory allocation, the overall size of this physical memory block, and the allocation table that indicates which memory areas are in use, which memory areas are free, and the size of each free region.

This section examines aspects of heap memory management through an example implementation of the malloc and free functions for an embedded system. In the example implementation, the heap is broken into small, fixed-size blocks. Each block has a unit size that is power of two to ease translating a requested size into the corresponding required number of units. In this example, the unit size is 32 bytes. The dynamic memory allocation function, malloc, has an input parameter that specifies the size of the allocation request in bytes. Malloc allocates a larger block, which is made up of one or more of the smaller, fixed-size blocks. The size of this larger memory block is at least as large as the requested size; it is the closest to the multiple of the unit size. For example, if the allocation requests 100 bytes, the returned block has a size of 128 bytes (4 units x 32 bytes/unit). As a result, the

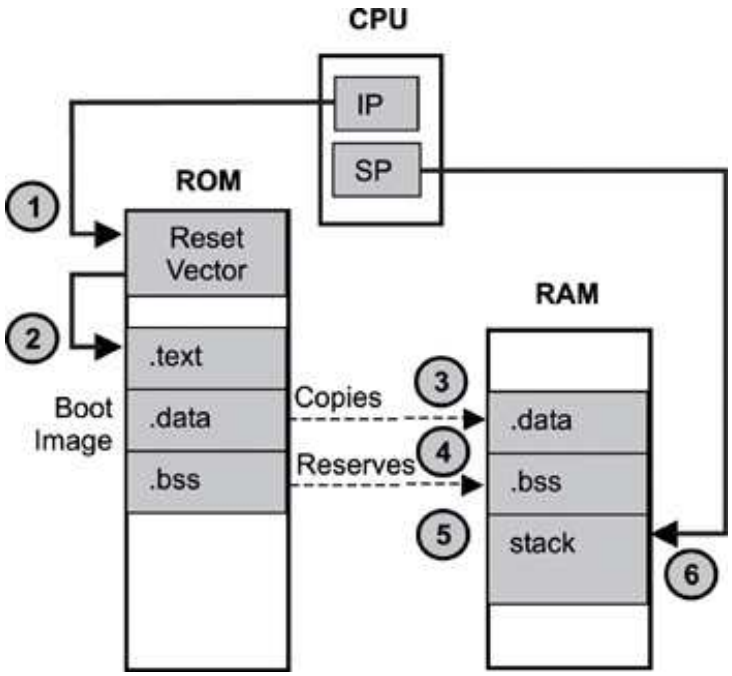


Figure 2.8: Example Memory Layout for an embedded application after being loaded by the OS.

requester does not use 28 bytes of the allocated memory, which is called memory fragmentation. This specific form of fragmentation is called internal fragmentation because it is internal to the allocated block.

The allocation table can be represented as a bitmap, in which each bit represents a 32-byte unit. Figure 2.9 shows the states of the allocation table after a series of invocations of the malloc and free functions. In this example, the heap is 256 bytes in size. After Step 5, the fifth and eighth blocks are free but all other blocks are occupied. Step 6 shows two free blocks of 32 bytes each. Step 7, instead of maintaining three separate free blocks, shows that all three blocks are combined to form a 128-byte block. Because these blocks have been combined, a future allocation request for 96 bytes should succeed.

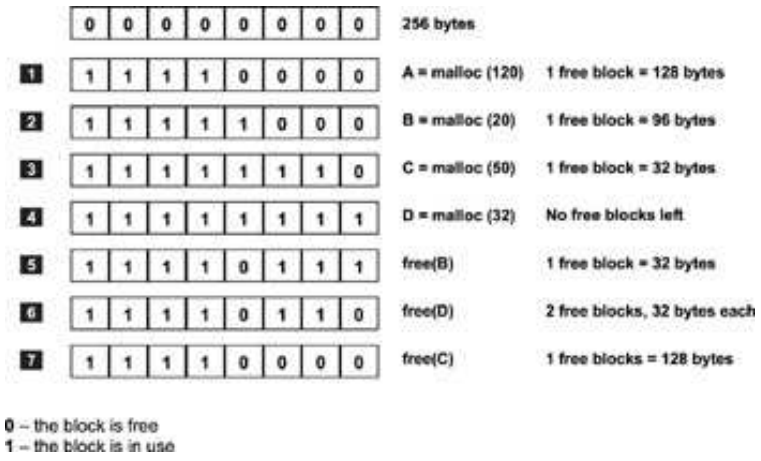


Figure 2.9: Example heap management system for tracking free and used heap chunks.

If Step 6 were to be another malloc request for a block of 64 bytes in length, it would fail because no contiguous blocks of memory were available, despite the fact that 64 bytes of total heap space are still free. The existence of these two trapped blocks is considered to be external fragmentation because the fragmentation exists in the table, not within the blocks themselves. One way to eliminate this type of fragmentation is to compact the area adjacent to these two blocks. Memory compaction occurs block by block to try and group all occupied heap memory blocks together in physical memory and generally continues until all of the free blocks are combined into one large chunk.

Several problems occur with memory compaction. It is time-consuming to transfer memory content from one location to another. The cost of the copy operation depends on the length of the contiguous blocks in use. The tasks that currently hold ownership of those memory blocks are prevented from accessing the contents of those memory locations until the transfer operation completes. Memory compaction is almost never done in practice in embedded designs. The free memory blocks are

combined only if they are immediate neighbors, as illustrated in Figure reffig:heap-example

Memory compaction is allowed if the tasks that own those memory blocks reference the blocks using virtual addresses. Memory compaction is not permitted if tasks hold physical addresses to the allocated memory blocks. In many cases, memory management systems should also be concerned with architecture-specific memory alignment requirements. Memory alignment refers to architecture-specific constraints imposed on the address of a data item in memory. Many embedded processor architectures cannot access multi-byte data items at any address. For example, some architecture requires multi-byte data items, such as integers and long integers, to be allocated at addresses that are a power of two. Unaligned memory addresses result in bus errors and are the source of memory access exceptions.

Some conclusions can be drawn from this example. An efficient memory manager needs to perform the following tasks quickly. First, it must determine if a free block that is large enough exists to satisfy the allocation request. This work is part of the malloc operation. It must also update its internal management information after requests. This work is part of both the malloc and free operations. Finally it must determine if the just-freed block can be combined with its neighboring free blocks to form a larger piece. This work is part of the free operation. The structure of the allocation table is the key to efficient memory management because the structure determines how the operations listed earlier must be implemented. The allocation table is part of the overhead because it occupies memory space that is excluded from application use. Consequently, one other requirement is to minimize

the management overhead.

2.6 Recursive Functions

Any function in a C program can be called recursively; that is, it can call itself. C and virtually all other programming languages in use today allow the direct specification of recursive functions and procedures. When such a function is called, the computer (for most languages on most stack-based architectures) or the language implementation keeps track of the various instances of the function (on many architectures, by using a call stack, although other methods may be used). The number of recursive calls is limited to the size of the stack. Each time the function is called, new storage is allocated for the parameters and for the auto and register variables so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Many examples of the use of recursion may be found as the technique is useful both for the definition of mathematical functions and for the definition of data structures. Many mathematical functions can be defined recursively, such as factorials, Fibonacci series, Euclid's GCD (greatest common denominator), Fourier Transforms and Newton's Method. In Newton's method, for example, an approximate root of a function is provided as initial input to the method. The calculated result is then used as input to the method, with the process repeated until a sufficiently accurate

value is obtained. Other problems can be solved recursively, such as games like the Towers of Hanoi or more complex ones like chess. In games, the recursive solutions are particularly convenient because, having solved the problem by a series of recursive calls, it is easy to find the path arriving at the correct solution. By keeping track of the move chosen at any point, the program call stack does the accounting automatically.

Recursive Example One simple recursive function is found in the mathematical formula used to compute the factorial of a number n:

```
if  n>1 :      n! = n * (n-1)!
    n<1 :      n! = 1
```

The function recursively expands until reaching a lower boundary when n is equal to one. The factorial computation can be expressed in C code using the following function:

```
int factorial (int number)
{
    if (number < 2)
        return 1;
    else
        return (number * factorial(number - 1));
}
```

By looking at this function, we see that the invocation depth of this function will increase linearly with the input value applied. Figure 2.10 gives an illustration of how the program stack would look after two different inputs are applied to the factorial function. The left figure shows the case where the factorial of three is calculated, and involves three total calls to the function. Similarly, the right figure shows a call depth of five invocations when calculating the factorial of five. From

these figures, it can also be observed that the each function invocation consumes an equal amount of stack space, which is known at compile-time. Unfortunately, even the best compiler analysis tools are unable to place a bound on the total size of stack memory allocated by this function at run-time, as it strictly depends on the inputs applied except in trivial cases.

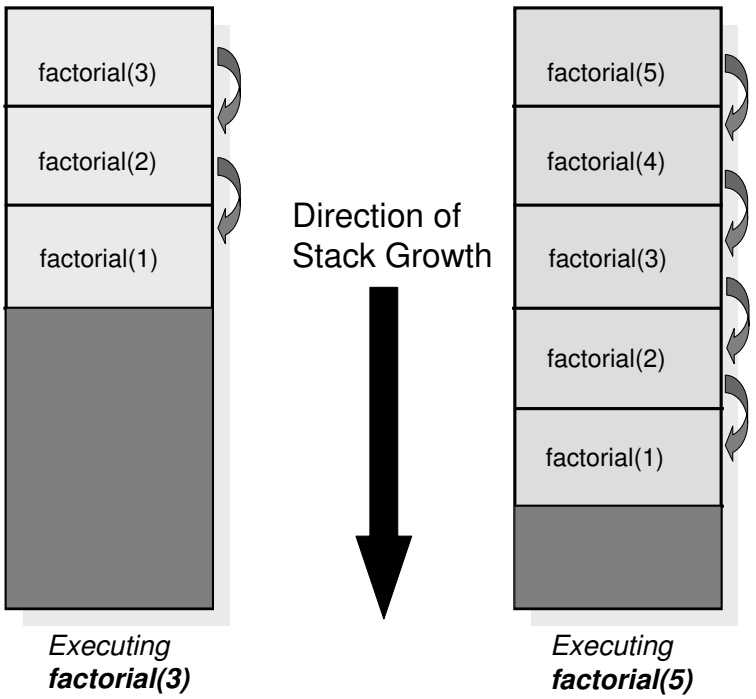


Figure 2.10: Stack growth of recursive factorial() function for two different invocations.

Chapter 3
Previous Work on SPM allocation
3.1 Overview of Related Research

This chapter will primarily focus on related research targeting embedded platforms that contain scratchpad memory along with a slower main memory and perhaps additional flash or ROM memories. We will begin with a discussion of research presented that performs static allocation decisions to take advantage of SPM for energy and/or run-time improvement in Section 3.2. These methods are able to statically allocate some portion of code, global and stack(non-recursive) variables for a program to SPM at run-time. Dynamic allocation methods for the same types of objects constitute the most recent thrusts in this area and are described in Section 3.3. Section 3.4 will present the only relevant research to allocating heap data to SPM while Section 3.5 will discuss methods aimed at converting heap data to stack data to possibly apply existing stack allocation methods. Section 3.6 will discuss variations on static and dynamic schemes targeted at helping designers choose hierarchies of SPM and cache memories when designing an embedded platform. These are generally variations on existing static and dynamic allocation methods to take into account heterogeneous latency and access costs when designing new memory organizations. Section 3.7 briefly discusses related work in the area of dynamic memory managers for heap data and how they apply to our own research. Finally, Section 3.8 completes the chapter with mention of any other related research that is worth mentioning in relation to our SPM allocation technique.

It should be noted that many high-end embedded platforms contain instruction and/or data caches in addition to SPM and other memories, although our techniques were not designed to take full advantage of those platforms. Instead, we attempt to avoid the use of caches altogether. The past decade has shown a great deal of related work on allocation approaches for both SPM and cache memories to avoid some of the energy/latency penalties incurred by poor cache performance. Our methods were designed to allow designers to only use SPM in embedded platforms instead of cache for better real-time guarantees, lower power consumption and reduced execution time. All three benefits are achieved through compiler-directed dynamic allocation techniques which provide the informed control that cache memories lack. As a final reminder, SPM allocation methods can help maximize both performance and power efficiency in an embedded system, regardless of what other memories that system contains.

3.2 Static SPM Allocation Methods

Several static methods have been proposed to allocate data to SPM. These methods make compile-time decisions to produce a static allocation of SPM among program objects at run-time. Since SPM does not have many of the drawbacks that caches do, static placements of important program memory objects are a very effective way of improving performance and reducing power consumption without requiring hardware redesign or massive additional software support. Indeed, for many small applications, static methods can perform almost as well as dynamic methods for certain SPM sizes, explaining the variety of recent investigations into this area.

Until very recently, most published static allocation methods only handled simpler program objects for optimized memory placement since optimal allocation solutions become more difficult as objects exhibit different lifetimes and more complicated execution behavior at run-time.

From the earlier overview on variable lifetimes and storage durations in languages such as C, global variables are the most suited to static memory allocations. Global variables are technically alive and have storage allocation throughout program execution, matching them to the scope that a static memory allocation scheme can handle. The earliest work on SPM allocation dealt only with allocation of global data objects for a program executed on an embedded system with only an SPM and main memory[126]. They use a static compiler program profile to predict the most profitable global objects to place in SPM and place all other program memory objects in main memory. The research presented in [109, 110, 111] is similar except that it assumes the presence of a data cache as well, and attempts to map main memory placements to avoid excessive cache conflicts for those global scalars and arrays which did not fit into SPM. As a natural progression to these placement methods for global variables, other static schemes have been proposed which also handle program code placement to SPM.

Program instructions for an embedded applications must also be stored in memory for loading and execution of the instructions, as well as accesses to the program symbol table, literal pools, constant declarations and other supporting binary data that are present along with executable machine instructions. For those embedded processors that possess SPM without an instruction cache, a program binary

will reside in some type of system memory. Generally most platforms may provide EEPROM, ROM, or some other type of permanent memory meant for reading program code during execution. For embedded systems that only contain main memory and SPM, software developers can also take advantage of SPM for code allocation as well as data variable allocation. A discussion of the benefits and complications involved with SPM code placement will appear at the end of Section 3.3.

Earlier SPM allocation methods were developed to statically allocate code objects in addition to global variables, since both have the same lifetime and storage duration and thus have similar allocation considerations. The research described in [17, 131] are related and describe a knapsack-based formulation approach to place code blocks and global variables to make best use of SPM as a viable cache alternative with better power and latency performance. Another scheme proposed in [10] uses a binary patching approach which attempts to redirect commonly executed code segments to SPM through a static mapping. A recent contribution in [61] extends the knapsack formulation for code placement by considering architectures with limited addressing modes or segmented memory spaces which would require more care in deciding SPM and main memory allocations.

SPM allocation methods progressed beyond just global and code objects with whole-program lifetimes, and have been extended to statically allocate stack objects with limited lifetimes and dynamic storage duration during program execution. Any stack-based programming language will use memory space to grow local variables allocated by function blocks as they become nested at run-time. Traditionally this space is placed in main memory to ensure it is large enough for program execution

without exhausting memory. Using stack space for temporary storage of information between program points is generally referred to as spill code generation. This is also one of the consequences of the modular object approach to binary generation used in most popular compilers, as functions must ensure that they preserve important information when calling a child function that will be needed after returning from that subroutine. Modern interprocedural analysis can alleviate some spill-code generation with whole program knowledge and by considering cross-procedure locality when performing register allocation to determine which program data can reside in registers as opposed to stack memory and thereby reduce the amount of spill-code generated.

Indeed, the benefit of having a small compiler controlled memory such as SPM was proposed for the desktop domain in [41] for reduction of spill code memory accesses, which account for a considerable portion of execution time in complicated applications. Their results were similar to those shown in [17] which confirmed the benefits of having a small, fast memory like SPM which is compiler controlled for better allocation of heavily referenced program objects without the complications of caches and higher level memory hierarchies. A very similar technique was also proposed by [94] precisely for the embedded domain which inspired [41] but is otherwise essentially the same proposed spill-code optimization to fast memory. Another method[127], refining the one proposed in [126], attempts to allocate global variables as well as partial or entire stack space for applications to SPM instead of main memory, as well as performing some pointer packing for efficient addressing between the two. This method employs an Integer Linear Programming(ILP) approach to

deciding memory allocations, which produces optimal results compared to knapsack and other heuristic methods. The benefits and disadvantage of ILP methods will be discussed at the end of section 3.3. These methods all still lack handling for limited lifetime variables, dynamic memory allocations and recursive function optimizations, although recently more robust methods have been investigated to better handle local variables.

Another interesting method was proposed by Angiolini et al in [9] and attempts to optimize the design process for an embedded platform which will include SPM. Their algorithm uses profile information to choose a set of SPM partitions and mappings for a chip design to improve the energy efficiency of an application through bank-selective power-saving features in the memory hierarchy. Their method is static and provides a simple mapping scheme to match program data access characteristics rather than automatic allocation of data to different types of memory. The approach is meant for hardware designers looking to design and customize an embedded processor (ASIP) design for energy performance using static program data placement methods and to take advantage of energy-saving techniques available in some embedded hardware.

The final set of static methods are those that can handle global variables and individual local stack variables while also accounting for their limited lifetimes. For those programs which do not use heap variables or recursive functions, the methods described in [16] produce ILP-based allocations to SPM for embedded systems with heterogeneous memory organizations. This provably optimal static method uses a fairly detailed ILP technique to arrive at exact static allocations for global and stack

variables, easily extended to also handle code segments as well. The ILP formulation works well in practice with commercial-level compilers such as GCC, although realistically solutions are limited to kernels and simple programs, as even heuristic methods may not converge for complicated program formulations with thousands of variables of variable lifetimes and with varying input dependencies. Luckily, a heuristic greedy allocation method that accounts for lifetimes was also presented in [16] which achieved very close performance to the ILP solution at a much reduced cost. This led another group to extend that work to also include support for caches using existing cache placement techniques[66]. This allowed their group to generate allocations automatically for program global and stack variables and achieve close to the optimal results obtained from the method in [16] for moderate sized applications using both SPM and cache.

3.3 Dynamic SPM Allocation Techniques

In order to best emulate a cache's ability to dynamically adapt to program locality during execution, dynamic methods are required to make optimal use of SPM. Dynamic methods are those which can change SPM allocation during runtime, attempting to capture more memory accesses to SPM than static methods are capable of. As an extension to their static global and code SPM allocation method in [131], the method presented in [130] allows them to dynamically allocate code segments at chosen program points to copy popular code segments into SPM when needed and out when no longer profitable. Their results show a large improvement compared to their previous static version, both using an ILP-based formulation.

A similar method using a knapsack based formulation was proposed in [33] for a compiler-based dynamic code allocator for SPM, although no implementation results were reported. The most recent work was reported in [117] in which a compiler managed dynamic instruction placement scheme is implemented in SPM. They investigated ILP formulations for allocation decisions but pursued a heuristic algorithm to achieve solutions for more complex programs that are problematic for ILP solvers. Their methods were primarily motivated to reducing the energy requirements for code executed compared to an instruction cache implementation. A very similar method was recently proposed in [1] which brings SPM code allocation methods to the desktop domain in an effort to approximate an instruction cache through the use of an SPM and a run-time instruction memory management module.

Another approach in a similar vein was proposed by Poletti et al. in [51]. In this publication they propose the additions of a mix of hardware and software techniques to manage SPM for embedded platforms at run-time. Their method targets high-end embedded processors with certain memory hardware and provides a high-level programmer API to access SPM and DRAM as separate main memory areas. Unfortunately this method leaves the burden of optimization on the programmer to make best use of the SPM available. This paper does, however, serve to illustrate software techniques most researchers use to transfer data back and forth from SPM and main memory when performing dynamic SPM allocations.

Besides dynamic code allocation, a few research groups have focused heavily on handling array variables for dynamic SPM allocation. The focus of Kandemir et all has been in the areas of array and loop transformation for locality

optimization[85] and has also been extended heavily into the embedded domain for SPM allocation[82, 83, 84]. Dynamic global and stack array placement in SPM through the use of Presburger Formulas was proposed in [83], a formulation that attempts to minimize copying of data between main memory and SPM needed to implement a dynamic allocation scheme. Their followup work makes use of added cost models but also tackles the problem of dynamic array allocations to SPM while minimizing transfer costs[84, 82]. The method in [84] can place global and stack arrays accessed through affine functions of enclosing loop induction variables in SPM. No other variables are placed in SPM; further the optimization for each loop is local in that it does not consider other code in the program. More recently, Absar et. all refined these methods to take further advantage of DMA-capable memories to reduce transfer costs and enable more dynamic array transfers to be profitable[2]. Of related interest are two other recent works that tackle the problem of array splitting for the purpose of providing smaller size granularity for SPM allocation algorithms. Affine array access optimizations [96] and non-affine array access optimizations[3] have been proposed to improve array handling inside whole-program optimization frameworks through compiler-implemented array and loop transformations. Another recent contribution in [75] proposed a language conversion tool for transformation of loops into structures more amenable to SPM array analysis and allocation techniques. Finally, research presented in [91] applies traditional register allocation methods to program arrays and performance a heuristic dynamic SPM allocation on promising arrays at compile-time.

Up until now, there have only been two compiler-directed methods published

that can dynamically allocate local, global and code objects to SPM to take advantage of limited lifetime and optimize all program objects present in a typical program. The first was our earlier method on dynamic allocation of local and global variables to SPM using compile-time decisions[136]; the final version of which is in [132]. This scheme handles all code, global and non-recursive stack program data, providing a way to account for changing program requirements at run-time, avoid software caching overheads and yield completely predictable memory access times, all while still having extremely low overheads. This method makes use of compiler analysis to drive a heuristic algorithm for whole program analysis of data variable behavior and optimal dynamic allocation decisions during execution. The method inserts code during compilation at appropriate program points to transfer selected data in and out of SPM to take advantage of program data locality. Results show that compared to the provably static allocation scheme for stack and global data, the dynamic method is considerably more powerful at reducing run-time and power consumption by taking full advantage of SPM. A full overview will be presented in the next chapter to illustrate how dynamic allocation of code, global and stack data can be performed as a complement to our new methods for heap and recursive stack data.

The recent paper by Verma et al [140] also implements a dynamic strategy similar to the previous heuristic method. Their approach is motivated by the ILP formulation for global register allocation shown in [54]. Both parts of the problem - finding what variables should be in SRAM at different points in the program and what addresses they should be assigned - are solved using ILP formulations. Code

is then inserted to transfer the variables between SRAM and DRAM. Being ILP based, their solution is likely to be optimal; however, ILP based solutions have some fundamental issues that limit their usefulness.

Problems with ILP-based Methods One, ILP formulations have been known to be undecidable in the worst case and in many practical situations are NP hard. Their solution times, especially for large programs can be exponential. Using ILP solutions is also constrained by issues like intellectual property of source code from different vendors, maintenance of the resulting large combined code and the financial cost of ILP solvers. Due to these practical difficulties of ILP solvers, it is very rare to find ILP solvers as part of commercial compiler infrastructures despite many papers being published that use ILP techniques.

Another drawback of the approach in [140] is that like global register allocation methods, the solution though optimal, is only so per procedure. In other words, the formulation does not attempt to exploit for SPM reuse across procedures. This might lead to data being needlessly swapped out even if retaining it in SRAM might be more beneficial across two procedures. Another important issue that is not addressed in [140] is the issue of correctness in the presence of pointers to stack and global data, something fully accounted for in both our previous scheme [132] as well as by the methods resented in this thesis. A final drawback of the scheme in [140] is that it does not discuss how a compiler would generate code to implement their ILP solutions in a typical embedded system.

3.4 Existing Methods For Dynamic Program Data

From a careful survey of all research related to the area of SPM allocation, both static and dynamic, for whole and partial program optimization, we have not found any published methods that can automatically optimize energy and run-time for all types of program and data objects, particularly heap and recursive-function stack objects. The method proposed in this thesis is currently the most robust among the existing technologies for automatic software optimization to take advantage of SPM. We were unable to locate any related work on recursive function handling for SPM, and in fact were unable to find any research on allocation of recursive-function stack data. Heap placement in SPM has been studied somewhat although only two papers directly deal with optimizing dynamic program data allocated on the heap, presented by Catthoor et all in [13, 51].

The work described in [51] was touched on the previous section and consists of implementing a tailored dynamic memory manager for embedded systems with DMA hardware. This is aimed at replacing standard malloc/free library implementations to provide integration with existing SPM allocation schemes. Their contribution was to make high-level APIs that programmers can use to create a new instance of the SPM manager at run-time for handling API calls for SPM heap allocation, while using the system malloc/free for main memory heap allocation. This process manages the SPM for allocation and movement to and from main memory with explicit DMA transfers to make it more amenable for use with existing SPM allocation schemes for code, global, and stack data. This allows them to provide programmers

with explicit tools to implement their own manual dynamic allocation operations when creating new programs, but does not provide any analysis or optimization to best allocate SPM memory among all program objects for automatic dynamic management with better performance. Their technique requires explicit hardware support for the DMA memory system they target, and simply provides high-level language versions of basic assembly functions for accessing that DMA hardware to transfer data to and from main memory. They also require invocation of a persistent memory manager to interface with the SPM, which is more cumbersome than simply placing SPM under standard library malloc/free control by modifying the OS or embedded code libraries if that approach is desired.

The followup work in [13] focused on improving the internal algorithms for their SPM-aware dynamic memory manager for increased energy savings compared to their earlier dynamic memory manager algorithm. While these methods are useful in enabling hardware-exposed optimizations to software developers seeking to use earlier SPM schemes, it is otherwise not capable of automatic optimization and allocation of SPM memory to account for heap data in a whole-program compiler optimization framework and has been superseded by the automatic dynamic SPM allocation methods presented by our group in [132] and [46].

Summarizing the above work, we can see that all the automatic static and dynamic allocation methods are restricted to code, global and non-recursive stack data. *As far as we know, our method is the first and only method to be able to place a portion of the heap data in SPM under automatic compiler control as well as the first able to handle and optimize recursive-function variables.* We have developed

both static and dynamic allocation methods for heap data allocations to SPM as well as the first static and dynamic method to also handle recursive functions for SPM allocation, and that can be applied to support other types of memory as well. Our methods employ completely predictable, compiler-controlled software insertions to control dynamic management of SPM and main memory for improved program performance without requiring additional hardware and which is adaptable to any memory organizations that includes SPM.

3.5 Heap-to-Stack Conversion Techniques

At first glance, it seems that recent work on converting heap data to stack data [26, 27, 36] or to stack-like constructs called regions [25, 59] may help in allocating heap data to scratch-pad. Here is some background on these methods. These methods use escape analysis [113, 142] to try to prove that a heap data structure is never accessed outside a certain procedure. If so, the heap variable can be placed on the procedure's stack frame, instead of the heap. The advantage of stack allocation is that the high overhead of heap allocation and de-allocation is avoided. Heap de-allocation is particularly expensive for object-oriented languages since it is done using garbage collection. In contrast, heap data on the stack is de-allocated at low cost when its corresponding procedure exits. One restriction with stack allocation is that it requires fixed-size heap variables, except in some cases when the data is on the frame on the top of the stack. Since this is restrictive, region-based schemes [25, 59] have been proposed for when heap data is of unknown size. Regions, like stack frames, are associated with procedures but are physically allocated

on the heap so that they can grow and shrink at runtime. A region is de-allocated when its corresponding procedure exits.

If some heap variables can be allocated to stacks or regions using these methods, then, we ask, can our global/stack method be used to allocate most heap variables to scratch-pad? Unfortunately this approach fails for most heap variables for two reasons. First, heap data structures with compile-time-unknown size cannot be allocated to scratch pad *even if they can be converted to stack or region allocation*. Dynamic data structures such as linked lists, trees and graphs almost always have compile-time-unknown size, and thus cannot be allocated to scratch-pad by stack/region conversion. Second, the fraction of heap data that is of fixed size, and can be converted to stack allocation using escape analysis, is small. Such data can be allocated to scratch pad using our stack/global method, but [36] reports that only 19% of the heap data in their benchmarks could be converted to fixed-size stack data. This low percentage is not surprising – most heap data is in dynamic data structures. Fixed-size heap variables occur mostly in object-oriented languages, where objects are often allocated on the heap so that they can be returned as results from their *method* (object-oriented function) of allocation. For most heap variables, since they are not of fixed size, stack or region allocation does not help in scratch-pad allocation.

In conclusion, allocating heap data to stack or regions reduces allocation and de-allocation overhead in all embedded systems, but cannot be used to allocate most heap data to scratch-pad.

3.6 Memory Hierarchy Research

The presence of SPM on low-power, low-cost computing devices such as today's typical embedded systems is a clear indication that most hardware and software developers feel that SPM is a better choice than caches for many applications. Other researchers have repeatedly demonstrated [10, 17] the power, area and run-time advantages of scratch-pad over caches, even with simple static allocation schemes such as the knapsack scheme used in [17]. Further, scratch-pads deliver better real-time guarantees than caches. Research into software-controlled cache memory management schemes has shown success in partitioning cache memories into a normal cache space as well as a more predictable equivalent scratchpad space[35], allowing designers more flexibility with traditionally hardware managed memory systems. In addition, our method is useful regardless of caches since our goal is to more effectively use the scratch-pad memory already present in a large number of embedded systems today [4, 28, 70, 71, 101, 102, 135], as well as show that proper SPM and compiler management can enable a software only system to outperform a dedicated cache for a given embedded application. In fact, this has prompted a great deal of research into using allocation optimization frameworks to help design better memory hierarchies consisting of different SPM-like memories, or even including some cache memories as well. The contributions of this thesis are useful regardless of the allocation method or framework chosen to exploit SPM or similar memory with or without the presence of caches, as it is the only known method to automatically allocate heap and recursive data to SPM, and can thus be either directly applied

or easily modified to apply to most other existing frameworks that deal with stack, global and code allocation.

The area of software/hardware co-design has been especially important in the field of embedded systems, where low-cost and low-power are the norm. Researchers in this field try to make the best use of software to perform as efficiently as possible in its interaction with processor hardware to create optimal deployed systems from the ground up. The performance benefits of SPM have prompted several researchers to look into hardware-driven solutions to improving program memory performance on embedded platforms. The problem of designing a suitable memory hierarchy for a target embedded application has been studied at the hardware level and resulted with exploration of custom memory hierarchies and organizations that include SPM and cache. Those that only include SPM designs generally attempt to provide the locality benefits of cache without its larger latency and power cost. They primarily accomplish this by suggesting different SPM sizes and configurations tailored for an application for maximal power savings, execution improvement or real-time guarantees. A brief overview of memory hierarchy design using SPM and related allocation studies follows, with a discussion of cache related techniques afterward.

There have been a few methods proposed that choose an SPM memory hierarchy and memory allocation to maximize the performance of a program without using caches. Kandemir et all presented their scheme for dynamic array allocation to either existing or an algorithmically calculated set of SPM memories for embedded benchmarks[80], although this is an adaptation of their work on dynamic array allocation to SPM in [84]. Another improvement upon their technique was

developed in [74], which updates the dynamic array allocation method in [80] by accounting for data reuse across dynamic allocation program points, as well as support for a hierarchy of SPM memories for better allocation placement compared to the single SPM case. In a similar vein, they have also recently proposed a method to apply data compression to SPM allocated data to increase SPM usage for existing allocation methods[106]. A recent publication from Wehmeyer et all describes a compiler-optimized method to partition SPM memory for static allocation of global and code objects for improved performance[144]. This is also an adaptation from their previous work on static allocation of code and global objects for single SPM systems[131]. Noting the trend among researchers, any of the previous related SPM allocation methods can be adopted for memory space exploration among multiple SPM memories for an embedded system for improved results assuming a designer is so inclined. Our static and dynamic methods for heap and recursive function handling will enable whole-program optimization for any such extended frameworks.

Given the rise in embedded systems research and usage, hundreds of different models and platforms are in production today. Generally the very low-end are nothing more than simple microcontrollers without any explicit memory organizations while middle and high-end embedded platforms generally have some kind of advanced memory system. Most include at least some form of internal memory for program or data storage, and some kind of external memory controller for larger storage requirements. A great many include some form of SPM or SRAM while some include only an additional cache hierarchy, and many manufacturers even include both. With the presence of both an SPM and cache on the same system, allocation

and placement can have a large impact on performance and several methods have been proposed to make best use of both. Building upon the earlier work of Panda in [109], Dutt et all describe an updated method in [49],that explores the choices available to designers in memory organizations. By exploring different scratchpad and cache configurations their method can decide on an appropriate choices for SPM and cache based on analysis and designed to cooperate with their allocation decisions. Their method involves mapping global variables onto SPM, applying array and loop optimizations for global arrays and mapping the rest of the program data onto cache memory for minimal cache conflict through careful address placement. They update their method in [110] to include a performance estimation framework on top of the memory exploration results that can be used to predict the performance of applications without prior implementation or simulation. A further follow-up from the group for cache-only memories that lack SPM was later described in [56].

The methods just described were primarily targeted at designers who may wish to explore optimal memory configuration and hierarchies from a hardware point of view, as well as providing the compiler framework to make better use of those memories. The majority of research in the SPM allocation domain instead focuses on maximizing performance of popular existing embedded platforms that already contain a determined amount of SPM, cache and other memories. By far the most popular area has been that of placing some subset of program data in SPM and the rest in the cache, with a host of refinements for actual memory assignment and cache pollution avoidance. Our own method can also be applied to any developed methods that take caches into consideration for memory layout, or more simply used

as the interface to main memory. Besides the works already mentioned[49, 110, 56], other methods have been proposed for systems containing both SPM and cache with different goals in mind. Verma et all proposed a similar update to their previous work in[131] which originally dealt with static SPM allocation of code and global objects. Their new method instead produces a static allocation of code blocks to SPM attempting to reduce the number of conflicts incurred by the instruction cache during execution, providing efficient instruction loading from both SPM and cache with less wasted cache activity[95, 141]. In the area of cache-conscious SPM allocation for program data, Xtream-Fit produces SPM allocations for constants and scalars while for other programs and code it employs energy conservation techniques targeting SDRAM accesses through a cache[116]. Their method is primarily directed at improving media application execution through cache prefetching and SPM allocation for popular global variables to avoid cache pollution.

As a natural benefit of SPM allocations instead of using cache memories to provide memory locality, studies have been conducted to show how SPM can impact real-time applications by providing better access guarantees. Many real-time systems require programs to be tightly bounded by their Worst-Case Execution Time(WCET) to ensure they can complete periodic tasks in a timely fashion. For this reason, most designers of real-time systems will avoid caches at all costs, despite their locality benefits. The variable delay in accessing memory through a cache hierarchy makes these systems unpredictable from a WCET point of view, and can make cache usage infeasible. The recent exploration by Marwedel et all in [97, 145] presents their earlier SPM allocation work in light of real-time concerns

and shows that using SPM memory allocation techniques can greatly improve the WCET bounds as well as power consumption bounds. Recently a method with very similar goals was also proposed in[65]. These studies further impress upon designers the benefits that SPM can produce when used instead of or in addition to cache memories for modern embedded systems.

Orthogonal Power Minimization Methods There exist several other complementary techniques aimed at reducing latency and run-time by placement to a particular memory module among those available in a heterogeneous memory hierarchy. These range in granularity from methods which design entire memory organizations to maximize application performance, down to detailed methods controlling individual sections of a single memory bank to reduce power consumption. Research for taking advantage of DRAM characteristics for efficient instruction sequencing, address calculation, bank management, etc, to reduce power and execution for systems has been popular[55, 38, 93, 44]. Another related optimization method attempts to perform data-layout optimizations, which affect at what addresses and in which order program variables should be mapped to memory banks in an embedded system, for reduced address calculation code, improved offset assignment for minimized addressing widths, among others[30, 37, 57, 34, 81, 79]. There also exists a great body of work relating to data and program transformations to improve cache locality, performance, energy consumption, or a combination of these and other goals. These cache-only memory organizations are all hardware managed and lack a controllable scratchpad although a wide variety of them exist[112, 118, 107, 90, 77]. A general overview of these types of optimization methods for embedded systems can also be

found in [108]. From reading the non-SPM allocation research, it becomes apparent that they are generally orthogonal to the SPM allocation methods for embedded systems described in the rest of this chapter and can thus be selectively combined with many of these approaches, and so will not be discussed further in this thesis.

3.7 Dynamic Memory Manager Research

As far as heap data allocation is concerned, there exists a large body of research in the desktop, and a very small amount in the embedded domain, that optimizes heap memory managers for better application performance. For the most part these memory managers are all orthogonal to our existing SPM allocation techniques, since these can be used to implement the system main memory heap manager. The SPM heap management is implicitly handled by our compiler method through insertion of wrapper code to replace the dynamic memory implementation being used with that embedded compiler. Our own method does not attempt to optimize the dynamic memory manager for main memory, and so any optimized implementation can be used to handle main memory. The advantage of such a large body of work in the desktop domain is that some of it can be applied to the embedded domain to make heap memory more efficient for embedded programmers.

Several methods exist for producing customized dynamic memory manager implementations for languages such as C which re-implement traditional malloc/free libraries tailored to that platforms expected workload. The major works in composing high performance uniprocessor memory allocators for C are those in [57, 143, 22, 21] all of which provide frameworks to produce tailored dynamic memory manager func-

tions from compiler and profile analysis. The usage of dynamic memory has traditionally been slim for embedded systems due to the generally poor performance of most traditional desktop dynamic memory managers ported to the embedded domain. Also, the non-deterministic nature of dynamic memory management functions prevent their use in real-time systems that must have bounded allocation and access times for memory. An overview is presented in [48] which surveys a variety of desktop management methods and presents an optimized version that allows usage in real-time systems. A similar method to bound allocation but with the use of added hardware was later suggested in [47]. Interest in making dynamic memory managers more cache-aware for allocation/deallocation decisions has prompted a number of papers that discuss different implementations to improve cache performance with dynamic memory used in a system [121, 122, 123, 18, 124, 98, 50]. Some of these ideas influenced the optimization of heap variable placement in my own algorithm, and were inspired by the works on heap object analysis and prediction. The concept of segregating heap objects by their request size as correlated with static and dynamic analysis of performance of our initial implementation has led to advances in the handling and allocation of unknown-size malloc allocation sites for my method.

3.8 Other Related Methods

Runtime methods such as software caching [100, 60] emulate a cache in SRAM using software. The tag, data and valid bits are all managed by compiler-inserted code at each memory access. Software overhead is incurred to manage these fields,

though compiler optimizes away the overhead in some cases [100]. Software caching schemes suffer from large overheads from inserted checks before every memory instruction, which are hard to optimize away especially for heap data. Lacking good methods for heap data, the current practice is to place all heap data to DRAM in systems with scratch-pad. Our proposed dynamic method promises to be the first to successfully place heap data in scratch-pad, and our earlier work on fully general dynamic stack,global and code allocation provides locality benefits without the sometimes extreme software caching overheads incurred by a system such as FlexCache. The Cool-Cache method proposed in [139] takes inspiration from earlier SPM allocation research and the FlexCache ideas to provide a hybrid software caching scheme. They make use of SPM to allocate popular global scalar data, and use an optimized software caching system to manage a larger SRAM bank. Their results show the benefit of avoiding software emulation of hardware cache operations by using SPM and compiler optimizations to try and overcome the crippling cost that software caching incurs.

Some software caching schemes have been proposed for desktops that use *dynamic compilation* [69] which changes the program at run-time in RAM. Most embedded systems, however, store the program in unchangeable ROM, and dynamic compilation cannot be used. Other software caching schemes have been proposed with different goals and/or non-applicable platforms [138, 20, 31, 119, 23, 76]. For lack of space, we do not discuss these further.

Our work is in some aspects also analogous to the offline paging problem first formulated by [19]. The offline paging problem deals with deriving an optimal page

replacement strategy when future page references are known in advance. Analogously we look at finding a memory allocation strategy when program behavior is known in advance. Offline paging, however, cannot be used for our purposes since it makes its page transfer decisions at run-time (address translation done by virtual memory), while we need to associate memory transfers with static program points.

Recursive Functions While a thorough search of all related research was performed, very little that related to the allocation of recursive function stack data exists as far as we were able to determine. This is not surprising, all other SPM allocation publications discussed in the first few sections either failed to mention recursive stack handling (only 4 mentioned heap data) or a very few simply referred to them as unhandled and left in main memory. We did find a large amount of research aimed at transformation of recursive functions for different approaches to optimization.

The most popular class of methods attempting to transform recursive functions into iterative functions is known as tail recursion (or tail-end recursion) optimization. Tail recursion is a special case of recursion that can be easily transformed into an iteration using different methods [24, 39]. Such a transformation is possible if the recursive call is the last thing that happens in a function. Replacing recursion with iteration, either manually or automatically, can drastically decrease the amount of stack space used and improve efficiency. This technique is commonly used with functional programming languages, where the declarative approach and explicit handling of state promote the use of recursive functions that would otherwise rapidly fill the call stack. Most modern compilers such as GCC support tail

recursion optimizations to reduce run-time and reduce stack overhead.

Tail recursion optimization has a long and rich history, prompting other researchers to look into ways to extend their usefulness to modern problems. One method performs procedure inlining to convert mutual recursion to direct recursion [86]. This allows use of optimization techniques that are most easily applied to directly recursive procedures, in addition to the well-known benefits of inlining. A recent method proposes complete recursive function inlining in restricted cases which aims at total transformation of the original function [134]. Transformation methods have also been studied converts certain general recursive functions into loops [62]. Finally, another recent publication [148] takes advantage of the significant performance benefits of recursive algorithms on both multi-level memory hierarchies and shared-memory systems. They found that recursive algorithms tended to have the data reuse characteristics of blocked-algorithm, blocked at many different levels. They target promising loops for conversion into recursive functions for optimized memory performance.

Chapter 5
Dynamic allocation of static program data

This chapter presents an overview of our earlier work in [132], which details the first compiler method to dynamically allocate code, global and stack (non-recursive) data to SPM using whole program analysis. This earlier work is part of the thesis of my co-researcher, Sumesh Udayakumaran, and is not a new contribution of this thesis. Nevertheless, my method for allocation of heap and recursive stack data builds upon the work in [132] for its code, global and stack handling methods. For this reason, this chapter presents that scheme as necessary background information to understand how whole program allocation to SPM occurs.

The concept of using heterogenous memory for optimal memory layouts has been researched heavily in the past few years, particularly in the case of embedded systems having SPM. The more recent research has presented powerful automatic compiler methods for the static and dynamic allocation of program data to minimize runtime and power. Unfortunately, many of these earlier methods focused on narrow optimizations using SPM which have only scratched the surface of the powerful compiler analysis and optimization tools available today.

Our preliminary research on SPM allocation was presented in [136] and accounts for changing program requirements at runtime, has no tags like those used by runtime methods, requires no run-time checks per load/store, has extremely low overheads and yields 100% predictable memory access times. Our completed research was presented in [132]. This publication explained our complete compiler

method for allocating three types of static program objects - global variables, stack variables and program code - to scratch-pad while still being to dynamically modify runtime allocation without excessive overheads. With tools in place to handle static program data, we were able to integrate them with our proposed research on dynamic program data and enable full program memory optimization for the first time.

5.1 Overview for static program allocation

A general outline of the dynamic allocation method in [132] for static program data follows. The compiler begins by analyzing the program to identify locations termed program points where it may be beneficial to insert code to copy a variable from DRAM into the scratch-pad. It is beneficial to copy a variable into scratch-pad if the latency gain from having it in scratch-pad rather than DRAM is greater than the cost of its transfer. A profile-driven cost model estimates these benefits and costs for a program. The compiler ensures that the program data allocated to scratch-pad fits at all times by occasionally evicting existing variables in scratch-pad to make space for incoming variables. In other words, just like in a cache, data is moved back and forth between DRAM and scratch-pad, but under compiler control, and with no additional overhead.

Key components of the method consist of the following. (i) To reason about the contents of scratch-pad across time, it helps to attach a concept of relative time to the above-defined program points. Towards this end, a new data structure is introduced called the *Data-Program Relationship Graph (DPRG)* which associates a unique

timestamp with each program point. (ii) A detailed cost model is presented to estimate the run-time cost of any proposed data transfer at a program point. (iii) A compile-time heuristic is presented that uses the cost model to decide which transfers minimize the run-time. The well-known data-flow concept of liveness analysis [11] is used to eliminate unnecessary transfers provably dead variables [1]are not copied back to DRAM; nor are newly alive variables in this region copied in from DRAM to SRAM [2]. In programs, where the final results (only global) need to be left in the memory itself, this optimization can be turned off in which case the benefits may be reduced. [3]This optimizations also needs to be turned off for segments shared between tasks.

There are three desirable features of the algorithm which can be readily observed. (i) No additional transfers beyond those required by a caching strategy are done. (ii) Data that is accessed only once is not brought into the scratch-pad, unlike in caches, where the data is cached and potentially useful data evicted. This is particularly beneficial for streaming multimedia codes where use-once data is common. (iii) Data that the compiler knows to be dead is not written out to DRAM upon eviction, unlike in a cache, where the caching mechanism writes out all evicted data.

This method is clearly profile-dependent; that is, its improvements are de-

[1]In compiler terminology a variable is dead at a point in the program if the value in it is not used beyond this point, although the space could be. A dead variable becomes live later if it is written to with subsequently used data. As a special case it is worth noting that every un-initialized variable is dead at the beginning of the program. It becomes live only when written to first. Further, a variable may have more than one live range separated by times when it is dead.

[2]The current implementation only does data-flow analysis for scalars and a simple form of array data-flow analysis that can prove arrays to be dead only if they are never used again. If more-complex array data-flow analysis is included then the results can only get better.

[3]Such programs are likely to be rare. Typically data in embedded systems is used in a time critical manner. If persistent data is required, it is usually written into files or logging devices.

pendent upon how representative the profile data set really is. Indeed, all existing scratch-pad allocation methods, whether compiler-derived or programmer-specified, are inherently profile dependent. This cannot be avoided since they all need to predict which data will be frequently used. Further this method does not require the profile data to be like the actual data in all respects so long as the relative re-use trends between variables are similar in the profile and actual data, good allocation decisions will be made, even if the re-use factors are not identical. A region's gain may even be greater with non-profile data if its data re-use is higher than in the profile data.

Real-time guarantees Not only does the method improve run-time and energy consumption, it also improves the real-time guarantees of embedded programs. To understand why, consider that the worst-case memory latency is a component of the worst-case execution time. This method, like all compiler-decided allocation methods, guarantees that the latency of each memory instruction is known for kernels and other programs with minimal input profile dependence. This translates into total predictability of memory system behavior, thus helping designers immensely with improving Worst-Case Execution Time(WCET). Such real time benefits of scratch-pad have been observed before, such as in [145].

Program Code It is separately shown how this method can also be easily extended to allocate program code objects. Although code objects are accessed more heavily than data objects (one fetch per instruction), dynamic schemes like this one are not likely to be applicable in all cases. First, compared to data caches, use of instruction caches is much more feasible due to their effectiveness at smaller sizes

and highly predictable, minimally input-dependent execution profiles. It is not uncommon to find use of instruction caches (but not data caches) in embedded systems like the Motorola STARCORE, MFC5xx and 68HC processors. Second, for low and medium-end embedded systems, code is typically stored in ROM/Flash. An example of such a system is Motorolas MPC500, MCORE and 6812. Unlike DRAM devices, ROM/Flash devices have lower seek times (in the order of 75ns-120ns, 20 ns in burst/page mode) and power consumption. For low-end embedded systems, this would mean an access latency of only one or two cycles, because they operate at lower clock speeds. For such low-end embedded systems using ROM/Flash where cost is an important factor, speeding up accesses to code objects is not as critical as optimizing data objects residing in DRAM. High-end systems such as the Intel StrongARM and Motorola Dragonball operate at higher clock rates, which consequently increases the relative cost of ROM/Flash access latency. The proposed extension for handling code would thus enable the dynamic method to be used for speeding up code accesses in such systems and increasing performance as the latency gap between SPM and code memory increases.

Impact The impact of this work will be a significant improvement in the cost, energy consumption, and run-time of embedded systems. The results in [132] show up to 39.8% reduction in run-time for this method for global and stack data and code vs. the optimal static allocation method in [16] also extended for code. With hardware support for DMA, present in some commercial systems, the run-time gain increases up to 42.3%. The actual gain depends on the SRAM size, but the results show that close to the maximum benefit in run-time and energy is achieved for

a substantial range of small SRAM sizes commonly found in embedded systems. Using an accurate power simulator, the method also shows up to 31.3% reduction in energy consumption vs. an optimal static allocation. This method does incur some code-size increase due to the inserted transfers; the code size increase averages a modest 1.9% for the benchmarks compared to the unmodified original code for a uniform memory abstraction; such as for a machine without scratch-pad memory.

The next few section will discuss the method for static program data in more detail.

5.2 The Dynamic Program Region Graph

The dynamic memory allocation method in [132] for code, stack and global program objects takes the following approach. At compile-time, the method inserts code into the application to copy program objects from DRAM into the scratch-pad whenever it expects them to be used frequently thereafter, as predicted by previously collected profile data. Program objects in the scratch-pad may be evicted by copying them back to the DRAM to make room for new variables. Like in caching, all data is retained in DRAM at all times even when the latest copy is in the scratch-pad. Unlike software caching, since the compiler knows exactly where each program object is at each program point, no run-time checks are needed to find the location of any given variable. It is show in [132] that the number of possible dynamic allocations is exponential in both the number of instructions and the number of variables in the program. The problem is almost certainly NP-complete, though there has not been

an attempt to formally prove this.

Lacking an optimal solution, a heuristic is used. The cost-model driven greedy heuristic presented has three steps. First, it partitions the program into regions where the start of each region is a program point. Changes in allocation are made only at program points by compiler inserted code that copies data between the scratch-pad and DRAM. The allocation is fixed within a region. The choice of regions is discussed in the next paragraph. Second, the method associates a unique timestamp with every program point such that (i) the timestamps form a partial order; and (ii) the program points are reached during run-time roughly in timestamp order. In general, it is not possible to assign timestamps with this property for all programs. Later in this section, however, a method is shown, that by restricting the set of program points and allowing multiple timestamps per program point, is able to define timestamps for all programs. Third, memory transfers are determined for each program point, in timestamp order, by using the cost-driven algorithm in section 5.3.

Deriving regions and timestamps The choice of program points and therefore regions, is critical to the algorithms success. Regions are the code between successive program points. Promising program points are (i) those after which the program has a significant change in locality behavior, and (ii) those whose dynamic frequency is less than the frequency of its following region, so that the cost of copying into the scratch-pad can be recouped by data re-use from scratch-pad in the region. For example, sites just before the start of loops are promising program points since they are infrequently executed compared to the insides of loops. Moreover, the loop

often re-uses data, justifying the cost of copying into scratch-pad. With the above two criteria in mind, program points are defined as (i) the start and end of each procedure; (ii) just before and just after each loop (even inner loops of nested loops); (iii) the start and end of each if statements then part and else part as well as the start and end of the entire if statement; and (iv) the start and end of each case in all switch statements in the program as well as the start and end of the entire switch statement. In this way, program points track most major control-flow constructs in the program. Program points are merely candidate sites for copying to and from the scratch-pad whether any copying code is actually inserted at those points is determined by a cost-model driven approach, described in section 3.

```
main () {
   if (...)   { proc-D()}   else   {...}
   proc-A()
   proc-B()
}

proc-A () {
   proc-C()
}

proc-B () {
   proc-C()
   while (...){Y =...}
}

proc-C () { X =...}

proc-D () { ...}
```

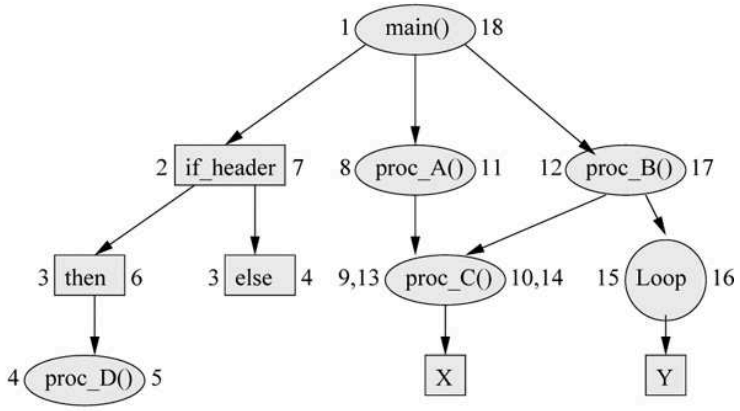


Figure 5.1: Example program on left with its DPRG representation on right.

Figure 5.1 shows an example illustrating how a program is marked with timestamps at each program point. Figure 5.1(a) shows the program outline. It consists of five procedures, namely main(), proc-A(),proc-B(), proc-C() and proc-D(), one loop and one if-then-else construct. The only program constructs shown are loops,

procedure declarations and calls, and if statements other instructions are not. Accesses to two selected variables X and Y are also shown.

Figure 5.1(b) shows the Data-Program Relationship Graph (DPRG) for the program in figure 5.1(a). The DPRG is a new data structure introduced to help represent regions for reasoning about their time order. The DPRG is essentially the programs call graph appended with new nodes for loops, if-thens and variables. In the DPRG shown in figure 5.1(b), there are five procedures, one loop, one if statement, and two variables represented by nodes. Separate nodes are shown for the entire if statement (called if-header) and for its then and else parts. On the figure oval nodes represent procedures, circular nodes represent loops, rectangular nodes represent if statement nodes, and square nodes represent variables. Edges to procedure nodes represent calls; edges to loop and if nodes shows that the child is in its parent; and edges to program object nodes represent memory accesses to that program object from its parent. No additional edges exist to model continue and break statements. The DPRG is usually a directed acyclic graph (DAG), except for recursive programs, where cycles occur.

Figure 5.1(b) also shows the timestamps (1-18) for all program points, namely the beginnings (shown on left of nodes) and ends (shown on right) of every procedure, loop, if-header, then and else node. The goal is to number timestamps in the order they are encountered during the execution. This numbering is computed at compile-time by the well-known depth-first-search (DFS) graph traversal algorithm. the DFS marks program points in the order seen with successive timestamps. The DFS is modified, however, in two ways. First, the DFS is modified to number then and else

nodes of if statements starting with the same number since only one part is executed per invocation. For example, the start of the then and else nodes shown in the figure both are marked with timestamp 3. The numbering of the end of if-header node (marked 7 in the figure) follows the numbering of either the then and else parts, whichever consumes more timestamps. Second, it traverses and timestamps nodes every time they are seen, rather than only the first time. This still terminates since the DPRG is a DAG for non-recursive functions. Such repeated traversal results in nodes that have multiple paths to them from main() getting multiple timestamps. For example, node proc-c() gets timestamps 9 & 13 at its beginning, and 10 & 14 at its end.

It can be seen that the timestamps provide a partial order rather than a total order. For example, the two possible alternatives for a simple if-then conditional block will never have an edge between them showing a relative order between these two regions. Instead, relative orderings can only be correlated with timestamps for those regions along an execution path. Timestamps are useful since they reveal dynamic execution order: the run-time order in which the program points are visited is roughly the order of their timestamps. The only exception is when a loop node has multiple timestamps as descendants. Here the descendants are visited in every iteration, repeating earlier timestamps, thus violating the timestamp order. Even then, we can predict the common case time order as the cyclic order, since the end-of-loop backward branch is usually taken. Thus we can use timestamps, at compile- time, to reason about dynamic execution order across the whole program. This is a useful property, and it has been speculated that timestamps may be

useful for other compiler optimizations as well that need to reason about execution order, such as compiler-controlled prefetching [92],value prediction [87] and speculation [29]. Timestamps have their limitations in that they do not directly work for Goto statements or the insides of recursive cycles; but workarounds for both are mentioned in section 5.4 for the method in [132] described in this chapter.

5.3 Allocation Method for Code, Stack and Global Objects

This section describes the algorithm from [132] for determining the memory transfers of global and stack variables at each program point. The next section shows how this method can be extended with some minor modifications to also allocate program code.

Before running this algorithm, the DPRG is built to identify program points and mark the timestamps. Next, profile data is dynamically collected to measure the frequency of access to each variable separately for each region. This frequency represents the weight of the edge from a parent node to a child variable. Profiling also measures the average number of times a region is entered from a parent region. This represents the edge weight between two non variable nodes. The total frequency of access of a variable is the product of all the edge weights along the execution path from the main() node to the variable.

At each program point, the algorithm then determines the following memory transfers: (i) the set of variables to copy from DRAM into the scratch-pad and (ii) the set of variables to evict from DRAM to the scratch-pad to make way for

incoming variables. The algorithm computes the transfers by visiting each program point (and hence each region) once in an order that respects the partial order of the timestamps. For the first region in the program, variables are brought into the scratch-pad in decreasing order of frequencyper byte of access. Thereafter for subsequent regions, variables currently in DRAM are considered for bringing into the scratch-pad in decreasing order of frequency-per-byte of access, but only if a cost model predicts that it is profitable to do so. Variables are preferentially brought into empty space if available, else into space evicted by variables that the compiler has proven to be dead at this point, or else by evicting live variables. Completing this process for all variables at all timestamps yields the complete set of all memory transfers.

Transfers are not performed blindly, and follow a cost-benefit analysis applied to the DPRG. Given a proposed incoming variable and one-or-more variables to evict for the incoming variable, the cost model determines if this proposed swap should actually take place. In particular, copying a variable into the scratch-pad may not be worthwhile unless the cost of the copying and the lost locality of evicted variables is overcome by its subsequent reuse from scratch-pad of the brought-in variable. The cost model used models each of these components to derive if the swap should occur.

The pseudocode representation of the main allocation algorithm is shown in Figure 5.2, and begins by declaring several compiler variables. These include V-fast and V-slow to keep track of the set of application variables allocated to the scratch-pad and DRAM, respectively, at the current program point. Bring-in-set, Swap-

```
Define                          /* The values of all of the quantities defined below change at each program point */
   Set V-slow                   /* Set of variables in DRAM at this point */
   Set V-fast                   /* Set of variables in the scratch-pad at this point */
   Set Bring-in-set             /* Variables to bring into the scratch-pad at this program point */
   Set Swapout-set              /* Set of variables for eviction to DRAM */
   Set Retain-in-fast-set       /* Set of variables to retain in the scratch-pad */
   Set Dead-set                 /* Set of variables in V-fast whose lifetimes have ended */
   float freq-per-byte[variable,region]   /* Access frequency per byte of variable in region in profile data*/
void        Memory-allocator()
1. initial-candidate-list ← Sort variables accessed in first region in decreasing order of freq-per-byte[variable, first region].
2. Assign V-fast and V-slow for first region by filling the scratch-pad in greedy order from initial-candidate-list.
3. for all timestamped program points in the application visited in partial order of their timestamps, starting at second region
4.      Swapout-set ← NULL_SET;   Bring-in-set ← NULL_SET;   Retain-in-fast-set ← NULL_SET
5.      Dead-set = Variables which are no longer alive at this program point
6.      Free-space = Free-space + sizeof(Dead-set)
7.      for all variables V accessed in this region in decreasing order of frequency-per-byte
                    /* In rest of code 'this region' ≡ region after current program point */
8.          if V ∈ V-slow
9.              if (sizeof(V) ≤ Free-space)                          /* V fits; no need to swap out variables */
10.                 Benefit-of-bring-in-V ← Find-benefit(V, NULL_SET)
11                  if (Benefit-of-bring-in-V > 0)
12.                     Bring-in-set ← Bring-in-set ⋃ {V}
13.                 endif
14.             else                                                 /* V does not fit; try to swap out variables */
15.                 Swapout-set-for-V ← Find-swapout-set(V)
16.                 if (Swapout-set-for-V ≠ NULL)
17.                     Swapout-set ← Swapout-set ⋃ Swapout-set-for-V
18.                     Bring-in-set ← Bring-in-set ⋃ {V}
19.                     Free-space ← Free-space + sizeof(Swapout-set-for-V) − sizeof(V)
20.                 endif
21.             endif
22.         else /* V ∈ V-fast */
23.             if (V not in Swapout-set)                            /* Has not been swapped out so far */
24.                 Retain-in-fast-set ← Retain-in-fast-set ⋃ {V}
25.             endif
26.         endif
27.     endfor
28.     V-fast ← V-fast ⋃ Bring-in-set − Swapout-set − Dead-set
29.     V-slow ← V-slow ⋃ Swapout-set − Bring-in-set − Dead-set
30.     Store V-fast and V-slow for this region.
31.endfor                                                            /* For all program points */
32.return

Set     Find-swapout-set(V)
33.Swapout-set-for-V ← NULL_SET
34.Swapout-candidate-list ← Sort variables in the scratch-pad in ascending order of size. In case of a tie, choose variable with
   the higher next-timestamp-of-access. Exclude variables that have become dead in this region (already removed in line 6).
35.Swapout-candidate-list ←
        Swapout-candidate-list − (Swapout-set ⋃ Bring-in-set ⋃ Retain-in-fast-set)   /* Update candidate-list*/
36.Size-required ← sizeof(V) - Free space
37.while (((Swapout-candidate ← next-element(Swapout-candidate-list)) ≠ NULL) and (Size-required > 0))
38.     Benefit-of-swap ← Find-Benefit(V, Swapout-candidate)
39.     if (Benefit-of-swap > 0)
40.         Swapout-set-for-V ← Swapout-set-for-V ⋃ {Swapout-candidate}
41.         Size-required ← Size-required − sizeof(Swapout-candidate)
42.     endif
43.end-while
44.if (Size-required > 0)                                    /* Could not find required space by swapping out */
45.     return (NULL)                                       /* Do not swap */
46.endif
47.return (Swapout-set-for-V)                                /* Found required space by swapping out */

int     Find-benefit(V,Swapout-candidate)
48.Latency-gain ← freq-per-byte[V, this region] ⋆ size(V) ⋆ (Latency_slow_mem − Latency_fast_mem)
49.Latency-loss ←
        freq-per-byte[Swapout-candidate, this region] ⋆ size(Swapout-candidate)* (Latency_slow_mem − Latency_fast_mem)
50.Migration-overhead ← Time for copying Swapout-candidate (if modified) to DRAM + Time for copying V to the SRAM
51.Benefit-of-swap ← latency-gain − latency-loss − Migration-overhead
52.return Benefit-of-swap
```


Figure 5.2: Algorithm for static program data allocation.

out-set and Retain-in-fast-set store their obvious meaning at each program point. Dead-set refers to the set of variables in V-fast in the previous region whose lifetime has ended. The frequency-per-byte of access of a variable in a region, collected from the profile data, is stored in freq-per-byte[variable, region].

The algorithm proceeds in the following manner. Lines 1-2 computes the allocation for the first region in the application program. For the first region, variables are greedily brought into the scratchpad in decreasing order of frequency-per-byte of access. Line 3 is the main for loop that steps through all the subsequent program points in timestamp order. At each program point, line 7 steps through all the variables, giving preference to frequently accessed variables in the next region. For each variable V in DRAM (line 8), it tries to see if it is worthwhile to bring it into the scratch-pad (lines 9-21). If the amount of free space in the scratch-pad is enough to bring in V, V is brought in if the cost of the incoming transfer is recovered by the benefit (lines 10-13). Otherwise, if variables need to be evicted to make space for V, the best set of variables to evict is computed by procedure Find-swapout-set() called on line 15 and the swap is made (lines 16-20). If the variable V is in the scratch-pad (line 22), then it is retained in the scratch-pad provided it has not already been swapped out so far by a higher frequency-per-byte variable (line 23-25). Finally, after looping through all the variables, lines 28 and 29 update, for the next program point, the set of variables in scratch-pad and DRAM, respectively. Line 30 stores this resulting new memory map for the region after the program point.

Next, Find-swapout-set() (lines 33-47) is called in line 15. It calculates and returns the best set of variables to copy out to DRAM when its argument V is

brought in. Possible candidates to swap out are those in scratch-pad, ordered in ascending order of size (line 34); but variables that have already been decided to be swapped out, brought in, or retained are not considered for swapping out (line 35). Thus variables with higher frequency-per-byte of access are not considered since they have already been retained in scratch-pad in line 24. Among the remaining variables of lower frequency-per-byte, as a simple heuristic small variables are considered for eviction first since they cost the least to evict. Better ways of evaluating the swapout set of least cost by evaluating all possible swapout sets are avoided to avoid an increase in compile-time; moreover we found these to be unnecessary since only variables with lower frequency-per-byte than the current variable are considered for eviction. The while loop on line 37 looks at candidates to swap out one at a time until the space needed for V has been obtained. A cost model is used to see if the swap is actually beneficial (line 38); if it is the swapout set is stored (lines 40-41). More variables may be evicted in future iterations of the while loop on line 37 if the space recovered by a single variable is not enough. If swapping out variables that are eligible and beneficial to swap out did not recover enough space (line 44), then the swap is not made (line 45). Otherwise procedure Find-swapout-set() returns the set of variables to be swapped out.

Cost model Finally, Find-benefit() (lines 36-43) is called in lines 10 & 38. It computes whether it is worthwhile, with respect to run-time, to copy in variable V in its first argument by copying out variable Swapout-candidate in its second argument. The net benefit of this operation is computed in line 51 as being the latency-gain minus latency-loss minus Migration-overhead. The three terms are explained as

follows. First, the latency gain is the gain from having V in the scratch-pad in the next region (line 48). Second, the latency loss is the loss from not having Swapout-candidate in the scratch-pad in the next region (line 49). Third, the migration overhead is the cost of copying itself, estimated in line 50. The overhead depends on the point at which the transfer is done. So the overhead of transfers done outside a loop is less than inside it. The algorithm conservatively chooses the transfer point that is outside as many inner loops as possible. The choice is conservative in two ways. One, points outside the procedure are not considered. Two, transfers are not moved beyond points with earlier transfer decisions. An optimization done here is that if variable Swapout-candidate in scratch-pad is provably not written to in the regions since it was last copied into the scratch-pad, then it need not be written out to DRAM since it has not been modified from its DRAM copy. This optimization provides functionality to the dirty bit in cache, without needing to maintain a dirty bit since the analysis is at compile-time. The end result is an accurate cost model that estimates the benefit of any candidate allocation that the algorithm generates.

Optimization One optimization was developed which ignores the multiple allocation decisions inside higher level regions and instead adopts one allocation inside the particular region. The static allocation adopted is found by doing a greedy allocation based on the frequency per byte value of the variables used in the region. Such an optimization can be useful in cases when transfers are done inside loops and the resulting transfer cost is very high. In cases where the method in [132] would guarantee that the high cost can be recouped, it might be beneficial to adopt a simple static allocation for the particular region. To aid in making this choice, the

method compares the likely benefit from a purely dynamic allocation with a static allocation for the region. Based on the result either the dynamic allocation strategy is retained or the static allocation used for the region.

Code Objects The above framework was also extended to allocate program code objects. There exist several key questions when performing code allocation that need to be answered. First, at what granularity should code objects be allocated(basic-block/procedures/files). Second, how should a code object be represented in the DPRG. Third, how is the algorithm and cost model modified. The first issue concerns the granularity of the program objects. As with data objects, the smaller the size of the code objects, the larger the benefits of scratch-pad placement is likely to be. Keeping this in mind, the granularity of code objects was originally proposed as having units of basic-blocks. However, code generation for allocations at such fine granularity is likely to introduce too many branch instructions while also precluding the use of existing linker technology for its implementation. Another drawback is increased complexity of profiling. A step up from using basic-blocks was chosen and the algorithm creates code objects on the basic of functions, with selective divisions for loop boundaries inside functions. New code object divisions are selectively created at loop boundaries since it is often profitable to place loops in scratch-pad to capture localized re-use. This optimization is based on function outlining (inverse of inlining where loops are instead outlined into procedures), and is available in some commercial compilers like IBMs XLC compiler. Both methods can yield code objects of smaller size but at vastly different implementation costs. For its ease of implementation, function outlining was chosen to provide program objects with fine

enough granularity to make allocation and code generation more feasible.

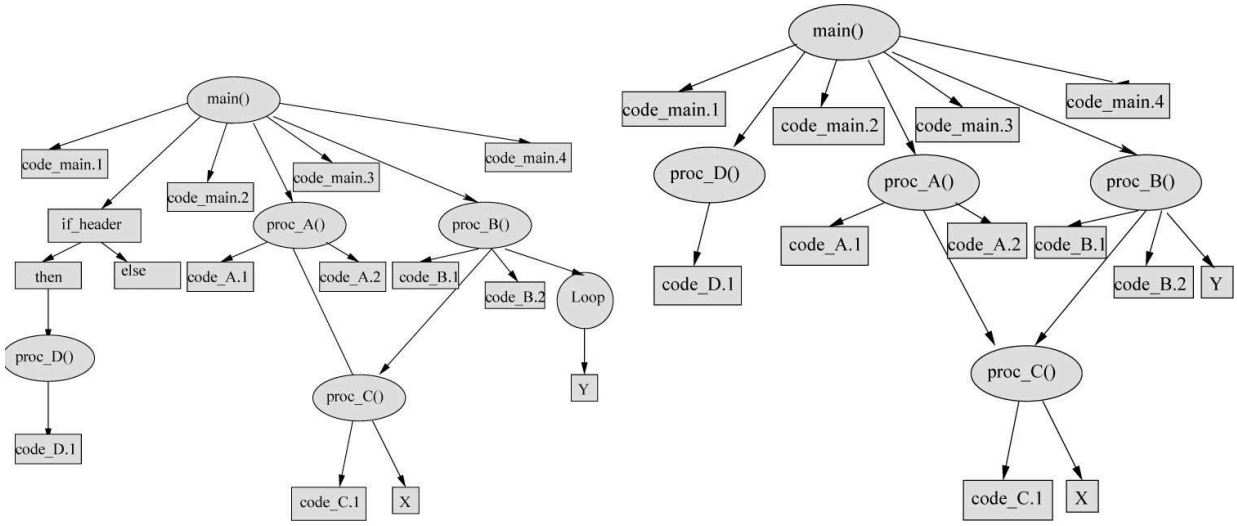


Figure 5.3: Left: Example DPRG with code nodes. Right: An example Coalesced DPRG.

The next issue to be handled is that of representing code objects in the DPRG. Since the choice of program objects is at the level of procedures (native or outlined),code objects are attached to parent procedures just like variables are (henceforth called code variable nodes). Figure 5.3 shows an example of a DPRG which also includes code objects shown as rectangular nodes. Every procedure node has a variable child node representing the code executed before the next function call. For example in figure 5.3 code A.1 represents all the instructions in proc A executed before the procedure proc C is called and code A.2 represents all the instructions in proc A executed after return from proc C until the end the proc A. An advantage of such a representation is that the framework for allocating data objects can be used with little modification for allocating code objects as well. As in the case of data objects, profiling is used to find for every node the frequency of access of each child code variable accessed by that node. For a code variable its frequency is given by

its corresponding number of dynamic instructions executed. The size of the code variable is the size of the portion of the procedure until the next call site in the procedure. A modified DPRG structure is also created in which non-procedure DPRG nodes have been coalesced into the parent procedure node. This new structure is named the Coalesced-DPRG. Figure 5.3 also shows the Coalesced-DPRG for the DPRG in Figure 5.3.

The original allocation algorithm described in the previous section is modified in the following manner. When a procedure node in the DPRG is visited, first we check if the procedure node can be allocated in the scratch-pad. Such an approach is motivated by the same considerations as for the choice of procedures over basic blocks. Using the original algorithm would have required using expensive profiling to find the frequency of the code variable in much smaller portions of code. To determine if a procedure node can be allocated to scratch-pad, it is helpful to use the Coalesced-DPRG. It suffices to find out if a hypothetical allocation (lines 4-30) done at the corresponding procedure node using the Coalesced-DPRG(using lines 7-21), would allocate the procedure node to the scratch-pad. If the procedure gets allocated to the scratch-pad then the available scratch-pad memory is decreased by the size of the procedure node. Then the algorithm proceeds with the rest of the pseudo-code explained in the previous section(lines 4-30) using the original DPRG. The only other difference is that the procedure node is ignored, that is it is neither considered for swap-in or swap-out, for that region. Thus the modified algorithm is able to allocate both data and code while retaining the same overall framework.

5.4 Algorithm Modifications

For simplicity of presentation, the algorithm in the previous two sections leaves some issues unaddressed. Solutions to these issues are proposed in this section. All the modifications proposed here are carried out by the algorithm presented before and are driven by the same cost model. They do not define a new algorithm. and GOTO's simply extend the functionality of the algorithm.

Function pointers and pointer variables that point to global and stack variables Function pointers and pointer variables in the source code that point to global and stack variables can cause incorrect execution when the pointed-to object is moved. For example, consider a pointer variable p that is assigned to the address of global variable a in a region where a is in SPM. Later if p is de-referenced in a region when a is in DRAM, then p points to the incorrect location of a, leading to corrupt data and probable program failure. We now discuss two different alternative strategies to handle this issue. An advantage of both these schemes is that both alternatives need only basic pointer information. However, pointer analysis information can also be used to optimize both the schemes and further reduce their overhead.

Alternative 1: The first alternative presented involves using a run-time disambiguator that corrects the address of the pointer variable. This alternative involves four steps. First, pointer analysis is performed to find the pointers to global/stack and code objects. Second, all statements where the address of a global/stack or a code variable is assigned, including when passed as a reference parameters, are

updated to to use the DRAM address assigned to that object by the compiler. This is not hard since all compilers identify such statements explicitly in the intermediate code. With such a reassignment, all pointer variables in the program refer to the DRAM locations of variables. The advantage of DRAM addresses of objects is that they are unique and fixed during the program objects lifetime; unlike their current address which changes every time the program object is moved between scratch-pad and DRAM. Note that only direct assignments of addresses need to be taken care, statements which copy address from one pointer to another do not need any special handling. The third step inserts code at each pointer de-reference in the program to perform run-time translation into its current address in case its resident in SPM for that region. This translation is done using a custom run-time data structure which, given the DRAM address, returns the current location of the program object. Since pointer arithmetic is allowed in C programs, the data structure must be able to look up addresses to the middle of program objects and not just to their beginnings. The data structure used is a height-balanced tree having one node for each global/stack and code object whose address is taken in the program. Each node stores the DRAM address range of the variable and its current starting address. Since recursive procedures are not allocated to the scratch-pad, each variable has only one unique address range. [4] The tree is height-balanced with the DRAM address as the key. The tree is updated at run-time when a pointed-to variable is transferred between banks and it is accessed through pointers before its next

[4]With pointer analysis information this translation can be replaced by a simpler comparison involving only the dram addresses of variables in the pointed-to set

transfer. Since n-node height-balanced trees offer O(log2N) lookup this operation is reasonably efficient. An advantage of both these schemes if that both alternatives need only basic pointer information. However pointer analysis information can be used to immensely, simplify both the schemes. Once the current base address of the program object in scratch-pad is obtained, the address value may need to be adjusted to account for any arithmetic that has been done on the pointer. This is done by adding the offset of the old pointer value from the base of the DRAM address of the pointed-to variable to the newly obtained current location. The final step that we do is that after the dereference the pointer is again made to point to its DRAM copy. Similar to when translating, the pointer value may need adjustments again to account for any arithmetic done on the pointer. It appears that the scheme for handling pointers described above suffers from high run-time overhead since an address translation (and re-translation) is needed at every pointer de-reference. Fortunately this overhead is actually very low for four reasons. First, pointers to global/stack and code objects are relatively rare; pointers to heap data are much more common. Only the former require translation. Second, most global/stack and code accesses in programs are not pointer de-references and thus need no translation. Third, even when translation and hence a subsequent re-translation is needed in a loop (and thus is time consuming) it is often loop-invariant and can be placed outside the loop. The translation is loop invariant if the pointer variable or aggregate structure containing the pointer variable is never written to in the loop with a address taken expression. [5] This was found to often be the case and in almost all situations

[5]Pointer arithmetic is not a problem since it is not supposed to change the pointed-to variable

the translation can be done before the loop. Consequently the retranslation can be done after the loop. Finally, one optimization is employed for cases where it can be conservatively shown that the variables address does not change between the address assignment and pointer de-reference. For these cases the current address of the program object, regardless of it is in the scratch-pad or DRAM, can be assigned and no translation is required. Such instances most trivially happen in cases when for optimized code generation, the address is assigned to a pointer just before the loop and then the pointer variable is used in the loop. For all these reasons, the run-time overhead of translation was found to be under 1the benchmarks by the much larger gain from the method in access latency. Finally, the memory footprint of the run-time structure was observed to be very small for the benchmark suite.

Alternative 2: A second alternative was taken from the work in this thesis for heap data allocation. It uses a strategy of restricting the offsets a pointed-to variable can take. The strategy proceeds in the following steps. First, a variable whose address is never taken is placed with no restrictions since no pointers can point into it. Address-taken information is readily available in most compilers; in this way, many global/stack variables are unaffected by pointers. Second, variables whose address is taken have the following allocation constraint for correctness: for all regions where the variables address is taken or the variable may be accessed through pointers, the variable must be allocated to the same memory location. For example if variable a has its address taken in region R1, and may be accessed through a pointer in region R5, then both regions R1 and R5 must allocate a to

in ANSI-C semantics.

the same memory location. This ensures correctness as the intended and pointed-to memory will be the same. The consensus memory bank for such regions is chosen by first finding the locally requested memory bank for each region; then the chosen bank is the frequency-weighted consensus among those requests. Regions in which a variable with address taken is accessed but not through pointers are unconstrained, and can allocate the variable anywhere. Currently, the results presented in [132] only explored alternative 1 for the benchmarks and future work is planned to compare both alternatives as well as hybrids of the two in an effort to maximize performance.

Join nodes A second complication with the allocation algorithm from [132] arises for any program point visited from multiple paths (hence having multiple timestamps). For these program points, the pseudo-code loop in line 4 from Figure 5.2 is visited more than once, and thus more than one allocation is made for that program point. An example is node proc C() Figure 5.3. These nodes with multiple timestamps are termed "join" nodes since they join multiple paths through the DPRG during program execution. Join nodes can arise due to many program constructs including (i)in the case of a procedure invoked at multiple call sites, (ii)at the end of conditional path or (iii) in the case of a loop. For parents of join nodes, considering the join node multiple times in the algorithm is not a problem - indeed it the right thing to do, so that the impact of the join node is considered separately for each parent. However, for the join node itself, multiple recommended allocations result, one from each path to it, presenting a problem. One solution is cloning the join node and the sub-graph below it in the DPRG along each path to the join node, but the code growth can be exponential for nested join nodes. Even selective cloning

is probably unacceptable for embedded systems. Instead, the strategy avoids all cloning by choosing the allocation desired by the most frequent path to the join node for the join node. Subsequently compensation code is added on all incoming edges to the join node other than for the most frequent path. The compensation code changes the allocation on that edge to match the newly computed allocation at the join node. The number of instances of compensation code is upper-bounded by the number of incoming edges to join nodes We now consider the most common scenarios separately.

Join nodes: Procedure join nodes The method chooses the allocation desired by the most frequent path to the procedure join node for the join node. Subsequently as discussed before, compensation code is added on all incoming edges to the join node other than for the most frequent path. Join nodes: Conditional join nodes Join nodes can also arise due to conditional paths in the program. Examples of conditional execution include if-then, if-then-else and switch statements. In all cases, conditional execution consists of one or more conditional paths followed by an unconditional join point. Memory allocation for the conditional paths poses no difficulty each conditional path modifies the incoming memory allocation in the scratch-pad and DRAM memory to optimize for its own requirements. The difficulty is at the subsequent unconditional join node. Since the join node has multiple predecessors, each with a different allocation, the allocation at the join node is not fixed at compile-time. The solution used is the same as for procedure join nodes and is used for similar reasons. Namely, the allocation desired by the most frequent path to the join node is used for the join node, just as above.

Join nodes: loops A third modification is needed for loops. A problem akin to join nodes occurs for the start of such loops. There are two paths to the start of the loop a forward edge from before the loop and a back edge from the loop end. The incoming allocation from the two paths may not be the same, violating the desired condition that there be only one allocation at each program point. To find the allocation at the end of the backedge, Procedure Find-swapout-set is iterated once over all the nodes inside the loop. The allocation before entering the loop is then reconciled to obtain the allocation desired just after entering the loop in this way, the common case of the back edge is favored for allocation over the less common forward edge.

Recursive functions The approach discussed so far does not directly apply to stack variables in recursive or cross-recursive procedures. With recursion the call graph is cyclic and hence the total size of stack data is unknown. Hence for a compiler to guarantee that a variable in a recursive procedure fits in the scratch-pad is difficult. The baseline technique is to collapse recursive cycles to single nodes in the DPRG, and allocate their stack data to DRAM. Edges out of the recursive cycle connect this single node to the rest of the DPRG. This provides a clean way of putting all the recursive cycles in a black box [6]. The method can now handle the modified DPRG like any other DPRG without cycles.

Goto statements The DPRG formulation in section 5.3 does not consider arbitrary Goto statements. This is mostly because it is widely known that Goto

[6]Recursive functions will not be considered in the future by Udayakumaran, as methods presented later in this thesis have been developed to allocate them.

statements are poor programming practice and they are exceedingly rare in any domain nowadays. Nevertheless, it is important to handle them correctly as valid language feature. Only Goto statements are being in question here; breaks and continues in loops are fine for DPRGs.

The solution to correctly handle Goto statements involves two steps. First, the DPRG is built and the memory transfers are decided without considering Goto statements. Second, the compiler detects all Goto statements and inserts memory transfer code along all Goto edges in the control-flow graph to maintain correctness. The fundamental condition for correctness in the overall scheme is that the memory allocation for each region is fixed at compile-time; but different regions can have different allocations. Thus for correctness, for each Goto edge that goes from one region to another, memory transfers are inserted just before the Goto statement to convert the contents of scratch-pad in the source region to that in the destination region. In this way Goto statements are handled correctly but without specifically optimizing for their presence. Since Goto statements are very rare, such an approach adds little run-time cost for most programs.

The DPRG construct along with the extensions in this section enable this method to handle all ANSI C programs. For other languages, structured control-flow constructs likely will be variants, extensions or combinations of constructs mentioned in [132], namely procedure calls, loops, if and if-then-else statements, switch statements, recursion and goto statements.

5.5 Layout and Code Generation

This section has three issues. First, it discusses the layout assignment of variables in scratch-pad. Second, it discusses the code generation for this scheme. Third, it discusses how the data transfer code may be optimized.

Layout assignment The first issue in this section is deciding where in the scratch-pad to place the program objects being swapped in. A good layout at a region should be able to place most or all of the program objects desired in the scratch-pad by the memory transfer algorithm in section 5.3. To increase the chances of finding a good layout, the layout assignment algorithm should have the following two characteristics. First, the layout should minimize fragmentation that might result when program objects are swapped out, so as to increase the chance of finding large-enough free holes in future regions. Second, when a memory hole of a required size cannot be found, compaction in scratch-pad should be considered along with its cost.

The layout assignment algorithm runs as a separate pass after the memory transfers are decided. It visits the regions of the application in the partial order of their timestamps. At each region, it does the following four tasks. First, the method updates the list of free holes in the scratch-pad by de-allocating the outgoing variables from the previous region. Second, it attempts to allocate incoming variables to the available free holes in the decreasing order of their size. The largest variables are placed first since they are the hardest to place in available holes. When more than one hole can be used to fit a variable, the best-fit rule is followed: the smallest

hole that is large enough to fit the incoming program object is used for allocation. The best-fit rule is commonly used for memory allocation in varying domains such as segmented memory and sector placement on disks [133].

Third, when an adequate-sized hole cannot be found for a variable, compaction in the scratch-pad is considered. In general, compaction is the process of moving variables towards one end of memory so that a large hole is created at the end. However, a limited form of compaction is considered that has lower cost: only the subset of variables variables that need to be moved to create a large-enough hole for the incoming request are moved. Also, for simplicity of code generation, compaction involving blocks containing program objects used inside a loop is not allowed inside the loop. Compaction is often more attractive than leaving the incoming program object in DRAM for lack of an adequate hole this is because compaction only requires two scratch-pad accesses per word, which is often much lower cost than even a single DRAM access. The cost of compaction is included in the layout-phase cost model; it is done only when its cost is recovered its benefit. Compaction invalidates pointers to the compacted data and hence is handled just like a transfer in the pointer-handling phase (section 5) of the method. Pointer handling is delayed to after layout for this reason.

Fourth, in the case that compaction is not profitable, the approach attempts to find a candidate program object to swap out to DRAM. Again, the cost is weighed against the benefit to decide if the program object should be swapped out. If no program object in the scratch-pad is profitable to swap out, the approach decides to not bring in the requested-incoming program object to the scratch-pad. Fortunately,

the results from [132] show that this simple strategy is quite effective.

Code generation After the method decides the layout of the variables in SRAM in each region, it generates code to implement the desired memory allocation and memory transfers. Code generation for the method involves changing the original code in three ways. First, for each original variable in the application (Eg: a) which is moved to the scratchpad at some point, the compiler declares a new variable(Eg: a_fast) in the application corresponding to the copy of a in the scratch-pad. The original variable a is allocated to DRAM. By doing so, the compiler can easily allocate a and a_fast to different offsets in memory. Such addition of extra symbols causes zero-to-insignificant code increase depending on whether the object formats includes symbolic information in the executable or not. Second, the compiler replaces occurrences of variable a in each region where a is accessed from the scratch-pad by the appropriate version of a_fast instead. Third, memory transfers are inserted at each program point to evict some variables and copy others as decided by the method. The memory transfer code is implemented by copying data between the fast and slow versions of to-be-copied variables (Eg: between a_fast and a). Data transfer code can be optimized; optimizations are described later in this section.

Since the method is dynamic, the fast versions of variables (declared above) have limited lifetimes. As a consequence different fast variables with non-overlapping lifetimes may have overlapping offsets in the scratch-pad address space. Further, if a single variable is allocated to the scratch-pad at different offsets in different regions, multiple fast versions of the variables are declared, one for each offset. The requirement of different scratch-pad variables allocated to the same or overlapping

offsets in the scratch-pad in different regions is easily accomplished in the back-end of the compiler.

Although creating a copy in scratch-pad for global variables is straightforward, special care must be taken for stack variables. Stack variables are usually accessed through the stack pointer which is incremented on procedure calls and decremented on returns. By default the stack pointer points to a DRAM address. This does not work to access the stack variable in scratch-pad; moreover the memory in scratch-pad is not even maintained as a stack! Allocating whole frames to scratch-pad means losing allocation flexibility. The other option of placing part of stack frame in scratch-pad and the rest in main memory requires maintaining two stack pointers which can be a lot of overhead. The easiest way to place a stack variable 'a' in scratch-pad is to declare its fast copy 'a_fast' as a global variable but with the same limited lifetime as the stack variable. Addressing the scratch-pad copy as a global avoids the difficulty that the scratch-pad is not maintained as a stack. Thus all variables in scratch-pad are addressed as globals. Having globals with limited lifetimes is equivalent to globals with overlapping address ranges. The handling of overlapping variables was mentioned in the previous paragraph.

Code generation for handling code blocks involves modifying the branch instructions between the blocks. The branch at the end of the block would need to be modified to jump to the current location of the target. This is easily achieved when the unit of the code block is a procedure by leveraging current linking technology. Similar to the case of variables, the compiler inserts new procedure symbols corresponding to the different offsets taken by the procedure in the scratch-pad. Then

it suffices to modify the calls to call the new procedures. The back-end and the linker would (without any modifications) then generate the appropriate branches. As mentioned earlier, outlining or extracting loops into extra procedures can be used to create small sized code blocks. For this optimization to work, the local variables that are shared between the loop and the rest of the code are promoted as global variables. These are given unique names prefixed with the procedure name. In the set of benchmarks, it was observed that the overhead due to these extra symbols was very small.

Reducing run-time and code size of data transfer code The method copies data back and forth between the scratch-pad and DRAM. This overhead is not unique to this approach hardware caches also need to move data between scratch-pad and DRAM. The code-size overhead of such copying is minimized by using a shared optimized copy function. In addition, faster copying is possible in processors with the low-cost hardware mechanisms of Direct Memory Access (DMA) such as in ARMv6 or ARM7. DMA accelerates data transfers between memories and/or I/O devices.

5.6 Summary of results

This paper presents compiler-driven memory allocation scheme for embedded systems that have SRAM organized as a scratch-pad memory instead of a hardware cache. Most existing schemes for scratch-pad rely on static data assignments that never change at run-time, and thus fail to follow changing working sets; or use soft-

ware caching schemes which follow changing working sets but have high overheads in run-time, code size memory consumption and real-time guarantees. The scheme presented in [132] follows changing working sets by moving data from scratch-pad to DRAM, but under compiler control, unlike in a software cache, where the data movement is not predictable. Predictable movement implies that with this method the location of each variable is known to the compiler at each point in the program, and hence the translation code before each load/store needed by software caching is not needed. The benefit of our method depends on the scratch size used. The method in [132] were implemented in the GCC v3.2 cross-compiler, targeting the Motorola M-Core embedded processor. After compilation, the benchmarks are executed on the public-domain and cycle-accurate simulator for the Motorola M-Core available as part of the GDB v5.3 distribution. When compared to a provably optimal static allocation, results show that this scheme reduces run-time by up to 39.8% and overall energy consumption by up to 31.3% on average for the benchmarks, depending on the scratch-pad size used.

Chapter 5

Dynamic program data

For simpler programs that use only static program data such as global and stack variables, many effective allocation methods have been proposed to take advantage of SPM for embedded systems. Chapter 3 presented a summary of all relevant SPM allocation methods. The previous chapter discussed the best existing SPM allocation method in detail. Surprisingly enough, after searching through more than two dozen papers presenting SPM allocation methods, we only found a few that mentioned heap data. These few did so only in the context that their methods lacked automatic compiler support for its handling. None were able to alter heap allocation at runtime for runtime and energy savings using existing dynamic SPM allocation methods.

In order to optimize dynamic program data with compiler methods, we have built upon ideas from previous allocation methods and have incorporated several new concepts. This chapter will begin with a discussion of dynamic program data including the motivation behind its use and the characteristics which make it hard to optimize. This is followed by a section which discusses the modifications we have made to the DPRG to incorporate dynamic program data in its representation. Finally we conclude the chapter with an introduction to our compiler method for dynamic allocation of dynamic program data using the enhanced DPRG, before

discussing it in complete detail in the next chapter.

5.1 Understanding dynamic data in software

Our methods will most interest those programmers designing modern software for embedded platforms that rely on high-level compiler languages for development. Compiler tools for dynamic program memory have lagged far behind that of static program code, stack and global data objects for embedded platforms. It is only in the past few years that modern embedded engineers have enjoyed advanced compilers capable of generating efficient machine code targeting embedded processors from high-level languages. With the benefits of our dynamic memory allocation methods, programmers can now make best use of Scratch-Pad Memory on their embedded platforms for a much wider range of software applications. Of course, even with the extensions proposed and implemented for our current compiler platform, there are classes of applications which make poor candidates for our methods. During the course of this research, we have observed a variety of development scenarios where our methods are most and least profitable. This section will discuss the observed program characteristics and their effect on our method's efficacy.

A program that dynamically allocates data at runtime is the single most likely candidate to benefit from our dynamic memory allocation method. Almost all programs will benefit from our previously developed methods that are able to dynamically allocate stack, global and code data to Scratchpad Memory. This becomes critical as target applications become more complex, consuming more storage space

at runtime and overflowing local register storage into main memory. Our current methods handle the complicated problem of managing runtime allocated dynamic data for optimal placement among heterogeneous memory banks. Dynamic memory allocation is an essential feature for high-level languages such as C and C++, where most advanced applications will employ some form of runtime memory management and until now has been strictly placed into main memory for other allocation schemes. *Dynamic program data can take two forms, one being program objects allocated from heap memory and the other being program objects grown recursively inside of self-referencing function calls.* This chapter will present an overview of dynamic program data and how our method views these program objects for optimization.

Heap Memory in C The C programming language normally manages memory either statically or automatically. Static-duration (global)variables are allocated in main (fixed) memory and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and are created and destroyed as functions are called and return. Both these forms of allocation are limited, as the size of the allocation must be a compile-time constant. If the required size will not be known until run-time. For example, if data of arbitrary size is being read from a file or input buffer, then using fixed-size data objects becomes inadequate.

The lifetime of allocated memory is also a concern for programmers. Neither static- nor automatic-duration memory is adequate for all situations. Stack-allocated data can obviously not be persisted across multiple function calls, while global data persists for the life of the program whether you need it to or not. In

many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly but more flexibly managed, typically by allocating it from a heap, an area of memory structured for this purpose. In C, one uses the library function *Malloc* to allocate a block of memory on the heap. The program accesses this block of memory via a pointer which malloc returns. When the memory is no longer needed, the pointer is passed to *Free* which deallocates the memory so that it can be reused by other parts of the program.

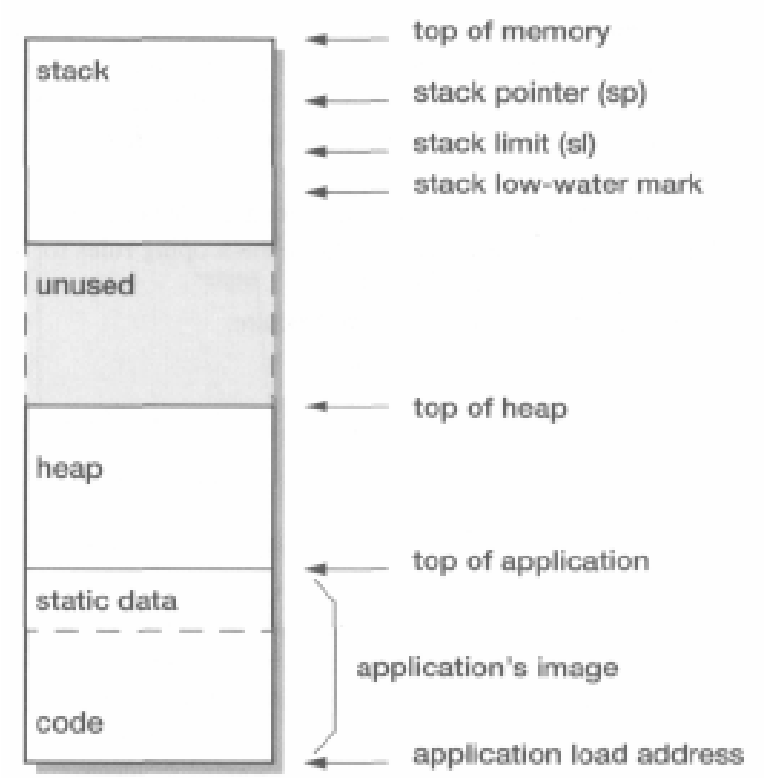


Figure 5.1: View of main memory for a typical ARM compiled binary application, showing heap, stack and global memory areas.

Figure 5.1 shows a view of main memory for an ARM binary executable that

has been loaded onto an embedded platform. Stack memory traditionally begins at the topmost available memory address and grows downward into lower memory addresses as the program call depth increases. Program code is typically loaded at the lowest memory segment available on an ARM platform, with static global data placed immediately above. The heap memory area begins at the lowest free memory address available for the system and grows upwards as objects are allocated at runtime.

The *malloc* function is the basic function used to allocate memory on the heap in C. Its prototype is:

```
void *malloc(size_t size);
```

This code fragment allocates *size* bytes of memory. If the allocation request fails, a null pointer is returned. If the allocation succeeds, a pointer to the block of memory is returned. This pointer is typically cast to a more specific pointer type by the programmer before being used.

Memory allocated via malloc is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer (that is, the block is said to be "freed"). This is achieved by use of the *free* function. Its prototype is

```
void free(void *pointer);
```

This code fragment releases the block of memory pointed to by *pointer*. *pointer* must have been previously returned by malloc or calloc and must only be passed to free once.

The standard method of creating an array of ten integers on the stack or as a global variable is:

```
int array[10];
```

To allocate a similar array dynamically, the following code could be used:

```
int *ptr = malloc(10 * sizeof (int));
```

There exist several other ANSI C heap management functions which programmers sometimes make use of. One alternative is to use the *calloc* function, which allocates memory and then initializes all allocated bytes to zero. Another function named *realloc* allows a programmer to grow or shrink a block of memory allocated by a previous heap function. Many variants exist among different implementation of the standard C libraries, particularly among embedded systems. In general, libraries targeted for embedded platforms usually use a compact and highly tuned implementation to manage heap memory.

Recursive Functions The concept of recursion is one essential for computer programmers in languages such as LISP, which is a language based mostly on recursive function calls and dynamic list data structures. Virtually all programming languages in use today allow the direct specification of recursive functions and procedures. When such a recursive function is called in C for example, the computer or the language implementation keeps track of the various instances of the function usually by using a call stack, although other methods may be used. Since many recursive functions reach a dynamic stack call depth, they are generally unbounded at

compile-time and not amenable to typical bounded memory allocation optimization strategies.

The Oxford English Dictionary recursively defines *recursion* as "The application or use of a recursive procedure or definition"! *Recursive* is defined as "Involving or being a repeated procedure such that the required result at each step except the last is given in terms of the result(s) of the next step, until after a finite number of steps a terminus is reached with an outright evaluation of a result." In software, recursion is when a function or method calls itself repeatedly (usually with slightly different input arguments). Recursion allows the writing of elegant and terse code for some algorithms although its programming complexity increases dramatically as the recursive function is redesigned to handle more possible situations. The increased complexity generally comes with a loss of automatic compiler optimization effectiveness because of the explosion of possible paths through a recursive function graph.

Recursion in computer programming defines a function in terms of itself. Recursion is deeply embedded in the theory of computation, with the theoretical equivalence of mu-recursive functions and Turing machines at the foundation of ideas about the universality of the modern computer. A good example application of recursion is in parsers for programming languages. The great advantage of recursion is that an infinite set of possible sentences, designs or other data can be defined, parsed or produced by a finite computer program. The popular Quicksort and Mergesort algorithms are also commonly done using recursion. Some numerical methods for finding approximate solutions to mathematical equations rely entirely on recursion.

In Newton's method, for example, an approximate root of a function is provided as initial input to the method. The calculated result (output) is then used as input to the method, with the process repeated until a sufficiently accurate value is obtained.

Dynamic Data Structures

Dynamically created data structures like trees, linked lists and hash tables (which can be implemented as arrays of linked lists) are key to the construction of many large software systems. For example, a compiler for a programming language will maintain symbol tables and type information which is dynamically constructed by reading the source program. Many modern compilers also parse the source program and translate it into an internal tree form (abstract syntax tree) that is also dynamically created. Graphics programs, like 3D rendering packages, also make extensive use of dynamic data structures. In fact it is rare to find any program that is larger than a couple of thousand lines that does not make use of dynamically allocated data structures.

For programmers dealing with large sets of data, it would be problematic to have to name every structure variable containing every piece of data in the code – for one thing, it would be inconvenient to enter new data at run time because you would have to know the name of the variable in which to store the data when you wrote the program. For another thing, variables with names are permanent – they cannot be freed and their memory reallocated, so you might have to allocate an impractically large block of memory for your program at compile time, even though you might need to store much of the data you entered at run time temporarily. Fortunately, complex data structures are built out of dynamically allocated memory, which does

not have these limitations. All a program needs to do is keep track of a pointer to a dynamically allocated block, and it will always be able to find the block.

There are some advantages to the use of dynamic storage for data structures. Since memory is allocated as needed, we don't need to declare how much we shall use in advance. Complex data structures can be made up of lots of "lesser" data structures in a modular way, making them easier to program. Using pointers to connect structures means that they can be re-connected in different ways as the need arises. Data structures can be more easily sorted, for example.

Dynamic program memory management is one of the main reasons that high-level object oriented programming languages such as Java have become popular. Java does not support explicit pointers (which are a source of a lot of complexity in C and C++) and it supports garbage collection. This can greatly reduce the amount programming effort needed to manage dynamic data structures. Although the programmer still allocates data structures, they are never explicitly deallocated. Instead, they are "garbage collected" when no live references to them are detected. This avoids the problem of having a live pointer to a dead object. If there is a live pointer to an object, the garbage collection system will not deallocate it. When an object is no longer referenced, its memory is automatically recovered. In theory, Java avoids many of the problems programmers have with dynamic memory. The cost, however, is in performance and predictability. Automatic garbage collection is usually much less efficient than adequate programmer managed allocation and deallocation. Also, garbage collectors tend to deallocate objects from a low level, which can further hurt performance. Finally, garbage collection severely degrades

real-time bounds for designers concerned with WCET. For embedded processors, the cost of managed high-level languages is simply too high and they are rarely used in practice.

Recursive Data Structures Dynamically allocated data is particularly useful for representing recursively defined data structures. These types of structures are common when working with high-level languages, and are ubiquitous in computer algorithms. Programmers often use a particular kind of recursive data structure called a tree to represent hierarchical data. A tree is recursively defined in graph theory as a root with zero or more subtrees. Each subtree itself consists of a root node with zero or more subtrees. A subtree node with no branches or children is a leaf node. A classic use of recursion is for tree traversal, where actions can be performed as the algorithm visits each node in the dynamically allocated tree.

The following example will illustrate use of tree data structure in a C applications. A tree can be implemented in various ways, depending on the structure and use of the tree. Let us assume a tree node consists of a scalar data element and three pointers for the possible children of the node:

```
typedef struct Tree_Node {
    int data;
    Tree_Node* left;
    Tree_Node* middle;
    Tree_Node* right;
}
```

In figure 5.2, we see a 5 level tree data structure labeled with individual nodes from A - O. Node A is the root of the tree since it is the only node with no parents and node G is a leaf node since it has no children. With these types of recursive

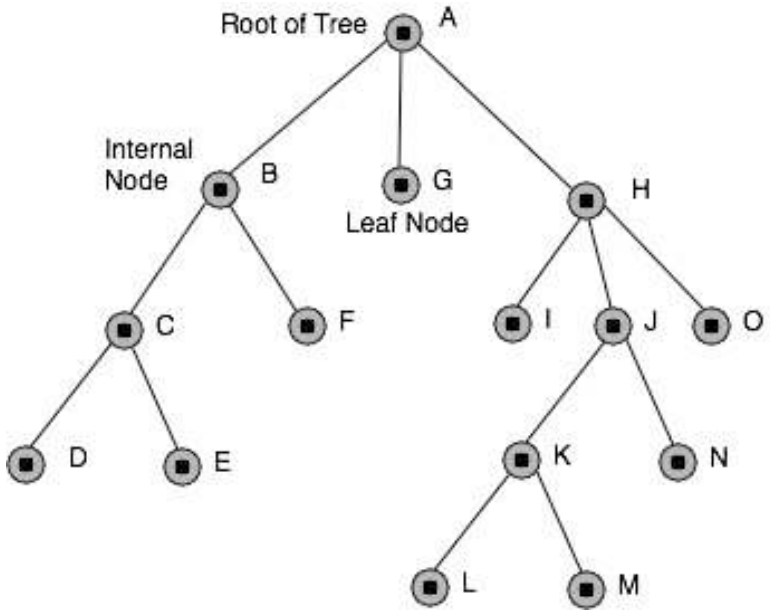


Figure 5.2: An example of a binary tree node data structure with a recursive data pointer.

data structures, recursive algorithms are often used to dynamically traverse the structures to perform operations on the data or structure itself. There are three types of recursive tree traversals: preorder, inorder and postorder - each defines the particulars of whether you work on a node before or after working on its children.

Given the tree pictured in Figure 5.2, let us walk through what would happen on an post-order traversal (DFS) of the tree. First we would call the postorder algorithm on the root node (node A), placing this method call on the stack. Node A has 3 children so we call the postorder function recursively on the first child (node B), pushing that call on the stack. Node B has 2 children, so we again call the postorder function on the first child (node C), placing that call on the stack. We then call the method on node Cs first child, node D. Now the stack is 4 function calls deep, but this final node is a leaf, so some processing is done to its data and

the deepest postorder function call returns. Now the algorithm is back in Node C, but the postorder function has moved on to its second child (node E), increasing the stack depth to 4 again. After the function finishes processing the lead node data, execution works its way recursively up and down across all children for all nodes. During this process, the program is using pointers to traverse the data structures, performing memory accesses for each visit to heap objects as well as the growing and shrinking stack objects.

Binary trees, linked lists, graphs and skip lists are only a few of the many kinds of recursive data structures frequently used by programmers. Unfortunately, dynamic data structures and even simple heap objects are problematic for optimization precisely because of their dynamic runtime behavior. The following section will discuss the problems with existing SPM allocation methods that optimize static program data allocation. As we will see in our results chapter, being able to analyze and optimize dynamic program data with compiler methods can have a dramatic effect on efficiency for programs at all levels of complexity.

5.2 Obstacles to optimizing software with dynamic data

Our methods will most interest those programmers designing modern software for embedded platforms that rely on high-level languages for development. As embedded devices become more prevalent, so has the sophistication of the software deployed on these devices. The increased sophistication generally requires more levels of abstraction in the software development layer. Instead of traditional assembly

languages, most modern embedded developers now employ high-level languages and compiler support in the embedded domain. One distinct advantage of high-level languages is the ease with which programmers can dynamically allocate memory to handle inputs or algorithms of widely varying sizes. Unfortunately, the performance of dynamically allocated memory has lagged far behind that of statically allocated program objects due to a number of reasons.

From the previous section we saw that dynamic data structures provide programmers with a powerful tool for flexible memory storage in their complex applications. This implementation flexibility is also the root for the fundamental difficulty in automatic compiler analysis of applications using dynamic program memory. Many accurate methods have been developed to statically predict the relative execution paths and access frequencies for simpler applications which rely on static program data allocation. For complex programs using dynamic data allocation, it is no longer sufficient to only perform a static analysis, since much of the program behavior will instead be input dependent. The very presence of dynamic memory usage in a program implies that the application expects program input of some sort that in unknown at design-time. The actual path an application takes at runtime and the relative access frequencies to data objects during its lifetime can vary dramatically for programs operating on dynamic data structures. From this we see that we must include comprehensive methods in our SPM allocation methods to ensure that it is able to optimize a given application across the most number of inputs and minimize the over-customization typical of profile-guided methods. We note that by necessity all locality enhancement schemes such as ours are inherently profile de-

pendent and this is not a problem unique to our method. Nevertheless, while other published research has glossed over this aspect, it becomes critical when dealing with larger applications and so must be addressed for our proposed methods.

The nature of dynamic memory implementation for C has been seen as both a blessing and a curse by most programmers due to its completely manual approach to dynamic memory management. As noted previously, C is the lowest-level "high-level" language in use today, which explains its overwhelming popularity for embedded design. C programmers have fine-grain control over dynamic memory allocation at runtime in their programs, but must also be careful to properly manage and make use of that dynamic memory. The actual implementation of the C library routines implementing heap management is also of frequently interest to embedded designers and should be investigated in parallel with orthogonal dynamic memory optimizations. The choice of a heap management algorithm can heavily influence the overhead and predictability for a platform using many programs with dynamic memory. Whenever possible, research in dynamic memory optimizations has striven to minimize overhead for dynamic memory management and improve safety and predictability whenever possible. Our own research in SPM management for dynamic data provides a customized memory management solution for embedded programmers with low overhead, predictable latencies and safe dynamic memory transfers for improved locality.

Finally, the largest concern for any memory locality optimization that performs dynamic runtime changes to memory is that of data access safety and correctness. This is a concern for the entire range of dynamic memory optimizations,

from automatic hardware methods such as caches to software methods like software caching. Because dynamic memory is accessed entirely through pointers in C and C++, pointer correctness much be maintained across the entire program for any changes made. This is avoided in more complex compiler languages such as Java, although not without incurring significant costs in efficiency and complexity. Our own methods were developed with memory safety as one of the cornerstones of its implementation. Using detailed static and dynamic program analysis, our compiler method enforces memory safety before modifying a program's memory allocations at runtime to improve access locality.

Comparison with existing dynamic memory optimizations

To illustrate the difficulties in optimizing dynamic program data for dynamic SPM placement, it is helpful to review the advances made in the similar research into optimizing the layout of dynamic data structures to improve cache performance. In general, the goal of any memory optimization is to improve the effectiveness of a computer memory system. The goal for a software optimizer is to help the memory system by improving the program locality, either temporal, spatial or both. To alter the temporal locality, an optimizer must be able to modify the algorithm of the program, which has only proved possible for certain stylized scientific code, where transformations such as loop tiling and loop interchange can significantly increase temporal and spatial locality. Unfortunately, many program structures are usually too complex to be transformed using these types of optimizations. This is a situation common in programs that manipulate pointer-based data structures and most schemes to optimize dynamic program memory placement have focused

on making such structures cache conscious.

Perhaps the most researched field concerning the optimized placement of dynamic program memory is that of data-layout optimization for cache hierarchies. Indeed, this has become the standard for dynamic memory investigation because almost all modern computer systems employ cache hierarchies in their design, in an attempt to dynamically exploit data locality among its running applications. By looking at the progress made in data-layout cache optimization, we can draw similarities between that field and and our own research on dynamic SPM management for dynamic program data.

Data-layout optimizations create a layout with good spatial locality generally by (i) attempting to place contemporaneously accessed memory locations in physical proximity (i.e., in the same cache block or main-memory page), while (ii) ensuring that frequently accessed memory cells do not evict each other from caches. It turns out that these goals make the problem of finding a good layout not only intractable but also poorly approximable [114]. The key practical implication of this hardness result is that it may be difficult to develop data-layout heuristics that are both robust and effective (i.e., able to optimize a broad spectrum of programs consistently well).

The hardness of the data-layout problem is also reflected in the lack of tools that static program analysis offers to a data layout optimizer. First, there appear to be no static models for predicting the dynamic memory behavior of general-purpose programs that are both accurate and scalable, although some successes have been achieved for small C programs [5]. Second, while significant progress has been made in deducing shapes of pointer-based data structures, in order to create a good layout

it is necessary to also understand the temporal nature of accesses to these shapes. This problem appears beyond current static analyzers even for powerful compiler packages. The inadequacy of static analysis information has been recognized by existing data layout optimizations, which are all either profile- guided or exploit programmer-supplied application knowledge. Although many of these techniques manage to avoid the problem of statically analyzing the program memory behavior by instead observing it at run-time, they are still fundamentally constrained by the difficult problem of selecting a good layout for the observed behavior [114]. Typically based on greedy profile guided heuristics, these techniques provide no guarantees of effectiveness and robustness. By observing which areas have proven difficult for other researchers working with dynamic program data, we have been able to incorporate a number of features into our own approach to overcome such problems.

5.3 Creating the DPRG with dynamic data

The previous sections presented material to help in understanding the characteristics that make dynamic program data both useful and difficult to work with. In this section, we begin the presentation of our own methods for optimizing dynamic data by introducing our compiler analysis framework. Our optimization methods operate on a modified version of the Dynamic Program Region Graph(DPRG), which was reviewed in the last chapter on SPM allocation for static program data. The DPRG is a combination of the control flow graph and data flow graph for a

particular application, augmented with dynamic profile information. This section will present details on the version of the DPRG used for our research on dynamic program data optimizations.

Our method defines regions and timestamps in the same way as for our method for global and stack data [132]. A region is a contiguous portion of code in which the allocation to scratch-pad is fixed. Boundaries of regions are called 'program points', and thus regions can be defined by defining a set of program points. Code to transfer data between scratch-pad and DRAM is inserted only at the program points.

Program points and hence regions are found as follows. Promising program points are (i) those after which the program has a significant change in locality behavior, and (ii) those whose dynamic frequency is preferably less than the frequency of its following region, so that the cost of copying into SRAM can be recouped by data re-use from SRAM in the region. For example, sites just before the start of loops are promising program points since they are infrequently executed compared to the insides of loops. Moreover, the loop often re-uses data, justifying the cost of copying into SRAM. With the above two criteria in mind, we define program points as *(i) the start and end of each procedure; (ii) just before and just after each loop* (even inner loops of nested loops); (iii) the start and end of each if statement as well as the start and end of each possible path of the evaluation dependent code blocks; and (iv) the start and end of each case in all switch statements as well as the start and end of the entire switch statement. These points correspond to the finest granularity of the compiler which is the assembly language level.

```
[example.c]

main() {
    if (...) {
        Ptr = func-A() }
    else {
        func-B(x) }
    ...
    return
}

func-A() {
    return malloc(4)
}

func-B(int x) {
    for (i=1 to x) {
        Y[i] = func-A()
    }
}
```

Figure 5.3: A sample program containing three functions and three variables.

To illustrate the DPRG and how we augment it to handle dynamic program data, let us assume our compiler encounters the following program:

The code fragment in Figure 5.3 contains a simple program outline for illustrative purposes. It consists of three procedures, namely *main()*, *func-A()* and *func-B()*. Procedure *main()* contains an if-then conditional block of code which chooses between the two functions *func-A* and *func-B*. The "then" code block assigns the pointer variable *Ptr* the value returned by procedure *Func-A*. Procedure *func-A* allocates 4-bytes of memory from the heap and returns a pointer to the allocated memory. The "else" code block instead calls procedure *Func-B* if that path through the if-then condition is taken. Procedure *func-B* contains a loop that we

name *Loop_1* and which makes accesses to the stack array variable *Y* by allocating heap memory for its array of pointers. Only loops, conditional blocks, procedure declarations and procedure calls are shown along with three selected variables Ptr, x and Y – other instructions and constructs are not. We will present more detailed presentation on some special cases for the DPRG in the next two chapters.

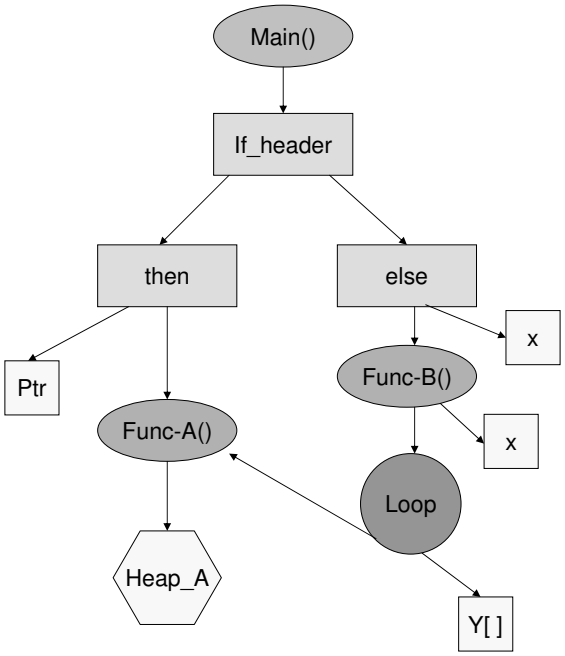


Figure 5.4: Static DPRG representation of the example code showing a heap allocation site.

Figure 5.3 shows the DPRG constructed during the initial phases of our optimization scheme for the same program example. The initial phase of our method conducts static compiler analysis to create a static version of our DPRG. This is augmented later with profile information from the application to provide dynamic access statistics and complete the DPRG. For very simple program code which does not take an input, the static and dynamic DPRG structures are almost identical

and so may be used interchangeably without coalescing into the final DPRG. The dynamic information concerning access frequency is carried inside the edge data structure between all nodes in the DPRG and comes mostly from run-time profile information. Static analysis is able to determine frequencies for some global,code and stack variable at compile-time. But, as with our example, often there is insufficient information from just the program code without knowledge of program inputs and runtime behavior. For heap data, the static analysis only reveals the heap allocation sites in an application, along with their allocation size without reliable access information or runtime behavior prediction. Dynamic program behavior information is gathered from runtime profiles, using a target input set where applicable. The example code in 5.3 shows the most common type of heap object allocation site, that of a known-size unit for every call. For heap data with an allocation size that cannot be bound at compile-time, we present methods in Chapter 7 to handle these types of objects.

In the DPRG shown in Figure 5.3, there are three procedures, one loop, one heap allocation site and one if-then condition construct. Separate nodes are shown for the entire if statement(called if-header) and for its then and else parts. The oval nodes in the figure represent procedures, circular nodes represent loops, rectangular nodes represent if statement nodes, square nodes represent global and regular stack variables and octagonal nodes represent heap objects. Edges in the diagram directed to procedures indicate a call to that function. Edges to loop and if nodes show that the node is the child to its originating parent. Edges to program objects represent memory accesses to that program object from its parent. the DPRG is usually

a directed acyclic graph (DAG), except for recursive programs which are handled specially and will be shown later.

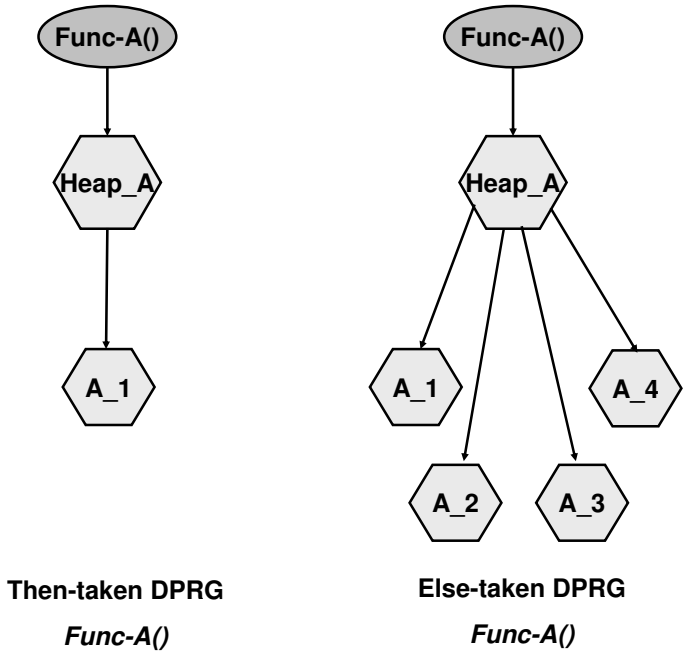


Figure 5.5: Detailed view of the DPRG for Func-A, including heap objects.

Of particular importance in the DPRG shown in figure 5.3 is the representation of a program's heap objects. The octagonal node in this figure represents a *Heap Allocation Site* in the program. This is defined as every unique segment of program assembly code existing in a compiled application which calls a library function to allocate space from the program heap. All individual heap objects allocated using the same allocation site are associated with this node in the DPRG. To better explain the way the DPRG handles heap object creation sites in a program, we present a more detailed picture in Figure 5.3 which shows the DPRG structure for *func-A* in greater detail.

Figure 5.3 depicts a detailed view of the DPRG for procedure *func-A* from

our example program, for the two alternate paths through the if-then conditional block at runtime. From this we see that func-A only performs a single call to the malloc library function to allocate 4 bytes of data and returns its address, no matter which branch is taken. This is the only location in Figure 5.3 which invokes a heap creation procedure, and is thus the only heap allocation site for the program. The left side of the figure shows the DPRG when the if condition is true, with only a single heap object allocated from this site named *A_1* . The right side of the figure shows the DPRG when the condition is not met, allocating 4 heap objects from this site, labeled *A_1* - *A_4*. From dynamic profile information, our compiler can estimate the number of objects which were allocated from this site during different visits to its program region for a program input. We use the creation and access frequency data from the program profile to create the overall heap allocation site node in the DPRG. We have also developed techniques to analyze multiple program inputs for such dynamic regions, used to reduce profile sensitivity and is discussed in 7.4.

When using the DPRG to optimize heap variables, we denote the memory assigned to each heap allocation site as being the profiled *heap bin size*. Throughout this thesis, a heap bin is simply a collection of memory locations corresponding to storage space allocated from the program heap. For allocation purposes, a heap bin is more specifically a set of 1 or more individual heap objects of the same size allocated from the same heap allocation site in a program. For example, we can apply our method to the example program assuming the program input leads to the execution of func-B in the if-then condition block. In this case, we see that the stack

array variable "Y" is used to store pointers to four heap objects, each allocated with a size of 4 bytes. We can say that the four heap objects created in this program region make up a heap bin for this heap allocation site with a total size of 16 bytes for this example. The bin is of a reasonable size with small individual heap objects accessed inside a loop, making it a likely candidate for SPM placement. Later we will see how our method determines which heap objects constitute a promising bin set for optimization purposes and how our method determines the final sizes for all heap bins in a program when accounting for dynamic SPM movement.

To uniquely identify each region in the DPRG, we mark each newly visited node during the runtime profile with a unique increasing timestamp. Internally, our compiler method actually uses a three integer index of the 3-tuple form (A,B,C) to uniquely identify each program point in the DPRG during compilation. We have adopted this form during development of our profiling tools to deal with programs using multiple C-language source files, common for realistic applications. In the index, A corresponds to the file number, B to the function number and C to the function marker number. Each is assigned as parsed during compilation of each function from each file that makes up a complete application. This allows us to properly track program regions at the assembly code level at which the program executes on an embedded processor, and correctly profile program regions for access activity.

As each program point is reached during execution, the compiler-assigned triple is associated with a runtime timestamp when the profiler first sees that region to make analysis and optimization passes simpler. We note that when a sim-

ple program is timestamped using static compiler analysis such as in our previous work [132], the relative ordering of timestamps can help illuminate relationships between different program points and access patterns for static program data. Our method incorporates dynamic program data which voids most of the existing static compiler analysis methods for reliable runtime behavior prediction. We note that not all program code regions contain executable code since compilers to not always generate useful code between marker boundaries as they occur in optimized assembly output. It should also be noted that we can have uniquely timestamped program regions between any two markers which can be reached using an edge on the DPRG, even if the markers come from different files or functions.

Recursive Functions The approach discussed so far does not directly apply to stack variables in recursive or cross-recursive procedures. With recursion the call graph is cyclic and hence the total size of stack data is unknown. For a compiler to guarantee that a variable in a recursive procedure fits in the scratchpad is difficult. The baseline technique is to collapse recursive cycles to single nodes in the DPRG, and allocate their stack data to DRAM. Edges out of the recursive cycle connect this single node to the rest of the DPRG. This provides a clean way of putting all the recursive cycles in a black box while we first describe our core method for heap variables. Section 7.2 will change the way we handle recursive functions by modifying both the DPRG representation of these functions as well as the core allocation algorithm explained in the next chapter.

Chapter 6

Compiler allocation of dynamic data

In this thesis we propose a framework that enables efficient, effective and robust data-layout optimizations for dynamic program data. Our compiler method finds a good dynamic program data layout by iteratively searching the space of possible layouts, using profile feedback to guide the search process. A naive approach to profile-guided search may transform the program to produce a candidate data layout, then recompile and rerun it for evaluation and feedback for further tuning. This leads to extremely long search cycles and is too slow and cumbersome to be practical. To avoid this tedious process, our framework instead iteratively evaluates possible dynamic layouts by simulating their program memory behavior using the DPRG. Simulation using the DPRG is also more informative than rerunning since it allows us not only to measure the resulting performance of the candidate data layouts, but also to easily identify the objects that are responsible for its poor memory performance. Having understood how program information is represented by the DPRG in the last chapter, we now move on to discussion of our allocation method which relies on DPRG information.

This chapter presents a discussion on the core steps of our allocation method for heap objects. Our compiler method operates in three main phases: (1) DPRG initialization and initial allocation computation, (2) iterative allocation refinement

and (3) code generation. Only the essential steps from our complete allocation algorithm for dynamic program data will be presented in this chapter to ease understanding of the concepts. Once our basic method for regular heap objects has been presented, the next chapter will go into further detail on the improvements developed over our core algorithm.

Section 6.1 presents an overview of our method for dynamic SPM allocation of heap objects. The preparation required for our main algorithm is described in Sections 6.2 and 6.3. Sections 6.4, 6.5, 6.6, 6.7, 6.8 and 6.9 detail the individual steps comprising the iterative portion of our allocation algorithm. Section 6.10 concludes this chapter by reviewing our methods to perform code generation of an optimized application binary. The next chapter will complete the presentation of our full method by presenting additional optimization modules developed to handle more complicated dynamic data.

6.1 Overview of our SPM allocation method for dynamic data

Figure 6.1 shows the core steps that our method takes to allocate all types of data – global, stack and heap – to scratch-pad memory. The proposed memory allocation algorithm is run in the optimizing compiler just after parsing and initial optimizations but before register allocation and code generation. This allows the compiler to insert the transfer code before the compiler code optimizations, register allocation and code generation passes occur for best efficiency. Although the main contribution of this paper is the method for heap, the figure shows how our

allocation method is able to handle global and non-recursive stack variables using findings from our previous research in [132] to give a comprehensive allocation solution. While our method uses many of the concepts presented in [132], our dynamic data allocation algorithm is completely original. The compiler implementation in this thesis was redesigned several times and evolved into a highly integrated optimization framework that properly considers all program variables for best overall placement to SPM using dynamic methods. However, our methods for handling heap data can also be applied independently with any other SPM allocation scheme for global and non-recursive stack data. For example, a developer may decide to split the total SPM space amongst heap and non-heap data, and separately apply different allocation methods for each variable type. We found that this separate handling approach tends to greatly reduce the benefit of SPM allocation by unduly limiting the flexibility of the allocation search space. The separation of optimization approaches was thus abandoned in favor of our more efficient comprehensive approach for whole program optimization to SPM presented in this thesis.

Here we overview the steps shown in figure 6.1. *Details are found later in the sections listed in the right margin for each step in the figure.* **Step 1** partitions the program into a series of regions during compilation while gathering static program information. The region boundaries are the only program points where the SPM allocation is allowed to change through compiler-inserted copying code. This step ends by adding the dynamic profile information from regions and variables to create the final DPRG. Step 2 performs an idealized variable allocation of available SPM for stack, global and dynamic data variables for all program regions using the final

Step 1 Create DPRG and prepare it for usage. /* Section 6.2 */
Step 2 Find idealized SPM allocation for global, stack and heap. /* Section 6.2 */
Step 3 Compute initial heap bin sizes. /* Section 6.3 */
Step 4 Compute consensus heap bin sizes. /* Section 6.3 */
/* Iteration loop start*/
Step 5 Allocation Feedback Optimizations(not used for first iteration). /* Section 6.8 */
Step 6 Transfer Minimization Passes. /* Section 6.5 */
Step 7 Heap Safety Transformations. /* Section 6.6 */
Step 8 Complete Memory Layout for global, stack and heap. /* Section 6.7 */
Step 9 /*Iteration loop end*/ /* Section 6.9 */
(a) If (estimated runtime of current solution < estimated runtime of best solution so far)
Update best solution so far.
(b) If (estimated runtime of current solution == estimated runtime of solution in last two iterations)
Goto Step 10. /* See same solution again ⇒ end search */
(c) If (number of iterations < THRESHOLD)
Goto Step 5. /* Otherwise end search and proceed to Step 10 */
Step 10 Code generation to implement best allocation found. /* Section 6.10 */

Figure 6.1: Algorithm for allocating global, stack and heap data to scratch-pad memory.

DPRG. Given a target SPM size, **Step 3** computes an initial bin size for each heap variable for each region based upon the relative frequency-per-byte of *all* variables accessed in that region. Variables of any kind with a higher frequency-per-byte of access are thus preferred when deciding initial heap, stack and global variable allocations. **Step 4** computes a single consensus bin size for each heap "variable" (allocation site) equal to the weighted average of the initial bin sizes for that variable across all regions which access that variable; weighted by frequency-per-byte of that variable in each region. **Step 5** begins the iterative loop and performs the feedback optimizations based on the results of the previous iteration's allocation findings (beginning with the second iteration). **Step 6** applies a set of transfer minimization passes to try and reduce the overhead from transfers in our dynamic SPM placement. **Step 7** performs a set of heap safety transformations to ensure

our allocation conforms to our heap bin allocation requirements. **Step 8** computes the memory address layout for the entire program which includes finding the fixed offset assigned to each heap bin at runtime. **Step 9** performs an iterative step on the algorithm. Step 9(a) maintains the best solution seen so far. Step 9(b) terminates the algorithm if the iterative search is making no progress. Step 9(c) is the heart of the iteration in that it repeats the entire allocation computation, but this time with feedback from the results of this iteration. After the iterative process has exited, **step 10** generates code to implement the best allocation found among all iterations.

Handling Global and Stack Data Our method as implemented is capable of allocating all possible types of program data, whether they be dynamic or static data types. The compiler infrastructure created for this thesis is able to dynamically place global, stack, heap and code objects to SPM at runtime and is not limited to just dynamic data, although the main contributions of this thesis encompass dynamic data types. While it is possible to apply only our techniques for dynamic data on top of another existing allocation scheme, for best performance our implementation actually tightly integrates handling for both types to best tradeoff SPM space for runtime gain. For all our passes, we attempt to allocate all variables with best FPB regardless of type. All passes were created to allow flexible allocation attempts using all types of variables, and taking advantage of their different properties, most notable liveness and pointer analysis results, to be able to dynamically allocate to SPM and main memory throughout program execution for most benefit. *The only differences between dynamic and static data allocation result from the different allocation behaviors, lifetime properties and access profiles seen in the DPRG for a*

program. With a firm understanding of how each object type can possible interact during program execution, it becomes much easier to develop a single complete infrastructure to perform total program allocation.

This thesis has culminated in the development of a robust framework that is able to account for these distinguishing features and handle all data types in one package. While there are no new contributions for code or global object allocation from this thesis, we do contribute a new method to allocate recursive stack data, which completes the list of allocatable stack objects. The compiler platform developed in this thesis makes use of our previous work on code, global and stack allocation in [132] for any difficulties or optimizations affecting static data types. Otherwise, this thesis only discusses the contributions for dynamic data handling except where it is necessary to mention static data handling in context. *It should be understood that the majority of our general allocation passes apply to code, global and stack variables as well, except when a pass is explicitly targeting heap data.* For example, transfer minimization is important for any type of variable, with the type simply helping to determine the flexibility with which a variable can be manipulated or allocated within the confines of the entire program.

6.2 Preparing the DPRG for allocation

Before engaging the main iterative body of our allocation methods for dynamic SPM placement, we must first make a few internal modifications of the DPRG to accommodate extra information. For each program region in the DPRG we will

also have a memory map of the SPM space available for the program in each region. This will allow us to keep track of which region variables are candidates for SPM placement for that iteration as well as available SPM space. Our allocation method prepares the DPRG for use by incorporating these memory maps and other reference structures to create a complex internal graph structure. This graph represents all DPRG nodes, including all static and dynamic variable and edge information for each node, as well as the node's SPM memory map and allocator feedback data. Once the complete allocator DPRG has been loaded, we finish processing by visiting each program region to find the region with the largest memory occupation, which in turn defines the maximum memory occupancy for the program.

After the DPRG has been loaded and prepared for use, we attempt an idealized allocation pass for the entire program in order to find a good starting point for our search. This is accomplished by first visiting each region in the DPRG and sorting its variable list by FPB for that region. Each variable with a positive FPB is processed in decreasing order and placed in SPM if the variable fits in the SPM free space. At the end of this pass, we can see an idealized view of our SPM allocation as execution progresses from region to region through the DPRG. This view represents the situation where the most frequently accessed variables (that fit) are resident in SPM for each region. Of course, this is an ideal because this placement is not concerned with the greater issues of consistency and transfer costs that plague dynamic allocation schemes. With an allocation goal in place, we proceed to the final step of our preparatory phase.

6.3 Calculating Heap Bin Allocation Sizes

Our next phase uses the program allocation derived in the last step to decide on realistic initial values for the heap bin sizes for each allocation site. The bin size assignment heuristic assigns a single bin size for each heap variable in two steps. First, each region requests an initial bin size for each heap variable considering only its own accesses (Step 2 in figure 6.1). Second, a single consensus bin size is computed for each heap variable (Step 3 in figure 6.1) as a weighted average of the initial bin sizes of regions accessing that variable. This section discuss these two steps in detail.

Before looking at the algorithm for bin size computation, let us consider two intuitions on which the algorithm is based. The first intuition is that *the bin size assignment heuristic assigns larger bins to sites having greater frequency-per-byte of access*. The reason is that the expected runtime gain from placing a heap bin in scratch-pad instead of DRAM is proportional to the expected number of accesses to the bin. Thus for a fixed amount of scratch-pad space, the gain from that space is proportional to the number of accesses to it, which in turn is proportional to the frequency-per-byte of data in that space. A second intuition is also needed: the constraint of fixed-sized bins implies that *even heap variables of lower frequency-per-byte should get a share of the scratch-pad*. This intuition counter-balances the first intuition. To see why this is needed, consider that according to the first intuition alone, a heap variable with the highest frequency-per-byte in a certain region should be given all the scratch-pad space available. This may not be wise because of the

fixed size constraint: doing so would mean a huge bin for the variable that would crowd out all other heap objects in all regions it is accessed, even those with higher frequency-per-byte in other regions. A better overall performance is likely if we 'diversify the risk' by allocating all bins some scratch-pad, even those with lower frequency-per-byte.

The initial bin size computation algorithm is shown in procedure **find_initial_bin_size**() in figure 6.2. It proceeds as follows. For every region in the program (line 1), all the variables accessed in that region are considered one by one, in decreasing order of their frequency-per-byte of access in that region (line 3). In this way, more frequently accessed variables are preferentially allocated to scratch-pad. For each variable, if it is a global or stack variable, then space is also reserved for it (line 5). This reserved space is an estimate, however – *the global or stack variable is not actually assigned to SRAM (scratch-pad) yet*; that is done after bin size assignment by Step 6 in figure 6.1. This design allows for our heap method to be de-coupled from the global-stack method, allowing the use of any global-stack method.

Returning to initial bin size assignment, if the variable v is heap variable (line 6 in figure 6.2) then an initial bin size is computed for it (lines 7-12). Only when the frequency-per-byte of the site in the region exceeds one (*i.e.,* there is reuse of the site's data in the region), is a non-zero bin size is requested (lines 8-10). Then, the bin size is computed by proportioning the available SRAM in the ratio of frequency-per-byte among all sites accessed by that region (line 8). Only sites that having $freq\_per\_byte(S_i, R) > 1$ are included in the formula's denominator. The bin size is revised to never be larger than the variable's total size in the profile data (line 9):

```
void find_initial_bin_size() {
1.  for (each region R in any order) do
2.      SRAM_remaining = MAX_SRAM_SIZE
3.      for (each variable v of any kind accessed in R sorted in decreasing frequency-per-byte(v,R) order) do
4.          if (v is a global or stack variable)
5.              SRAM_remaining = SRAM_remaining - size(v)
6.          else {                              /* v is heap variable */
7.              if (freq_per_byte(v,R) > 1)     /* if variable v is reused in R */
8.                  initial_bin_size(v,R) = SRAM_available × freq_per_byte(v,R) / Σ_{all accessed variables u_i in R} freq_per_byte(u_i,R) which are > 1
9.                  initial_bin_size(v,R) = MIN(initial_bin_size(v,R), size of v in profile data)
10.                 initial_bin_size(v,R) = next-higher-multiple-of-heap-objects-size(initial_bin_size(v,R))
11.             else                            /* no reuse in variable v in R */
12.                 initial_bin_size(v,R) = 0
13.             SRAM_remaining = SRAM_remaining - initial_bin_size(v,R)
14. return

void find_consensus_bin_size() {
15. for (each heap variable v in any order) do
16.     consensus_bin_size(v) = (Σ_{all R} initial_bin_size(v,R)×freq_per_byte(v,R)) / (Σ_{all R} freq_per_byte(v,R))
17. return
```

Figure 6.2: Bin size computation for heap variables.

this heuristic prevents small variables from being allocated too-large bins. Finally, the bin size is revised to be a multiple of the heap object size (line 10), to avoid internal fragmentation inside bins.

Finally, a single final bin size is computed as a consensus among the initial bin size assignments above, as shown in procedure **find_consensus_bin_size**() in figure 6.2. For each heap variable v, the consensus bin size (line 16) is computed as the weighted average of the initial bin size assignments for that site across all regions that access S, weighted by the frequency-per-byte of that variable in that region. Through experimentation with other methods, we found that a weighted average across all program regions generally gave us the best starting point for heuristically finding an optimal SPM allocation.

6.4 Overview of the iterative portion

With our preparatory steps completed, we can now apply the bulk of our allocation algorithm to arrive at a realistic allocation with best performance improvement. The next four sections will present the passes which constitute the iterative portion of our algorithm for heap memory allocation. Additional modifications have also been developed to handle different dynamic program objects as well as to optimize handling of specific scenarios for robust performance. Those optimizations not essential to our core approach will be presented in the next chapter to avoid unnecessary confusion while explaining our essential allocation steps.

Our heap bin consensus step allowed us to bind the heap bins for all heap allocation sites in a program, giving each a known size for use in our allocator. With this in place, we can now analyze heap objects more easily because all now have memory occupancies which can be used for determining the best distribution of limited SPM space among all program objects. To refine our allocation from an ideal and impractical starting point to a realistic and efficient solution we make use of two basic concepts in our iterative core. The first concept we keep in mind is that greedy search algorithms tend to fall into valleys, so we must incorporate methods to constantly perturb the search space to explore different optimal optimization valleys using efficient heuristics. The second important concept to keep in mind is that it is impossible to give all variables their desired SPM allocations without incurring excessive overhead in transfers, so a delicate balance must be maintained among all variables for all program regions.

6.5 Transfer Minimizations

The first iterative step shown in Figure 6.1 performs allocation modifications based on feedback from the previous iteration's results. Because this is not meaningful until the second iteration, this will be discussed at the end of this section in context and we instead begin by discussing our Transfer Minimization Passes. These passes are primarily aimed at reducing the costs incurred by the memory transfers required to implement a chosen dynamic allocation scheme. Before presenting these minimization passes, we first mention the other methods we use to reduce the overhead due to transfers.

Reducing runtime and code size of data transfer code Our method copies heap bins back and forth between SRAM and DRAM. This overhead is not unique to our approach – hardware caches also need to move data between SRAM and DRAM for the same reasons. The simplest way to copy is a **for** loop for each bin which copies a single word per iteration. We speed up this transfer in the following three ways. First, we generate assembly-level routines that implement optimized transfers suited to the block size being copied. Copying blocks of sizes larger than a few words are optimized by unrolling the **for** loop by a small constant. Second, code size increase from the larger generated copying code are almost eliminated by placing the code in a memory block copy procedure that is called for each block transfer. Third, faster copying is possible in processors with the low-cost hardware mechanisms of Direct Memory Access (DMA) (*e.g.,* [28, 45]) or pseudo-DMA(*e.g.,* [102]). DMA accelerates data transfers between memories and/or I/O devices. Pseudo-DMA accelerates

transfers from memory to CPU registers, and thus can be used to speed memory-to-memory copies via registers. Despite the reduced cost for transfers gained from these methods, our allocation scheme strives to keep these as low as possible to achieve the best performance available.

Reducing the number of transfers made at runtime The first compiler pass we apply to lower the cost of transfers is one which does so by attempting to reduce the number of transfers required at runtime. It is based on the set of passes originally developed for our previous work on dynamic stack and global allocation 5 which have been modified to account for dynamic program data as well. These include all passes which perform special analysis on procedure and conditional join nodes as well as loop node optimizations. When we have excessive transfers in regions inside of loops for example, poor dynamic placement drives up the transfer costs and voids any benefits from SPM placement. For all three types of nodes, we attempt to optimize for the most frequent case to reduce transfer overhead while placing useful objects of all types in available SPM. These nodes correspond to the program regions most likely to be problematic when trying to find a good solution across an entire program. Another optimization in this pass is inspired from our research on static allocation and attempts to find the best static solution for problematic program regions. When our dynamic allocation solution for regions has an SPM locality benefit that is about equal to its transfer costs, we generally choose the static allocation to improve the chances that dynamic allocations will benefit other regions. Another optimization ensures that unnecessary transfers are removed for regions where a variable is considered dead by compiler analysis.

Lazy leave-in optimization The second of our transfer minimization passes is our Lazy Leave-In optimization. This optimization is applied per variable and looks at the variable access frequency and SPM placements for all regions in the candidate DPRG allocation. Our default behavior is to copy a bin out to memory in regions where it is not accessed. Instead in some cases, it may be profitable to leave a bin in scratch-pad even in regions where it is not accessed, if the cost of copying the bin out to DRAM exceeds the benefit of using that scratch-pad space for other variables. With this optimization there are no heap safety concerns since there is no correctness constraint for regions which do *not* access a variable.

During refinement, we know what other variables were assigned to the space evicted by a bin. If the profile-estimated net gain in latency from these other variables is less than the estimated cost of the transfer, then the compiler lazily leaves the bin in scratch-pad and does not bring the other variables in. The implementation of this optimization relies on an estimated gain vs. estimated cost comparison as applied to contiguous groups of regions, but this time the groups are of those regions which do *not* access a bin. This optimization expands the peephole optimizations for particular region nodes to minimize transfers for candidate allocations for improved SPM benefit.

6.6 Heap Safety Transformations

At the end of our iterative pass for refining the current candidate allocation, we must perform some validation steps before solving for a valid memory address

layout and estimating the total program runtime and energy costs. An essential requirement for correctness in our method is that we must not allow the possibility of incorrect program pointers as a result of our dynamic allocations. At the end of our iterative search, our algorithm must ensure that the bin offset and size never changes in SPM for all regions in the program where heap objects from that site may be accessed. While stack and global variables also suffer from the same problem, pointers to those data types are rare and generally easier to analyze. Because heap data is entirely accessed through pointers, and the C language does not enforce strict pointer data typing, this becomes a serious concern for program safety when performing runtime data movement.

Our methods make use of the full range of static and dynamic analysis available to modern compiler developers. We create a modified version of the DPRG that also contains full alias and pointer analysis information provided by our compiler analysis passes. We maintain constant bin size across regions for each heap allocation site throughout our algorithm as a requirement for operation. Our layout pass verifies that pointer safety is maintained for all regions where optimized heap bins may be accessed. We use the information obtained from pointer analysis across all program regions where a variable is live, and with this information are able to mark regions with a variable's access guarantees. Our method is able to determine if a variable will, will not, or may be accessed in a particular region, which in turn dictates the flexibility we can have with required transfers of heap data across program regions.

The first part of this transformation pass uses the DPRG with access guarantees to evaluate all heap allocation sites which have been optimized in at least one

region, in decreasing order of their total FPB. Each bin is examined for the entirety of its lifetime and its placement benefits and transfer costs are evaluated. We analyze its current candidate allocation and modify the allocation DPRG so that the bin is placed in SPM in all regions where we can not guarantee it is not accessed. We note that like a few of our other iterative transformations, we allow about 20% extra SPM space for overfitting in passes which attempt to bring in more variables to SPM to lower costs. We do this to allow a wider but reasonable search space for better allocations, and rely on our final validation and layout passes to trim the candidate allocation to fit into available SPM.

At the end of this heap transformation pass, all of the heap bins optimized for this program will have guarantees for safe program behavior at runtime. As a final note, even the best current pointer analysis methods often are unable to make certain guarantees on the majority of heap variables in a program, which tends to make our allocations conservative in nature. Improvements in the predictability of memory accesses for pointers in the C language pointer will only improve the performance of our approach.

Indirection optimization Our second heap transformation pass is applied directly after our first safety transformation. We noticed that an opportunity for improving our algorithm can arise because of the following undesirable situation. In our strategy it is possible that in a certain region, the cost of copying a bin into scratch-pad before that region can exceed the gain in latency of accesses in the region. Leaving the bin in DRAM in such a case would improve runtime, but unfortunately *this violates correctness because of the fixed-offset requirement of our*

method. To see why correctness is violated consider that a heap bin must be in the same memory bank in *all* regions that access it, as otherwise, if the bin remained in a different memory bank in a some of those regions, pointers into objects in the bin might be incorrect. Thus the optimization of leaving the bin in DRAM cannot be applied.

Fortunately, there is a way to make this optimization legal. It is legal to leave the bin in DRAM when it is not profitable to copy it into scratch-pad at the start of a region, provided, in addition *all accesses to the bin in the region are translated at runtime to convert their addresses from scratch-pad to DRAM addresses*. Doing so will ensure that all pointers to the bin – which are really invalid since they point incorrectly to scratch-pad – will become valid after translation. Scratch-pad addresses can be translated at runtime to DRAM addresses by inserting code before every memory reference that adds to each address the difference between the starting memory offset of the bin in DRAM and SRAM. In this way, a level of indirection is introduced in addressing, and hence the name of this optimization. We present a method for doing so in our discussion of our previous methods for stack and global data 5.4.

One may wonder why the indirection optimization is used as an optimization, and not the default scheme. Recall than the default scheme ensures that a bin is allocated at the same offset in SRAM whenever it is accessed, and uses indirection only as an exception. The reason that indirection is not used as the default is that indirection has a cost – extra code must be inserted before every access to check if its address is in SRAM and if so, to add a constant to translate it to a DRAM

address. This extra code consumes run-time and energy. It is therefore profitable to apply indirection only if the cost of the transfer exceeds the overhead of indirection. For regions where a bin is frequently used, the opposite is true – the overhead, which increases with frequency of use, will increase and often far exceed the cost of transfer; so indirection is not profitable. Since regions where bins are accessed frequently make up most of the run-time the default behavior should match their requirements; indirection is used sparingly for other regions where its overhead is justified.

The indirection optimization (step 4 in figure 6.1) is applied as follows. For every heap variable v in the program in any order, the compiler looks at all groups of contiguous regions in which v is accessed. For each such group of regions, it estimates whether the cost of copying the bin into scratch-pad at the start of the group (a known function of the size of the block to be copied) is justified by the profile-estimated gain in access latency of accesses to v in the group. If the estimated cost exceeds the estimated gain, then the transfer of the bin to scratch-pad is deleted, and instead all references to v in the group are address-translated as described in the previous paragraph. The address translations themselves add some cost, which is included as an increase in the estimated cost above. A consequence of the indirection optimization is that scratch-pad space for a some bins is freed in address-translated regions.

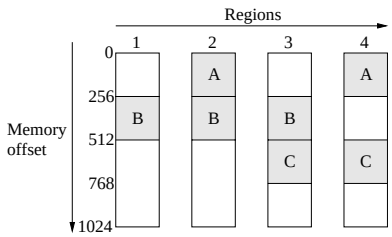
6.7 Memory Layout Technique for Address Assignment

At the end of our iterative allocation steps, the next step in our method is to compute the layout of all variables in scratch-pad memory (Steps 5-7 in figure 6.1). The layout refers to the offsets of variables in memory. Computing the layout is done in three steps. First, the layout of heap variables is computed (Step 5). Second, the placement of global and stack variables is computed (*i.e.,* which global and stack variables to keep in scratch-pad is decided) (Step 6). The placement takes into account the amount of space remaining in scratch-pad after heap layout. Third, the layout for global and stack variables is computed by allocating such variables in the spaces remaining in scratch-pad after heap layout. *This section only discusses how the heap layout is computed.* The placement and layout of global and stack variables is independent of this paper, and any dynamic method for global and stack variables can be used, such as [132].

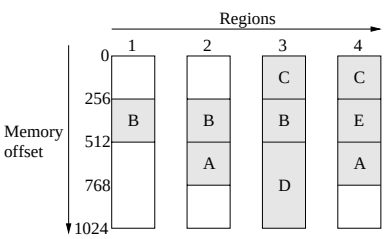
Before we compute the heap layout, it is instructive to consider why heap layout is done before the placement and layout of global and stack variables. The reason the heap layout is done first is that *the layout of heap bins is more constrained than that of global and stack variables.* In particular, heap bins must always be laid out at the same offset in every region they are accessed. Thus, allowing the heap layout to have full access to the whole scratch-pad memory increases the chance of finding a layout with the largest possible number of heap variables in scratch-pad. The less constrained global and stack variables, which typically can be placed in any offset [132], can be placed in whatever spaces that remain after heap layout.

Site	Bin size (bytes)	Regions accessed
A	256	2,4
B	256	1,2,3
C	256	3,4
D	512	3
E	256	4

(a)



(b)



(c)

Figure 6.3: Example of heap allocation using our method showing (a) heap Allocation sites; (b) greedy compiler algorithm for layout – D does not fit; and (c) layout after backtracking – D fits.

Placing heap data first does ***not*** *mean, however, that heap variables are preferentially allocated in scratch-pad.* Global, stack and heap variables were given an equal chance to fit in scratch pad in the initial bin size computation phase. At that point, we had already reserved space for global and stack variables of high frequency-per-byte.

The layout of heap bins is computed as follows. Finding the layout involves computing the *single fixed offset* for each bin for all regions in which the bin's site is accessed. Further, different bins must not conflict in any region by being allocated to the same memory. We use a greedy heuristic that allocates bins to scratch pad in decreasing order of their overall frequency per byte of access, so the most important bins are given preference. Each bin is placed into the first block in memory that is free for all the regions accessing the bin's site. Figure 6.3(b) shows the result of the greedy heuristic on sites A-C listed in figure 6.3(a). This heuristic can, however, fail to find a free block for a bin. Figure 6.3(b) shows this situation – the bin for D

cannot be placed since no contiguous block of size 512 is available in region 3.

To increase the number of bins allocated to scratch-pad, we selectively use a back-tracking heuristic whenever the greedy approach fails to place a bin. Figure 6.3(b) shows how the greedy heuristic fails to place bin D. Figure 6.3(c) shows how D can be placed if the offset choices for A and C are revisited and changed as shown. To find the solution in figure 6.3(c), our back-tracking heuristic *tries to place such a bin by moving a small set of bins placed earlier to different offsets.* This heuristic is written as a recursive algorithm as follows. To try to place a bin that does not fit, it finds the offset in scratch-pad at which the fewest number of other bins, called *conflicting bins*, are assigned. Then it recursively tries to move all the conflicting bins to non-conflicting locations. If successful, the original bin is placed in the space cleared by the moved conflicting bins. The recursive procedure is bounded to three levels to ensure a reasonable compile-time. Four levels increased the compile-time significantly with little additional benefit. An example of this method is when block D cannot be placed in figure 6.3(b). The offset with the minimum number of conflicts (2) is 512, and the conflicting block set is C. Thus block D is placed at offset 512 by moving C, which in turn recursively moves block A. The conflict-free assignment in figure 6.3(c) results. Of course, even this recursive search may fail to place a bin – in this case the corresponding heap allocation site places all its data in DRAM.

When we are unable to find a suitable chunk of SPM to assign a heap bin for all accessed program regions, we apply one final method. This approach attempts to split the heap bin into smaller chunks at the granularity of an individual heap

object slot. This heuristic gathers the free chunk list across all DPRG regions where the variable is live. If splitting of the heap bin among the free chunks is possible, we attempt that allocation and distribute the heap bin among the memory blocks. If no free space is available, we apply a two-level swapping heuristic that attempts to free up any memory blocks of the appropriate size among regions with the most SPM contention. We observed that a combination of swapping and splitting is usually able to place most of the profitable heap bins into SPM which would not otherwise have fit.

Although we usually layout heap variables first, for some instances it becomes advantageous to modify this somewhat. For our algorithm, we have modified this step so that it alternates the search order each time the layout algorithm is applied. All searches are always ordered by a variable's overall FPB. For many situations, placement of heap objects first and them stack and global objects is generally a good idea, since it allows the harder to place heap objects more freedom for placement. Unfortunately, sometimes this can also cause poor initial allocations which must be refined over multiple iterations. This can happen when less frequent heap objects crowd out much more frequent stack and global data. We attempt to avoid this overspecification by having the second iteration instead search for variable layouts purely by FPB, regardless of object type. By alternating the search space, we help refine our iterative search for the lowest cost allocation.

At the end of the layout pass, we now have a realistically implementable dynamic SPM allocation for an entire program. Although some of our transformations allowed overallocation of available SPM for some DPRG regions, our layout pass

ensures the chosen allocation reflects a realistic implementation for the target SPM and trims out excess variables. It is important to note that throughout the layout pass, we keep track of those variables which succeeded and failed different portions of the allocation process. We make use of all such feedback information from our previous allocation iteration to guide the initial transformation for our next iteration. The following section will discuss our feedback-driven allocation refinements.

6.8 Feedback Driven Transformations

On the second iteration of our algorithm, there is now feedback information from the last allocation pass that guides us in how to modify variable placements for the next iteration. Using this information, we apply the following two refinement passes.

Heap Bin Resizing This pass attempts to improve the relative distribution of heap bin sizes using feedback from the previous iteration. In this pass, we begin by obtaining a list of heap variables which were considered for allocation in the previous iteration. This list is processed in decreasing order of overall FPB. For each heap variable, we attempt to refine its assigned bin size. If a variable was placed successfully in SPM during the last iteration, we attempt to grow the size of the bin. If a heap variable failed allocation, then we attempt to decrease its bin size. This increase or decrease of each individual bin will change the relative distribution of SPM space among all heap objects. Finally, before actually resizing each slot, we examine the DPRG to ensure that their is enough SPM space in each affected

region where the heap variable can possibly be accessed. For this step we allow a 20% overhead for SPM storage at each region to allow for allocation exploration and will be rectified in the final memory layout step.

For each allocation site that has a known size, we increment or decrement a heap bin using the *heap slot size*. We see that here, a heap bin is made up of an integer number of these heap allocation slot units. As each heap variable is processed, those which fail to be allocated over successive iterations will have their attempted SPM allocations reduced to zero. Similarly, successfully placed variables are allowed to instead grow to their maximum profiled size as an upper limit. We have also added a further optimization for programs which use large numbers of small heap objects. For these programs, our bins are resized using exponentially grown slot increments. Since our algorithm is designed for fast results, having a low iteration count tended to constrain our heap bin resizing pass when relying on a strictly linear approach to slot size modifications.

Variable Swapping This optimization is applied primarily to global and stack variables, although in the next chapter we see that it is also applied to code variables as well. This is a last-chance optimization that is applied to those variables with a non-zero FPB that have failed allocation the past three consecutive iterations. As the name implies, we apply a heuristic algorithm that attempts to swap a combination of variables having a lower FPB than those unsuccessful variables with a high FPB. For this we go through all regions where the variable is accessed in DRAM, and build a list of SPM variables that we may swap out for each region to make room for the DRAM variable. We also keep a tally of the total benefits and transfer

costs for each region for all variables affected by these possible swaps. At the end we attempt to choose the most common set of variables which satisfied the allocation for each region with a reasonable cost-benefit value. Because this is a rather lengthy process, we only apply this when a variable has failed three consecutive allocation attempts, after which we wait another three iterations before re-attempting variable swapping.

6.9 Termination of Iterative steps

From looking at the iterative steps, we can see that our transformation passes serve to perturb the search space to evaluate different possible allocations, while our iterative passes serve to minimize transfers and find the best possible dynamic SPM placement. This serves to guide our iterative search toward better rather than less efficient solutions, while ensuring that our approach does not fall into repetitive allocation attempts without forward progress. Once the iterative portion is complete and we have a candidate layout, the algorithm reaches the end of its iterative loop and can take several paths from there.

Although the scratch-pad allocation algorithm is essentially complete after layout assignment, there is an opportunity to do better by iteratively running the entire algorithm again and again, each time taking into account feedback from the previous iteration. This iterative process is depicted in step 9 of figure 6.1. Step 9(a) maintains the best allocation solution seen amongst all iterations seen so far. Step 9(b) terminates the algorithm if the iterative search is making no progress.

Step 9(c) jumps back to step 2 to repeat the entire allocation computation, but this time with feedback from the results of the previous iteration.

The iterative process described above is a heuristic, and is not guaranteed to improve runtime at each iteration. Indeed, the runtime cost may even increase, and after some number of iterations it often does, so for this reason, rather than use the results of the last iteration as the final allocation, the solution with the best estimated runtime among all iterations is maintained. It is fairly straightforward to estimate the runtime cost of a solution at compile-time by counting how many references are converted from accessing DRAM to scratch-pad by a given allocation for the profile data.

A desirable feature of our iterative approach is that it is *tunable* to any given amount of desired compile-time by modifying the threshold for exiting the iterations in step 9(c). In practice, however, we found that we find close to the best solution in a small number of iterations; usually less than three to five iterations. Thereafter it may get worse, but that is no problem as the best solution seen so far is stored, and worse solutions are discarded. After exiting the iterative process (step 10 of figure 6.1), the best solution seen so far is implemented.

6.10 Code generation for optimized binaries

Once the best candidate allocation layout has been selected by our algorithm, it must be implemented through changes to the program code before proceeding with the rest of the compilation process to create an executable. For code, stack and

global variables, we use the same method as used in our previous method, discussed in 5.5. This consists of declaring an SPM version for each optimized variable in the program and then using this version in all places where the variable has been allocated to SPM. Of course, we also insert transfer code at all appropriate regions as well as memory address initialization code for all affected variables.

Heap memory is managed in C through library functions, and hence must be handled differently than code, global or stack data to implement a dynamic SPM allocation. Luckily, our constant-size bin and same-offset assignment constraints make this portion relatively straightforward and has three main aspects. First, the memory transfers are inserted where-ever the method has decided. Second, addressing of heap variables does not need modification since they are addressed (usually through pointers) in the same way. Third, calls to malloc() are replaced by calls to a new wrapper function around malloc(). This wrapper first searches for space in fast memory for that bin using a new free-list for fast memory. This free-list contains a list of the memory locations our allocator has assigned this bin for SPM, generally contiguous. An argument is passed to the wrapper malloc specifying which site is calling it. In this way the wrapper becomes aware of which site is calling it, so that it can look in the bin free list for that site. If space cannot be found in that sites bin free list, then malloc is called on the original unified free-list in slow memory. The code for malloc() is the same for fast and slow memory, but works on different free lists. Similar modifications are made for other heap memory allocation routines such as calloc() and realloc(). Free() functions are also modified to release the block to the either the sites bin free list or the unified DRAM free list, depending on the

address of the object to be freed.

After all the heap library functions in the program have been modified to match the chosen optimized layout, the appropriate transfer code is inserted at all region boundaries for heap bins. As mentioned previously, these consist of calls to optimized transfer functions tailored to the particular platform capabilities. The best performance is observed with those processors able to take advantage of DMA transfers to reduce latency and power costs. We have implemented transfer functions which make use of software, pseudo-DMA and DMA methods, all of which will be discussed in Chapter 8.

Chapter 7

Robust dynamic data handling

After exploring the best ways to dynamically allocate heap data to SPM, we proceeded to create other methods to handle those dynamic program objects our base scheme does not handle. These methods enable allocation of other previously unhandled program objects and complete the spectrum of possible compiler decided program allocations. Our first section will give a brief overview of the optimizations employed throughout this research that are directly relevant to our method's performance. The second section discusses our method for handling of heap allocation sites which create objects of an unknown size at compile-time. Our third method presents our algorithm for dynamic allocation of recursive stack functions to SPM. Finally, the last section is dedicated to exploring our method for profile sensitivity and how we minimize its effects for our method.

7.1 General Optimizations

While our primary research focused on finding the best methods for handling dynamic program data, in the process we also applied other general optimizations to the problem of minimizing runtime and energy consumption for embedded applications. Because our method is compiler based, we primarily focused on improving our compiler platform as much as possible. To ensure that our methods would be

applicable to the widest range of embedded products, we also strove to implement the best simulation platform possible for evaluation.

Our original research on heap allocation to SPM was conducted based on the Motorola MCore embedded processor. At the time, we used the best tools available for which we were able to obtain source code for modification, which were GCC 2.95 and GDB 5.1. The simulator included with GDB was contributed by Motorola for the Mcore, and was a very basic functional simulator for that CPU, which we augmented with memory organizations and power models to emulate a complete platform. Unfortunately, that version of GCC was also the first to offer a back-end supporting the MCore. This back-end was a bare-bones implementation, including little optimization and generally poor quality code generation. Also, Newlib (the only system library distribution available for the Mcore) contains a minimal C library implementation for the Mcore and only the most essential system stubs were implemented, preventing many complex applications from being compiled or simulated. The general lack of consumer and manufacturer support for this processor is the main reason the MCore is now considered to be on the lower end of the embedded processor spectrum, albeit still popular for simpler deployments. To summarize, the poor development platform available for the MCore limited our research due to a severe lack of compilable and executable benchmarks, poor code generation and crippled compiler optimizations(leading to redundant use of excessive stack and global objects which should have been optimized out).

Because we aim to optimize the allocation of dynamic program objects, it is vital that we have a development and simulation infrastructure which allows ex-

ecution of applications using dynamic data. It became apparent after thorough analysis of our preliminary results in [46] that our previous platform was severely limited in many ways. We thereafter dedicated some time to finding an embedded platform more representative of the current embedded market on which to evaluate our allocation methods, as well as one that enjoyed better development support for academic research. After a thorough exploration of academic publications, compilers, embedded processors and internet websites, we came to the conclusion that the ARM processor family offered the best resources to carry out state of the art research into compiler-driven memory allocation. ARM processors enjoy continual support from their manufacturer, have a large open-source following in the development world, and have cores which are designed to be backwards compatible for greater longevity in the marketplace and academia.

A large portion of academic publications concerning memory optimizations for embedded systems chose a processor from the ARM product family to evaluate their methods. Most researchers either used a complete silicon compiler and silicon simulation framework or used a software compiler and software emulation platform. Full-blown silicon development packages are all proprietary and were not a viable option for this research. They are commonly used to evaluate hardware modifications to existing ARM processor designs instead of for use with purely software-based optimizations. Among the software-based approaches, a large portion made use of the Simplescalar simulator ported for the ARM architecture.

We first attempted to use the SimpleScalar-ARM simulation platform due to its integrated energy models for simulated execution of ARM applications. Sim-

plescalar implements a more realistic processor simulation in that it uses the ARM binary description interface to decode and execute binary files. This is what many processors running an OS on an ARM cpu use, and allows the use of optimized C libraries tailored to embedded platforms, such as ucLibC. This makes a good pairing for industrial-strength compilers from vendors aimed at software engineers creating real-world products. In the open-source world however, support for this format in GCC was problematic. We were able to compile many applications using ucLibC with GCC but were unable to execute them on the SimpleScalar simulator properly, due to incompatibilities with system hooks and low-level calls. Instead, we found that a better solution was obtained when we paired GCC with GDB and the associated open-source C library, Newlib. After much searching and evaluation, we were able to obtain the highest levels of compiler performance, broadest range of compilable and executable benchmarks as well as the best simulation environment to evaluate our method using only freely available tools.

The ARM family enjoys broad support in both its simulation and the range of compiler optimizations available to take advantage of its hardware design. The highly active compiler community support for this architecture was one of the deciding factors for redeveloping our development framework for the MCore. This was most apparent in the continual contribution of high quality compiler optimizations which began with the inception of Code Sourcery to the GCC project in 2000. Since then, there have been significant improvements in GCC such as the transition to an SSA intermediate form, addition of ARM hardware information to the GCC compiler and incorporation of cutting edge compiler optimizations tailored to take best

advantage of the ARM architecture. Code Sourcery and other contributors have improved the performance of the GCC compiler to levels which rival industrial-strength compilers, something unheard of in the embedded world for open-source compilers.

The many improvements in GCC in general as well as in its ARM back-end has allowed us to pursue more aggressive compiler optimizations, for tighter code and smaller memory footprints for executables. Producing more efficient code in both execution latency and memory occupation is of primary importance for any embedded system, and whenever possible orthogonal methods should be applied. Our switch from the MCore to the ARM showed that our method was even more beneficial for scenarios where a modern industrial-strength compiler has optimized static program data as much as it can. Because compilers can only currently analyze static program data for optimal placement, even the best methods can only reduce the costs due to accessing static program data at compile-time. When static program data has been optimized as much as possible through existing methods, this only serves to raise the importance of any remaining, unoptimized dynamic program data, which will then consume a correspondingly larger ratio of the applications runtime and energy usage. Any further optimizations targeting static program objects will yield diminishing returns for an increased level of effort at compile-time, prompting the shift to handling dynamic program data efficiently.

Like any industrial-strength compiler, GCC is a very complicated software package and required a large time investment to yield results. The C compiler alone consists of hundreds of thousands of lines of code, with tens of thousands dedicated to the ARM architecture alone. To take full advantage of its different strengths,

we have tightly integrated our methods throughout the compilation process, detailed in our chapter on methodology. Worth noting is that with the increased complexity of the compiler analysis and optimization passes, we were able to tailor our methods for application at the highest levels of available compiler optimization. The most complex compiler optimizations deal with program transformation on a large scale, something which is problematic for memory analysis and optimizations of code. Understanding of these advanced compiler analysis and transformation passes were instrumental in the development of both our base approach and of our other optimizations presented in this chapter.

7.2 Recursive function stack handling

While expanding our set of benchmarks, we discovered that a great deal of programs that use heap memory also tended to employ recursive functions in their core processing. This was problematic since there are no existing methods to handle recursive functions for memory allocation purposes and is rarely even mentioned in published SPM research. Our previous research into static program data [132] also recognized the problem that recursive functions present for existing static program data methods. Just like heap data, recursive function data has also always been left in main memory by previous SPM allocation schemes. By applying concepts developed in our heap memory methods to recursive function data, we devised the first method able to analyze and optimize recursive functions to SPM as part of our complete optimization framework.

In order to handle recursive functions and optimize their stack data we treat these functions similarly to our method for heap variables. Recursive functions can contain zero or more bytes of stack allocated storage upon entry into the function at runtime used for register-spilled storage. Recursive functions contain self-referencing calls, and so can continue to call itself at runtime, invoke more instances of itself and continue growing the allocated stack space indefinitely. This behavior places them in the category of dynamically allocated program data, just like heap memory, because neither can usually be analyzed and bounded at compile-time. The function stack frame (allocated each time the procedure is called) is fixed at compile-time for recursive procedures, granting each invocation instance the same fixed-size property that we take advantage of for known-size heap allocation sites.

Handling recursive functions required some changes to the way we construct and process the DPRG for a compiled program. Profiling these functions correctly required careful code generation changes for region marker insertions used to track individual function invocations at runtime. Our previous SPM allocation method collapsed recursive function DPRG instance nodes into a single node which was then left in DRAM. Our current method also collapses all possible nodes resulting from varying runtime invocation depths into a single recursive variable node corresponding to the entire recursive region, with information on the individual instances invoked at runtime. For functions inside recursive regions, we do not allow individual SPM placement of stack variables, and instead treat them in terms of their entire allocated function stack frame. This allows us to maintain the same fixed-size slot semantics that we use for heap data allocation in our iterative algorithm, treating

each invocation of a recursive function stack frame the same way we would treat the allocation of a heap object at a heap allocation site. Once our recursive region nodes and variable instance information has been added, we treat recursive variable nodes just like heap variables for optimization purposes, with the same restrictions on placement in regions where that variable is accessed or could be accessed through a pointer.

Figure 7.2 shows a sample DPRG representation of a recursive function. In this figure we can see that the function is represented by a single node, and contains a function stack variable for each invocation captured by the runtime profile. Like heap objects, we use runtime behavior to number each created program object according to its relative time of allocation. Thus, the original invocation to recursive function FuncRec would be labeled FuncRec_1, the next recursive invocation would be FuncRec_2, and so on while the recursive stack depth increases through self-calls.

There is one significant difference between individual heap objects and recursive stack objects, and that lies in their de-allocation order. Individual heap objects are allocated in some runtime order, but can be de-allocated at any time throughout the program, in any order desired. Recursive function stack frames must be de-allocated in reverse order of their creation, so each stack frame is de-allocated once that invocation has exited and returned to its parent function. We take advantage of this restriction to make better guarantees on predicting the accesses to different instances of recursive variables and correlation of different depths to access frequency of those variables.

With a way to analyze and optimize recursive function variables in place, we

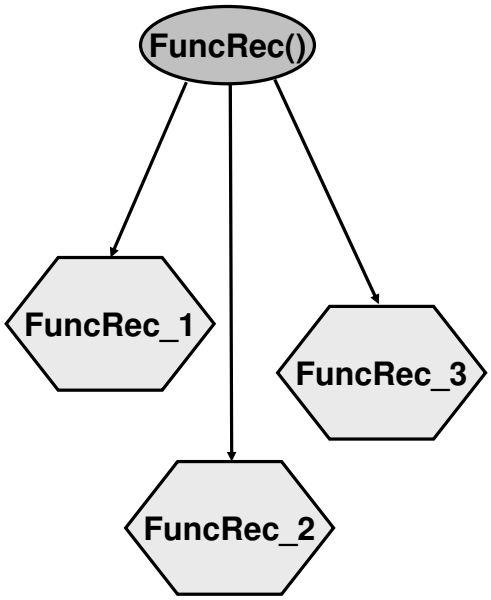


Figure 7.1: DPRG representation of a recursive function FuncRec() with three invocations shown.

can apply our SPM allocation method to the augmented DPRG for affected programs. *Having fixed-size stack frames as its basic allocation unit (or slot), we treat recursive functions like heap variables in deciding relative SPM allocations.* Code generation for SPM-allocated recursive functions is also more similar to our method for heap variables than our method for non-recursive stack and global program variables. Instead of assigning this stack function a global symbol with two copies for its possible locations in SPM and DRAM, each optimized invocation of the function must still reserve enough space to store a stack pointer address for use when swapping the current function in and out of SPM. We also create a small global variable which contains the current call depth for each recursive function. We insert a few lines of code at the beginning and end of each recursive function cycle to increment the counter upon entry and decrement the counter upon exit, as well as to check

when a stack pointer update must be performed.

For example, let us assume our allocator gives a recursive function three slots of memory in SPM, with each slot corresponding to the stack frame for a single recursive cycle invocation. Part of the call-depth counter code inserted into each optimized region performs a check on that counter after the region has been entered and the counter incremented to the current call-depth. For the first three invocations of the recursive function, the stack pointer will be updated to the address in SPM assigned for that stack frame by our allocator. Afterward, the original program stack pointer to main memory in DRAM will instead be used for placement of the remaining, unoptimized stack frames. This allows to selectively allocate certain recursive data to SPM while not having to optimize the entire possible depth at runtime. If more than one function were to be involved, each would use the same counter to determine each function's allocation to SPM as part of that overall cycle.

Figure 7.2 shows an example of a binary tree, a popular data structure used in many applications that is often traversed using recursive functions. By looking at the relative accesses to the stack variables as the recursive functions traverse this tree, many applications generally fall into uneven frequency distributions across the different recursive function invocations. In Figure 7.2, the data structure shows the majority of memory accesses occurring near the root of the tree, and the nodes tend to receive a decreasing ratio of accesses as they grow further away from the root. Other applications with different recursive data structures may exhibit the opposite behavior, where the majority of memory operations occur at the leaves of a structure in the deepest invocation depths of recursive functions. The majority

of the recursive benchmarks in our suite that made use of such directed growth data structures did so with recursive functions that were weighted according to their depth. For functions which exhibited a strong correlation between call depth and stack accesses, we were able to take advantage of the linear allocation and de-allocation property for recursive function and their stack-allocated data. By selecting which counter depths will trigger the assignment of an SPM stack address upon, we can target our SPM placement to those stack frames at depths which were profiled to be the most profitable at runtime. While trivial for top-heavy data structure accesses, this becomes very useful for optimizing bottom-heavy structures and placing frequent stack data in SPM, even at the bottom of long recursive call depths.

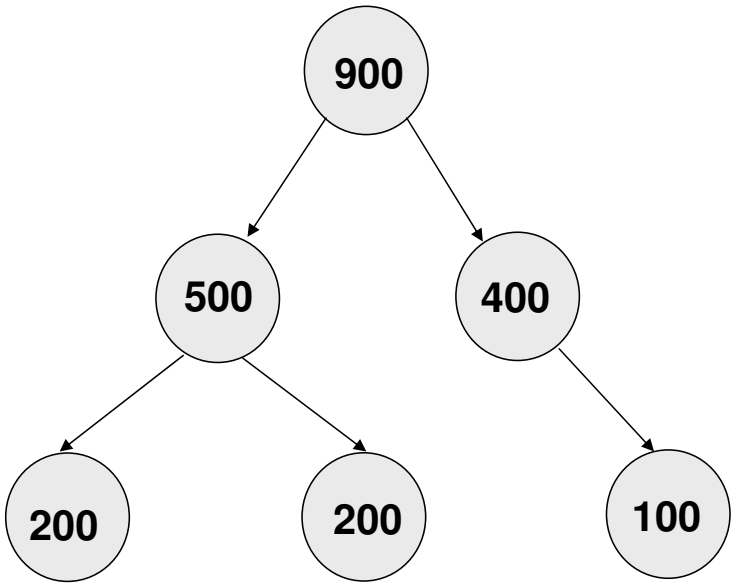


Figure 7.2: Binary Tree with each node marked by its access frequency for use by allocation analysis.

Recursive function nodes in our DPRG represent an entire recursive cycle, even if that cycle is made up of more than one function. More complex programs may employ a series of functions that make up a single recursive cycle when represented as an unmodified DPRG. During the creation of our final DPRG, we collapse all complete recursive cycles detected into single DPRG nodes, even if they span several functions. These nodes encompass the entire amount of stack memory allocated by a single invocation of an entire cycle. Recursive cycles which span more than a single function are rare due to their design difficulty, and our own large benchmark set only exhibits a single program with multi-function recursive program regions, that being BC, a popular program for recursion-based mathematical expression evaluation. However, to support the full range of possible programs , we developed and implemented support for multi-function recursive regions as well as single-function regions.

Once we were able to analyze and optimize programs with recursive objects, we found that many programs which made heavy use of recursive functions also used these functions in conjunction with heap allocated dynamic data structures. Moreover, these truly dynamic functions tended to allocate a larger proportion of unknown-size heap variables than non-recursive applications. This prompted further development of our methods to handle unknown-size heap objects for dynamic SPM placement.

7.3 Compile-time Unknown-Size Heap Objects

When dealing with heap allocation sites, there are situations where the requested size at runtime is not fixed for that site throughout the program. Programs with such unknown-size heap variables present problems for our base method, because we no longer have the niceties given by our treatment of heap variable instances in terms of heap bins made up of known-size heap allocation slots. Fortunately, for many simpler programs this does not impact allocation significantly because these sites tend to allocate objects with a low FPB, making them poor choices for placement. For more complicated programs, such as those employing recursive algorithms and recursive dynamic heap structures, unknown-size sites become more common and more important with respect to their FPB. To overcome this problem, we first developed an optimization which attempts to transform certain heap allocation sites to fix their allocation size at compile-time. We then incorporated handling of the remaining unknown-size sites into our method for allocation to SPM and have collected preliminary results of its performance. These types of program objects are fundamentally the most difficult for a compiler to analyze and predict, presenting new challenges for researchers in this field.

Heap data in complicated applications is generally dependent on an application's input data and results in an unknown-size heap allocation site for several common program scenarios. We will briefly discuss the different scenarios where we observed unknown-size allocation sites from the many benchmark suites we sampled to help the reader understand when and why these types of allocation sites are used

by programmers.

Typical uses for unknown-size allocation sites The first and perhaps most common type of site with unknown-size at compile time is for those which allocate a few small chunks of memory for temporary program use at runtime. This is common in many of the benchmarks we compiled for variables such as temporary file names, temporary node names, id tags, temporary swap variables and other variables which are created and destroyed without being accessed more than a few times for references at different program regions. Because these types of variables tend to be infrequently accessed and of unknown size, they are generally low priority candidates for SPM placement for any memory allocation scheme and so can be safely ignored for special allocation effort.

The second type of unknown-size site resides in programs that contain internal memory management routines for runtime application use. This kind of site is very common in complex desktop programs such as JPEG, M88kSim and GCC and usually are the only program locations where a call to one of the C heap creation functions is found. These sites tend to allocate extremely large amounts of memory for program use (once at initialization and then when more system heap memory is required), with the application designed to allocate from that chunk using internal heap management routines. These kinds of applications show a large investment in programmer development to optimize memory usage internally and are not directly compatible with fine-grain methods such as ours. Also, many of these applications were far too complex and library dependent to reasonably be considered embedded software, and would more likely benefit from cache optimizations geared for their

target workstation platforms. Our benchmark suite only contains programs with basic memory management, minimal library requirements, reasonable memory occupancies(less than 8 MB max and typically less than 300kb), no user interaction, and no allocation sites of this second type.

The third common usage for unknown-size sites is for the allocation of input data for program lifetime, usually in the form of a large array which constitutes more than 15% of the total program memory occupancy. This is common for any application which performs data processing on text or media data, such as dictionaries, images, movies and numeric arrays. We have included many benchmarks which allocate compile-time known and unknown-size arrays for program processing, such as GSM, Susan, Mpeg2Decode, Mpeg2Encode, Imp, Bzip, Spiff, and Heapsort. Most of these programs allocate large the large arrays and although they are frequently accessed, their large size with respect to other program variables makes them poor candidates for SPM placement when our target platform can only use an SPM with a size 5-25% of the total program memory footprint. For those programs that contain unknown-size sites with very low FPB values and/or which made up more than 25% of the data occupancy, these sites are never placed into SPM, and instead our scheme places smaller and more valuable heap objects from other more profitable heap allocation sites.

The first three common uses for unknown-size heap allocation sites were all for situations where it was either impossible or unprofitable to place objects from these sites into SPM for an embedded system. The final three types are typically moderately sized and make better candidates for placement into limited SPM space for an

allocation method. Many of our predominantly known-size site benchmarks contain a few of the next few types of sites, and our unknown-size and recursive benchmark subsets tend to make heavy use among the final three kinds of unknown-size sites. Following the discussion of the final three types, we will present an overview of our methods for handling these three kinds of unknown-size heap allocation sites for dynamic SPM placement.

The fourth type of site is perhaps the single most common unknown-size site across all the programs and even textbooks dealing with heap memory allocation. This site is characterized by being the allocation site inside a simple wrapper function which many programmers use to allocate memory safely. Remember that the C system library does not guarantee that there will be enough system memory available to satisfy every heap allocation request. Because of this, many programmers place heap allocation requests inside a function which makes the request and then verifies that the pointer corresponded to a valid memory location. Generally allocation failure causes a program exit for embedded applications to avoid memory corruption. This common programmer practice of placing heap allocation calls inside of simple validation functions is often described as being malloc wrapper functions, because of the simple validation function wrapped around the heap allocation call. As we will see shortly, wherever possible we apply a transformation that attempts do fix the allocation size of these sites through cloning and inlining, allowing us to apply our fixed-size heap allocation site methods.

The fifth typical use for unknown-size sites is for small temporary array or simple data structures which operate on sections of the large input arrays allocated

by our third type of heap site. These are common in many media processing applications such as GSM and Susan, and often only allocate a few different size objects from these sites no matter what inputs are applied. Although these are unknown-size to our compiler analysis, many programs tend to use these sites as centralized allocation program locations for a few hard-coded data structures used for internal processing support. Those applications which used unknown-site allocation sites but which always allocated the same objects for different inputs can be considered known-size and are so grouped for our benchmark experiments. Those applications which allocate different temporary array objects for different inputs are one of the primary targets for our general unknown-size allocation support.

Finally, the sixth type of unknown-size site are those used by programs employing randomized dynamic data structures, such as trees or graphs, which are heavily input dependent for creation, processing and intermediate modification. These are common among benchmarks which rely on recursive function algorithms to traverse dynamic structures and add nodes in integer multiples during runtime processing. Also common is the allocation of many small objects used for token processing with iterative algorithms such as in the Spiff benchmark, which are created and destroyed throughout program execution. While these types of variables tend to exhibit mediocre FPB ratios at best, they are usually only of small benefit for SPM placement but important nonetheless to complete a robust compiler-directed dynamic SPM allocation method for embedded platforms.

Optimizing Unknown-size variables Many application suitable for embedded systems employ algorithms of moderate complexity which are usually more specific

for their target logic than desktop applications. Many of these use heap data, but do so in an organized and type-specific fashion, resulting in fixed-size heap allocation sites with very specific program purpose and typically predictable behavior across individual objects from that site. For more complex programs or for situations where heap data is managed collectively to reduce heap library interface program locations, we found that unknown-size allocation sites were more common. As we noted in the last few paragraphs, many of the objects allocated at such unknown-size sites tend to either be very small or very large, with low access frequency, thus making them poor candidates for any memory optimization scheme. From analysis of the different ways programs used moderate size unknown-size allocation sites with more frequently accessed objects, we developed a few different methods for handling the more promising of these unknown-size heap variable sites for placement into SPM.

Once our basic method for known-size heap objects was completed, we turned to adding support for the other types of dynamic program data. In the previous section on recursive function handling, we discussed our methods for handling recursive DPRG nodes and bounding their dynamic memory consumption through fixed-size stack frames similar to our fixed-size heap bin slots corresponding to individual heap object allocations. Our upgrade in compiler infrastructure also allowed the development of similar methods for unknown-size heap allocation sites. The most important compiler pass that we modified was the function inlining pass, a pass which clones the contents of a called function into its calling code location to avoid the invocation cost of a new function stack frame as well as its setup and return code. This is best explained with the help of an example program situation

```
[unknown.c]

main() {
   if (...) {
      VarA = func-A(4) }
   else {
      VarA = func-A(32) }
   ...
   Return
}

func-A(int size) {
   Return ( malloc(size) )
}

func-B(){
    VarB = func-A(20)
}
```

Figure 7.3: Example of an unknown-size heap allocation site inside of a program.

where this is useful.

Looking at Figure 7.3 we can see an example program that contains a heap allocation site of unknown-size inside of procedure *func-A*. Depending on a conditional statement, either a heap object of size 4 or size 8 bytes is requested using the same allocation site inside this procedure and could be called with different sizes from other procedures, making it a compile-time unknown allocation site. This is what we term a malloc wrapper, albeit the most basic one possible for explanatory purposes. From seeing the popularity of this heap interface approach in different applications, we focused on this scenario and attempted to take advantage of this behavior. GCC, like many compilers, is essentially a function-by-function compiler, and relies on a linker to create a final executable binary to tie together all the sepa-

rate code modules corresponding to functions in a typical multi-source C program. With later versions of GCC and the improved compiler analysis passes for the ARM, we were able to take better advantage of interprocedural knowledge to analyze malloc wrapper functions such as this one and how they are called from other functions in a program. We modified our compiler to analyze all functions which contained a heap creation call of unknown-size, which were smaller than 30 lines of code and which appeared in multiple program regions from the program call-graph. Those which matched our criteria had the wrapper function calls replaced by an inlined copy of the body of that wrapper function. Using this compiler pass, we were able to transform many unknown-size sites into known-size allocation sites for better guarantees and optimizations possible from our allocation method for known-size heap variables.

For those program locations which we are unable to transform to inlined known-size allocation sites, we have developed additional modifications for our iterative allocation method for their support.

Impact on our method Unknown-size heap allocation sites are treated like known-size sites when analyzing a program and optimizing its allocation, with only a few important differences. For unknown-size sites, the DPRG contains no static information, and all the information concerning allocated objects and their access frequencies comes directly from profile data. Variable instance nodes for that site remain the same as for regular heap nodes, although unlike known-size sites these will contain allocated objects of different sizes in the program DPRG. Initial heap bin and consensus computations are accomplished the same way for these objects as for

known-size objects in our iterative method. The primary difference in the consensus bin is that it represents a single amorphous chunk for that site as opposed to being made up of an integer number of known-size optimized heap objects. The use of a single contiguous chunk is enforced throughout our algorithm and dictates the approach to code generation for these sites. Lacking individual instance slots for this kind of heap bin, optimized variables of this type are assigned a more general kind of malloc wrapper. This wrapper does not restrict allocations to fixed allocation slot sizes, and instead treats the bin as a general free pool of memory allocated with a simple first-fit algorithm. While this allows us to optimize profitable unknown-size heap variables, the lack of a consistent allocation unit causes many problems for our current iterative approach.

Our allocation method is easily extended to support unknown-size heap objects for analysis and allocation purposes. Unfortunately, the extensions remove many of the heap variable properties which we take advantage of in our approach. The first apparent problem is the issue of fragmentation upon repeated allocations and freeing of heap space from amorphous heap bins. Because we no longer have fixed-size allocation requests, we can not guarantee that this heap bin will not become internally fragmented over time. Fortunately, this is not a problem our technique alone incurs, as its a problem typical of all heap manager algorithms. For long running benchmark applications, a programmer would simply investigate either a more advanced malloc-wrapper creation scheme, or use one of several already published schemes that tries to avoid this very problem for heap management.

The other problem with not having fixed size slots in our bins is that we lose

fine-grain control over allocation of heap objects by size, which is usually directly related to their program data type in C. For example, suppose there exists a heapsite that tends to allocate varying sizes of heap objects, of size M, N, and O 90% of the time with a wide variety of other sizes the other 10% fo the time, and only objects of size M were deemed to have an FPB worth placing in SPM. Without a runtime method of controlling which particular objects are allocated to SPM for this unknown-size site, we are less able to guarantee the the most beneficial type of heap objects allocated at this site will actually be placed into SPM at runtime. Specialized methods are needed to filter incoming requests dynamically and guess which objects allocated at runtime are the ones will receive the most accesses and should thus be allocated to SPM first. We noticed that in many cases for such optimized sites, very often heap objects which are actually placed in runtime with our First-Come-First-Served are those with low FPBs, with the truly profitable objects being allocated at much later program timestamps. This is important because objects allocated with the same size in C programs tend to also be of the same type, and often only certain types have a large proportion of total memory accesses for a program. This was the motivation behind our compile-time transformation to inline and create separate heap allocation sites for a program, each with a fixed-size, from an originally unknown-size site. This gives us finer control over objects by site for SPM allocation, raising the guarantees that the objects we with to place in SPM at runtime are actually placed into SPM.

Finally, the lack of a consistent base unit of allocation for a heapsite prevents the majority of our iterative optimizations from being effective. First, the heap

bin consensus pass for initial allocation assigns all unknown-size sites with a single amorphous bin of memory for possible SPM placement. Lacking smaller units which make up this bin, this site receives a fixed size allotment of available SPM and cannot be changed by any of our refinement methods. Thus, we cannot apply heap bin resizing except for the trivial case where we do or do not place the entire consensus bin into SPM for a particular iteration. As a tradeoff, we do apply our variable swapping pass to unknown-size heap variables in our feedback refinement step, a step usually used with only stack and global variables. Unknown-size sites also tend to use a large variety of pointers to different data types, making pointer analysis particularly obscure for these variables, and greatly reducing our access guarantees for different program regions to minimize transfers. Finally, our layout pass is unable to split these bins into smaller chunks when this site conflicts with other variables with higher FPBs. Without a guaranteed fixed unit of allocation for this site, splitting the bin into smaller pieces may render it completely unusable by any allocation requests at this site, resulted in unused and wasted SPM space.

Possible Improvements Handling unknown-size allocation sites robustly and optimally for all cases is most likely an unsolvable problem because of their completely dynamic nature. In fact, even after much research and studying of these structures in typical programs, our most successful methods for optimizing this type of variable is to transform them into more predictable known-size heap allocation sites. Otherwise, our best results for beneficial allocation of unknown-size heap objects were for those applications in which always allocated a set of known-size objects across different inputs. For the truly dynamic unknown-size sites, our allocation

extensions improved performance somewhat, but mostly prompted the development of our profile sensitivity reduction approach. This will be covered in the next section, but first we will present a discussion on other investigations we undertook to improve allocation for these kinda of objects as well as future areas of research for further improvement.

One method we looked into was specializing our malloc wrappers for unknown-size sites to filter and only place those sizes which were profiled to be profitable into SPM. Although we experimented with several versions of this approach, we found that many programs exhibited behavior for different inputs which caused the filtering to fail. For example, one program showed 3 different common allocation sizes for one input (one very profitable), and then showed 3 completely different common allocation sizes for a different input. By specializing our wrappers for the first three expected sizes(and only allowing one size into SPM), when this optimized application was run with the second input, no objects from that site had matching sizes and none were placed into SPM, resulting in completely wasted SPM space. In a similar vein was an attempt to track the size requests as well as the time-ordering of different requests from the same site throughout program execution. This was done on the observation that for some unknown-size sites, there seems to be more usage for earlier or later requests for certain size objects, according to profile results. This also tended to over-specialize the generated malloc wrappers however, again causing problems with many applications and was not used for our final implementation.

Like caches, our performance is also affected by a program's dynamic behavior. When an optimized heap allocation site creates more objects than fit into the

available SPM space, the rest of the objects will be placed into main memory. If for this program the most accessed variables were those that reside in main memory, and not those placed into SPM, then we reap little or even negative benefit from having placed them into SPM(transfers not justified by the accesses). We experimented with constraining SPM placement through temporal tracking, similar to our recursive method. For example, if a program were to allocate a binary tree from the root down, the shallowest levels of the tree would be placed into SPM for an optimized site with limited bin slots. If this program performed most of its accesses to objects allocated later in time, our method might build a wrapper that placed the first ten requests to main memory at that site, and then attempt SPM. This worked well when used with the same profile. For applications which exhibited the wide access distribution among heap objects from the same site, this tended to fail miserably for different inputs however.

Instead of overspecializing for these cases, we focused effort on methods to reduce our profile sensitivity and make better general predictions using full profile information. Our next section presents the final contribution of this research concerning our methods to reduce the profile sensitivity of our proposed allocations for programs which exhibit a great deal of profile-dependent memory and execution behavior, typical of those using a large proportion of unknown-size objects.

7.4 Profile Sensitivity

This section will discuss the issues involved with making general allocation decisions based on a combination of static analysis and dynamic execution characteristics. When dealing with dynamically allocated data, static analysis is insufficient and generally of very little use in predicting which objects should be preferentially allocated to faster rather than slower memory areas. Lacking comprehensive static compiler analysis tools, methods dealing with dynamic program data must instead rely on comprehensive dynamic profiling of the program using typical inputs. Some dynamic program variables are more predictable than others by their nature, and there are many programs with variables whose runtime behavior is completely determined by a choice of program inputs, making them unpredictable for an unknown input. For any allocation scheme which seeks to optimally place dynamic program data at runtime, it is vitally important to reduce the sensitivity of the program allocation scheme to the choice of program profiles analyzed. Comprehensive analysis methods greatly increase the chance that a chosen allocation will work well for the majority of expected program inputs.

For some applications profile sensitivity is not a problem, as their allocation and execution patterns vary minutely when different program inputs are applied. For other applications however, there is an intrinsic dependence between input data and a heap allocated object's behavior at runtime. For example, if we have a program that allocated a dynamic data structure such as a binary tree to represent an input set, the shape, size and access behavior for the trees produced by different

program executions may change greatly depending on the input used. For some applications, the heap objects allocated at a particular heapsite may not only behave differently for different inputs, but some also vary widely for the same input. This could cause serious problems if our allocator makes the poor assumption that a program heapsite will request a homogeneous heap variable repeatedly from the same site, when in reality it may request different sizes depending on the input. The previous section detailed our methods for reducing profile sensitivity for unknown-size heap allocation sites from the same input set. The following paragraphs detail our expanded methods to further reduce our method's sensitivity to the profile training data needed to guide our allocation process for input-dependent programs.

Reducing profile sensitivity is traditionally accomplished using static, dynamic and most often a combination of both static and dynamic compiler analysis approaches. For dynamic program data such as heap or recursive stack frames, static analysis with basic dynamic profile information are sufficient to make good predictions for programs with generally unchanging program behavior for possible (and non-trivial) program inputs [1]. As program complexity increases, generally so does the dynamic memory usage, with a corresponding increase in a program's memory allocation patterns and complexity of transformations(eg: larger, more convoluted data structures for complex processing such as in databases). As with any profile-based scheme, the larger the profile sample, the more information will be available for a prediction scheme to take into account. Any compile-time scheme such as ours

[1]Program inputs which cause errors or other program termination are not considered useful for optimization purposes because these are not normal application behavior

is essentially predicting future program behavior and attempting to place certain portions of the program into SPM for locality benefits. The best allocation results come from schemes which closely tailor program memory placement to match profiled usage, and for which the predictions prove consistently accurate.

The importance of reducing profile sensitivity became most apparent after analyzing the performance of our allocation method on applications with truly unknown-size allocation sites as well as its general performance when non-profile input program data was applied and performance measured. We focused our time on developing methods to overcome these problems by applying the same concepts we used for our recursive and unknown-size extensions. We studied in particular those situations where our method made bad predictions, resulting in little or negative runtime gain, and tried to find ways to avoid as many of these bad decisions as possible. We had some success with transforming unknown-size allocation sites to separate them into different known-size sites with more predictable behavior. For all other situations, we decided that the only effective way to make the best general predictions for programs with statically unpredictable behavior was to increase our knowledge of its dynamic behavior and combine it with what static information our compiler is able to obtain. This led to our modification of the way we build the DPRG for applications to reduce our profile sensitivity.

To better predict dynamic program behavior, more information on its previous behavior for different situations is generally the only useful source available to compiler designers. With increased information comes the price of greatly increased cost for an allocation algorithm to store and process that information efficiently. This

can be seen in our choice of the DPRG to represent a program under compiler analysis for SPM optimization. This structure contains control-flow and dynamic access information, but discards execution-path information with detailed cycle-accurate behavior recording, which would greatly increase storage and processing requirements. While such temporal information is important for other optimizations, we found that the increased complexity entailed by path-based profiling provided little additional and generally applicable insight into optimal allocations to limited SPM space at runtime. This concept of collapsing path-based information in favor of total program access behavior for our regular method was adopted when we developed methods to reduce our profile sensitivity for input-dependent applications.

Reducing profile sensitivity requires a better understanding of how a program will behave with respect to its input data. Accordingly, we have modified our own DPRG to capture the important additional information we need to better predict how a candidate allocation will benefits an application for the broadest number of input scenarios. We first begin by capturing separate DPRG structures for each program using individual program inputs. This provides static and dynamic behavior information in a compact representation of different ways a program can behave when it executes. Theoretically, the only way to account for all possible program behavior is to sample all possible program behavior at design-time, similar to how programmers perform validation experiments on their code designs. In practice this proves to be generally impossible, and infeasible for most embedded engineers to perform in practice, leading them to choose safer and less invasive optimization methods. Ideally, a good compiler analysis and optimization scheme will be able to

use the minimum amount of dynamic information possible, reducing analysis time and complexity, but generally also reducing its overall accuracy as well. Our entire method tries to create an effective and efficient heuristic analysis and optimization tool to deduce an optimal memory allocation for a particular program, even if that program exhibits dynamic and hard to predict behavior at runtime. A key observation is that it is often more desirable to give more SPM space to predictable and less profitable variables, then hoping for highly accurate predictions for more profitable (but much less predictable) dynamic program objects.

The DPRG was created as a tool for efficient static and dynamic information storage for a program. While this is sufficient for many programs, dynamic data requires some modifications to the DPRG to better account for dynamic program behavior across different inputs, both in terms of its possible execution path and data access pattern. Knowing that we need a way to better predict program behavior, and lacking static analysis methods applicable to these dynamic program variables, we instead gather more dynamic profile information for use in a new DPRG structure. By looking at those programs which were very input-sensitive, we saw that in many cases it was very hard for even a trained human to correlate a particular input to how those unpredictable variables would behave. Instead, we theorized that instead of over-specializing our methods to try and find useful patterns, we would be better served by instead minimizing the possibility of poor allocations by placing less importance on unpredictable variables, despite their FPB. We found that by using profile inputs that behaved quite differently, and combining the information from both inputs, we were able to better automatically identify which dynamic

```
[unknown.c]

main() {
   if (...) {
      VarA = func-A(4) }
   else {
      VarA = func-A(32) }
   ...
   Return
}
...
```

Figure 7.4: Example of an unknown-size heap allocation site inside of a function.

allocation sites were predictable and deserved more SPM placement.

Let us return to the same example program presented in Figure 7.3 to see how we should modify our DPRG to account for different program profiles resulting from different program inputs. The main function is extracted and redisplayed in Figure 7.4. For this function, we assume that with the first program input applied, the if condition is taken, and VarA is assigned 4 bytes of space from the heap and is very frequently used throughout the program. Now with a second program input, the if condition was not taken and VarA is instead assigned 32 bytes of space from the heap and is barely accessed throughout the remainder of the program execution. If we use only the DPRG generated from the first input, running the second input on the optimized application would show suboptimal performance. This is because our allocation was based on the first input, where VarA was smaller and more frequently accessed, and placed into SPM by the compiler. Using the second input reveals that the optimized heap bin should not have been placed in SPM and instead its space

should most likely have been given to another variables for increased benefit. Indeed, this becomes even more serious for realistic heap sites that allocate a large number of heap objects regardless of program input. Instead, our DPRG should reflect as many different program profiles as possible for those applications which are shown to be input dependent.

We take the following steps to prepare our DPRG for representation of multiple, possibly conflicting program execution profiles resulting from different program inputs. First we prepare a copy of the static program DPRG from the compiler, listing all program regions and any statically discoverable variables and access frequencies. We then process each program region by collecting the access information for all variables alive for that region from all available DPRGs corresponding to different program inputs. Our master copy, or averaged-DPRG, then updates the region node being processed with averaged information using all access statistics from the various profile DPRGs. Each region node is assigned an average value for the frequency that the region is accessed, and all edges visiting that region also receive average frequency assignments. Each region node also contains every variable node alive for that region which appears in any input DPRG. Each variable in that region is also averaged in terms of its access frequency among all DPRGs read in. In concept, this is similar to how we handle repeated program visits to a region at runtime which exhibit different memory access behavior for different execution paths, or even the same execution path. By not overspecializing our approach to such a fine-grained, time-dependent view, our heuristic method is better able to trade-off SPM allocation among all program objects by looking at the broader picture.

NODE	DPRG 1 FREQ	DPRG 2 FREQ	AVG FREQ
REGION_1	100	200	150
Heap-A	100	100	100
Heap-B	40	40	40
Stack_1	100	10	55
Global	20	200	110

To illustrate how this averaging process works, let us refer to a simplified DPRG for a sample program region called Region_1. This figure only lists the nodes and frequencies for that region, and lists the data collected from two different program inputs as well as the value for the averaged DPRG created. From the table, we can see that this region was visited 100 times for the first program input and 200 times for the second program input, receiving an average value of 150 for its access frequency in our averaged-DPRG. We see the same methods applied to all variables alive in that region. Of particular interest are variables Stack_1 (a stack variable from this region) and Global (a global variable). An SPM allocation derived solely from the first program input would place Stack_1 into SPM and leave Global in DRAM, while one based on the second profile would do the opposite. Which one is the right decision? Using the averaged-DPRG, our allocation algorithm is better able to guide its decisions based on the average trend among all variables for different program inputs.

The creation of the averaged-DPRG is similar in many respects to the way we create a single DPRG for an application and its static and dynamic behavior for a single program input. The same program regions appear in all DPRGs regardless of program inputs, the only differences are in the contents of the region nodes and

the edge frequencies between region nodes for each DPRG. Common conflicts are resolved as follows. The first apparent conflict occurs when a variable does not appear inside the same region for two different DPRGs. This can only occur for dynamic program variables such as heap objects or recursive stack frames and is not a problem. The variable will appear in the averaged-DPRG, but with zero access contribution from each of the DPRGs in which that variable did not exist. The other problem of having a variable appear in the same region, but with different sizes has already been addressed in the previous section for unknown-size objects, and is a problem unique to heap variables.

By averaging the contents of several DPRGs, our method is given much more information on the typical program behavior observed for different inputs. Very little variation is seen for many applications which have very predictable program behavior as well as dynamic data usage, and so any DPRG can be used to optimize these applications for general inputs. When significant differences between DPRGs are observed for different inputs, we can immediately identify which program regions and variables are input-dependent and for which allocation predictions may be inaccurate. By averaging the DPRG structures together from the different inputs, we are able to strengthen the relative importance of predictable variables which retain constant FPB values, while tending to decrease the relative FPBs of those variables which large input-dependence and profile variation. This is also helpful for programs containing unknown-size allocation sites, which tend to be the hardest to predict accurately using a single input profile.

In conclusion, reducing profile sensitivity is important for any allocation scheme,

because they are all inherently profile dependent. Using our averaging approach to creating a representative program representation in an averaged-DPRG helps guide our iterative search toward placement of those variables which are consistently shown to be of benefit, regardless of program input. To achieve best results, a designer should pay special attention to programs which exhibit significant behavior variation and attempt to capture a representative set of program inputs for the expected range of common behaviors. Our method is able to achieve profile-insensitive results using only a single program input for about half of our benchmark suite. Using only a single additional profile input with our averaging technique reduces our profile sensitivity by a further quarter of our total benchmark suite. As with any other approach, for best prediction of dynamic program behavior, the widest range of possible profiled input information should be collected and fed into our memory allocator.

Chapter 8

Methodology

To evaluate the effectiveness of our SPM allocation techniques, we have implemented our methods using readily available open-source compiler tools and tested them using a set of free benchmark applications on open-source simulation tools. Despite being open-source, the chosen packages possess many of the desirable features present in industrial development software for code analysis and optimization. While software support is not available in the open-source world for all embedded platforms, the compiler used for this research produces code of industrial quality for the targeted platform, as well as providing a fairly accurate simulation framework for benchmark execution and power statistics. Although tools provided by commercial vendors generally produce more efficient executables and more accurate simulations, they are also very heavily copyrighted, rendering access to compiler source close to impossible. The proposed method instead relies on a modern open-source development infrastructure including a compiler, assembler and linker for development, along with an open-source platform simulator for profiling and execution. Our methods were designed to make use of a variety of software-hardware interfaces targeting the widest range of embedded hardware platforms that would benefit from our techniques. Similarly, we have chosen a large set of open-source applications to compile with our methods and evaluate their execution and power characteristics. Our cho-

sen combination of hardware and software platforms have the advantage of being in common use today, enjoy industry-level open-source support for development and simulation, and have been instrumental in the development of the dynamic memory allocation methods applicable to a wide range of embedded applications.

Section 8.1 will describe the class of embedded hardware platforms targeted by this research, chosen for its popularity in industry as well as academia. Section 8.2 will discuss the general software requirements for implementation of our dynamic allocation methods. Details on the compiler software developed and implemented for this research will be presented in the Section 8.3. Section 8.4 will discuss the simulation engines employed to evaluate our method's impact on execution time and power consumption for benchmarks. Sections 8.5, 8.6 and 8.7 will discuss the benchmark set chosen for simulation, the typical classes of applications observed during investigation and finally present the relevant execution and allocation statistics for the set.

8.1 Target Hardware Platform

This section describes our target embedded system platform, one which is representative of a large number of the popular chips in use today. Generally, our method is useful on any typical commercial processor platform that includes a compiler-exposed scratchpad (or SRAM-like) memory along with other heterogeneous memories that can be controlled or accessed through processor instructions. While our method is potentially useful on a very large number of existing hardware

platforms, the research described in this document was tested with the Advanced Risc Machines Ltd.(ARM) v4 Instruction Set Architecture(ISA) CPU family. This is a good example of a popular RISC CPU family that supports various memory configurations along with efficient compiler and ISA interfaces. Our methods are equally applicable on the ARM or any other CPU supporting scratchpad memory with a compiler-friendly ISA, allowing improved performance at a reduced energy cost for programs using dynamic memory.

The ARM V4 is the earliest ISA version supported by this research, which is still currently used in a number of processor cores. This is currently the oldest ISA version still in production and being supported by ARM. They also produce the ARMv4T, ARMv5TE-J, ARMv6, and ARMv7 core families, comprising their current processor offerings. While the actual hardware implementation of their cores may vary and evolve, the ISA implementation remains consistent and ensures programmers will enjoy the same or better level of hardware instruction depth as the core design evolves. In general the market leaders in embedded processors have been those that have produced energy/performance efficient cores and have kept a consistent well-designed ISA as their hardware designs evolved. By targeting our methods toward common embedded hardware configurations that employ a standardized, compiler-accessible ISA interface, we ensure our methods are directly applicable to a very large range of current and future processors.

The individual processor core we target in our research is the Intel StrongARM SA-1100 [73], a processor that has been heavily investigated in academia. The Intel StrongARM was developed for portable and embedded application requirements

requiring low power consumption while maintaining reasonable performance. It consists of a 32-bit ARM pipelined RISC(Reduced Instruction Set Computer) processor with adjustable speeds between 60MhZ at 0.8V to 206 MHz at 1.5V. The adjustable speeds are consistent with the power-management modes available to the device, aimed at saving energy when the processor enters long idle or sleep modes of operation. As with the majority of the ARM processor implementations, it contains general purpose hardware to implement memory controllers and I/O devices among other useful peripherals for complete platform deployment.Our research targets the Intel StrongARM processor platform in particular, operating at 1.5V and with a 206MHz clock speed.

This particular core implementation was chosen as a model representative of commercial embedded cores and because of the power and latency information available for use in simulation. More detailed information available concerning our chosen embedded hardware and its hardware models will be presented at the end of this chapter.

Since ARM provides a variety of synthesizable cores, an end-user can generally expect a choice in configurable fast memory devices to deploy on the core in the form of Scratch-Pad Memory(SPM) or Tightly-Coupled Memory(TCM), and also in the form of instruction and data caches. Many of their cores also include a variety of DRAM, FLASH and ROM modules when required on-chip and the broad majority implement on-chip memory controllers for off-chip memory modules. This research uses the Intel StrongARM SA-1100 core configured with common embedded memory configurations to evaluate the effectiveness of our allocation methods. In

order to capture the widest selection of embedded platform configurations, we have evaluated and simulated various sizes of ScratchPad, Cache, SRAM, DRAM and ROM memory modules. While most of our results focus on ScratchPad and DRAM as typical memory configurations, our methods are also applicable when other types of heterogeneous memory is present on the same platform. Further details on the particular configurations explored will be presented in the simulation section as well as the results and analysis chapters.

Although our current research is based on an ARM embedded system, our methods were first developed and applied on the more primitive Motorola MCore processor family. The Motorola Mcore is a 32-bit RISC microprocessor designed for low power operation between 2.7 and 3.6 volts at 33MHz. This microprocessor comes packaged with 8-32 Kb of ScratchPad Memory, 0-256 KB of Flash Memory, and external memory controllers for up to 32MB in the form of DRAM, ROM, Flash or other memory peripherals. The Mcore ISA contained standard memory load and store instructions, as well as versions that operated on multiple registers at once, allowing pseudo-DMA operation when loading from memory banks. Lacking cache memory support, the MCore processor proved to be a perfect target for our SPM allocation methods.

Unfortunately, lack of adequate compiler tools led us to switch to the ARM platform to best evaluate our compiler allocation methods. The open-source GCC compiler available for the MCore was inferior and produced almost unoptimized code. We observed the compiler produce redundant memory operations, lacked library support for realistic applications and generally employed few compiler op-

timizations and working executables. For the MCore applications we were able to evaluate, the benefits of our system were consistent with the improvements observed on the ARM platform after accounting for the disparity in platforms. A short section in the Results chapter will discuss results obtained on the MCore platform. By examining these combined results, we find that our methods are equally applicable to low-end processors such as the Mcore and mid-level or high-end processors such as those in the ARM family, employing varying levels of software and hardware support.

8.2 Software Platform Requirements

Before implementing our allocation strategy, we surveyed a variety of embedded platforms to determine common hardware features and typical software interfaces. By choosing an entirely software driven approach to memory allocation, our method is readily applicable to a large number of embedded platforms, without requiring the costly hardware redesign and modification other methods require. Because our method is entirely compiler-based, memory and processor instructions are deciding factors in the software implementations possible. High-end embedded processors generally contain specialized memory instructions and hardware which would make the mechanics of our dynamic allocation methods considerably more efficient. Instead of focusing on high-end processors with advanced memory hardware/software, we instead looked for the most common embedded instructions in current use across low and middle range processors with heterogeneous memory

hardware. This section will discuss the software considerations when applying our methods to a particular embedded platform, both in terms of requirements from the ISA for allocation as well as requirements for the development platform on which our method will be applied.

The proposed allocation methods attempt to improve performance by dynamically allocating and transferring program objects using common instructions, without requiring additional hardware or esoteric ISA support. As a bare minimum, our methods requires the presence of basic read and write memory instructions able to access data at the different memory banks. The compiler methods can make use of the same data transfer instructions used for program operation to implement dynamic memory allocation to SPM as well as slower system memory. In order to do this efficiently across different embedded platforms, our methods can be tailored further according to the software and instructions available for a particular architecture. Our method is able to make use of optimized memory transfer methods such as Direct Memory Access(DMA) instructions or pseudo-DMA software methods.

As opposed to hardware caches and other forms of hardware memory managers, our methods can use any improved software exposed interfaces such as DMA for the lowest overheads for our allocations. DMA support is available in many embedded processors and consists of an enhanced software-controlled memory controller able to operate in parallel with the CPU and perform memory transfers more quickly than through serialized instruction-based word transfers. We have developed compiler methods for both the ARM and MCore that make use of DMA transfer functions to transfer data more efficiently than when using standard load

and store instructions. For our results, we model DMA operation at a cycle cost of N*DRAM_LATENCY/4 cycles per N words plus a 10 instruction overhead from the DMA instruction code executed for each transfer, along with associated power costs for memory and instructions.

Many embedded processors that do not support DMA do support some form of pseudo-DMA, a term we use to describe software methods that speed up data transfer over the basic word-level load/store method. Pseudo-DMA methods include those which use instructions that can read or write to a range of registers from external memory, taking advantage of DRAM data pipelining of sequential accesses. These can also include methods that use structured loops for large data transfers which take advantage of interrupts to better tolerate long-latency DRAM operations while processing instructions in the background. These and many other software-based methods exist today, many of which can be used to further decrease the overhead of dynamic data transfers incurred by our allocation methods but none of which are necessary to its success. We apply pseudo-DMA operations only to transfer blocks of 24 bytes or larger, since smaller blocks do not overcome the instruction overhead needed by the transfer method. For our results, an N-word transfer requires N*DRAM_LATENCY/2 cycles for the dram access cost plus an additional cycle for every 4 words transferred. Similar to DMA, pseudo-DMA is implemented using structured functions, so each transfer also adds a 10 instruction overhead for the processor instructions required, along with associated power costs for memory and instructions.

Regardless of the ISA features available for a hardware platform, some form

of compiler or development framework is required to automatically perform dynamic memory allocation during program execution. Our very first static allocation method relied on a compiler to produce assembly versions for all benchmark C source files, after which we would scan the assembly for memory operations affecting memory objects identified by reading the assembly code. To perform profiling our allocator would add assembly files containing custom profiling functions as well as assembly calls to these functions at program locations that accessed, allocated or de-allocated memory for program objects. Allocation was similarly performed with our method by parsing and modifying the assembly files just before final compilation into binary executables. The benefit of this approach was the ability to perform profiling and static allocation without requiring access to or modification of a compiler, simulation package or hardware platform. Unfortunately, this approach also lacks the benefits of compiler passes such as those used to identify and allocate all program objects at the smallest granularity as well as alias analysis passes to enable dynamic allocation in the presence of pointers. Furthermore, the software inserted profiling dramatically slowed execution and its effects on benchmark simulation and execution speed were difficult to distinguish without access to a modifiable simulator or advanced hardware testing platform, prompting us to look at open-source development platforms.

The majority of this research was conducted using the Gnu Compiler Collection(GCC) open-source compiler framework, the Gnu DeBugger(GDB) open-source simulator and an entirely open-source software benchmark library, modified to implement our memory allocation schemes. The use of open-source software allowed

us much greater flexibility in seamlessly applying our allocation methods to produce an enhanced development platform with minimal impact to the typical development process. Access to the source code that compiles and simulates an ISA was crucial to building an industrial strength compiler method that performs efficiently for the majority of applications and places no additional burden on the programmer. The following section will detail the modifications we have made to the compiler, while the final section in this chapter will discuss the simulation software used to evaluate our allocation scheme.

8.3 Compiler Implementation

As for any compiler-based method, the choice of compiler platform is crucial in determining how much program information is available and to what degree the compiler may be modified to perform advanced passes. Since complex methods require tighter coordination with the compiler, we focused on compilers which were open-source or for which we could obtain source code to properly integrate our allocation techniques. As part of our initial investigation, we evaluated several different compiler alternatives targeting different processor platforms before choosing the open-source GCC compiler for the ARM. Our search was restricted to open-source platforms only after we failed to obtain access to any commercial compilers for academic research. Being open-source was not the only requirement for our dynamic memory allocation method, and this section focuses on the reasons GCC was chosen for our research and its impact on our success. The four major compilation

steps comprising our method will be discussed to provide insight into the additions and modifications required to implement our research using the GCC C-compiler. Finally, this section ends with a brief discussion of any difficulties that arose while using this particular compiler and development platform.

GCC C Compiler

Our current dynamic memory allocation scheme is based on the most recent Gnu Compiler Collection(GCC) C-language compiler targeting the ARM embedded processor ISA. The ARM continue to be a popular ISA for embedded developers and benefits from full open-source support through one of their divisions, Code Sourcery Ltd. Code Sourcery, in combination with a large developer community, has contributed a wide variety of optimizations and software solutions aimed at producing efficient application code using GCC. With the support of ARM, Code Sourcery, and countless users, the latest GCC 4.x family contains many of the features found in high-end commercial compilers and has contributed to the success and validity of our compiler-based research.

The two biggest reasons we chose GCC for this research was due to both its high level of development support for the target platform as well as the breadth of compiler interfaces available to us in an open-source package. As this research was developed, our methods were integrated and adapted to successive evolutions of GCC, including two major version releases beginning with version 2.95 and up to the current version 4.1 package. The Motorola MCore was first supported in the 2.9x release with a very poor code generation backend and even worse library support, after which no further improvements were made for the MCore. On the

other hand, the ARM branch of GCC has had excellent support and has evolved with the compiler to include all the modern features of GCC as well as a high quality code generator, very good development library support and other software features to take advantage of ARM hardware. Further information on the current status of GCC can be found on the Gnu website[53] for ARM and all other supported platforms. While we expect our techniques to perform even better on commercial compilers, we were unable to obtain access to source code for any industrial compiler packages for the ARM platform, typical of all commercial compilers.

Figure 8.2 gives an overview of the typical steps the GCC compiler takes when compiling an application from source files. While we have taken advantage of several of the optimizations present in GCC, we will only discuss in detail those modifications essential to our process. Although over a dozen are listed, the steps shown represent only a small number of all the compiler passes performed inside GCC when creating an application executable. Details concerning individual steps GCC performs can be found in [128].

GCC Compiler Modifications In general, GCC was made for portability and runs on a variety of host machines, generating code compliant with the target machine specification provided by a vendor. The compiler begins by reading in a source file, and then proceeds to parse and compile the source code in that file function by function to produce an assembly equivalent for that file. For multi-source programs, this happens for each file in the program and then handed off to the linker to produce a final object binary. Only those steps which are directly affected by our method will be detailed, avoiding common compilation steps such as parsing, code

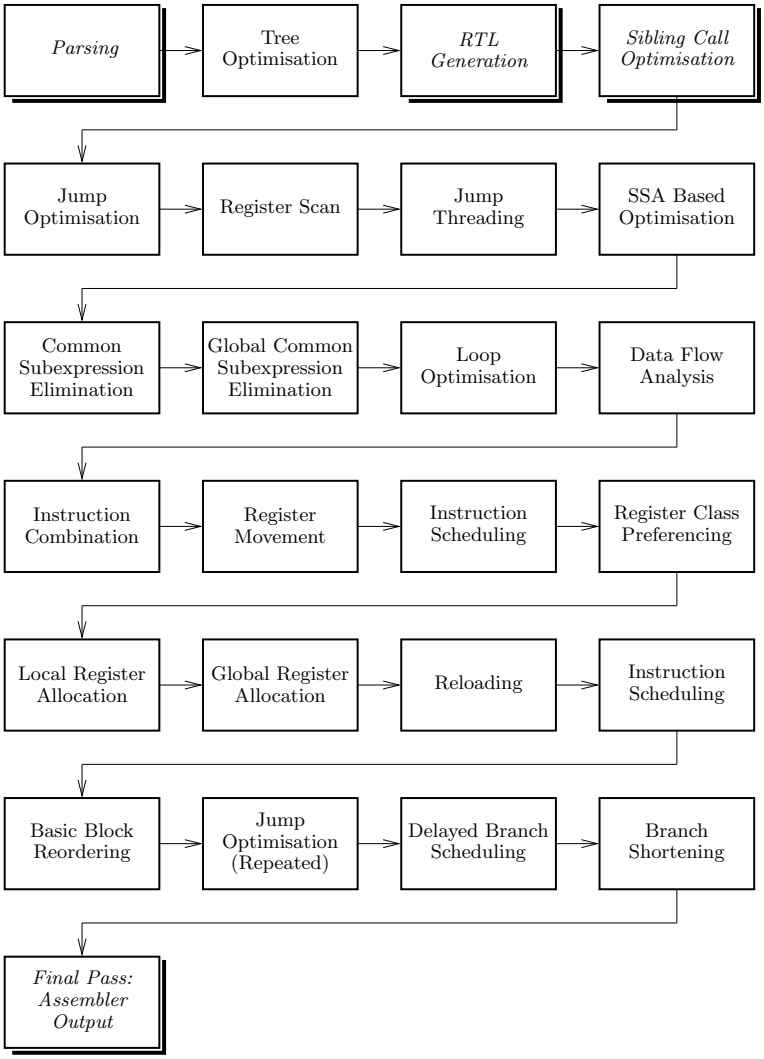


Figure 8.1: Detailed view of GCC compiler flow for a typical application.

optimizations and specific back-end optimizations for the target processor.

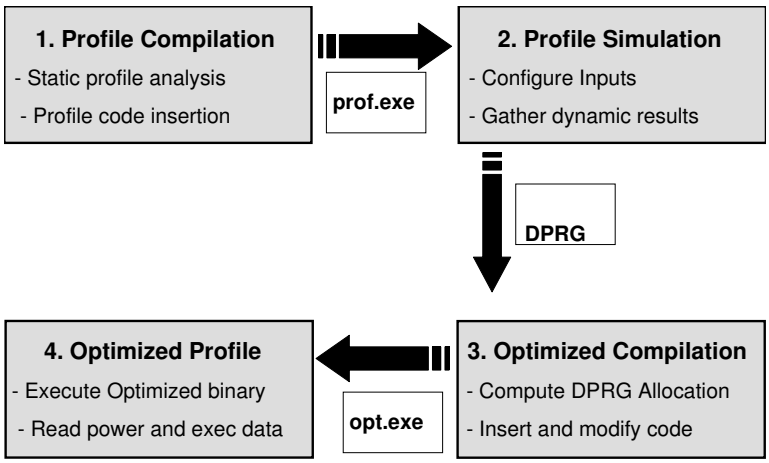


Figure 8.2: Four stages comprising the results collection process for SPM allocation experiments.

Figure 8.2 shows a high-level representation of the steps applied to each benchmark to obtain the results presented in this thesis. The first stage involved compiling the target application into an executable containing profile code for use in the simulator, during which it gathers static profile information on variables and program control flow. The second stage executes the instrumented binary on an accurate platform simulator using desired program inputs to generate dynamic information concerning the program memory allocation and program execution behavior. The second stage ends by annealing all static and dynamic program information gathered so far into the application DPRG. The complete DPRG is used in the third stage as an input to the SPM memory allocation process, which processes the profile information to guide its allocation process. After iterating through different solutions, the best one is chosen and implemented by recompiling the application using the dynamic layout calculated. Finally, the fourth stage concludes the process

by executing the optimized binary, after also instrumenting it with profile code, to check on the performance of the chosen SPM allocation for the program.

Static and Dynamic Profiling The earliest change made to the compiler is immediately after the compiler has completed its optimizations on the low-level Register Transfer Language(RTL) representation of the function currently being compiled. At this point, we can build the Control-Flow-Graph information augmented with variables and static analysis of variable frequency by examining code generated and relevant control structures such as loops or conditional paths. This information is dumped for each function from each file compiled and later reconstituted to provide a static DPRG representation used in our optimization pass. The static profile information will also be used during simulation to match local variables with their assigned stack pointer offsets once a function is entered, for accurate variable profiling. By waiting until the end of the compilation process, we know which variables the compiler was able to allocate to registers, and which were placed on the program stack and are candidates for SPM allocation.

Immediately after we gather the static profile information, we then scan through the chain of RTL nodes representing the current function. This low-level form closely corresponds with the assembly instructions which will be produced shortly by the final step of the backend. It is at this point that we can perform our own code insertions used to mark the start and endpoints of regions corresponding to those used in the DPRG representation. These markers are also associated with unique program points for use by the simulator to gather dynamic profile statistics as well as for optimization purposes. At this point we also collect pointer analysis informa-

tion provided through new alias analysis modules added to the GCC 4 release. The alias analysis information is used to determine regions where a heapsite may or will not be referenced. These regions are then used to define the limited lifetime region sets for heap instances allocated from that call site.

Profile Collection Once the profile-augmented binary has been produced, it is then fed into the ARM simulator for execution with a variety of representative inputs. By collecting the dynamic DPRG information through these profile runs, we can produce a coalesced DPRG that contains the static and dynamic program information used by our allocation pass to decide which variables are most profitable to place in SPM. Since we are mostly concerned with dynamically allocated memory objects placed on the heap, this is an essential step in predicting average trends for a program that may exhibit widely varying behavior due to input dependence. Using the static dprg created during compilation as the basic framework, dynamic execution information is added to the structure as the program executes. Upon loading the binary, the simulator reads the binary file descriptor used in ELF format binaries to extract information concerning code, text and data segments. These segment descriptors contain the global symbol table, declarations of all library and user functions present in the program as well as the addresses corresponding to each variable type. Heap variables are discovered as calls to the relevant allocation functions are intercepted with the size and assigned address information. Edge information between dprg regions is also discovered dynamically as the program encounters profile markers during execution. Despite the large amount of information collected during profile runs, the simulation speed does not degrade enormously through the

use of efficient data structures for looking up address accesses caused by memory instructions.

SPM Allocation The resulting dprg is then used to determine which regions and variables should be allocated to SPM dynamically or statically to minimize overall execution and energy costs for the target program. The detailed explanation of this process is discussed in Chapter 6.

Optimized Compilation Using the results of the SPM allocation pass, we now proceed to recompile the target program with the allocation decisions arrived at. Those stack variables that were promoted to global variable status for SPM allocation purposes are so modified when the program has been described in its initial tree form before RTL generation. Optimized stack variables are replaced by either DRAM or SRAM labels corresponding to that variable in all regions where it is present. This is also done for global variables that were optimized for any program regions. Once the compiler lowers the tree structures into RTL code, we can then replace optimized heap allocation sites with calls to wrapper functions that contain information on the bin assignments for use at runtime in generating addresses. Similarly, we can now also place at needed region transition points the code necessary to perform memory transfers for optimized variables between SPM and DRAM at runtime. The necessary code for the optimized heap allocation functions is also included as part of the compiled binary, ensuring that no unforeseen optimization passes are performed on this supporting code. Finally if code allocation was also optimized, then the final assembly files are modified to implement the code allocation scheme chosen, after which the code is linked into a final program binary.

Results Collection The optimized binary can now be executed on the simulator which has a specified address area assigned as the SPM bank with associated access and latency costs. The simulator can also enable or disable the presence of a cache hierarchy for all memory accesses to those address spaces marked as cacheable in the simulator. A detailed report is presented which specifies variables and their memory access types, information on transfers, information on stalls due to memory or instruction delays, cache performance and other execution data.

8.4 Simulation Platform

To estimate the effectiveness of our compiler-based SPM allocation scheme, a standard ARM embedded platform is the only real requirement for executing optimized executables and extracting power and execution data. Lacking access to laboratory and ARM development hardware, we have chosen a software simulator augmented with execution and power models tailored to our chosen ARM hardware platform. Indeed, for most embedded developers, cross-platform development is the standard, as it offers access to more powerful compiler analysis and optimization tools.

CPU Core

To simulate our target CPU, we use the ARM simulation framework available in the Gnu Debugger (GDB) software along with several modifications to accurately model the Intel StrongARM's execution and power characteristics. The simulation package in GDB was originally contributed by the ARM corporation, and consists

of a simulator (ARMulator) that emulates the complete ARMv4 ISA. We further tailored the simulation engine using execution latency specifications from the Intel StrongARM Technical Manual[73] and instrumented the simulator with additional custom-built profiling modules. To accurately model power consumption during program execution, we incorporated the research presented in [125], a paper describing power estimation research conducted using the Intel StrongARM processor. By drawing from these different resources, our research hardware model for the Intel StrongARM processor, operating at 1.5V and with a clock rate of 206MHz, produces highly accurate execution information for the benchmark results presented in the next chapter.

Scratch-Pad Memory (SRAM) In addition to the cpu core, we have modeled several types of memory common to embedded systems and have primarily focused on Scratch-Pad Memory. Scratch-Pad Memory is usually present on embedded platforms as a small user-controlled SRAM memory module that is built directly into the processing chip for low-latency and low-power access by the CPU. Our modified simulator is capable of mapping different simulated memory modules such as SPM to sections of its addressable memory space. When simulating memory instructions, our execution simulator model includes the cost of the CPU access to internal memory by the processor pipeline, while the power and latency costs for activity inside the SRAM memory module is calculated using a modified version of the Cacti hardware estimation tool. We have adopted a similar approach as that presented in [17],[131] and [83], in which we simplify the CACTI estimation model for a single-level, direct-mapped cache to match a ScratchPad memory module.

Our scratchpad memory model is calculated using CACTI [147] for various common sizes using a 0.5u technology size at 1.5V to match processor specifications. CACTI is a popular transistor-level estimation tool for parametrized cache designs that estimates the circuitry required for all cache components and their time and power requirements. SPM is modeled in CACTI as an SRAM memory array with its accompanying sense amplifiers, column multiplexers, output driver circuitry, row and column circuitry logic and decoder module. Using circuit models, CACTI is able to generate area, power and latency requirements for SPM configurations of various sizes and types by removing from its estimation process the extra circuitry needed only for a complete cache system.

Cache Hierarchy In addition to standard SRAM memory modules, our simulator includes support for cache memory hardware in both its power and execution models. While primarily the domain of desktops, caches have become more and more common as embedded processors have increased in complexity. Designed to take advantage of dynamic memory locality, they are useful for many applications although at a higher energy and area cost than SPM. The cache behavior for our simulator is modeled during instruction execution by directing all cache memory accesses to the Dinero IV[137] cache simulation software. Dinero IV is a popular academic highly functional and highly configurable cache access simulator integrated into our simulator memory hierarchy to provide latency information for cache memory traffic. For our estimates on power, area and latency characteristics for a given cache configuration, we have integrated the original CACTI [147] transistor-level cache simulator. In general, we model a direct-mapped single-level data cache for

our benchmark comparisons with configuration details available in the relevant results discussion sections. Similar to our SPM model, the cache memory used for this research is based on 0.5u technology size operating at 1.5V to match our ARM processor statistics.

DRAM On-chip memory like SRAM is fast and consumes less power than cache or DRAM memory designs, but it also consumes a large amount of silicon area and is much more expensive to design than the ubiquitous DRAM memory module. The most basic embedded processors usually include a small amount of fast on-chip memory or registers for common operations, hence the term Scratch-Pad Memory, along with instruction memory to hold program code. Middle and High-level processors generally include a much larger amount of off-chip memory in the form of ROM, Flash, SRAM or most commonly DRAM for application data exceeding internal register and SPM storage. DRAM is cost-effective, standardized and available in very large sizes, making it the main memory of choice for most embedded platforms requiring balanced read and write access to data. For our experiments the simulated ARM CPU accesses external DRAM memory through its memory mapped controller using various software control methods to perform read and write operations for benchmark data. We have incorporated the DRAM power estimation model provided by MICRON [78] for their external DDR Synchronous DRAM chip [99]. Using this model inside our simulator allows us to calculate power consumption characteristics for DRAM chips of different sizes. Our simulation model also accounts for aggressive energy and latency saving techniques commonly employed in embedded platforms in the course of simulating memory transactions to DRAM for

benchmark applications.

Other Memory Types Besides various types of SRAM and DRAM memory, embedded systems generally contain other types of memory such as Flash and ROM memory. Many platforms contain some form of ROM memory to serve as non-volatile storage and contain firmware and other system level software needed for regular platform operation. Processors performing identical tasks during their lifetimes typically also store program code in ROM as long as it is always read and never written during execution. As embedded processors have become more powerful, they generally have looked to newer memory technologies such as Flash to provide more flexible alternatives, which is really a re-programmable version of ROM memory known as EEPROM(Electrically Erasable Programmable Read Only Memory). We also note that their power consumption was comparable to SRAM for reads but was much more costly than DRAM for writes, making it a good choice for data that changes very little during execution, such as instruction code or data constants used frequently for applications, but a poor choice for compiler-controlled dynamic data allocation techniques. As we have looked at its application for code storage in other work, we have not considered it as a practical choice for meaningful benchmark results containing heap data.

8.5 Benchmark Overview

While drawn from a large number of sources, the benchmarks used for evaluation of our compiler methods were chosen because they all perform significant

dynamic data allocation, require only standard C-language libraries and resemble programs typically executed on embedded systems. Each benchmark included in our suite performs at least some amount of heap memory allocation for the inputs profiled. In general, many of the programs chosen for study are memory intensive; the runtime instruction mix is made up of 1-30% memory instructions for all applications. All benchmarks were compiled using full optimization levels with the Redhat Newlib C library [52] using the GCC ARM compiler described above. Wherever possible, we have chosen both programs and inputs typical of what may usually execute on an embedded platform, striving to keep memory occupancy limited to less than a few megabytes and a runtime that averages a few million processor cycles. For all applications we have two program input sets for generation of results, and whenever possible have chosen the two inputs whose profiles were least alike among those available.

8.6 Benchmark Classes

In order to understand how our methods can improve program performance, it is helpful to group the benchmarks into sets that exhibit certain characteristics. During the course of our research into optimizing dynamically allocated memory objects for embedded systems, we have found three main types to be considered. We have found it necessary to create extensions to our main method to handle the difficulties each type entails. Below we discuss the distinguishing traits for each type according to their dynamic data allocation characteristics.

The first type are those benchmarks in the *Known-Size Heap* set and denotes benchmarks with almost all heap objects allocated with a compile-time known size. These are the simplest of the dynamic memory types to analyze for SPM allocation purposes. With object sizes for heap allocation sites fixed at a compile-time known size, our method is able to apply our full range of heap allocation optimizations, including bin resizing. These additionally have better access guarantees since all objects allocated with the same known-size also tend to be of the same type for C applications and exist at unique code locations for individual handling. While every benchmark in our suite contains known-size allocation sites, this subset contains those compiled benchmarks for which the majority of heap accesses are to known-size sites but which may also contain unknown-size sites with a low FPB which are too large to be placed in SPM.

The second type are those in the *Unknown-Size Heap* set and refer to those applications which mainly use heap objects that come from heap allocation sites with a compile-time unknown size. This is common in applications that create wrapper functions around their heap function calls to detect errors, but makes it more difficult to allocate individual objects by fixing their location in code. Luckily, the GCC ARM compiler used in this research is capable of function inlining which we have used to very effectively eliminate simple wrappers and turning many such sites into compile-time known size sites. More complicated heap allocation interfaces (common in user memory managers) will significantly deteriorate the effectiveness of our method and we avoided benchmarks which employed a completely internal management solution(such as JPEG). We have developed extra methods to han-

dle unknown-size heap allocation sites as part of our overall allocation framework. This subset of our benchmark suite contains those applications which primarily use unknown-size allocation sites which are frequent enough and similar enough in size to other variables to be considered good candidates for SPM allocation. These benchmarks also tend to use a few very large heap objects of unknown-size which do not make good candidates for SPM placement.

The third type of benchmark is the *Recursive Stack* set and refers to those benchmarks that contain some form of heap data but also contain recursive stack objects. Recursive stack objects are the memory objects allocated on the program stack by user functions containing self-referencing function calls. Because of their dynamic nature, and compile-time unpredictability in terms of behavior or memory consumption, recursive function stack variables are usually ignored and unhandled for memory existing allocation schemes. We have developed the first methods for handling of recursive variables after finding that a large proportion of heap-using benchmarks also relied on recursive functions. This subset from our benchmark suite consists of those benchmarks which contain recursive functions forming part of the core program and which are executed at runtime. A few benchmarks in other subsets may contain one or two recursive functions, but these are usually for debugging or verification and are never executed in practice. We note that a few benchmarks may appear in both the unknown-size and recursive benchmark subsets for results if those benchmarks make significant use of both types of dynamic program data at runtime.

8.7 Benchmark Suite

To fully evaluate the performance of our dynamic heap allocation methods, we have drawn our benchmark applications from a large number of freely available software distributions. We have applied our approach to applications ranging from simple kernel programs to typical desktop software. We include programs from popular academic suites such as PTRdist, MediaBench [89], Olden [32], MIBench [58] and from other public-domain benchmark suites such as DhryBench [146], MallocBench and FreeBench, SciMark2, McCat [63], PROLANGS and the benchmark suite from the LLVM compiler distribution [88]. We include those benchmarks from each of these suites which uses heap memory, compiles using only standard libraries and which would be present on an embedded system. For benchmark suites aimed at higher end computer platforms such as MIBench, we were unable to choose most of the heap-using applications since they were either desktop oriented or used unsupported library calls. Also included are a few other applications such as the popular desktop compression program Bzip, a Huffman encoding application and the Sorting benchmark, which performs common sorting algorithms on dynamically allocated arrays. All of the programs used for this research are freely available as C source files and almost all are in their original untouched form. A few programs were slightly changed to remove unsupported system or library calls not necessary for the core algorithm, had their I/O redirected to files instead of interactive terminal streams or have had minor changes to remove unnecessary verification and debug code if it was problematic. Figures 8.3-8.5 show details for all individual benchmarks including

information on their allocation behavior and other notable characteristics.

There are several characteristics of interest when looking at each of the benchmarks we have collected and optimized. Of primary importance is the percentage of memory accesses made to heap objects for a program, which will impact how much benefit we may receive with proper allocation. Figures 8.3-8.5 show the most relevant statistics from the benchmark applications evaluated in this thesis. Also of interest are the dynamic memory characteristics of each heap-using program as well as how much total memory a program occupies. Programming styles vary widely, and the benchmarks included in our research are almost entirely untouched so that we may evaluate our methods against a wide range of programming styles. We apply our techniques wherever possible without manually changing application behavior to better suit our allocation strategy.

Benchmark	Dhrystone	Gsm	Huffman	KS	Susan	Chomp	Dhrystone-Float	Dijkstra
Suite	DhryBench	MediaBench	Misc	PtrDist	MiBench	ProLangs Suite	LLVM Suite	MiBench
Class	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size
Total Data Size	10884	17728	3296	29324	378110	2596	21496	45060
Global and Stack Size	10788	17064	352	28820	370324	188	21392	40916
Heap Data Size	96	664	2944	1168	17774	2408	104	4144
Memory Instructions	880056	19361186	244029	146655	5227928	9927	730058	5725441
Non-memory instructions	1250075	114472920	332942	383589	22417964	34751	2400084	13338524
Heap Accesses	440013	7472622	206705	88016	4891730	6648	60014	684945
Global and Stack Accesses	440043	11888564	37324	58639	336198	3279	670044	5040496
Cycles (ALL SPM)	2130132	133834216	576971	530245	27645954	44678	3130143	19063966
Cycles (ALL DRAM)	18851216	501698656	5213522	3316709	126977760	233291	17001264	127847368
Heap Allocation Sites	2	2	4	5	4	10	2	2
Heap Object Instances	2	2	4	146	4	227	2	260
Cycles (Baseline 5%SPM)	11440410	285801079	4504577	2824498	120924402	171750	6170466	44082140
Cycles (My Method 5%SPM)	3080170	169807181	3012241	1151146	27981532	104699	5030200	36443095

Benchmark	EKS	FFT	FT	Heapsort	Lists	LluBenchmark	Matrix	Misr
Suite	McCat Suite	MiBench	PtrDist	LLVM Suite	LLVM Suite	LLVM Suite	LLVM Suite	ProLangs Suite
Class	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size	Known-Size
Total Data Size	43368	16724	5860	6468	1052	11188	416	408
Global and Stack Size	424	308	156	60	56	52	56	276
Heap Data Size	43248	16416	5704	6408	996	11136	360	132
Memory Instructions	23449790	456105	21768	86429	1110605	95749	129732	1307100
Non-memory instructions	23912454	619111	84654	188439	1252821	319304	312762	3892246
Heap Accesses	7506460	78856	14601	61152	1100400	95616	129113	578043
Global and Stack Accesses	15943330	377249	7167	25277	10205	133	619	729057
Cycles (ALL SPM)	47362244	1075218	106422	274868	2363426	415053	442494	5199346
Cycles (ALL DRAM)	492908256	9741251	520014	1917019	23464922	2234284	2907402	30034246
Heap Allocation Sites	5	6	7	1	3	3	6	1
Heap Object Instances	107	6	318	1	3	1345	18	11
Cycles (Baseline 5%SPM)	189984984	2573520	383841	1436771	23271121	2283610	2907402	29806028
Cycles (My Method 5%SPM)	169408630	2417720	374880	1436771	20478121	2057299	2194712	28551952

Figure 8.3: Benchmark Suite Information Part 1 of 3.

Benchmark	Objinst	PiFFT	Sorting	Anagram	BC	BH	Bisort	Cfrac
Suite	LLVM Suite	FreeBench	Public Domain	PtrDist	PtrDist	Olden	Olden	MallocBench
Class	Known-Size	Known-Size	Known-Size	Recursive	Recursive	Recursive	Recursive	Recursive
Total Data Size	72	29732	1344112	73306	73306	14164	2076	2042
Global and Stack Size	40	2888	104	21266	21266	3808	552	1648
Heap Data Size	32	26844	1344008	52080	52080	10356	1524	394
Memory Instructions	56093	8852440	2015467.875	145811	145811	1858190	48385	68624
Non-memory instructions	147490	9252740	6103929	521913	521913	1985257	62135	114936
Heap Accesses	56083	2633845	743068	70193	70193	126054	16271	20493
Global and Stack Accesses	10	6218595	1272400	75618	75618	1732136	32114	48131
Cycles (ALL SPM)	203583	18105180	8119397	670100	670100	3843447	110520	183582
Cycles (ALL DRAM)	1269350	186301552	46413288	3485653	3485653	39149056	1029835	1487856
Heap Allocation Sites	4	12	3	2	2	4	1	8
Heap Object Instances	4	12	3	6	6	82	127	29
Cycles (Baseline 5%SPM)	1269350	71763841	22510353	2051457	4254868	27109175	773230	615322
Cycles (My Method 5%SPM)	1269350	71754018	8119409	2041577	3318833	9250543	773230	615322

Benchmark	Epic	Health	Imp	MST	Patricia	Perimeter	Qbsort	TreeAdd
Suite	MediaBench	Olden	McCat Suite	Olden	MiBench	Olden	McCat Suite	Olden
Class	Recursive	Recursive	Recursive	Recursive	Recursive	Recursive	Recursive	Recursive
Total Data Size	726568	13512	381992	13956	36688	17580	28228	12700
Global and Stack Size	328294	532	952	256	428	864	508	424
Heap Data Size	398274	12980	381612	13700	36480	16716	27720	12276
Memory Instructions	710255	103794	13936346	61009	66049	78311	101916	45045
Non-memory instructions	2852333	149570	42485832	160751	148870	137766	346324	76774
Heap Accesses	168066	14526	9097271	19025	55519	17064	80810	1025
Global and Stack Accesses	542189	89268	4839075	41984	10530	61247	21106	44020
Cycles (ALL SPM)	3562592	253367	56422192	221760	214920	216079	448240	121819
Cycles (ALL DRAM)	17057512	2225510	321213056	1380931	1469870	1704026	2384644	977674
Heap Allocation Sites	3	3	14	5	6	1	5	1
Heap Object Instances	51	829	267	1058	2736	597	3185	1023
Cycles (Baseline 5%SPM)	9378062	625188	229270600	595604	1303753	697729	2098887	224762
Cycles (My Method 5%SPM)	8365316	518645	182380563	501269	637575	534519	1516496	140933

Figure 8.4: Benchmark Suite Information Part 2 of 3.

Benchmark	TreeSort	Trie	TSP	Voronoi	Yacr2	AllRoots	Bzip	EM3D
Suite	LLVM Suite	McCat Suite	Olden	Olden	PtrDist	ProLangs Suite	mercial Application	Olden
Class	Recursive	Recursive	Recursive	Recursive	Recursive	Unknown-Size	Unknown-Size	Unknown-Size
Total Data Size	87084	30380	3196	15208	44030	652	1077462	35104
Global and Stack Size	75084	4176	928	1708	892	264	80332	272
Heap Data Size	12000	26294	2268	13500	43498	408	997130	34832
Memory Instructions	60846	433905	42365	42152	9906423	24962	1294819	104633
Non-memory instructions	133316	803043	66172	37048	25121946	70547	2066947.875	241600
Heap Accesses	19000	81860	20698	10895	5930953	5004	327825	44236
Global and Stack Accesses	41846	352045	21667	31257	3975470	19958	966994	60397
Cycles (ALL SPM)	194162	1237972	108541	79230	35028368	95789	3361767	347002
Cycles (ALL DRAM)	1350236	9501623	913552	880688	223250432	575387	27963328	2349640
Heap Allocation Sites	5	8	1	4	26	2	5	15
Heap Object Instances	1000	1318	63	119	60	8	7	1030
Cycles (Baseline 5%SPM)	701972	8286630	814849	803170	155184341	378961	9592000	1202097
Cycles (My Method 5%SPM)	386860	3543926	796606	709680	80513743	378961	9479026	1079832

Benchmark	Mpeg2Decoder	Mpeg2Encoder	SciMark2-C	Sgefa	Spiff
Suite	MediaBench	MediaBench	SciMark2 Suite	LLVM Suite	LLVM Suite
Class	Unknown-Size	Unknown-Size	Unknown-Size	Unknown-Size	Unknown-Size
Total Data Size	13364	913468	250152	15100	816954
Global and Stack Size	11188	4092	436	4232	806280
Heap Data Size	2176	909376	249716	10872	10674
Memory Instructions	1581194	7949331	7692309	1114431	24635
Non-memory instructions	2781366	36662352	11324136	3302422	57678
Heap Accesses	297088	2963228	5239318	1014114	3810
Global and Stack Accesses	1284106	4986103	2452991	100317	20825
Cycles (ALL SPM)	4362567	44630864	19016446	4416857	83056
Cycles (ALL DRAM)	34405388	196032576	165170320	25591122	565238
Heap Allocation Sites	4	11	11	5	1
Heap Object Instances	10	25	211	77	385
Cycles (Baseline 5%SPM)	20310996	101296615	153849468	23957654	191565
Cycles (My Method 5%SPM)	19547310	97914603	124965953	23055154	186796

Figure 8.5: Benchmark Suite Information Part 3 of 3.

Chapter 9

Results

This chapter presents results by comparing our method for dynamically allocated data against the usual practice of placing such data in DRAM, for a variety of compiler and architecture configurations. For comparison, we use the best existing compiler-directed SPM allocation scheme for global and non-recursive stack data from [132], since there exists no other automatic compiler methods to optimally allocate dynamic data using either static or dynamic memory placements. Both our method and the dynamic method for global and stack variables in [132] are implemented in the same GCC ARM compiler and simulation framework. All applications are compiled automatically using full optimization levels without requiring the user to specify anything other than the SPM space available for data allocation on the target platform. An external DRAM with 20-cycle latency and an internal SRAM (scratch-pad) with 1-cycle latency is simulated in the default configuration. The default configuration has an SRAM size which is 5% of the total data size in the program. The total data size for a program is the maximum memory occupancy during the course of its execution and not simply a sum of the total data objects allocated throughout its lifetime. The DRAM size, of course, is assumed to be large enough to hold all program data. Instruction code is placed in SRAM by the compiler for all experiments to remove its influence from SPM allocation. However, for

completeness we also present a study that considers code allocation in Section 9.6.

9.1 Dynamic Heap Allocation Results

9.1.1 Runtime and energy gain

Figure 9.1 compares the normalized runtime from our method versus from the existing practice of placing all heap data in DRAM. For each benchmark the SRAM size is the same in both configurations – 5% of the combined global, stack and heap data size in that program. Without our method, this SRAM is used only by global and stack data; with our method the SRAM is shared by global, stack and heap data. In both cases, global and stack data is allocated by the best existing method for global and stack data, which is the one in [132]. *The figure shows that the average runtime reduces by 22.3% by using our method for the exact same architecture.* The large improvements show the potential of our method to reduce runtime of the application beyond the state-of-the-art today.

Why do we do well? Before looking at additional experiments, it is insightful to look at why an improvement of 22.3% can be obtained with an SRAM of only 5% of the total data size of the program. We identify three reasons. **First**, it is well-known that a small fraction of frequently used data usually accounts for a large fraction of the accesses in the program. This is often referred to informally as the *ninety-ten rule* [64]: on average 10% of the data accounts for 90% of the accesses. Consequently, we find that our method is able to place the most frequently used heap variables, either fully or in large part, in SRAM – this is verified later in

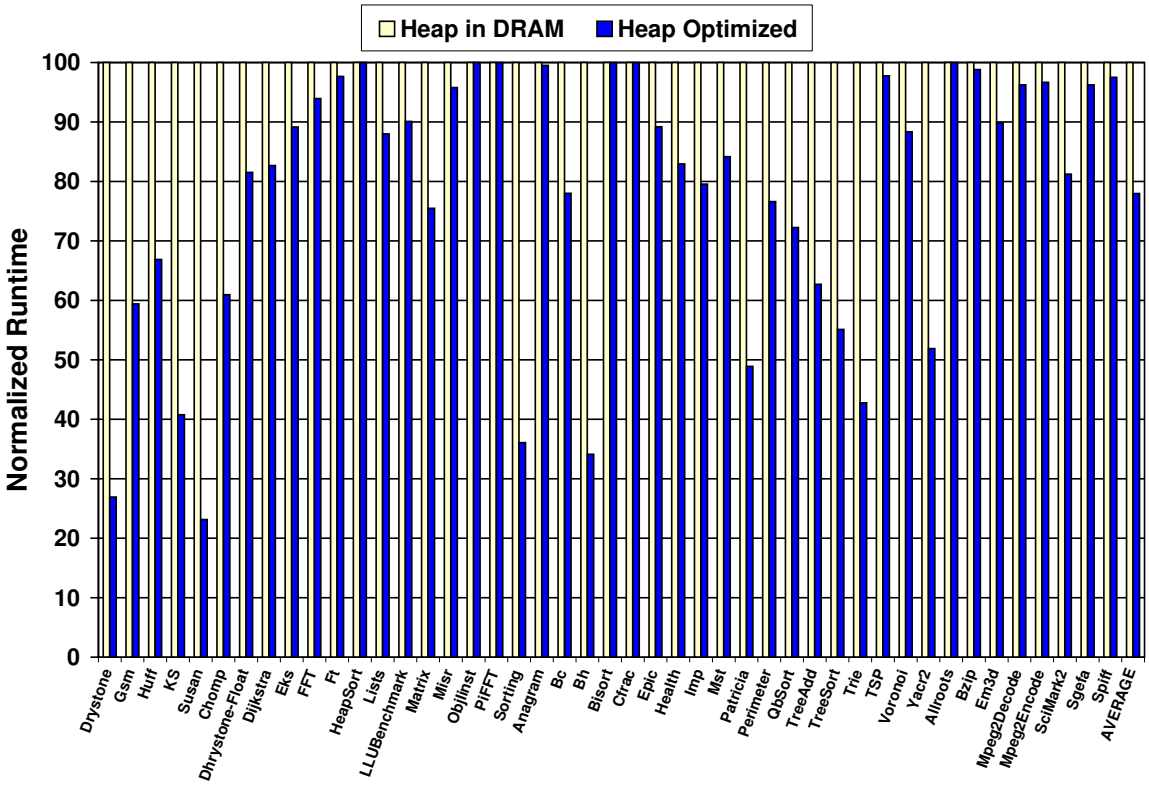


Figure 9.1: Runtime gain from using our method vs. allocating heap data in DRAM.

figure 9.5. Without our method, they all go to DRAM, which is much slower. A **second** reason for the significant improvement is that our earlier global and stack method, and our current heap method, are both *dynamic*. Thus even though the SRAM is 5% of the total data size, by sharing this space across different frequently used data variables in different regions, *it is possible to place more than 5% of the frequently used data in SRAM*. Note that a scratch-pad improves performance for the same reasons as a cache. Just as even a small cache can significantly improve run-time [64], it is not surprising that a small scratch-pad can too. **Third**, many of the benchmarks selected have a significant fraction of their accesses going to heap, and thus our method to optimize for heaps does well. For other benchmarks where the fraction of heap accesses is zero or small, heap allocation is, of course, not a

problem, and hence, it does not need SRAM placement.

Here we look at two examples from our benchmarks which illustrate why some heap data is accessed frequently. First, *Susan* is a typical image-processing application which stores the image in a large stack array. It performs iterative smoothing on small chunks of the image at a time; the chunks and associated look-up tables needed for smoothing are stored on the stack and on the heap. Because smoothing accesses each pixel many times, the most-frequently used chunk data is allocated to SRAM, but the infrequently used large image array and less-frequently used chunk data are placed in DRAM. Second, *Huffman* performs Huffman encoding, meant for data compression. Here most of the program data is on the heap and there is little other data. There are four heap variables of total sizes 60 bytes, 260 bytes, 14Kb, and 4Kb. The first is used to store the encoder structure; the second holds the coded bits of the character currently being encoded; the third stores the alphabet used; and the fourth holds the current block array used in encoding. The last two are somewhat frequently used with frequency-per-byte of about 150 but only a small portion of them fit in SRAM. The first two, however, are very highly used with frequency-per-bytes of about 40000 and 1000, respectively, and our method is able to place them in SRAM.

Energy gain Figure 9.2 compares the energy consumption of application programs with our method for heap data versus placing heap data in DRAM. The figure shows that *we measure an average reduction of 26.7% in energy consumption* for our applications by using our method vs. placing heap data in DRAM. This result demonstrates that our approach has the potential to not only significantly

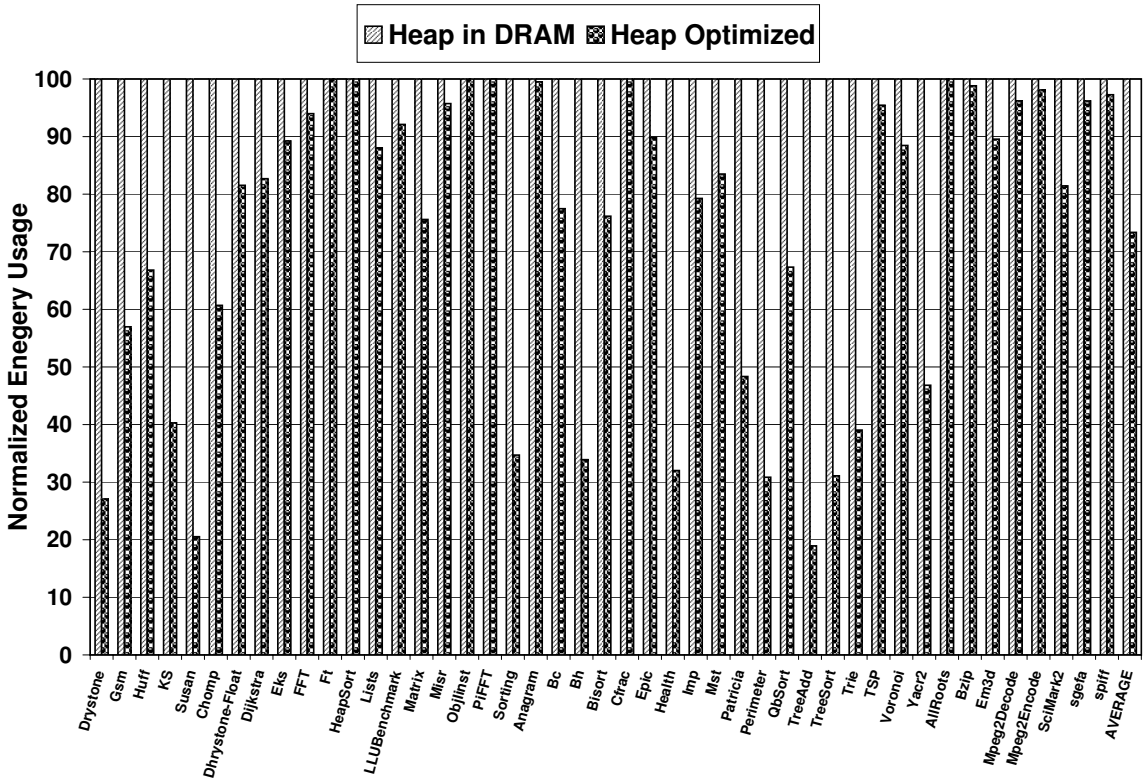


Figure 9.2: Improvement in energy consumption.

improve runtime, but also energy consumption. We see that the improvement in execution costs is reflected in the reduced power consumption for optimized applications, a correlation commonly seen by researchers working in compiler optimization. While our method primarily seeks to reduce runtime, this also generally leads to a proportionate reduction in the energy consumption of the system for most applications.

Comparing Figures 8.3- 8.5 against 9.1, we see that our method does not always show more benefit for applications with a larger ratio of heap data among all program data. In general, our method will tend to improve the performance of an application in proportion to the percentage of the program data is heap data. If heap data makes up a larger percentage of the program footprint, there is a much

higher chance that heap objects will exist that are important and frequently used at runtime. Looking at heapsort, we see that the 98% of its memory footprint is due to the single heap allocation site in the program that allocates a single large array, that is also input dependent and of unknown-size. A single heap object makes up almost the entire allocation requirements for this program in the form of an array. Without having an SPM large enough to hold 98% of the memory footprint, this site will never be optimized and will never be placed to SPM. These types of scenarios are also common in streaming media applications, such as Mpeg2Dec and MPeg2Enc, where our method is unable to place their larger, unknown-size heap allocation sites into SPM due to lack of available space. Further, even if enough space in SPM were available to consider storing such large heap objects, it is unlikely that their access frequency will be high enough in comparison to other smaller objects to ensure that placing the larger objects in SPM will be profitable.

9.1.2 Transfer Method Comparison

The results in figure 9.1 and the rest of the paper use DMA for the memory transfers in our method. In our preliminary research, we measured that the gain from our method as compared to the baseline method from [132] only reduced slightly by a few percent with all-software transfers, although this was from observing only five benchmark applications. After refining our own methods, as well as greatly expanding our suite of applications, we now see the opposite behavior when comparing transfer methods. Figure 9.3 shows the changes in normalized runtime

gain when comparing our method to the baseline method from [132] for our entire benchmark set. We can see that on average, our benchmarks show an increase in the normalized runtime gain as we used less efficient software methods. When both allocation methods use software transfers, there is an average runtime gain of 24.12%. Pseudo-DMA transfers show an average of 22.99% gain and the default DMA transfer method gives an average of 22.03% gain. Regardless of the direction of change, both our method and the baseline for global and stack [132] rely on memory transfers; a change in the transfer mechanism affects both and changes their ratio only slightly.

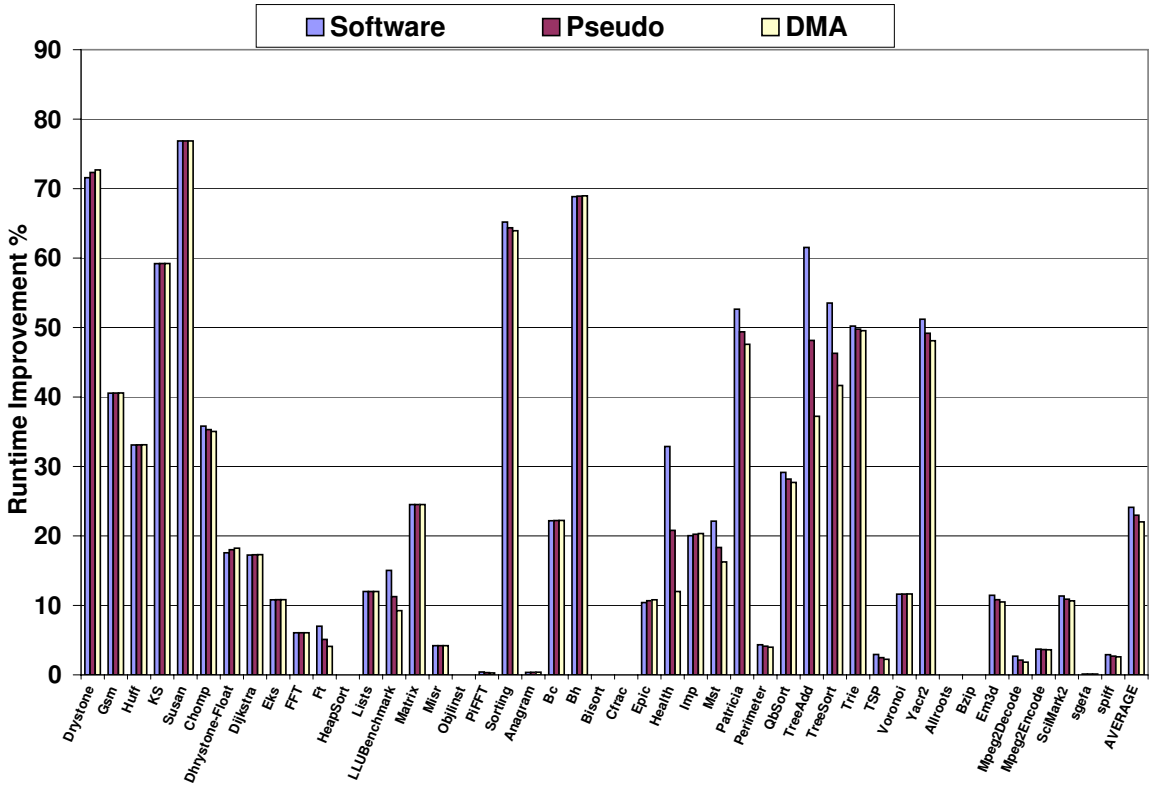


Figure 9.3: Change in runtime when comparing our method against the baseline using different transfer methods

To implement a dynamic allocation scheme, both our method and the method in [132] rely on available processor methods for copying data to and from different memory locations. Regardless of the mechanism to perform the transfers, the transfer themselves are important to both methods, as well as any other dynamic allocation scheme, such as those implemented by cache memories. When comparing two dynamic methods, it is important to keep in mind that both will have the same total amount of SPM space available. The two techniques can tradeoff their allocation and transfer decisions, generally causing either more or less transfers when we allocate dynamic data objects in addition to static data. Sometimes transfers increase when stack and global variables of importance must be in SPM for some regions, but swapped more heavily to place heap variables, or vice versa. Sometimes transfers decrease because the placement of heap variables precludes the placement of stack and global variables which were incurring transfers. This unpredictability is typical of memory allocation problems, requiring sophisticated methods, analysis and program profile information for best prediction of performance at runtime.

Looking at the results for the Health application, we see that the use of better transfer methods decreases the normalized gain from our method compared to the baseline method. This is because the transfers are used much more often for stack and global variables in the baseline method, than in our allocation with the heap variables in the improved method. Sometimes we can improve performance by allocating heap objects to SPM instead of attempting to dynamically allocate more stack and global variables, which in turn incur more transfers at runtime. This causes the ratio between the two methods to change as the transfer costs shift for

each individual transfer method, affecting the runtime improvement ratios between the two methods. Let us assume that the stack and global allocation had a large number of transfers, and those are sped up using DMA, but the heap allocation had many less transfers. The ratio comparing the two methods would thus decrease as the stack and global allocation costs are decreased at a greater rate than the heap results. Usually what we see is that when we allocate heap data, it consumes SPM space which would have been given to less profitable stack and global variables fitted in using transfers. We generally do not transfer heap variables in and out of SPM as often, due to safety constraints on possible pointer accesses and the high cost of transferring entire bins. Stack and global objects tend to be smaller and much safer to dynamically transfer at runtime with safety guarantees.

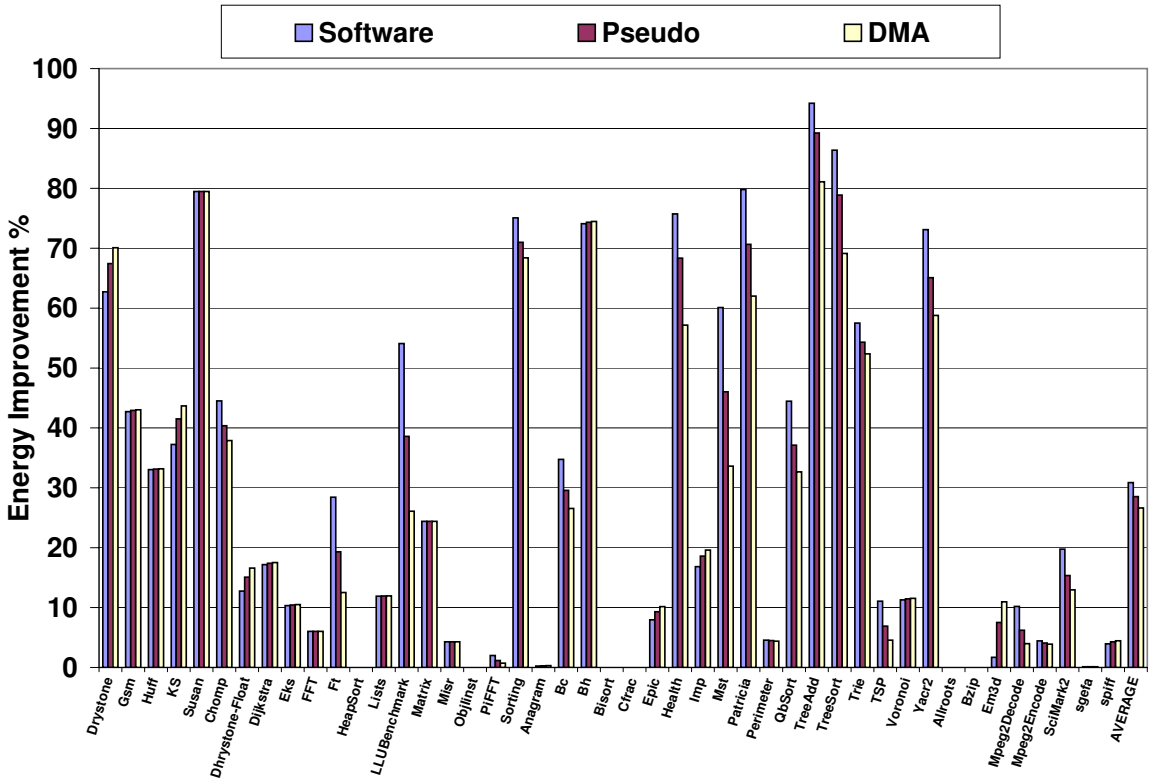


Figure 9.4: Improvement in power consumption comparing our method against the baseline using varied transfer methods

9.1.3 Reduction in Heap DRAM Accesses

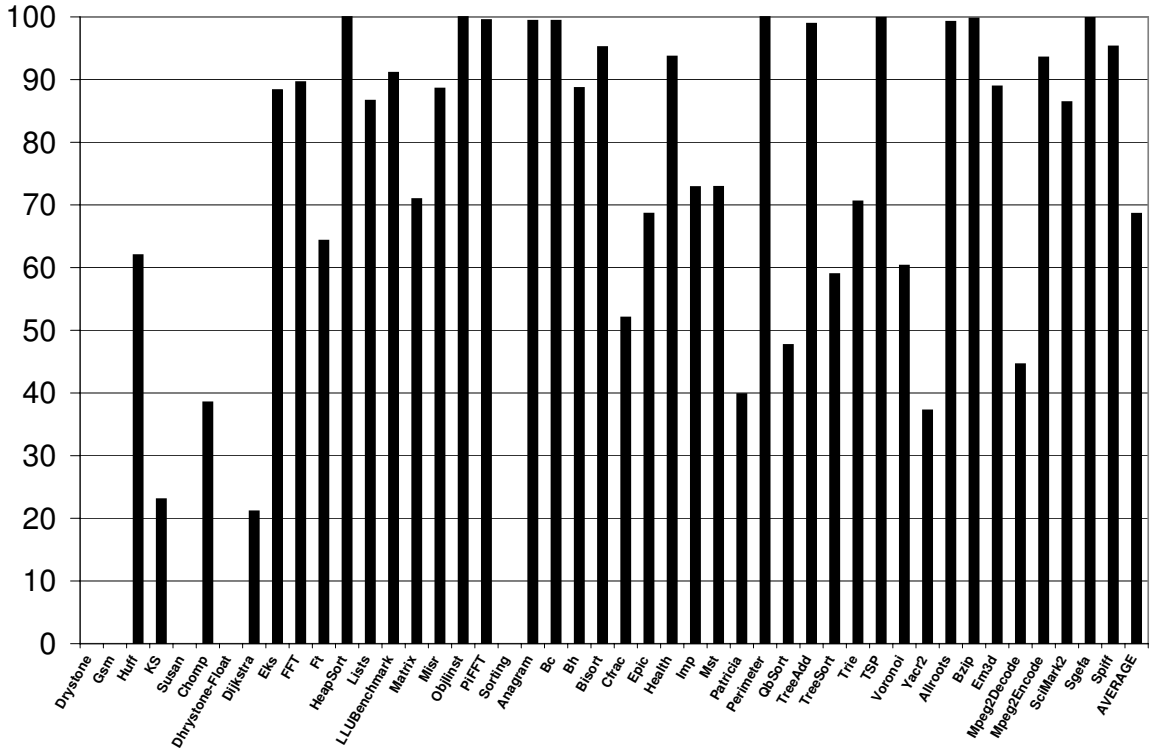


Figure 9.5: Percentage of heap memory accesses going to DRAM for each benchmark.

Figure 9.5 shows the net reduction in percentage of memory accesses to heap data going to DRAM because of the improved locality to SRAM afforded by our method. The number of DRAM accesses is increased by the transfer code but is reduced much more by the increased locality afforded by the SRAM bins. Considering both effects, the average net reduction across benchmarks is a very significant 68.6% reduction in heap DRAM accesses. Analyzing the results shows that our method was able to place many important heap variables into SRAM without involving transfers, explaining the high reduction in DRAM accesses for heap data and showing the benefit of the our allocation methods. This was correlated with a small increase in transfers for less important stack and global variables, which were

evicted to make room for the more frequently accessed heap variables allocated. In this figure, any applications without a bar indicates that our method was able to allocate all accessed heap variables into SPM of 5% data size.

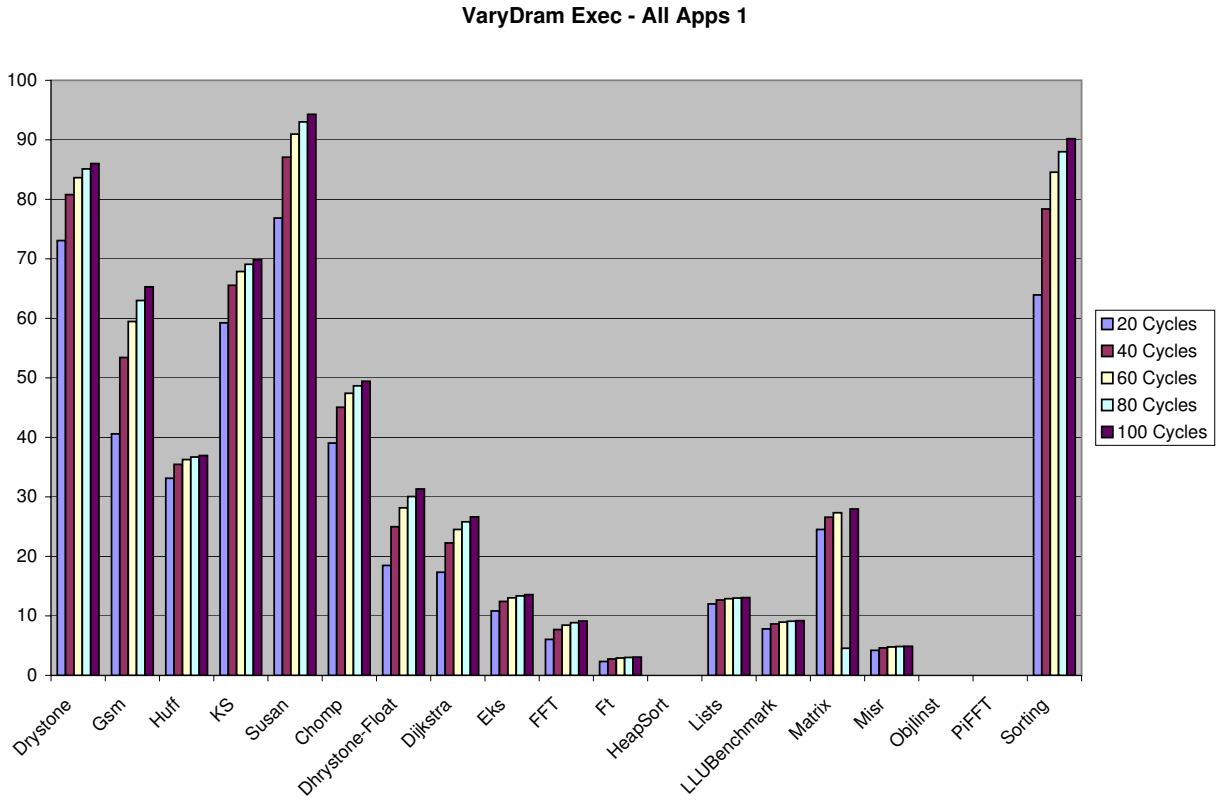


Figure 9.6: Effect of varying DRAM latency on runtime gain from our method (Part 1).

Figures 9.6 and 9.7 show the effect of increasing DRAM latency on the average runtime gain from our method for the entire benchmark suite. Since our method reduces the number of DRAM accesses, the gain from our method is greater with higher DRAM latencies. This is especially true when we are able to place most heap data in SRAM, and the larger latency affects mostly the stack and global DRAM data. The figure shows that the average runtime gain from our method versus heap allocation in DRAM increases from 22.3% with a 20-cycle DRAM latency to 27.8% with a 100-cycle DRAM latency. We can compare this figure against Figure 9.5 to

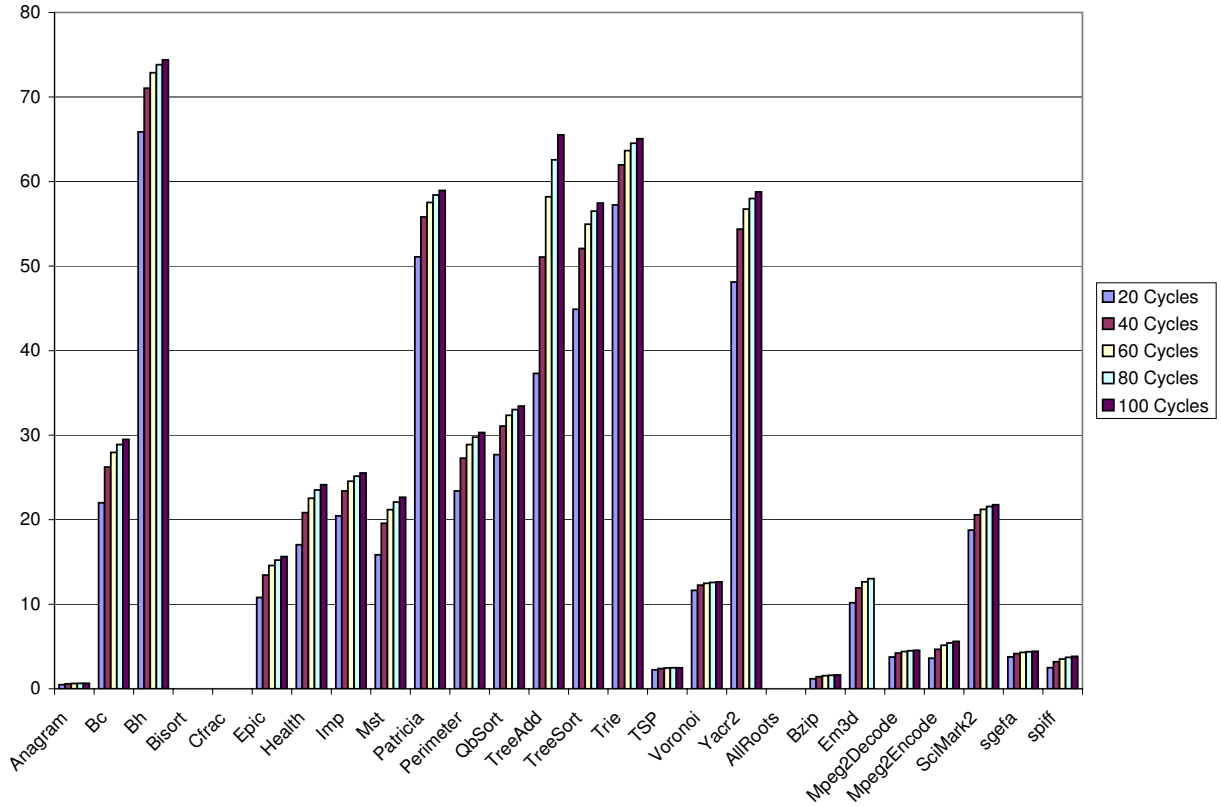


Figure 9.7: Effect of varying DRAM latency on runtime gain from our method (Part 2).

see how the increased DRAM latencies only make a difference for those applications where a significant number of heap accesses went to DRAM after optimization.

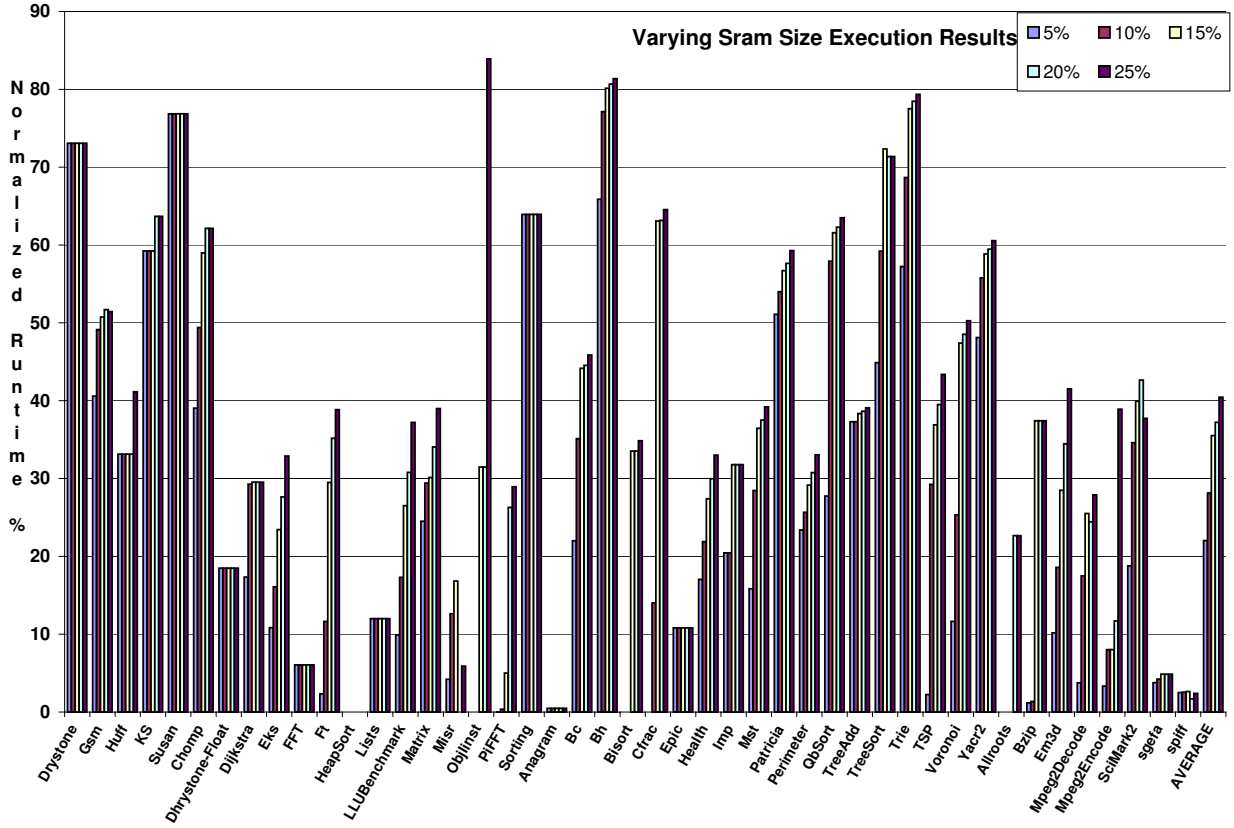


Figure 9.8: Effect of varying SRAM size on runtime gain from our method, where SRAM size is expressed as percentage of total data size for application.

9.1.4 Effect of varying SPM size

Figure 9.8 shows the effect of increasing SRAM size on the percentage gain in runtime from our method. The SRAM size is expressed as the percentage of the total data size for the application. The runtime gain from our method varies from 22.3% to 40.5%, when the scratch-pad size percentage is varied from 5% to 25%. From this we see that increasing the SRAM space beyond 5% gives only a relatively small additional benefit on average. This is because of only a small fraction of the

program data is frequently used. A similar effect is seen for caches: a very large cache does not yield much better performance than a moderately sized cache [64].

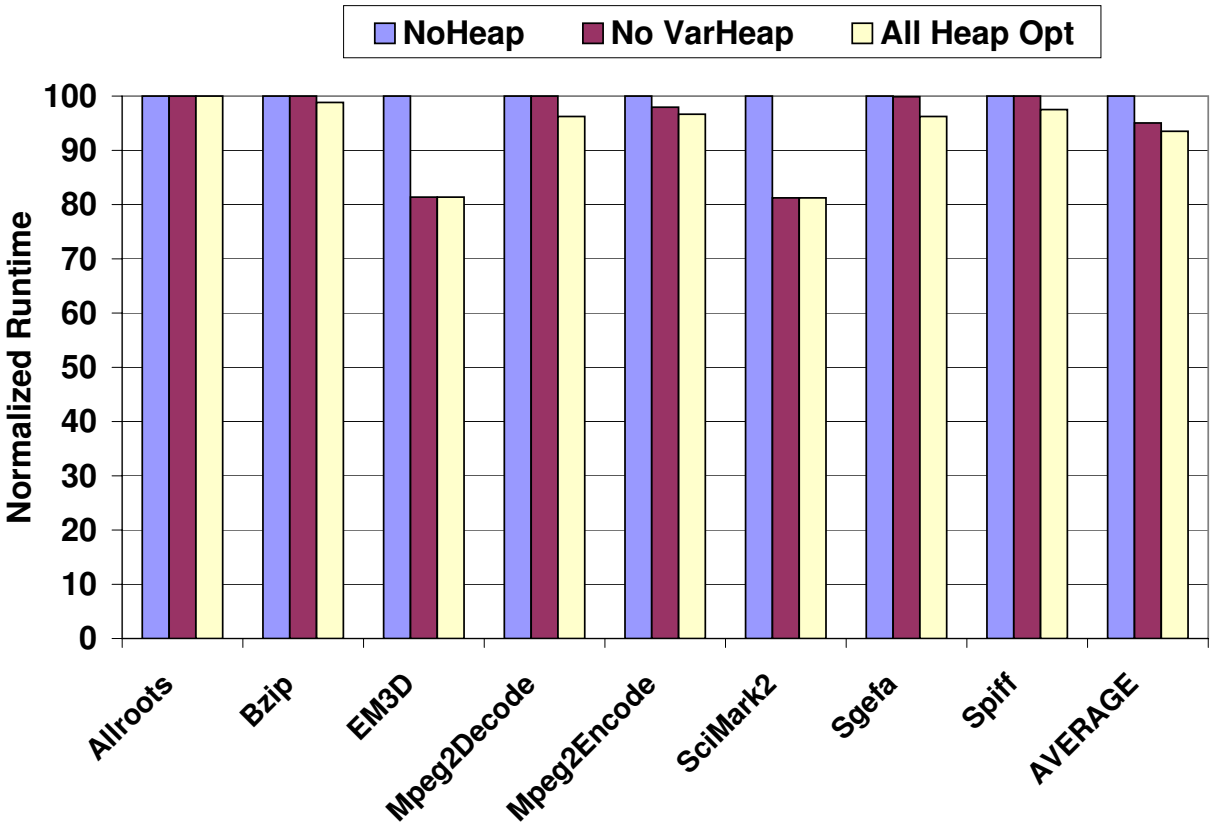


Figure 9.9: Normalized runtime for benchmarks with unknown-size heap allocation at 5% SRAM.

9.2 Unknown-size Heap Allocation

This section compares the performance of our method with and without support for heap allocation sites of a compile-time unknown size. These types of sites are much more difficult to optimize since there is no consistent unit of allocation for all heap objects created at that site. Figure 9.9 shows the normalized runtime for applications with unknown-size heap allocation sites using the default SPM size of 5%. The first bar is for the case where only stack and global data is allocated

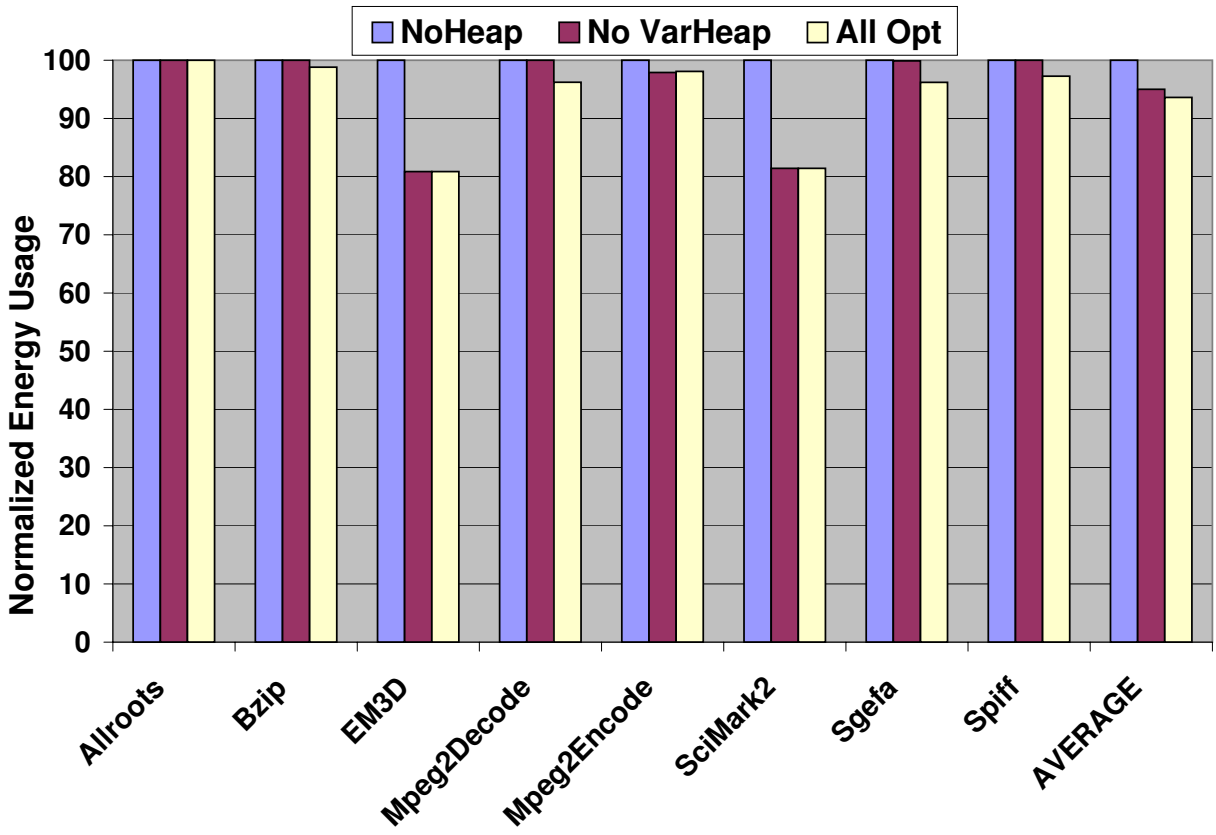


Figure 9.10: Normalized energy usage for benchmarks with unknown-size heap allocation at 5% SRAM.

to SPM, to which the other two bars are normalized for comparison. The second bar depicts the situation where we leave unknown-size heap objects in DRAM but optimized all other data types to SPM. The third bar shows the result when we allocate all supported data types to SPM. The difference between the first bar and second bar shows the improvement obtained from allocating known-size heap objects to SPM in addition to stack and global, while the difference between the second and third bars shows the separate improvement due to the addition of unknown-size heap objects. From Figure 9.9, we see that on average a 1.5% decrease in runtime is obtained when we allocate unknown-size heap objects in addition to known-size heap objects to SPM. Figure 9.10 shows the normalized enery usage for the same experiment, also averaging a 1.5% decrease in energy consumption when allocating unknown-size heap objects.

Unlike known-size heap allocation sites, our method is currently unable to apply the heap-resizing pass to unknown-size sites, since there is no unique allocation size for all objects at that site. Also, with different heap objects being created at the same site, our method has a much lower guarantee that the individual objects placed in SPM at runtime were the ones most beneficial for allocation using profile information. Figure 9.11 expands on this experiment by varying the SRAM size from 5% to 25% to reduce allocation pressure from the more predictable known-size heap allocation sites. From this figure, we see that the runtime improvement from handling unknown-size allocation sites rises from 1.5% at 5% SRAM, to 5.3% at 25% SRAM. Figure 9.12 shows the normalized energy usage obtained for the same experiment, with the energy consumption reduced by 1.5% at 5% SRAM to 7.8% at

25% SRAM. These results show that our unknown-size heap allocation extension is useful for handling these types of allocation sites although much more complicated methods are necessary to fully exploit these allocation patterns as we do for recursive stack and known-size heap objects.

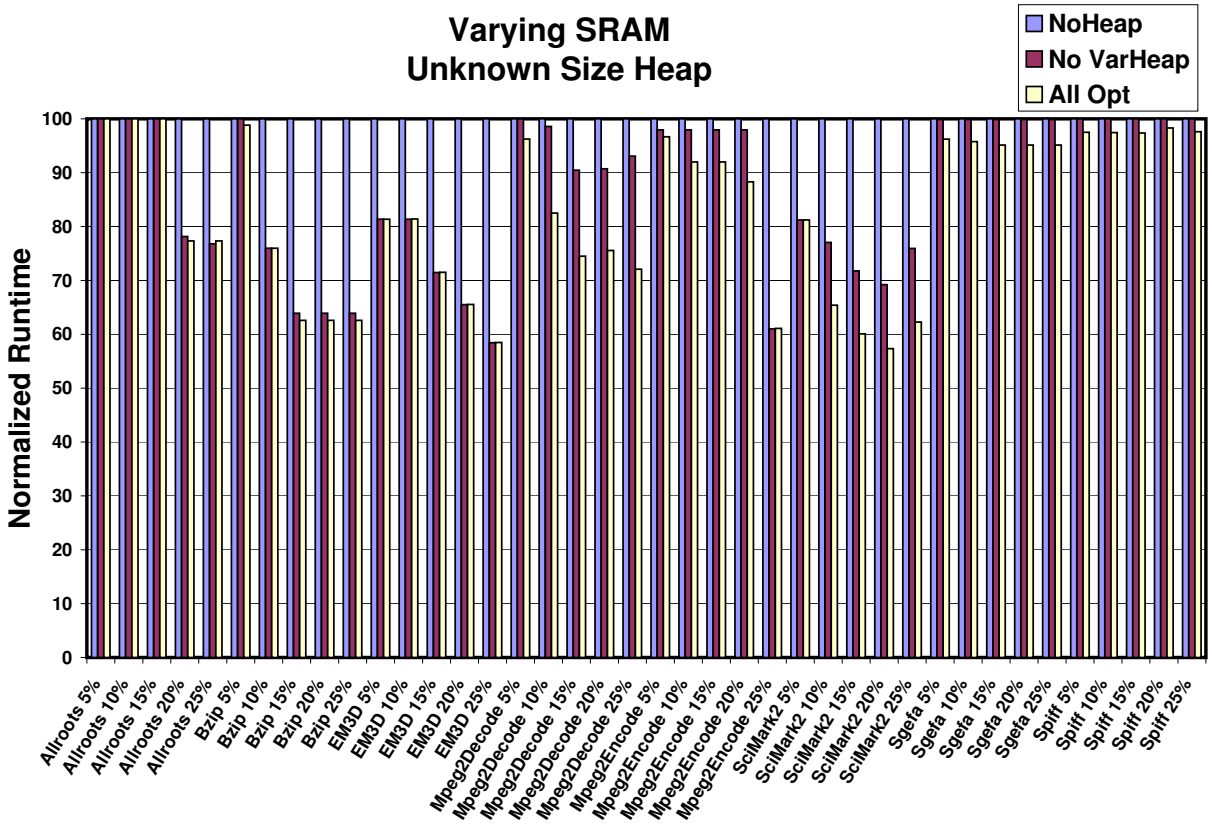


Figure 9.11: Normalized runtime for benchmarks with unknown-size heap allocation for varying SRAM sizes.

9.3 Recursive Function Allocation

In this section, we examine the effectiveness of our method for dynamically allocating the stack data from recursive functions to SPM. With no other published research related to allocation of recursive functions to SPM, our methods for stack

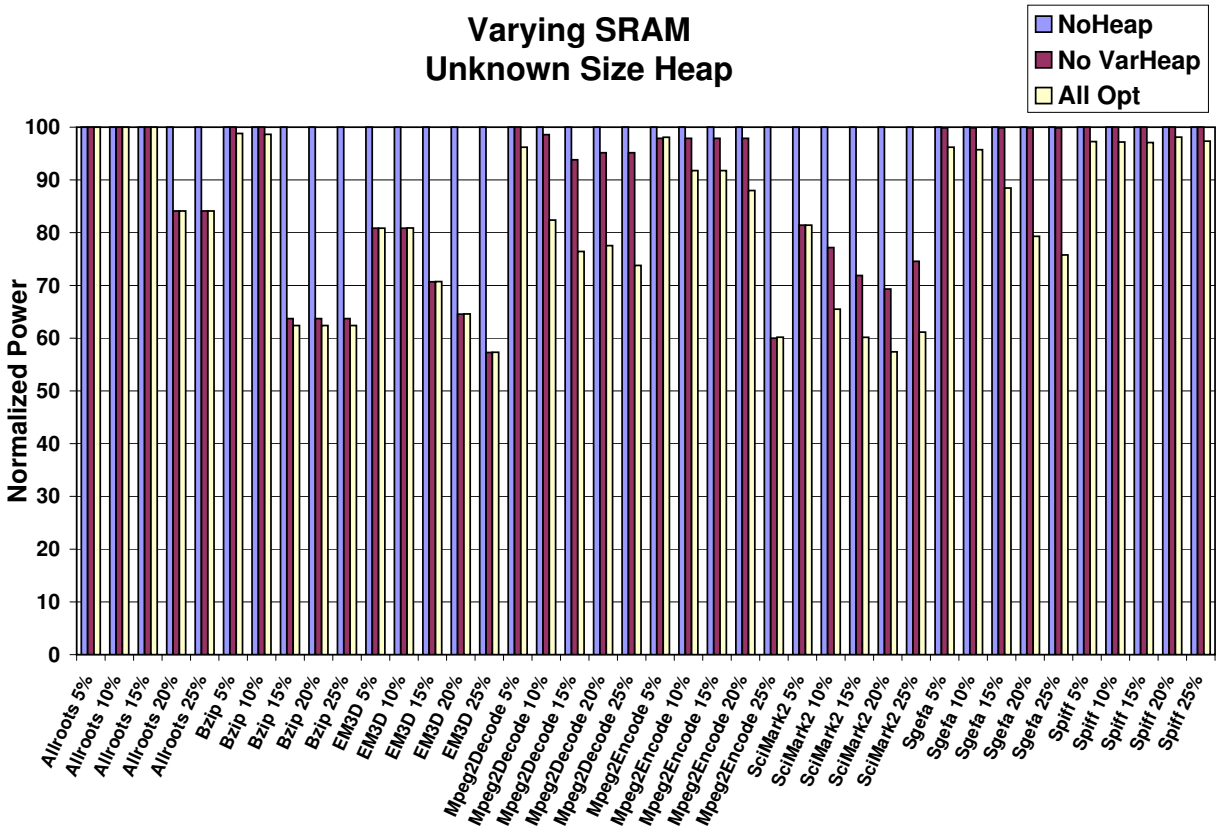


Figure 9.12: Normalized energy usage for benchmarks with unknown-size heap allocation for varying SRAM sizes.

data are as novel as our methods for heap allocation and we compare its performance against the best known SPM allocation scheme from [132]. We have developed these methods for handling recursive stack data due to its similarity to heap data, in that its total size is generally unbounded at compile-time and it is allocated at runtime dynamically instead of at compile-time. The results presented from these experiments show the benefit achieved by allocating recursive stack data to SPM by comparing allocation scenarios where a combination of stack, global and heap data are also allocated using our methods. Figure 9.13 shows the normalized runtime for applications with recursive stack function using different allocation schemes with the default SPM size of 5%. The first bar is for the case where only global and non-recursive stack data is allocated to SPM, to which the other three bars are normalized for comparison. The second bar shows the runtime when we allocate global, recursive and non-recursive data to SPM while leaving heap data in DRAM. The third bar shows the case when we leave recursive stack data in DRAM but allocate global, non-recursive stack and heap data to SPM. The final case is where we apply our full allocation methods and allow all data types to be considered for SPM allocation.

By comparing the first and second bars for each application in Figure 9.13, we see on average a 13.8% decrease in runtime when we allocate recursive stack data as well as non-recursive stack and global data. Comparing the first and third bars reveals a 11.8% reduction on average in runtime when heap is allocated to SPM with non-recursive stack and global data. Finally, looking at the difference between the first and fourth bars reveals the total contribuion from being able to allocate

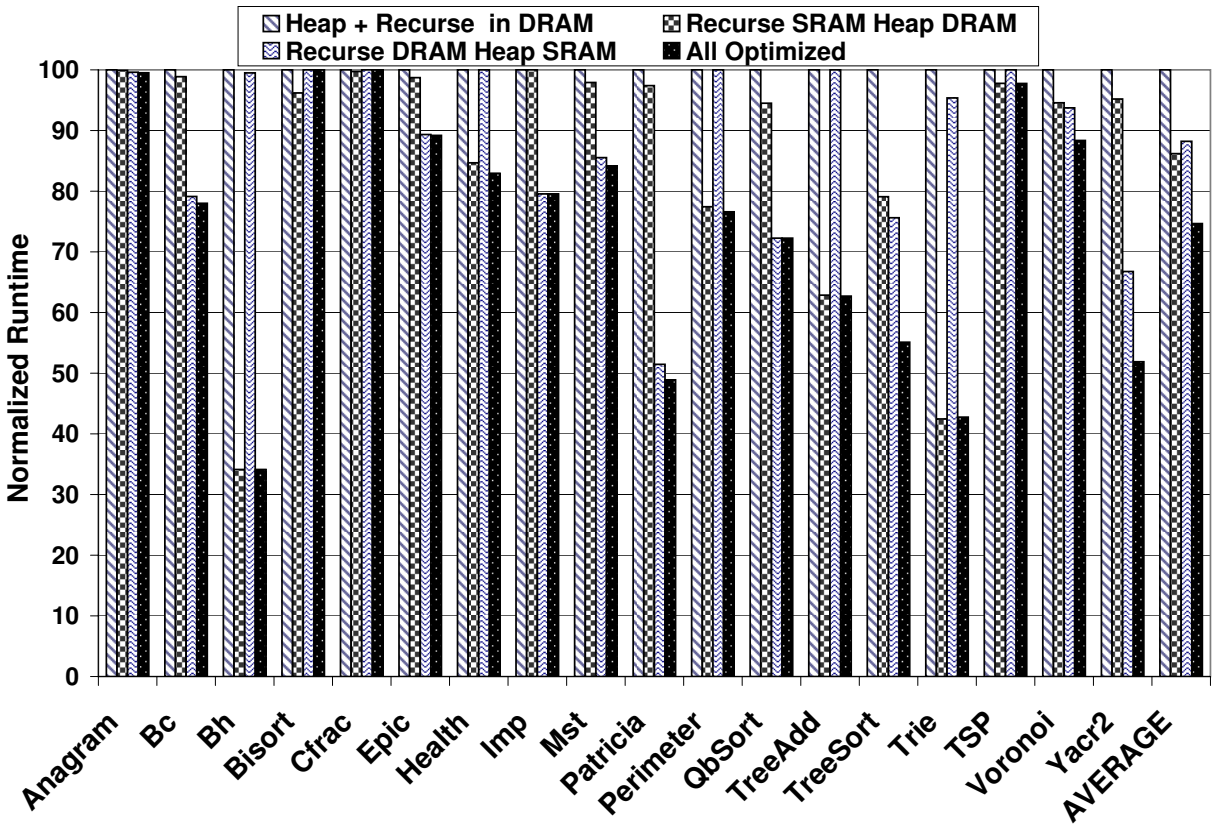


Figure 9.13: Normalized runtime for benchmarks comparing recursive function allocation.

both recursive stack and heap data to SPM in addition to non-recursive stack and global objects. On average, we are able to reduce runtime by 25.4% by applying our combined allocation methods for heap and recursive stack data. Looking at individual benchmarks shows that not all benchmarks have similar contributions from the two additional allocation methods, with some benchmarks heavily favoring either recursive stack or heap data in terms of access frequency. The overall results serve to reinforce the importance of being able to optimize the allocation of dynamically allocated objects for applications which make significant use of these methods.

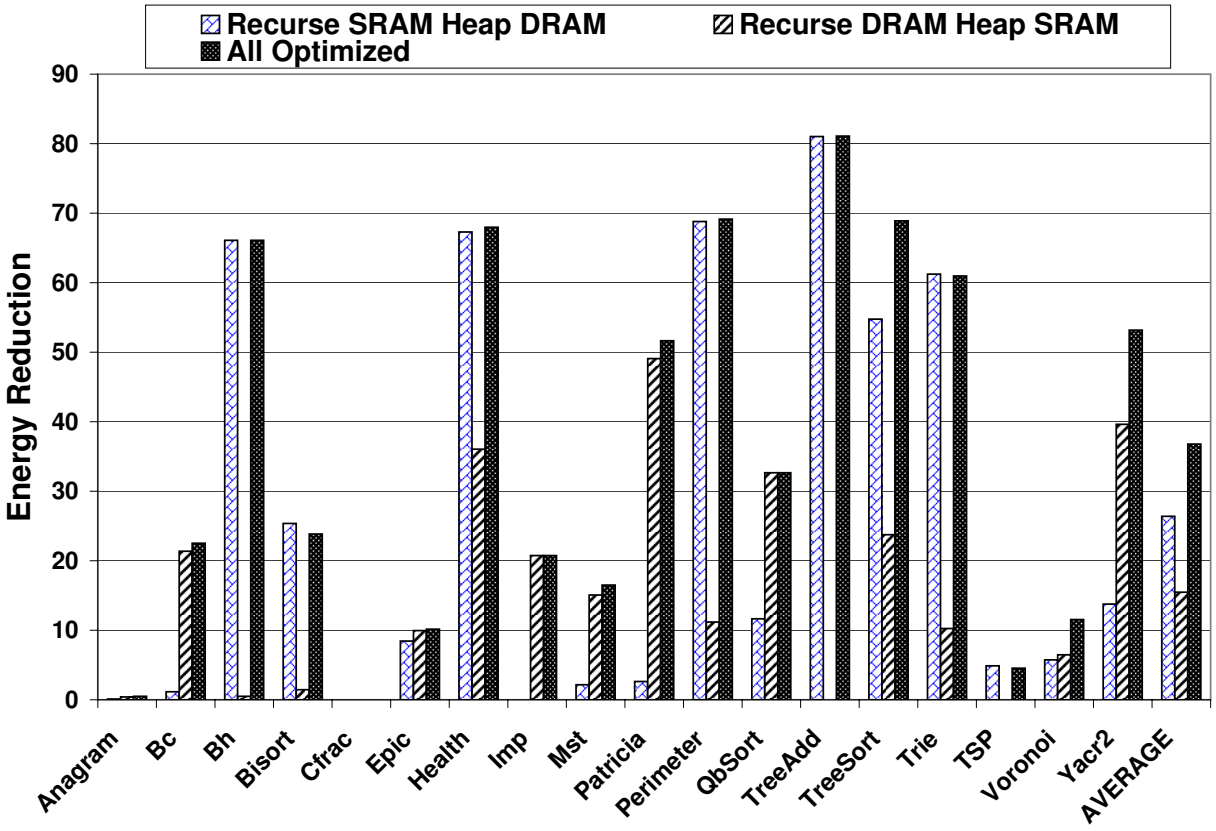


Figure 9.14: Reduction in energy usage for benchmarks comparing recursive function allocation.

Figure 9.14 shows the energy reduction obtained for the same experiment on the recursive benchmark set. On average, power consumption was reduced by

26.4% when our recursive stack allocation method was added to the baseline allocator. When we instead apply only our heap allocation method in addition to the default allocator, we see a reduction of 15.5% in energy consumption on average. Combining all allocation methods yields a total average reduction of 36.8% in energy consumption. Looking at individual applications shows that the contributions of our two additional methods are not additive and instead depend on which objects were optimally allocated to SPM in each instance. Treesort is a good example, where its energy decreases by 54.8% with recursive stack allocation and by 23.7% with heap allocation enabled. With all allocation methods applied, only a 68.9% reduction is seen, showing that SPM space is flexibly distributed among those variables it is able to best place for each scenario.

In an effort to further separate the effects from each of our proposed allocation methods for these applications, we also present the same experiment, but with an SPM size that is 25% of the total application memory occupancy. Figure 9.15 shows the runtime improvement results for this experiment. By increasing the SPM size and reducing the allocation contention among the different supported data types, this figure shows a clearer picture of the demarcation between our two additional methods for those applications that use both heap and recursive stack data. On average there is a 18.3% reduction in runtime when we allow recursive function optimization and a 25.2% reduction in runtime when we instead allow heap object optimization. When both methods are applied to this benchmark set, the average runtime is reduced by 46.7% across all benchmarks. Looking at Treesort again, we see that the reduced allocation pressure allows a clearer look at the indi-

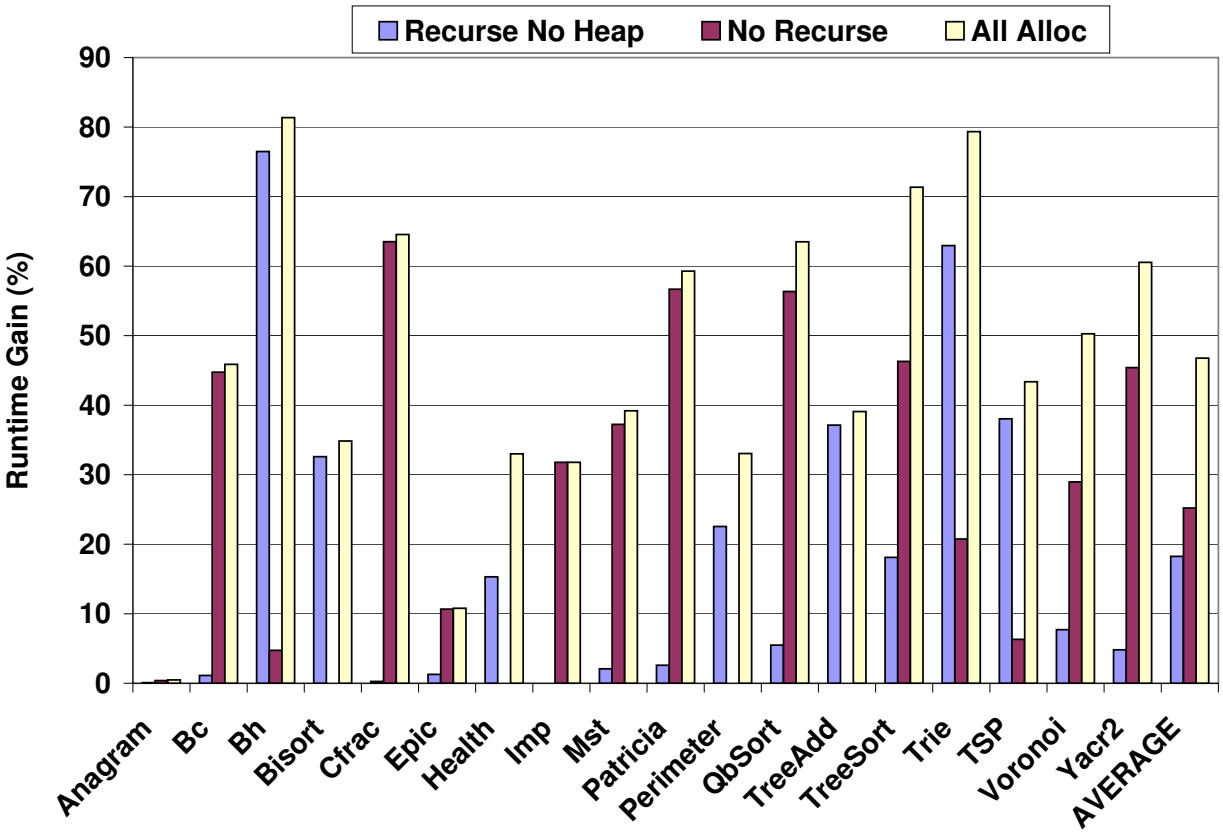


Figure 9.15: Normalized runtime for benchmarks comparing recursive function allocation at 25% SPM.

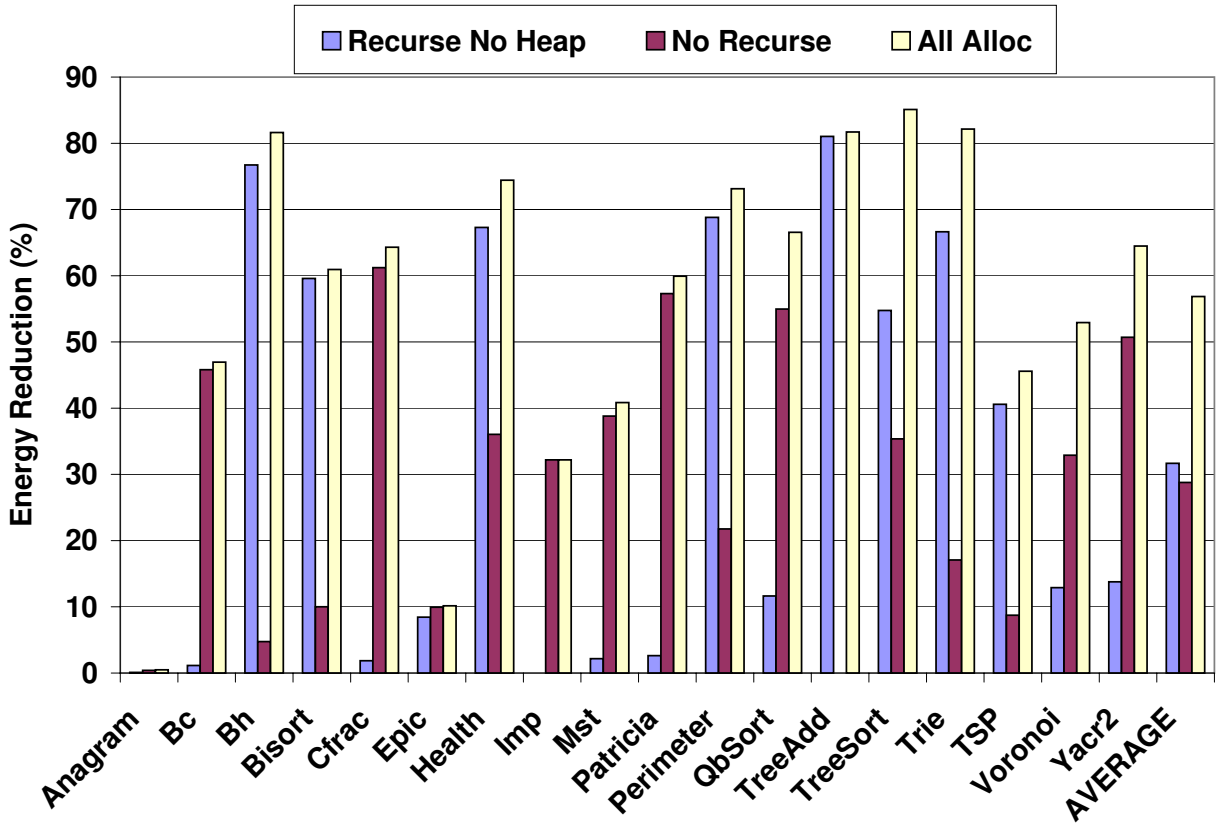


Figure 9.16: Reduction in energy usage for benchmarks comparing recursive function allocation at 25% SPM.

vidual contributions from both methods applied individually or in unison. Results on the reduction in energy consumption for this larger SPM size are presented in Figure 9.16. On average, enabling recursive allocation reduces power consumption by 31.7% compared to the default SPM allocation method. When heap allocation is instead enabled, we observe a 28.8% decrease in energy consumption. When all allocations are allowed, an average energy reduction of 56.9% is obtained, with both allocation methods able to contribute in more independent manner than at smaller and more constrained SPM sizes.

9.4 Comparison with caches

This section compares the performance of our method for scratch-pad memories (SPM) versus alternative architectures using either caches alone; or cache and SPM together. It is important to note that our method is useful regardless of the results of a comparison with caches because there are a great number of embedded architectures which have SPM and DRAM directly accessed by the CPU, but have no data cache. Examples of such architectures include low-end chips such as the Motorola MPC500 [105], Analog Devices ADSP-21XX [6], Motorola Coldfire 5206E [103]; mid-grade chips such as the Analog Devices ADSP-21160m [7], Atmel AT91-C140 [14], ARM 968E-S [12], Hitachi M32R-32192 [67], Infineon XC166 [72] and high-end chips such as Analog Devices ADSP-TS201S [8], Hitachi SuperH-SH7050 [68], and Motorola Dragonball [104]. We found at least 80 such embedded processors with no D-cache but with SRAM and external memory (usually DRAM)

in our search but have listed only the above eleven for lack of space. These architectures are popular because SPMs are simple to design and verify, and provide better real-time guarantees for global and stack data [145], power consumption, and cost [10, 131, 141, 17] compared to caches.

Our dynamic SPM allocation method shares similarities with a cache memory design but also has some important differences. Like caches our method gives preference to more frequently accessed sites by allocating them larger bins in SPM. Another advantage of our method is that it avoids copying infrequently used data to fast memory; a cache copies in infrequent data when accessed, possibly evicting frequent data. One downside of our method is that a cache retains the used subset of a heap variable in SRAM, while our method retains a fixed subset. Nevertheless, it is interesting to see how our method compares against processors containing caches.

We compare three architectures (i) an SPM-only architecture; (ii) a cache-only architecture; and (iii) an architecture with both SPM and cache of equal area. To ensure a fair comparison the total silicon area of fast memory (SPM or cache) is equal in all three architectures and roughly equal to the silicon area of the SPM in section 9.1 (which holds 5% of the memory footprint for each benchmark). Since cache must be a power of two in size and Cacti has a minimum line size of 8 bytes, the sizes of caches are not infinitely adjustable. To overcome this difficulty we first fix the size of cache whose SPM-equivalent in area holds the nearest to 5% of the data size. Then an SPM of the same area is chosen; this is easier since SPM sizes are less constrained. For an SPM and cache of equal area the cache has lower data capacity because of the area overhead of tags and other control circuitry. Area and

energy estimates for cache and SPM are obtained from Cacti [40, 147]. The cache simulated is direct-mapped (this is varied later), has a line size of 8 bytes, and is in 0.5 micron technology. The SPM is of the same technology but we remove the tag memory array, tag column multiplexers, tag sense amplifiers and tag output drivers in Cacti that are not needed for SPM. The Dinero cache simulator [137] is used to obtain run-time results; it is combined with Cacti's energy estimates per access to yield the energy results.

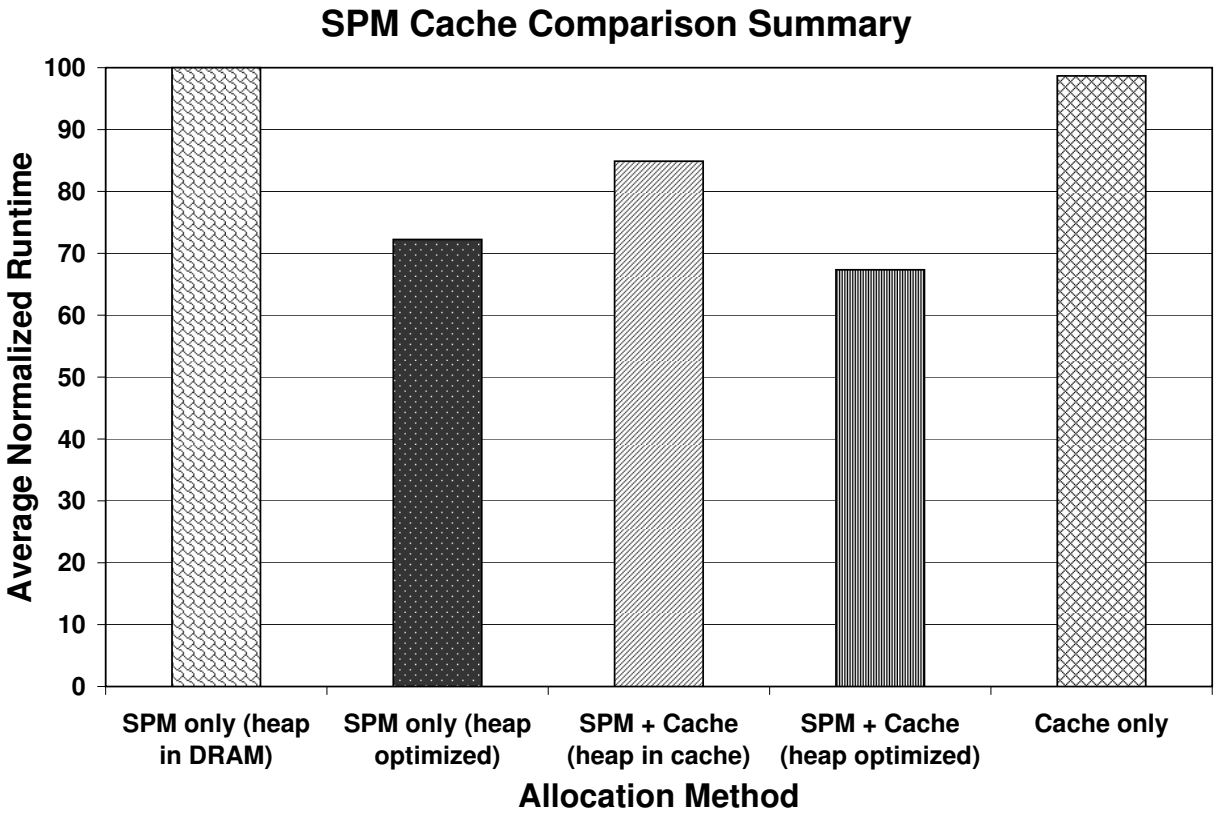


Figure 9.17: Average normalized run-time for architectures containing different combinations of SPM and cache.

Figure 9.17 shows the average normalized run-times for different architecture and compiler pairs, obtained by averaging the results for each scenario across all benchmarks. The first bar is without our heap and recursive stack allocation meth-

ods for the SPM-only design, against which the other bars are normalized. The second bar shows the runtime for the SPM-only design when we apply our allocation methods, though global and non-recursive are allocated to SPM in both scenarios. The third and fourth bars are similar to the first and second, except that for these two we have an SPM and cache available for use on the same system. The third bar shows the results when we allocate global and non-recursive stack objects to SPM and let the cache handle all DRAM accesses, including all heap variables which are left in DRAM for this scenario. With a cached DRAM present, both the transfers required for our methods as well as standard DRAM memory accesses are accelerated through the cache. The fourth bar corresponds to the case when we apply our full SPM allocation scheme to all data objects, and let the cache handle all DRAM accesses made, again improving transfers and accesses to DRAM. The fifth and final bar is for the cache only architecture where all data resides in DRAM and accessed through the cache only.

From the results shown in figure 9.17, we see that the first and fifth bars are almost equal, showing that the baseline SPM allocation method for global and non-recursive stack data performs only slightly worse (1.8%) than the cache-only platform. This was also shown to be the case in the results presented in [132] when the baseline SPM method was compared against a cache for a different benchmark set. The scenario where our method was applied on the Cache + SPM platform obtained the best performance improvement of 33%. The other Cache + SPM scenario where we do not apply our method showed a much smaller performance improvement of 15.4% over the baseline. Finally, the scenario where our allocation scheme is

applied to an SPM-only platform performed the second best with a 28.4% improvement in runtime compared to the baseline, and a remarkable 26.6% improvement on average over the cache-only architecture.

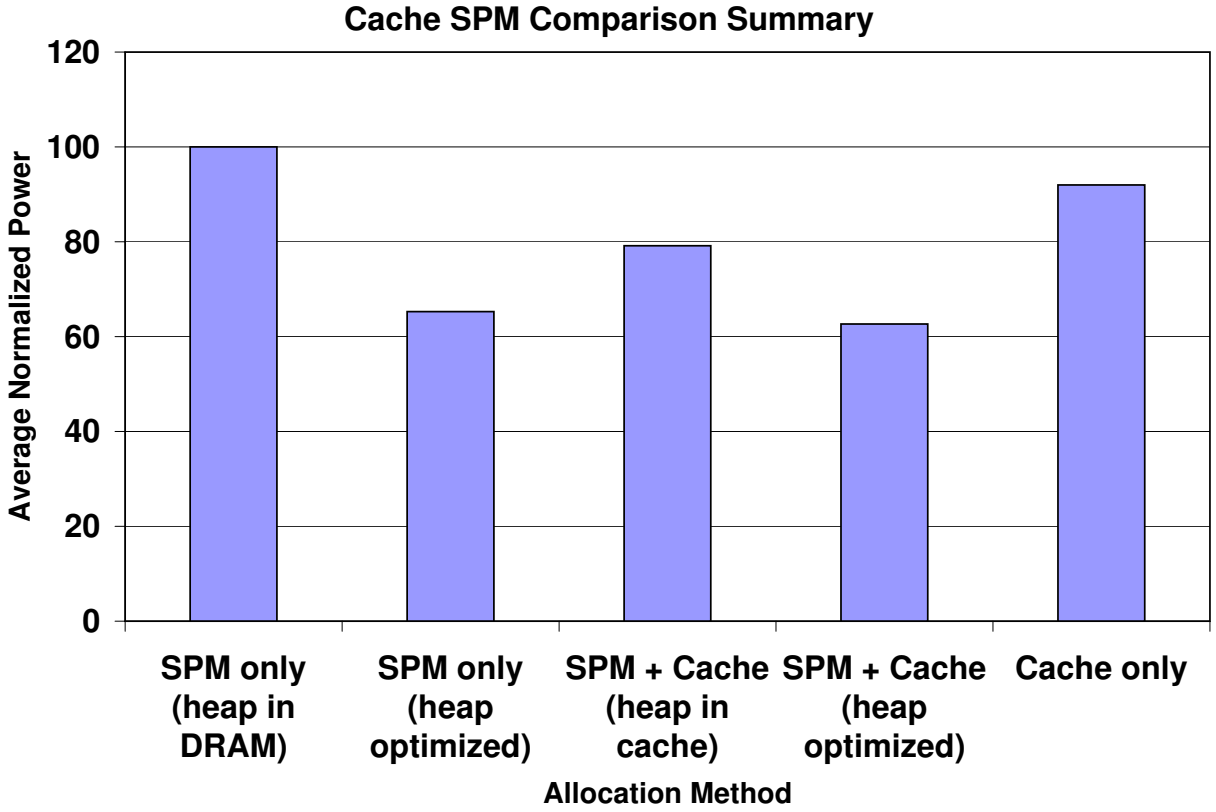


Figure 9.18: Average normalized energy usage averaged across all benchmarks for different architecture/compiler pairs.

Figure 9.18 shows the normalized energy consumption averaged across all benchmarks for the same configurations as in figure 9.17. In this figure we see that the cache-only scenario performs 8.5% better than the baseline in terms of energy consumption, which is higher than the 1.5% improvement in runtime. This is mainly due to the more efficient cache interface to DRAM which reduces power by pipelining cache line loading and decreasing idle processor cycles while DRAM operations complete. Looking at the third and fourth bars for the SPM+Cache sce-

narios, we see that the best energy savings of 37.6% is obtained when our method is applied. When our method is not applied for these platforms, the energy savings drop to 21.1%. The final scenario where we apply our method to an SPM-only platform achieves an impressive 35.3% savings in power.

It is interesting to analyze the strengths and weaknesses of our method vs. caches in the light of these results. From careful analysis of individual benchmark results (available in Appendix A), we have found that in many cases, caches simply do not perform well for dynamically allocated program data, particularly at smaller sizes where cache conflicts are more common. Comparing the results of the SPM + cache scenarios, they show that caches generally have a much harder time with heap data (due to their random creation and access patterns) than with stack and global data. The most common use of heap memory in applications is for allocation of dynamic data structures and for allocation of memory arrays for processing. Dynamic and recursive traversal of such data structures is often unlocalized and pointer reference chains tend to access non-sequential memory locations, both problematic for caches. Caches, on the other hand, perform best by localizing sequential memory accesses from applications such as a media encoders and are also able to localize accesses to individual variables too large to place in SPM. From the individual benchmark results, we see that applications such as the MPEG encoder and decoder perform better using only a cache than when using our method for SPM, mostly because these applications operate on very large variables which we are unable to fit into SPM.

Looking further into benchmark statistics, we also observed that most pro-

grams which used both heap and recursive functions also tended to do so for creation and traversal of dynamic data structures such as lists, trees and graphs. Furthermore, when the runtime stack for a recursive function is viewed as a stacked memory array, most recursive functions also tend to make most of their memory accesses at either the deepest or shallowest levels of recursion. Our method is able to select which invocations of a recursive function are placed in SPM and allowed to evict other variables. Caches, on the other hand, must transfer a cache line from DRAM to SRAM for every access miss incurred. Often in recursive functions, the entire recursive stack frame will be loaded into SRAM, evicting more useful data and deteriorating the performance of cache-only systems.

In conclusion, the results in figures 9.17 and 9.18 show that our method outperforms a cache-only architecture and also provides better run-time and energy in an SPM + cache architecture. We believe it is remarkable for a compile-time method for heap data to out-perform a cache – something many thought was not possible. There are also two other advantages of SPMs over caches not apparent from the results above. First, it is widely known that for data accesses, SPMs have significantly better real-time guarantees than caches [145, 131, 16]. Second, other researchers and our own simulations have repeatedly demonstrated a significant energy and run-time savings for platforms using only an SPM allocation scheme rather than relying on a hardware cache system [10, 131, 141, 17].

9.5 Profile Sensitivity

In this section we present several experiments to determine the consistency of our method across different inputs. Because our method is profile-based, it is important to quantify the degree to which the chosen profile affects program performance for different inputs. We begin by presenting allocation results when using an input set different from the one used in the rest of this chapter. For each benchmark, we were able to obtain or create at least two different input sets. Due to the difficulty in obtaining more input sets for all benchmarks, the large amount of time required for simulation and the large number of experiments present, we have limited our experiments to only two inputs. Instead, for those cases where we possessed more than 2 inputs we chose the two exhibiting the most profile variation. The input sets used were deemed reasonable for an embedded platform, and those benchmarks or inputs requiring very large amounts of storage were discarded. All results presented in this section were obtained using the default simulation parameters unless otherwise noted.

Figure 9.19 shows the normalized runtime results of our method when applied to the benchmark suite using the second of the input data sets available. By comparing these individual results to the original ones from the first input in Figure 9.1, we can make several observations. We see that some benchmarks have static allocation and execution patterns which do not vary much for different inputs. This tends to be the case for computation benchmarks and kernels such as Dhrystone, Dhrystone-Float, EKS, Heapsort, Lists, LLUBenchmark, Objinst, PiFFT and Sort-

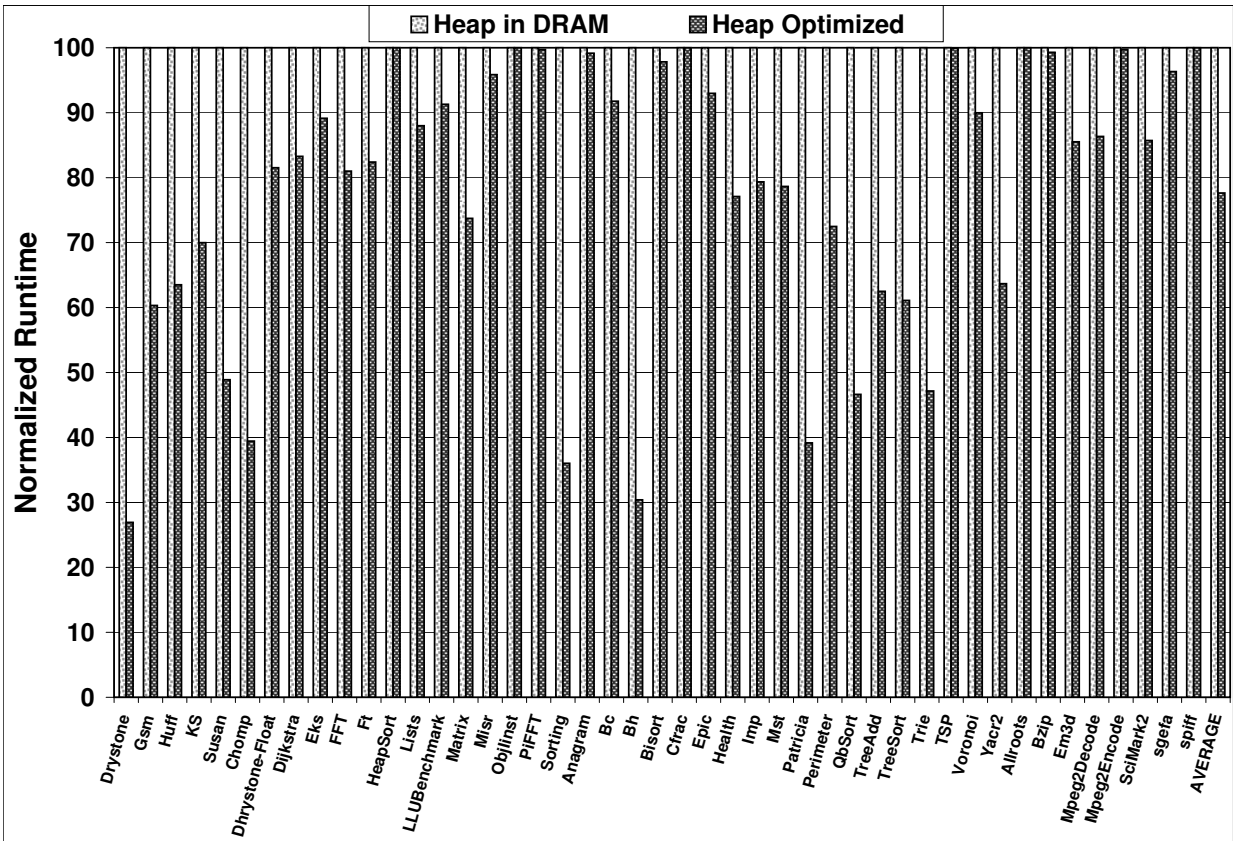


Figure 9.19: Normalized runtime using for second benchmark input set.

ing. All of these accept iteration counts or array sizes as their main input, have known-size heap allocation sites and have very predictable allocation and execution patterns from compile-time analysis. Other benchmarks are highly input dependent and their dynamic allocation and execution profile varies dramatically for different inputs. This can be observed by the relative shift in normalized runtimes for applications using different inputs such as Chomp, Dijkstra, BC, Cfrac, Patricia, QBSort, TSP, Allroots, Bzip, Mpeg2Decode, Mpeg2Encode, Scimark2 and spiff. We note that all of these applications involved randomized access of dynamic data structures highly dependent on the particular input set used. Many of these applications make use of unknown-size heap allocation sites and recursive functions, further indications of their dynamic input dependence.

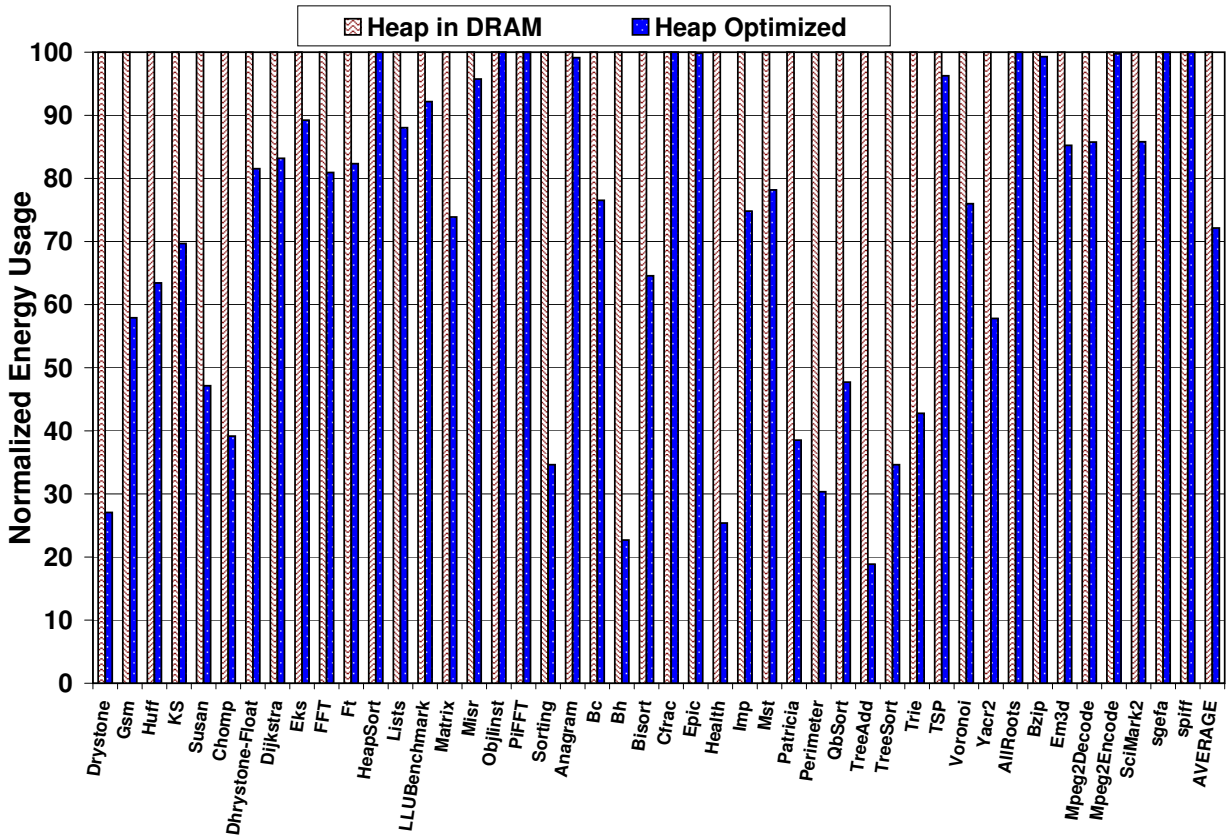


Figure 9.20: Normalized energy usage for second benchmark input set.

Figure 9.20 shows the energy usage results for the benchmark suite using the second of the input data sets. From this figure we can see the energy improvement closely follows the runtime improvement from each benchmark and our method is beneficial for both runtime and energy for most applications using different inputs.

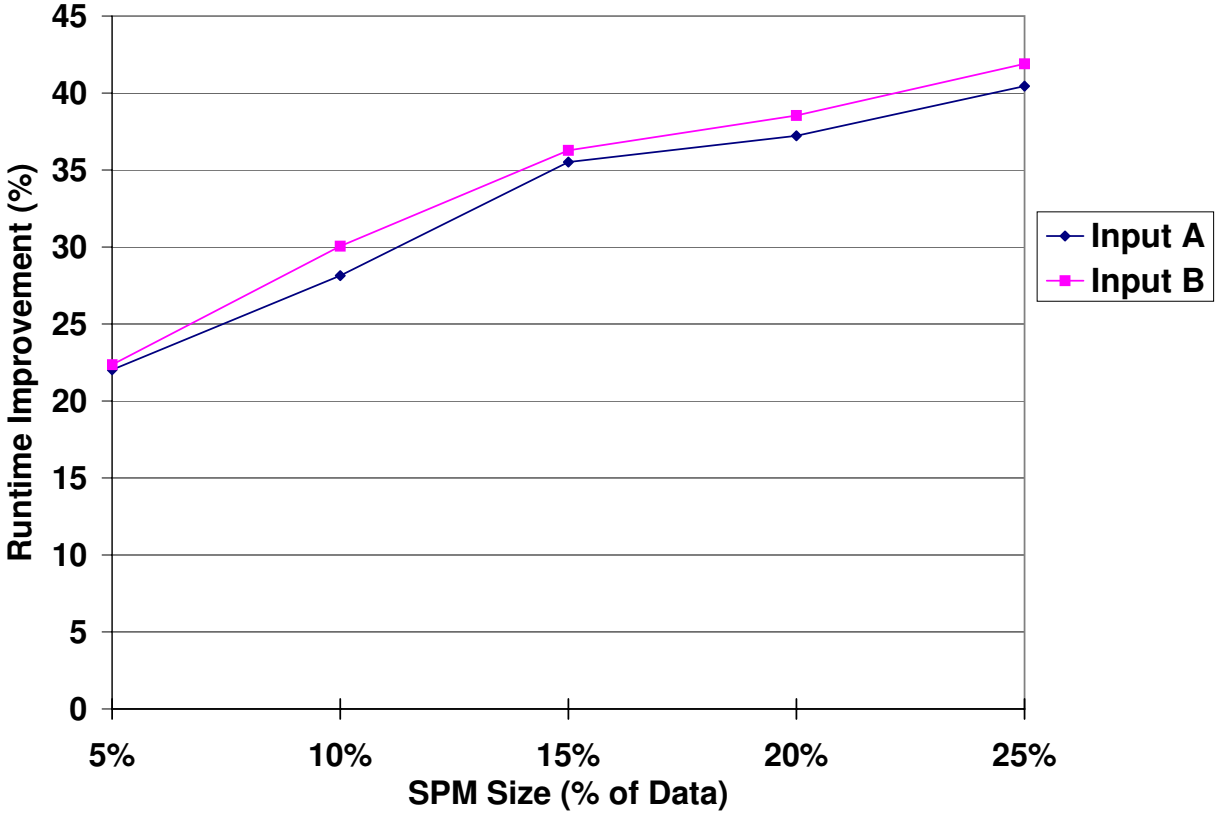


Figure 9.21: Average runtime benefit for both input sets at different SPM sizes.

In figure 9.21, we present a summarized view of our method's benefit for two different input sets as we vary the SPM size from 5% to 25% of the application memory footprint. From this figure we can see that despite the fluctuations in individual benchmarks from different inputs, the benefits of our method are consistent when the suite is examined as a whole. Our method is able to analyze dynamic profiles from an input set and tailor the application for best performance on a target plat-

form with SPM. The average runtime improvement is almost identical at 5% SPM for both input sets. As SPM size is increased, the average runtime remain within a few percent of each other. A very similar situation is observed from the average energy savings results graphed in Figure 9.22.

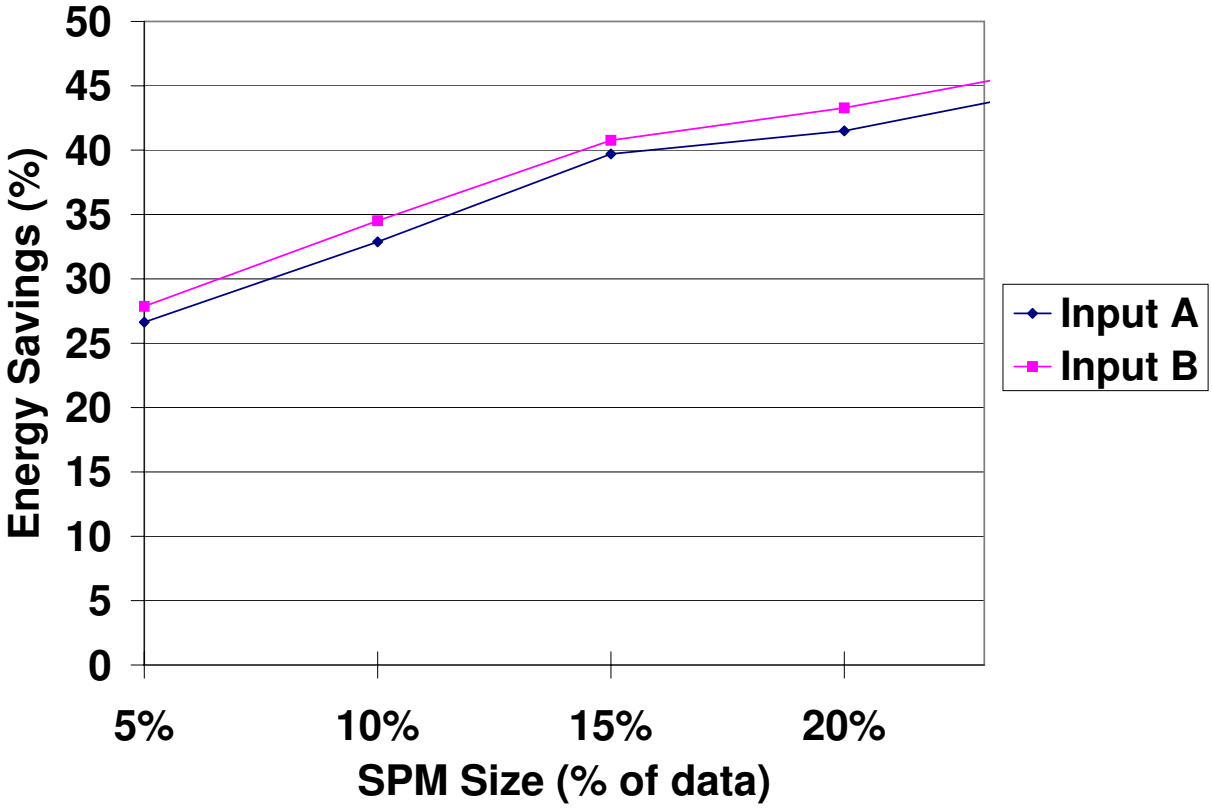


Figure 9.22: Average energy savings for both input sets at different SPM sizes.

9.5.1 Non-Profile Input Variation

Having shown that are method is able to analyze and optimize an application for a given input set, we also wish to see how well our method performs on a non-profiled input set. This becomes particularly important for those applications whose runtime behavior is demonstrably input dependent. To isolate the subset of profile sensitive applications, we conducted two experiments using the two input data sets

A and B. For the first experiment, we used input A to generate a program profile for our method and then evaluated its performance when input B was used with the optimized binary. The second experiment simply reversed the order the inputs were applied from the first. The subset of benchmarks with results that varied more than 1% among the different profile and optimization pairs make up our profile-dependent benchmark set. All other benchmarks were observed to have predictable runtime execution patterns, changing only slightly in terms of their program execution and allocation distribution for different inputs.

Figure 9.23 shows the runtime gain comparison results for the first experiment where we examine the profile dependence based on input A. The first bar shows the scenario we use profile information from input A to optimize and also gather improvement results. The second bar shows the case where we optimize based on input A's profile, but gather results using input B. Figure 9.24 shows the same scenario for the second experiment with the input order reversed. Looking at these two graphs shows us which applications are input dependent and quantified the effect that profile dependence has on our basic approach.

Our final experiment applies our profile averaging optimization, which attempts to combine profile information from different program inputs to decide on the best allocation strategy for the common case. We apply our optimization to those benchmarks identified as being input-dependent and which showed more than 1% variation in the previous experiment. Figure 9.25 shows the runtime improvement for three different scenarios. The first bar shows the runtime improvement when we profile and optimize with Input A. The second bar shows the improvement

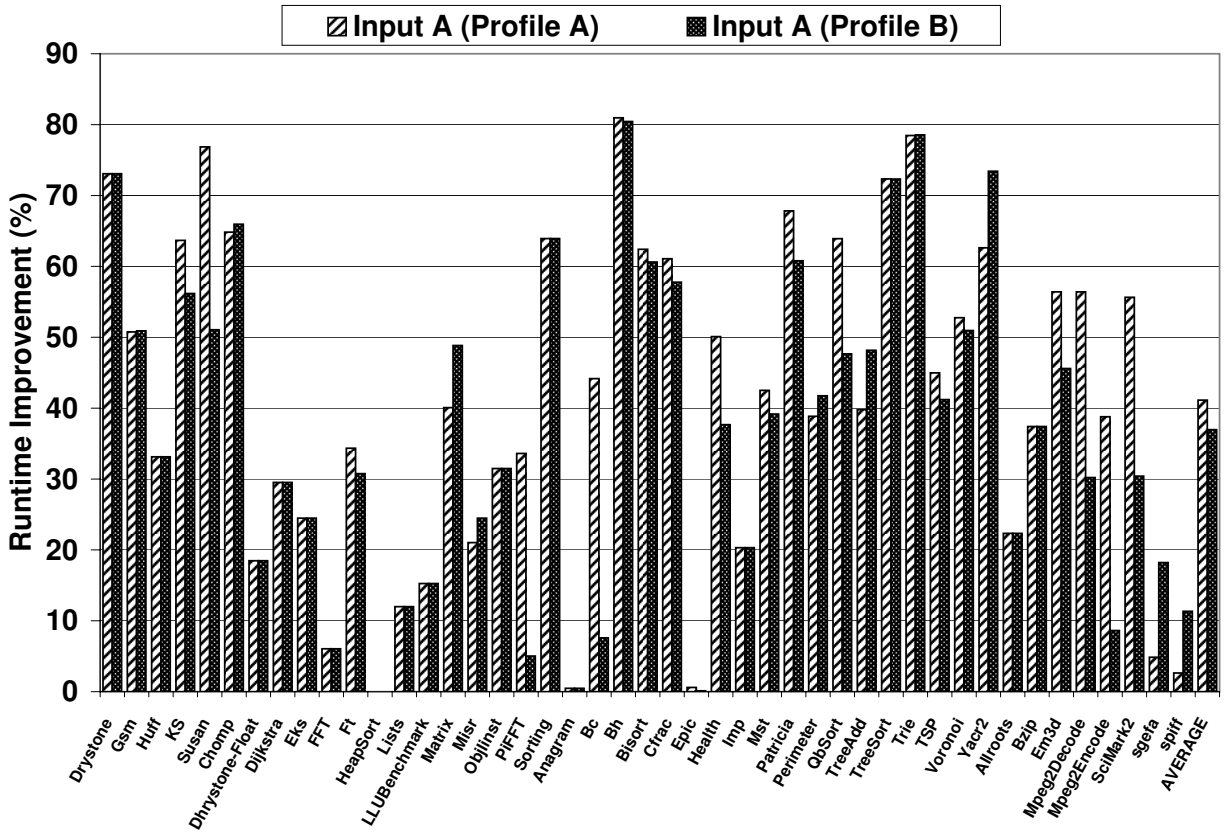


Figure 9.23: Normalized runtime showing profile input sensitivity.

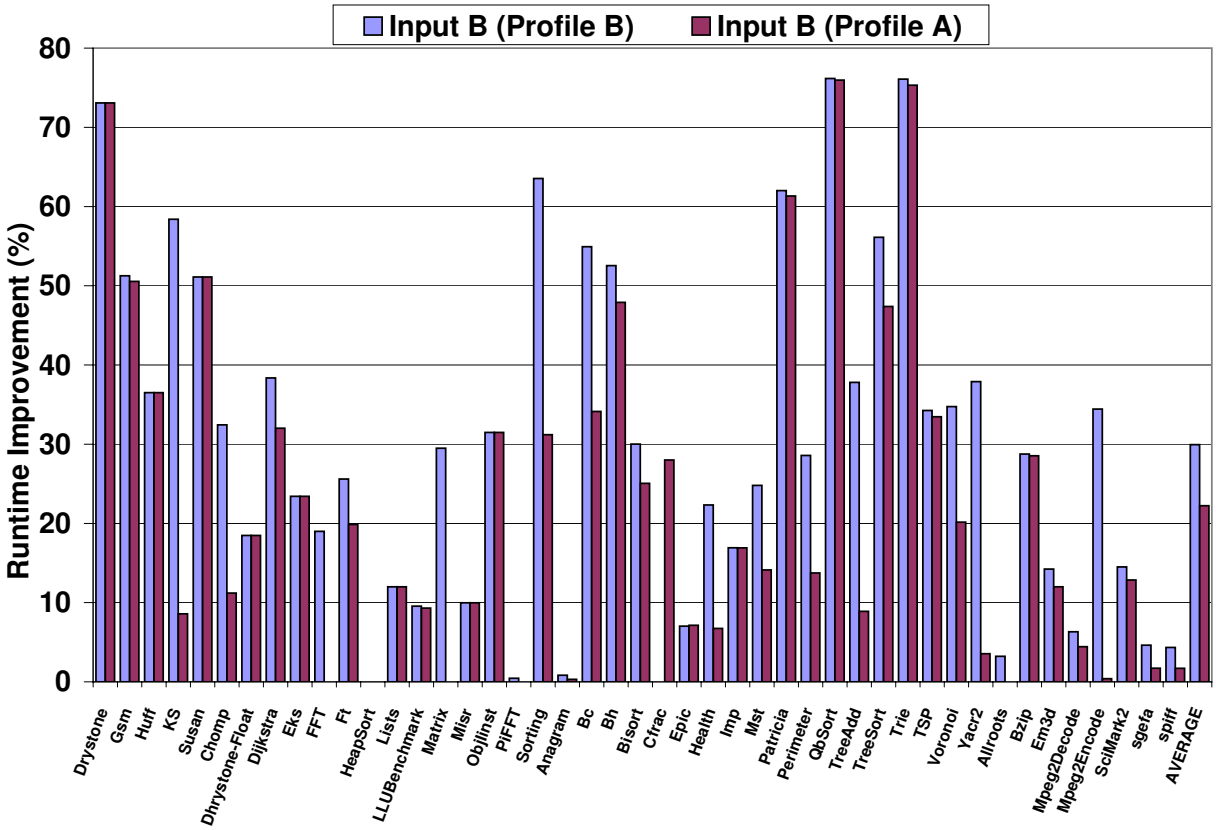


Figure 9.24: Normalized runtime showing profile input sensitivity.

when profiling with input B but using input A for results. The third bar shows the improvement when we apply our averaging method to profiles from inputs A and B and use Input A to obtain the results. We present similar results in Figure 9.26 except these are based on input B instead of A. Looking at both sets of results, we find that our averaging optimization is able to greatly reduce the profile sensitivity from our allocation performance. We find that with as little as two profile inputs, we can still achieve on average a runtime improvement of 36.3% regardless of inputs for the profile-dependent benchmark subset

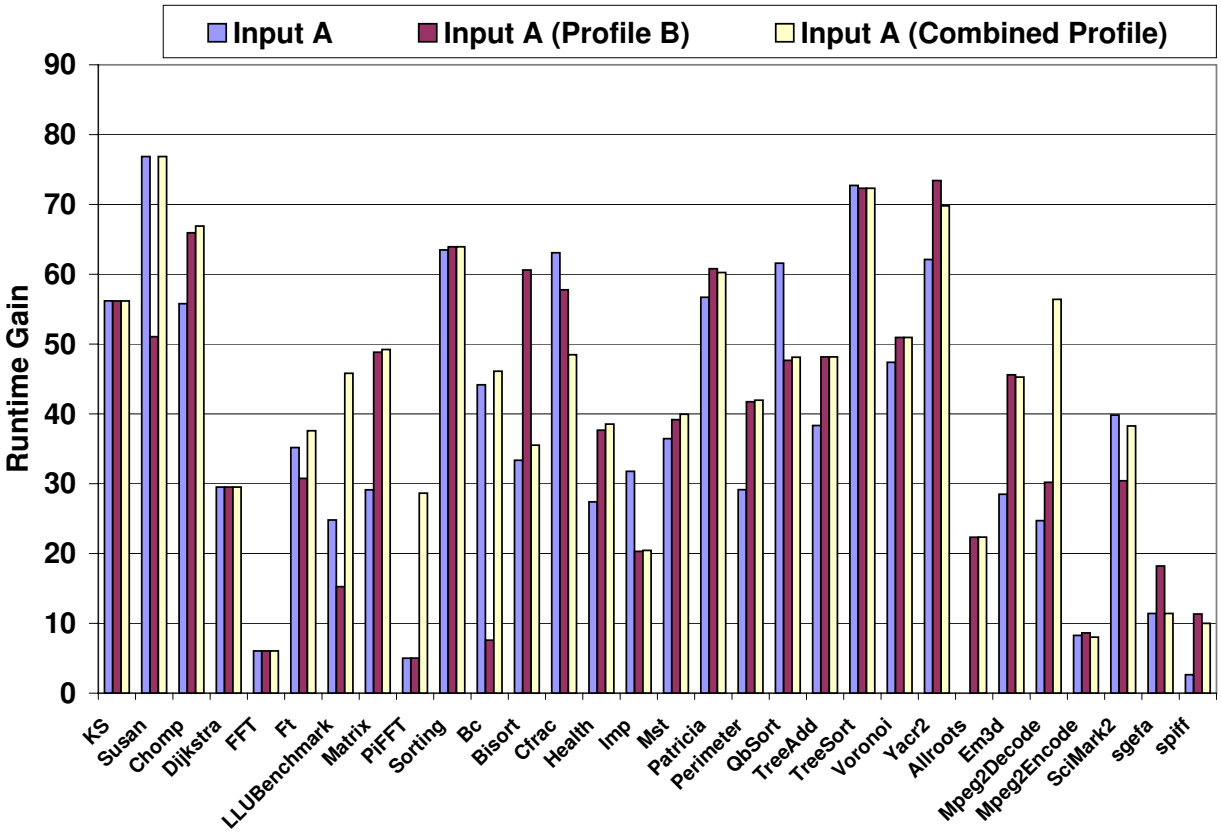


Figure 9.25: Improvement in runtime with and without the profile averaging optimization for Input A.

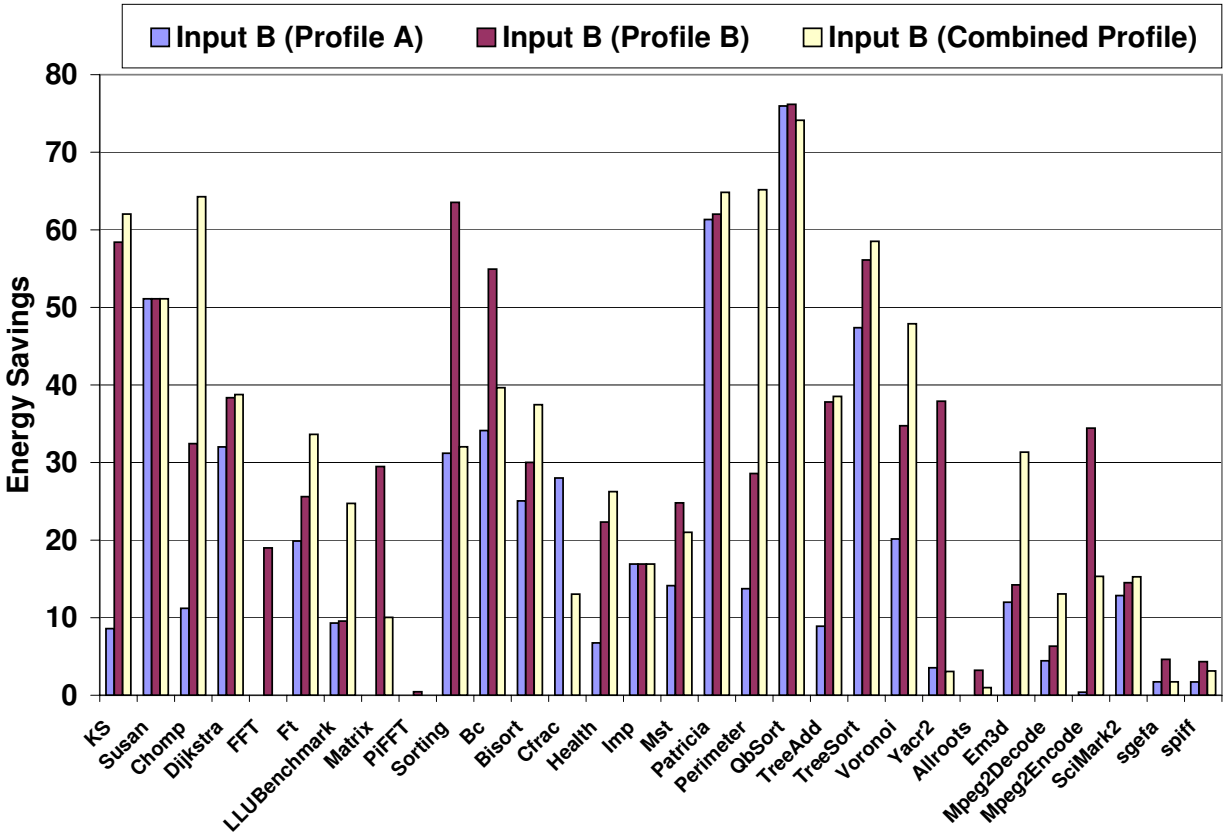


Figure 9.26: Improvement in runtime with and without the profile averaging optimization for Input B.

9.6 Code Allocation

Traditionally, code placement is usually one of the best ways a designer can improve embedded platform performance. This is reflected in the observation that instruction caches are much more common than data caches on existing embedded systems, because program instruction accesses tend to be highly predictable at run-time. Unlike program data, which can have very unpredictable execution profiles, program instructions are very amenable to exploitation of predicted access behavior. Even if a program makes almost no memory accesses during execution to operate on program data, the processor will be constantly reading instructions throughout program execution and generally do so in a linear manner with branches to different code segments.

Precisely because code allocation has been studied and exploited in so many different ways, in this thesis we focus on the more difficult problem of data allocation. Almost all mid-level and high-end embedded processors contain some form of fast memory meant to be used for code placement. This can come in the form of an instruction cache, Flash EEPROM, regular ROM or even SRAM. Because self-modifying code is rare in practice, a read-only memory like ROM is popularly used to store static program code like firmware or other permanent applications. Flash has lately become a popular choice to load instructions into for execution, due to its lower read latency and reduced power consumption compared to DRAM, with the added flexibility of writable memory. For these reasons, it follows that SPM is most beneficial for data placement, with code allocation handled through a number of

complementary designs in embedded systems. Embedded processors contain some form of fast memory meant to be used for code placement. This can come in the form of an instruction cache, Flash EEPROM, regular ROM or even SRAM. Because self-modifying code is rare in practice, a read-only memory like ROM is popularly used to store static program code like firmware or other permanent applications. Flash has lately become a popular choice to load instructions into for execution, due to its lower read latency and reduced power consumption compared to DRAM, with the added flexibility of writable memory. For these reasons, it follows that SPM is most beneficial for data placement, with code allocation handled through a number of complementary designs in embedded systems.

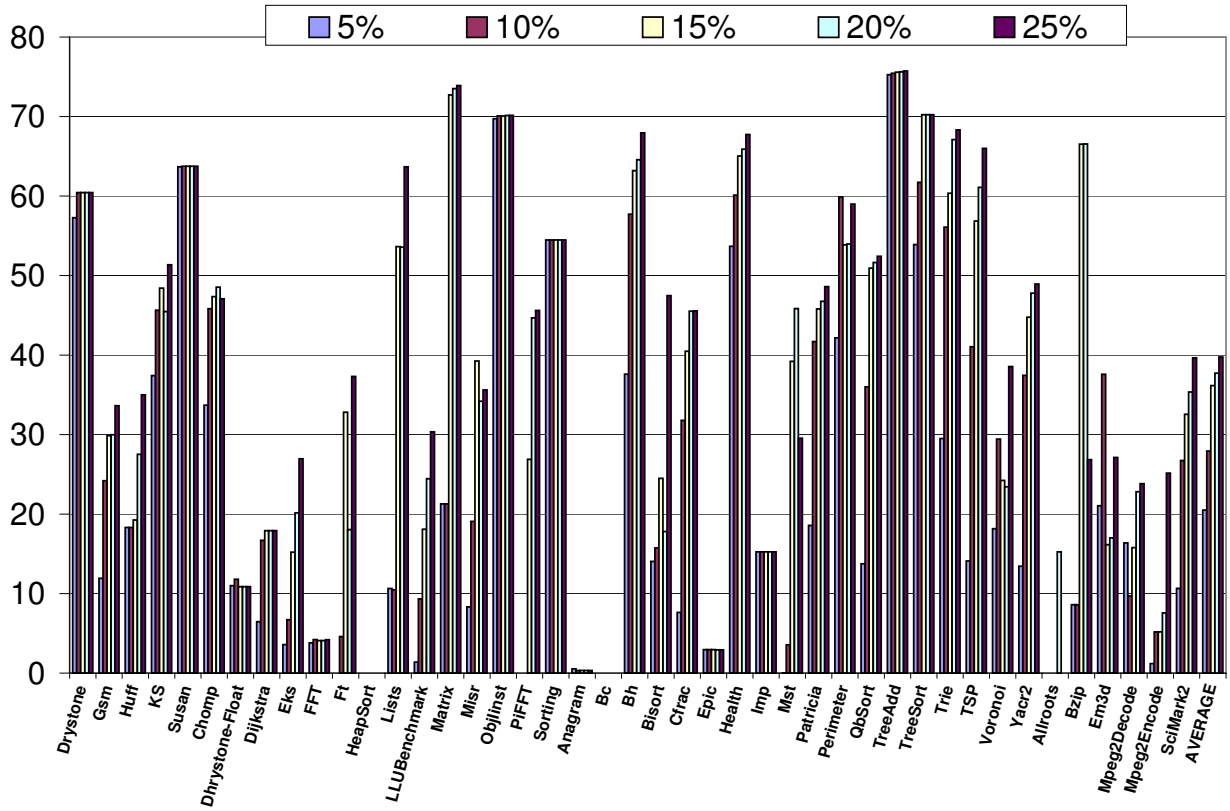


Figure 9.27: Runtime gain when code as well as global, stack and heap data are allocated to SPM of varying sizes.

Figure 9.27 presents the runtime gain from our method when code objects are also considered for dynamic allocation to SPM, compared to the baseline method in [132]. In this figure, results from the benchmark set are shown for different sizes of SPM, from 5% to 25% of the total data size for each program. We see that on average, when code objects are included by our scheme as well as the baseline scheme, our relative improvement in execution is very similar to the default situation where code is assumed to reside in SRAM. We see that at 5% SPM size, our method shows a 20.5% decrease in runtime compared to the baseline case, when both methods also allocate code to SPM with other objects. Without code allocation to SPM, we saw a similar decrease of 22% in runtime when comparing our method to the baseline at 5% SPM size. This correlation continues throughout the sizes of SPM explored. At 25% SPM our method shows a 39.76% runtime gain when code is included, compared to a gain of 40.45% for our default scenario where all code resides in SRAM. Figure 9.28 also presents the energy results from our code allocation experiment, and closely follows the results of our execution results. Including code objects in the list of allocatable objects for dynamic SPM methods only tends to put more pressure on placement at small SPM sizes, but otherwise presents no problem for our scheme or the baseline method.

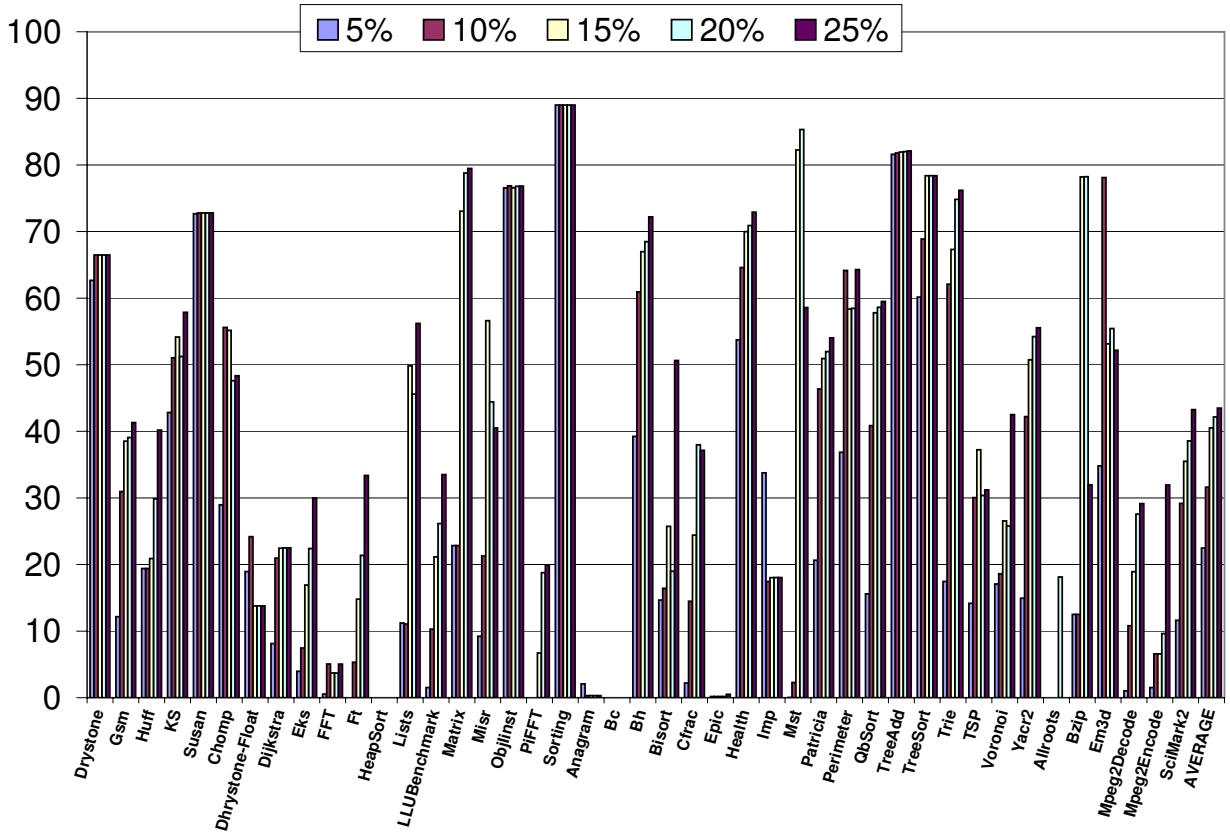


Figure 9.28: Energy savings when code as well as global, stack and heap data are allocated to SPM of varying sizes.

Chapter 10

Conclusion

This thesis presents the first-ever compile-time method for allocating a portion of a program's dynamic data to scratch-pad memory. Compile-time placement of dynamic data in scratch-pad is complicated by two factors. First, the size of dynamically allocated structures is usually data-dependent and thus is not knowable at compile-time. Consequently it is difficult guarantee at compile-time that a given dynamic data object will fit in scratch-pad. Second, moving heap data between scratch-pad and DRAM, required by all dynamic allocation methods, results in pointers pointing into a moved block to become invalid after movement, violating correctness.

The presented method for allocating dynamic data to scratch-pad solves the above problems as follows. First, the problem of unknown size dynamic data structures is solved by placing only a fixed-size portion of the structure, called a bin, in scratch-pad memory. Second, the problem of invalid pointers upon movement of bins is solved by ensuring that the bin location is the same at all program points where the heap structure is accessed. However, for better scratch-pad utilization, bins can be moved to other locations at program points where the heap structure is not accessed. More frequently accessed dynamic data is allocated a larger bin in scratch-pad to improve runtime. With our method, all types of global, code, stack and heap

variables can share the same scratch-pad. When compared to placing all dynamic data variables in DRAM and only global and stack data in scratch-pad, our results show that our method reduces the average runtime of our benchmarks by 22.3%, and the average power consumption by 26.7%, for the same size of scratch-pad fixed at 5% of total data size. Furthermore, our method is shown to outperform equivalent sized cache memory organizations for applications relying heavily on dynamic data during execution. Finally, our method is able to minimize profile dependence issues through careful analysis of dynamic profile information.

This chapter contains extra results, statistics and data concerning our experiments for SPM allocation techniques.

10.1 Primary Heap Allocation Results

This section provides further results for each of our proposed optimizations and the effect of varying the SRAM size dedicated to each benchmark application.

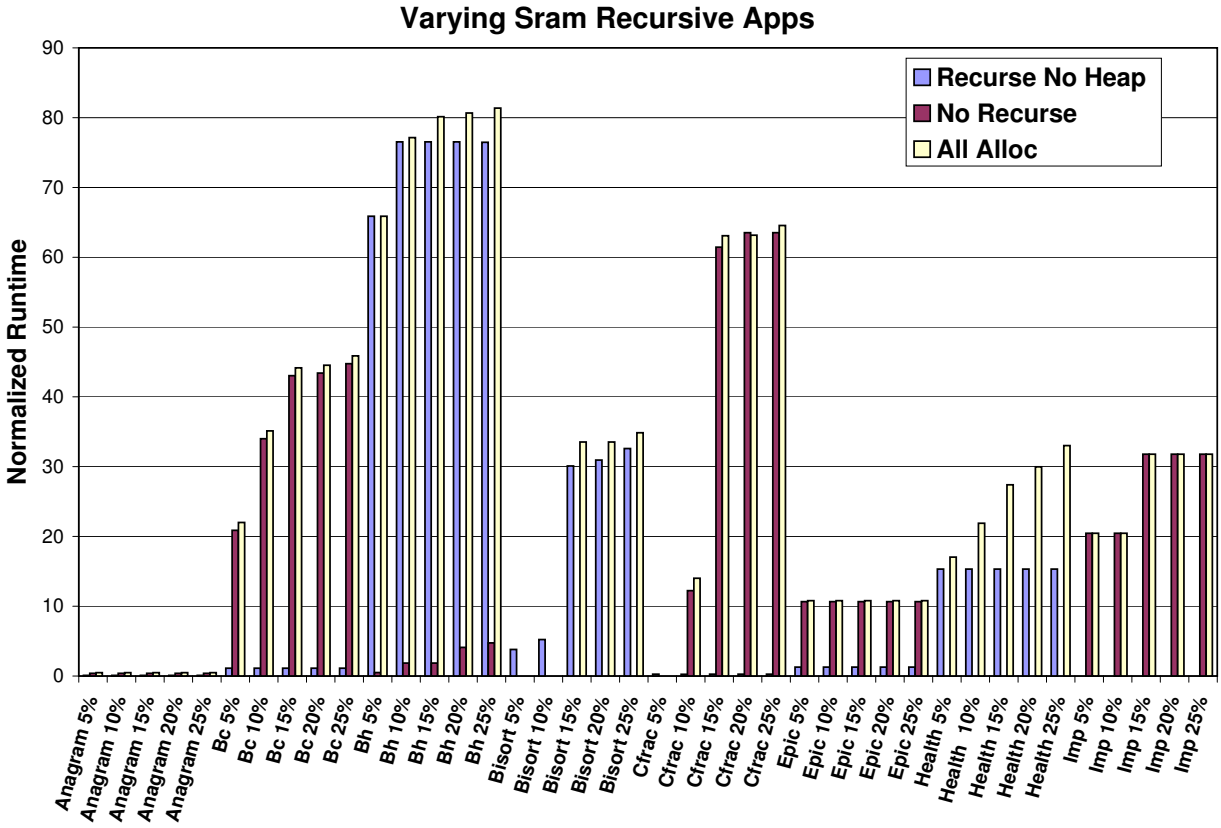


Figure 10.1: Normalized run-times for recursive applications when SRAM size is varied as a percentage of the program data size(Part 1).

Figures 10.1 and 10.2 shows the normalized run-time results for the recursive benchmark set while varying the total SPM size as a percentage of the benchmark's total working data size.

Figures 10.3 and 10.4 shows the normalized energy usage results for the

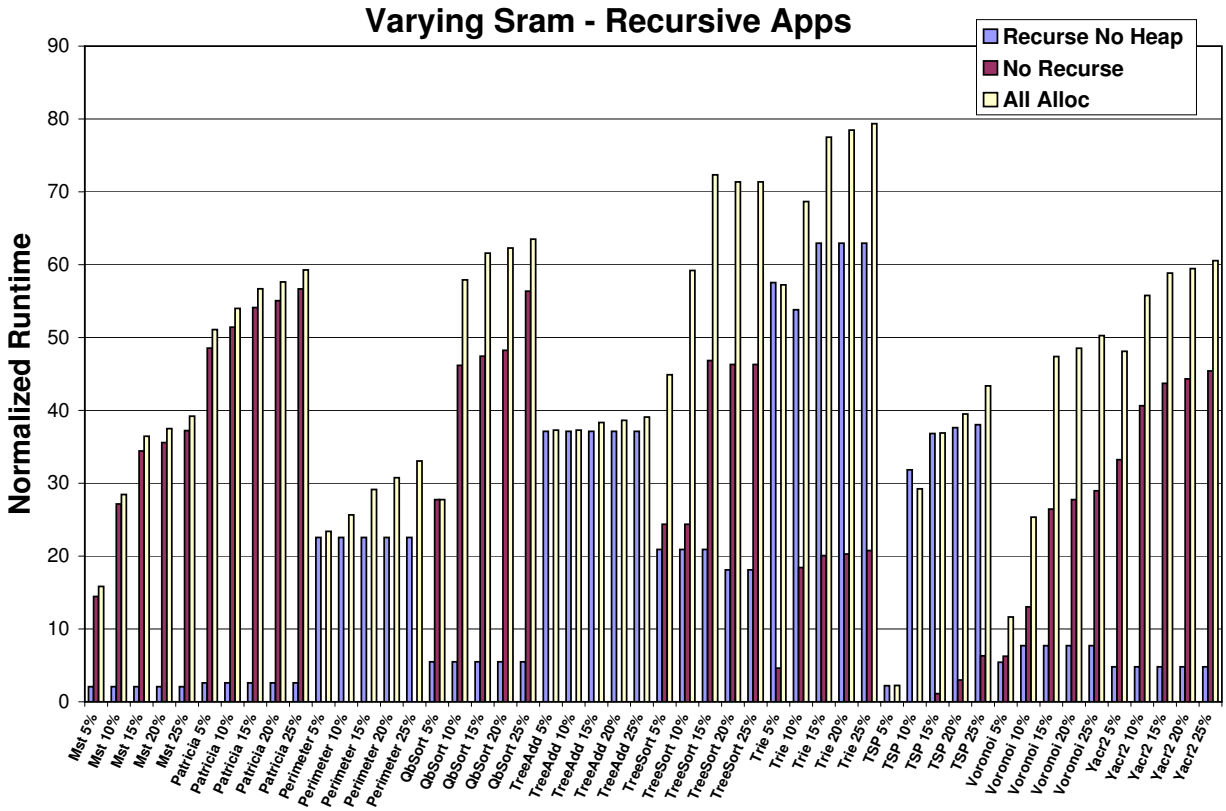


Figure 10.2: Normalized run-times for recursive applications when SRAM size is varied as a percentage of the program data size.

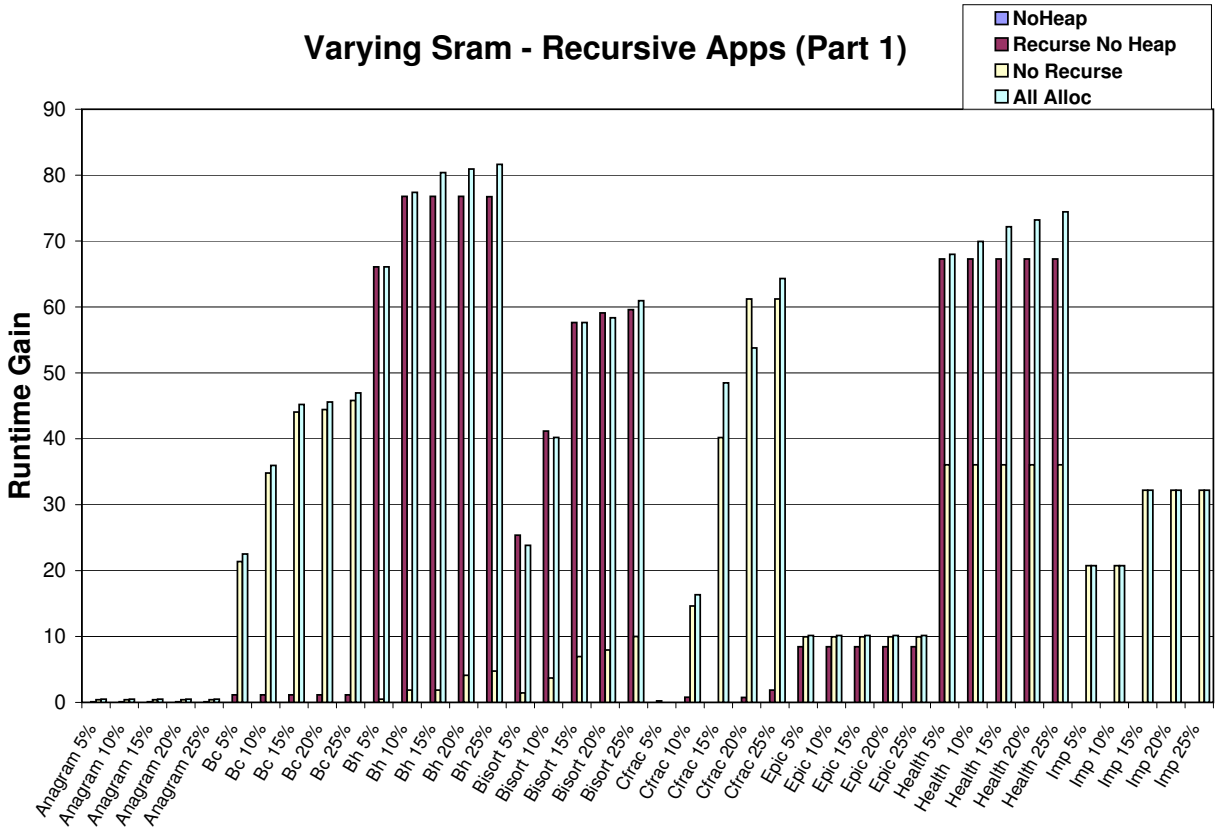


Figure 10.3: Normalized energy usage for recursive applications when SRAM size is varied as a percentage of the program data size.

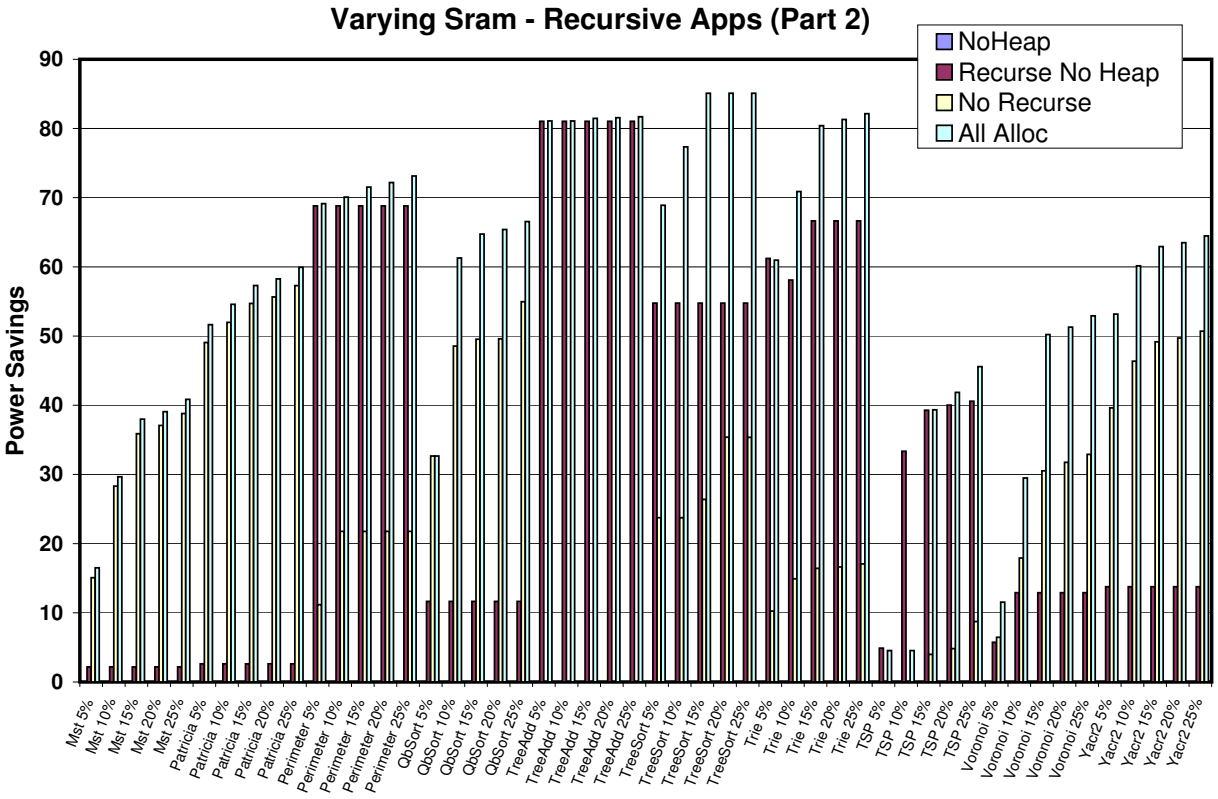


Figure 10.4: Normalized energy usage for recursive applications when SRAM size is varied as a percentage of the program data size.

recursive benchmark set while varying the total SPM size as a percentage of the benchmark's total working data size.

10.2 Cache Comparison Results

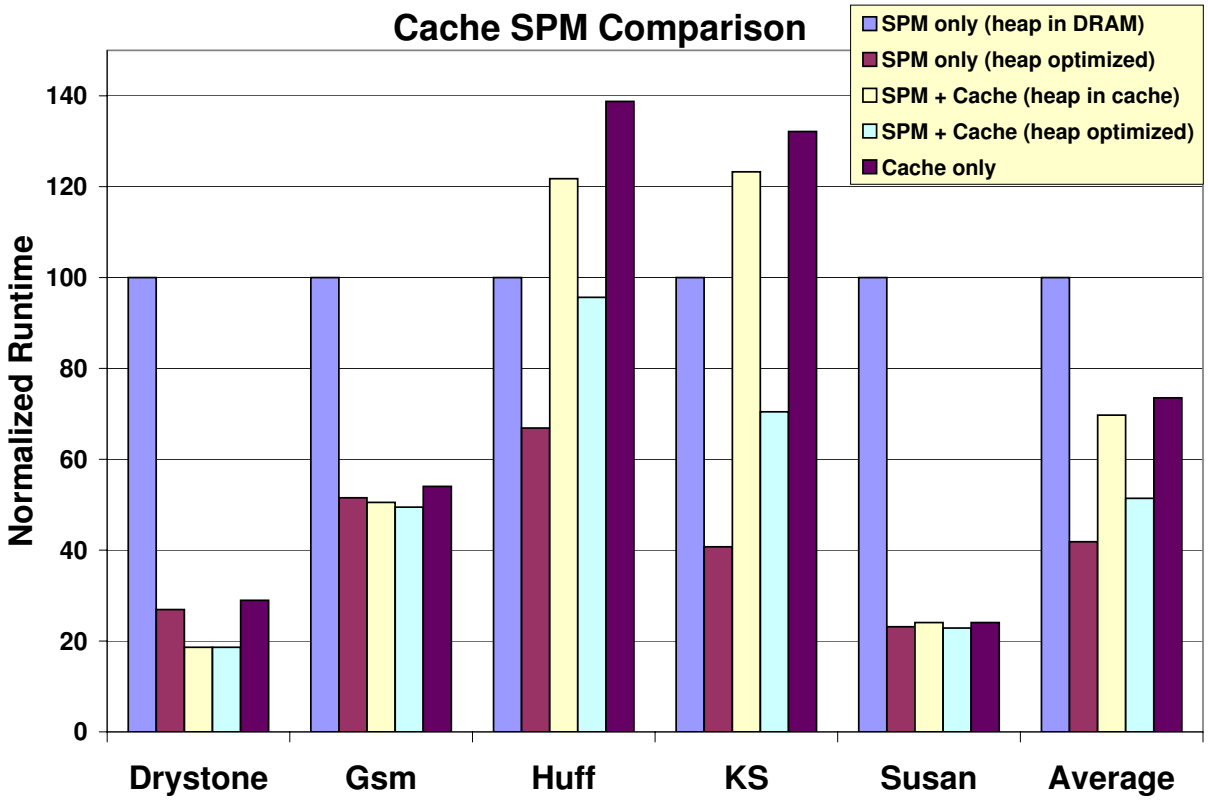


Figure 10.5: Normalized run-times from the original JEC benchmark set for architectures containing different combinations of SPM and cache.

Figure 10.5 shows the normalized run-time results averaged across our original JEC benchmark set from exploring the different memory configuration and compiler optimizations available.

Figure 10.6 shows the normalized run-time results averaged across our regular heap applications from exploring the different memory configuration and compiler optimizations available.

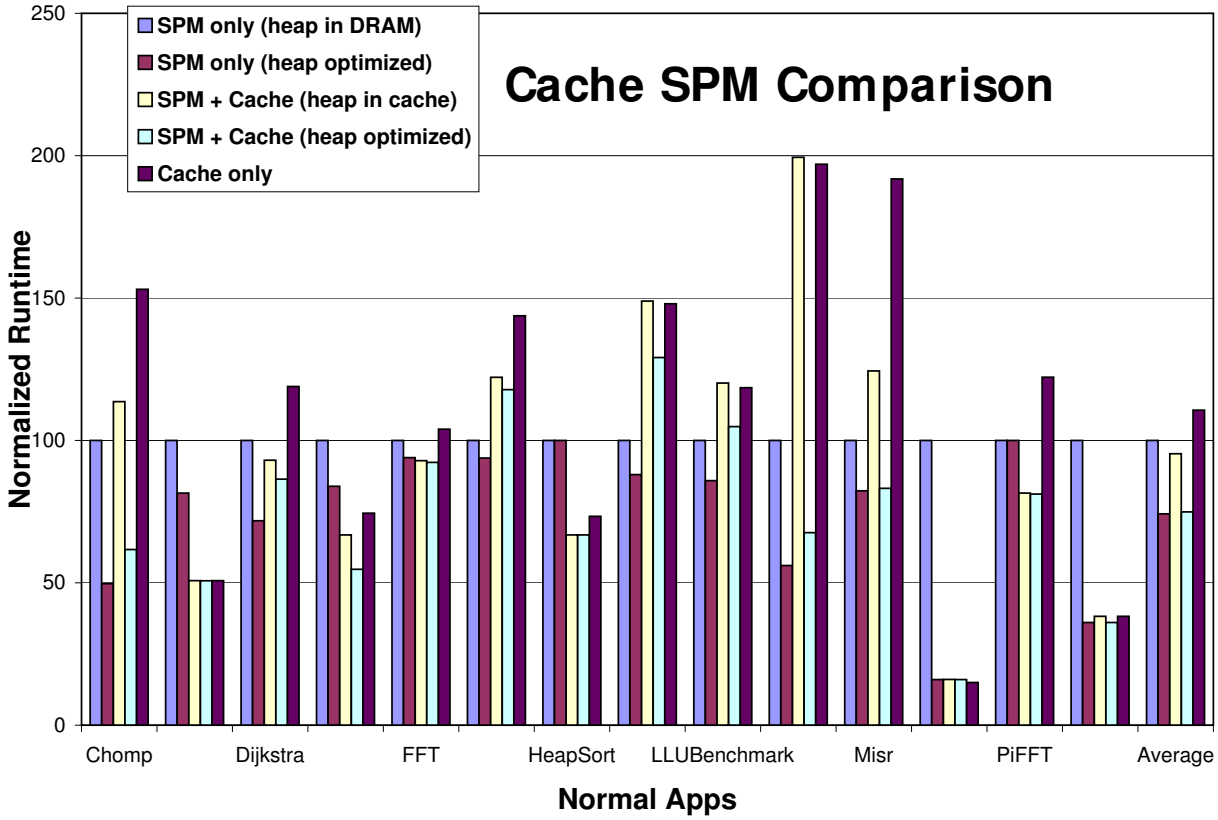


Figure 10.6: Normalized run-times from the regular heap applications set for architectures containing different combinations of SPM and cache.

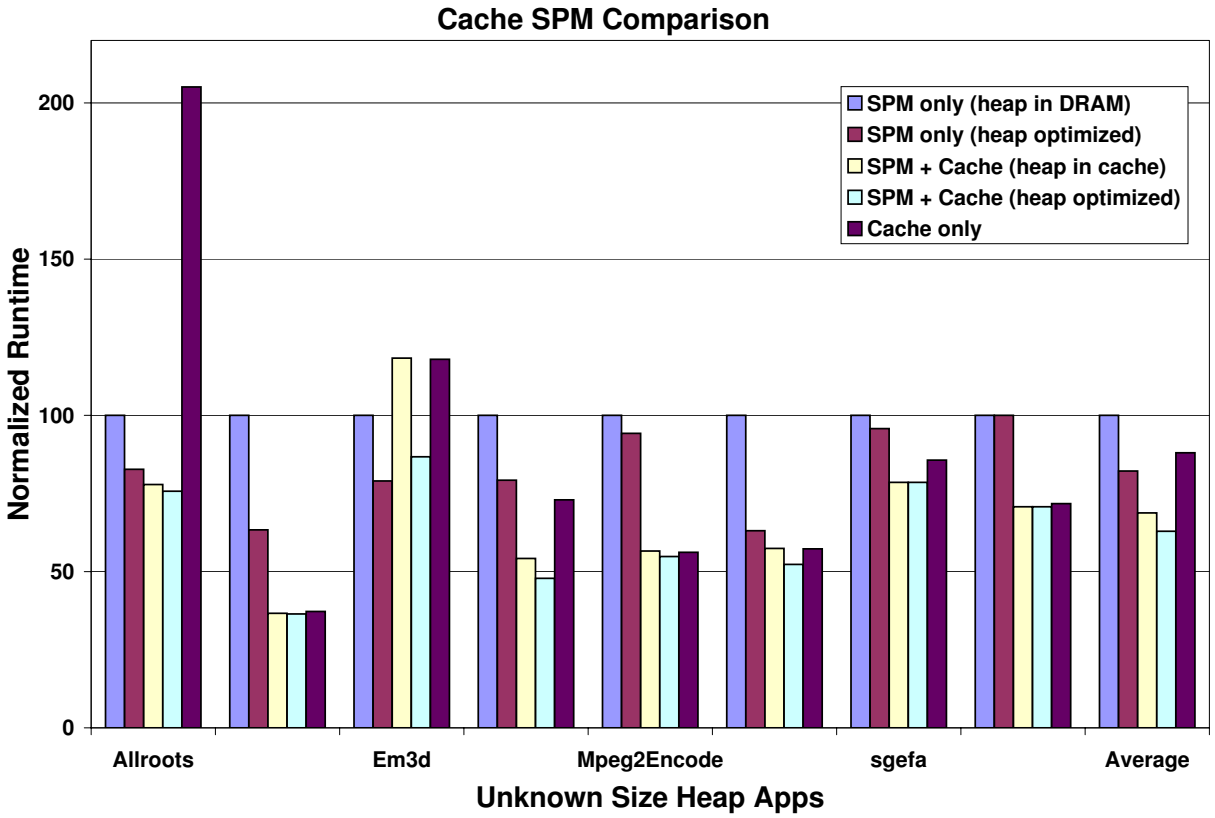


Figure 10.7: Normalized run-times from the unknown-size heap applications set for architectures containing different combinations of SPM and cache.

Figure 10.7 shows the normalized run-time results averaged across our unknown-size heap applications from exploring the different memory configuration and compiler optimizations available.

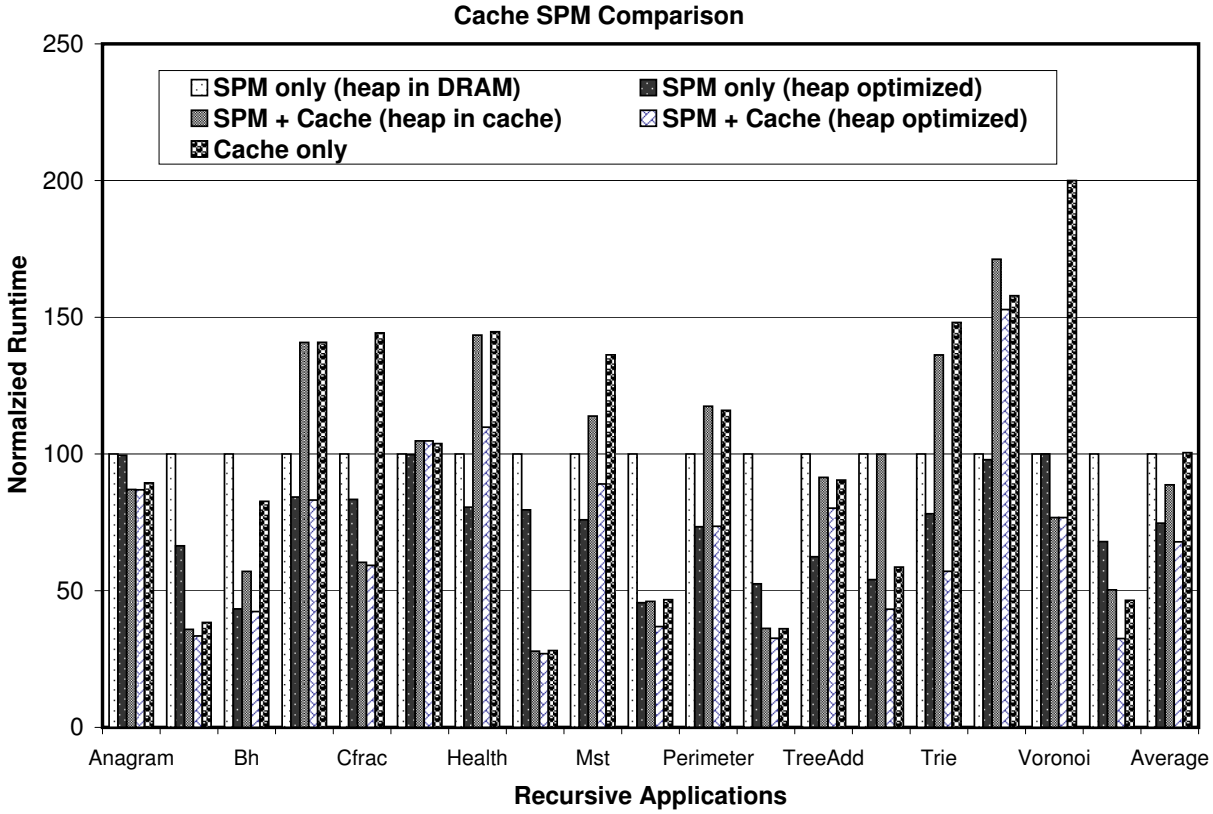


Figure 10.8: Normalized run-times from the Recursive benchmark set for architectures containing different combinations of SPM and cache.

Figure 10.8 shows the normalized run-time results averaged across the recursive benchmark set from exploring the different memory configuration and compiler optimizations available.

Figure 10.9 shows the normalized energy usage results for the JEC benchmark set from exploring the different memory configuration and compiler optimizations available.

Figure 10.10 shows the normalized energy usage for regular heap applications

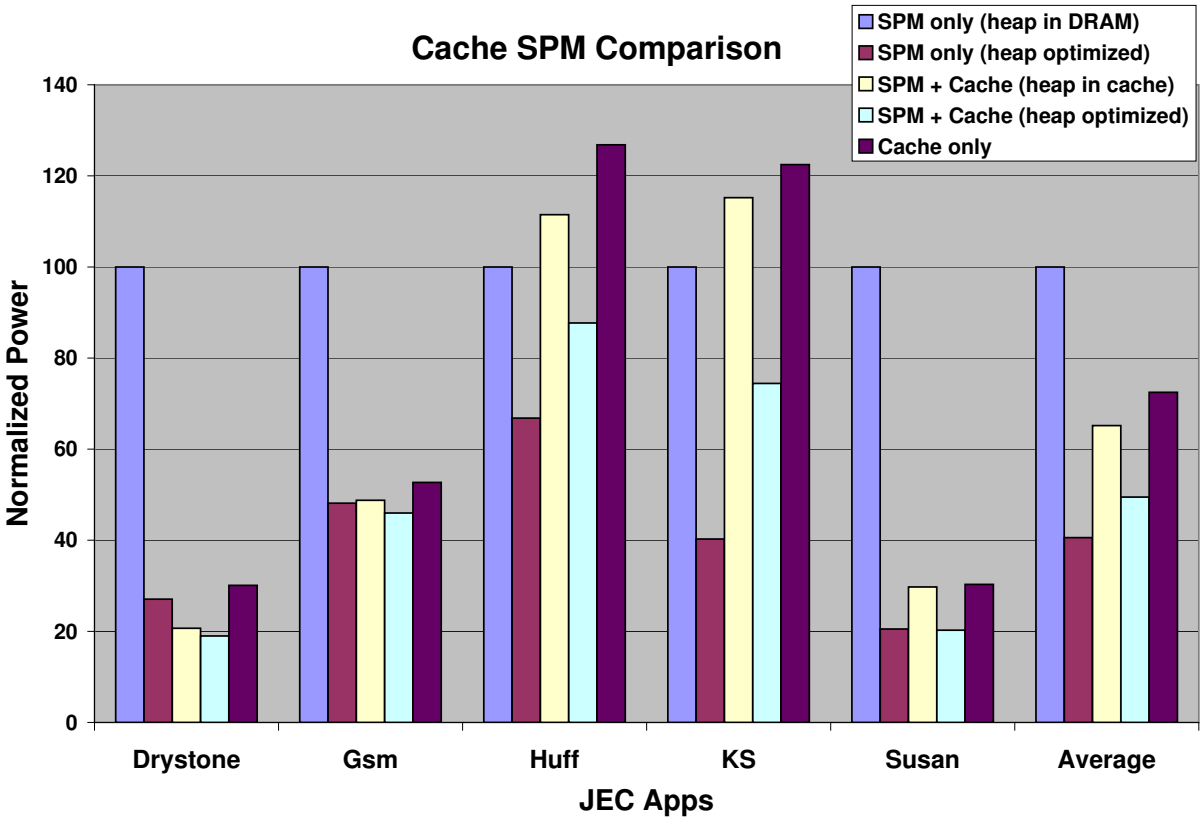


Figure 10.9: Normalized energy usage from the original JEC benchmark set for architectures containing different combinations of SPM and cache.

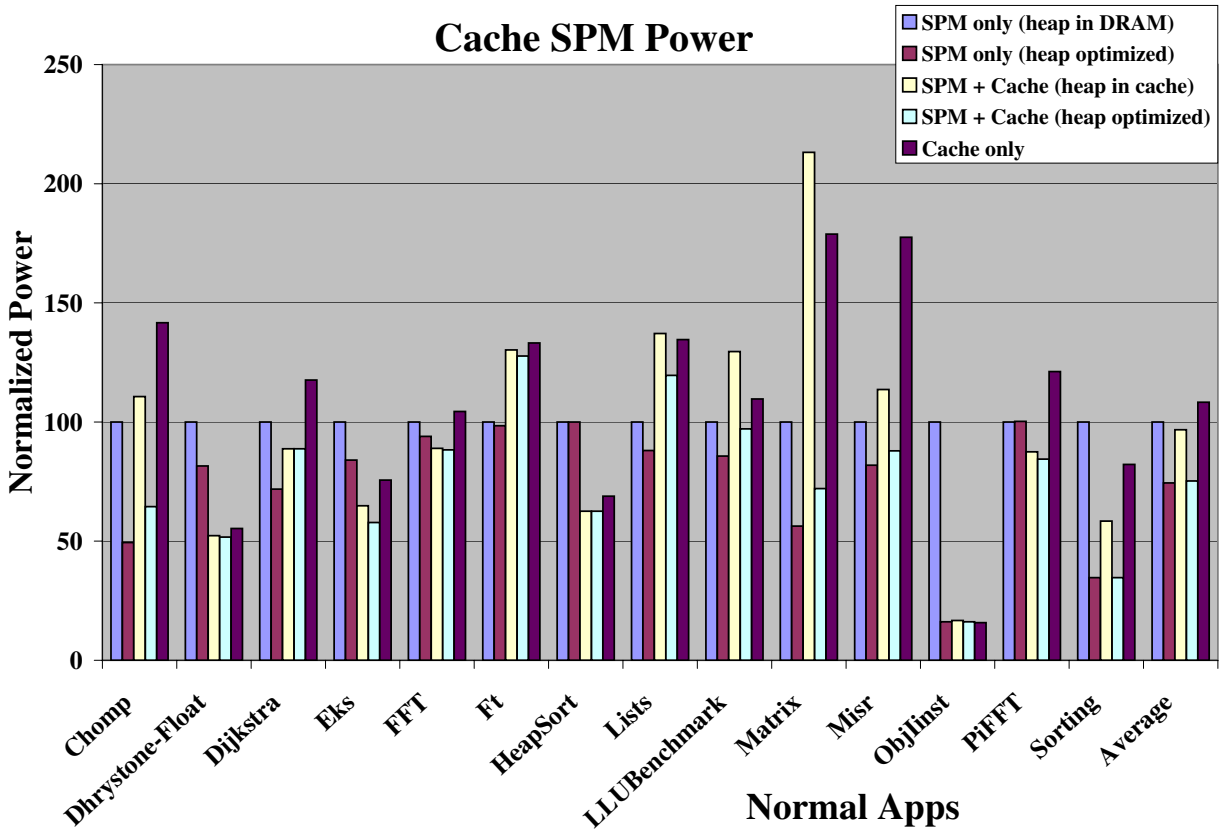


Figure 10.10: Normalized energy usage from the regular heap applications set for architectures containing different combinations of SPM and cache.

from exploring the different memory configuration and compiler optimizations available.

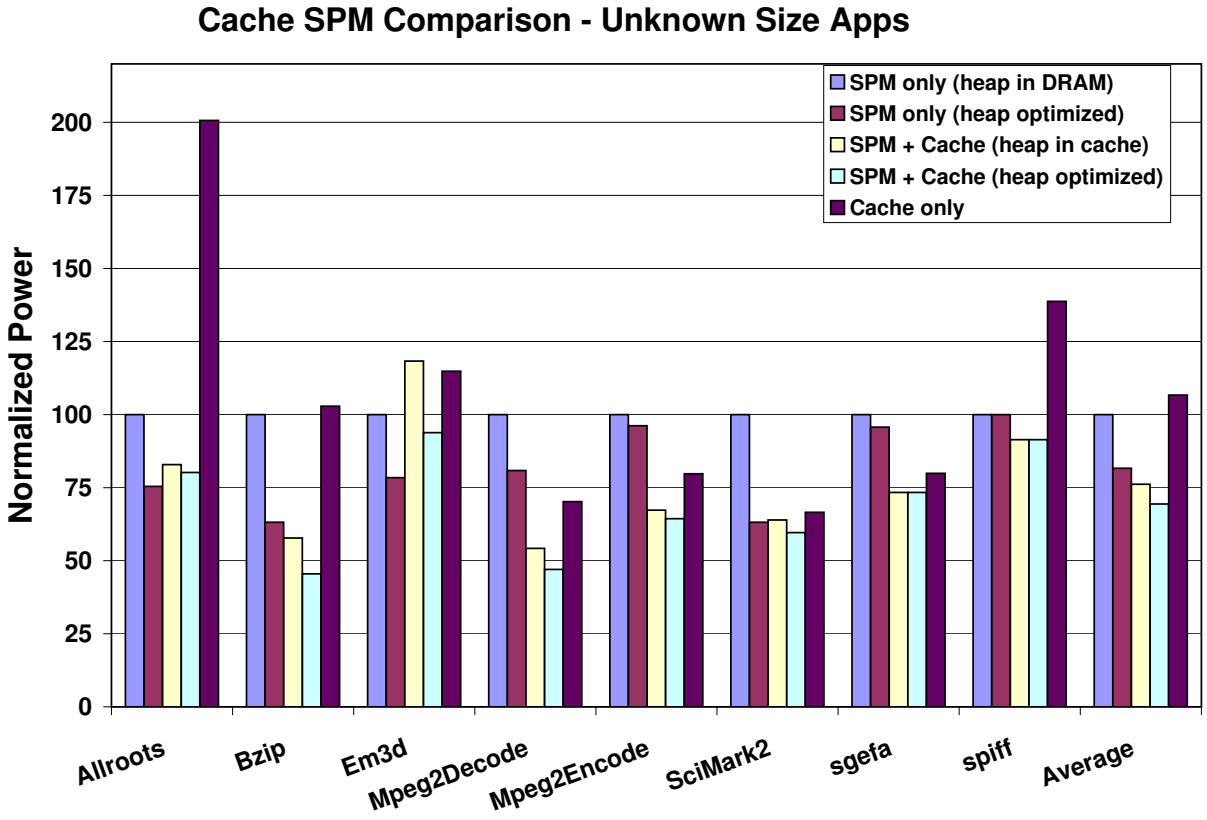


Figure 10.11: Normalized energy usage from the unknown-size heap applications set for architectures containing different combinations of SPM and cache.

Figure 10.11 shows the normalized energy usage results for the unknown-size heap applications from exploring the different memory configuration and compiler optimizations available.

Figure 10.12 shows the normalized energy usage results averaged across the recursive benchmark set from exploring the different memory configuration and compiler optimizations available.

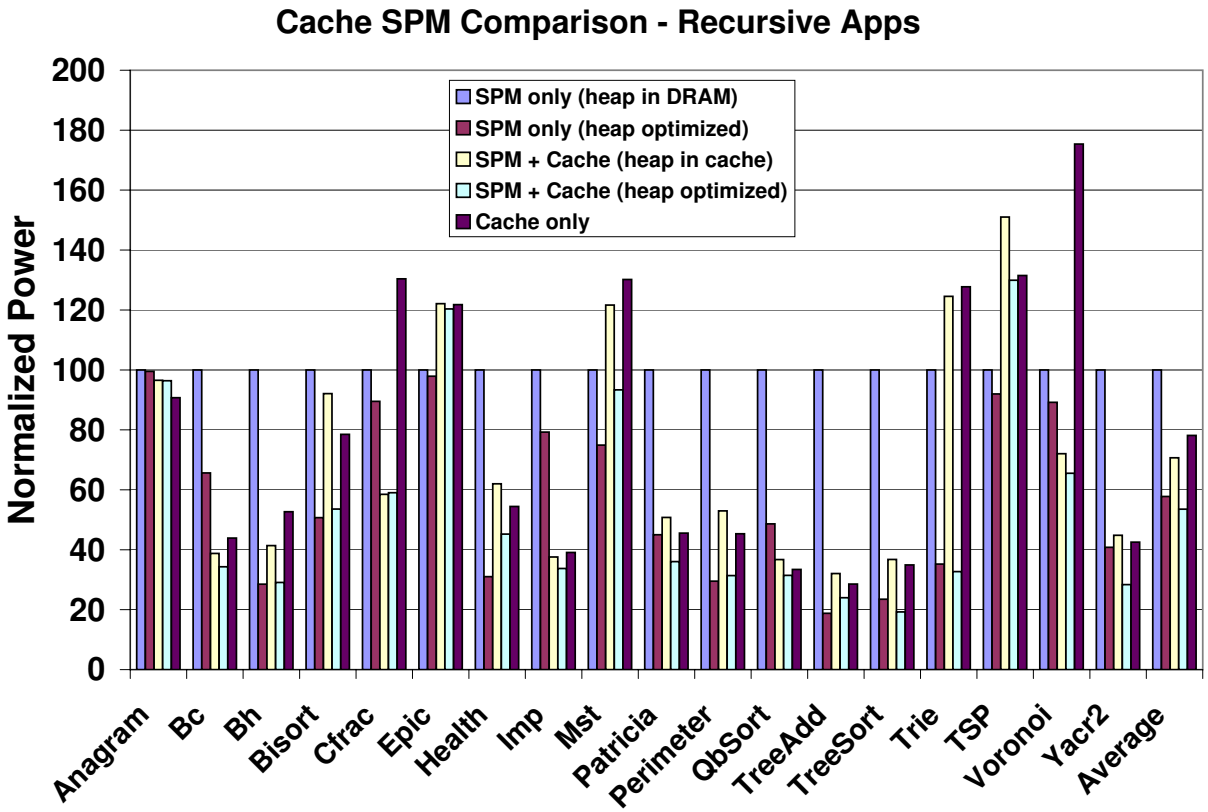


Figure 10.12: Normalized energy usage from the Recursive benchmark set for architectures containing different combinations of SPM and cache.

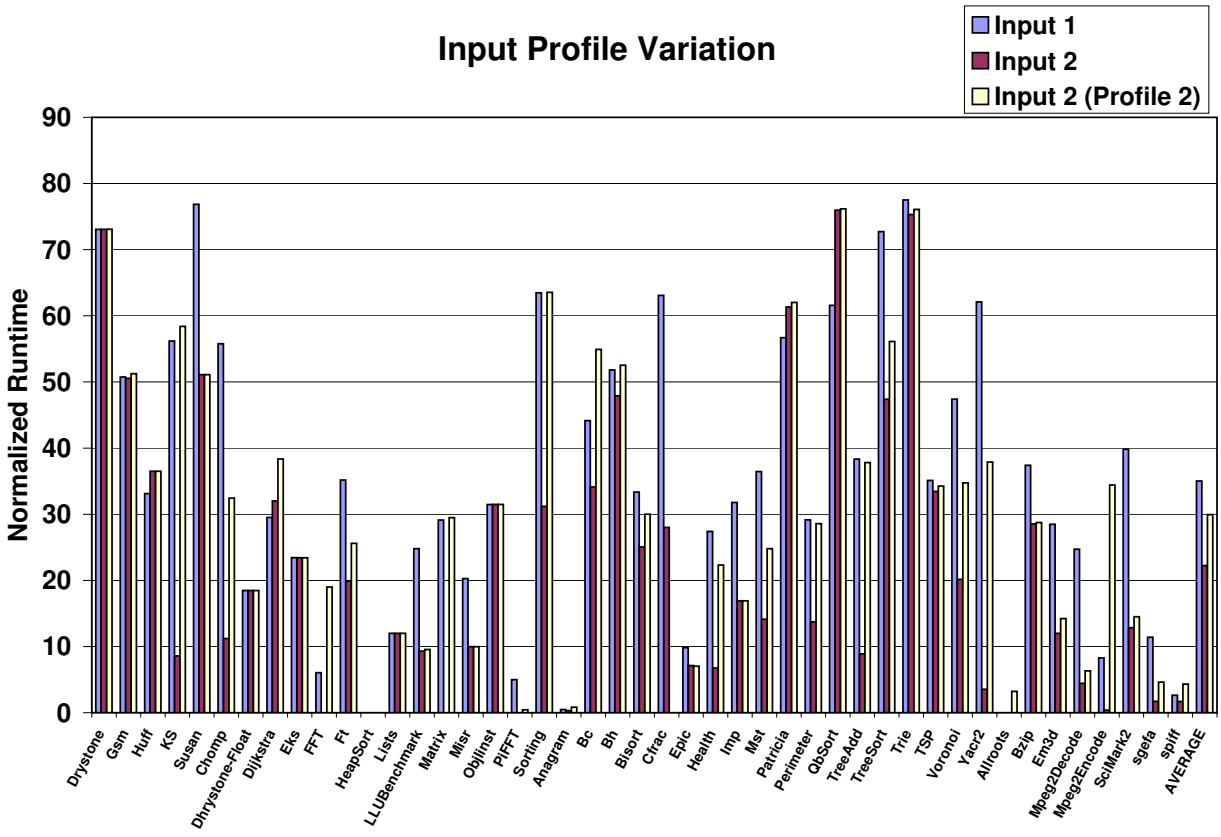


Figure 10.13: Normalized runtime showing profile input sensitivity.

10.3 Profile Sensitivity Results

This section contains further results from the profile sensitivity study performed in Chapter 9, where both experiments have Input Set A applied first followed by input set B to measure variation. Figure 10.13 shows the runtime gains from our profile sensitivity experiments, with detailed results for all applications.

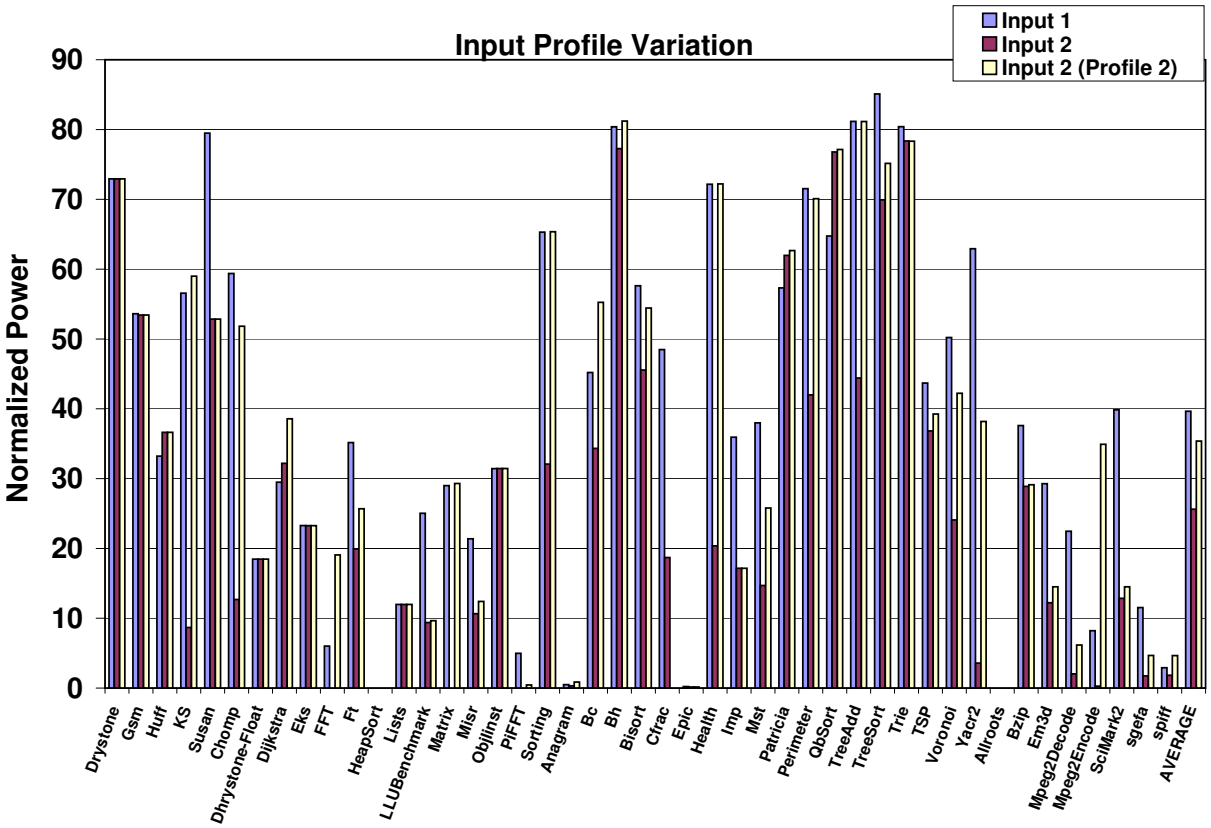


Figure 10.14: Normalized energy usage showing profile input sensitivity.

Figure 10.13 shows the energy savings from the same experiment on profile sensitivity experiments.

Figure 10.13 shows the runtime gains from our profile sensitivity experiments,except that we reverse the order of the inputs(B first, then A).

Figure 10.13 shows the energy savings from the same reversed experiment on

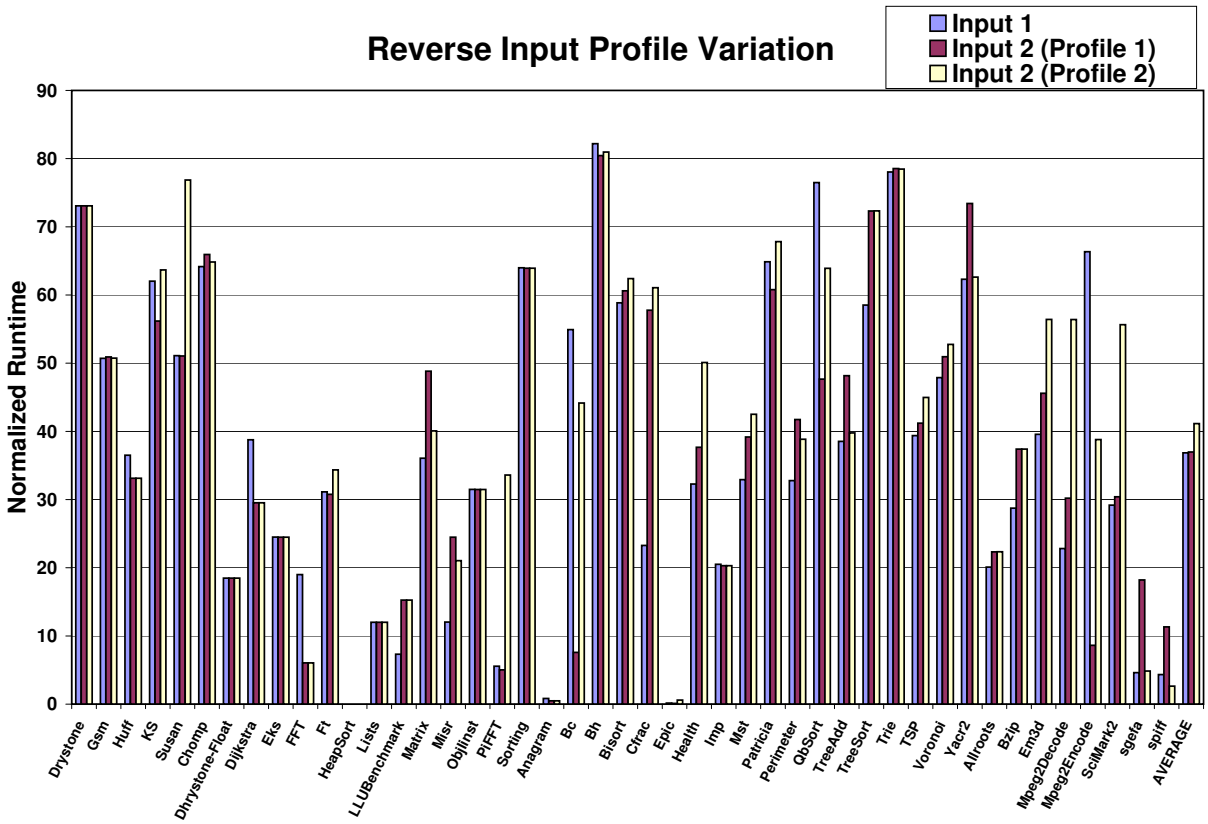


Figure 10.15: Normalized runtime showing profile input sensitivity, with the inputs applied in reverse order.

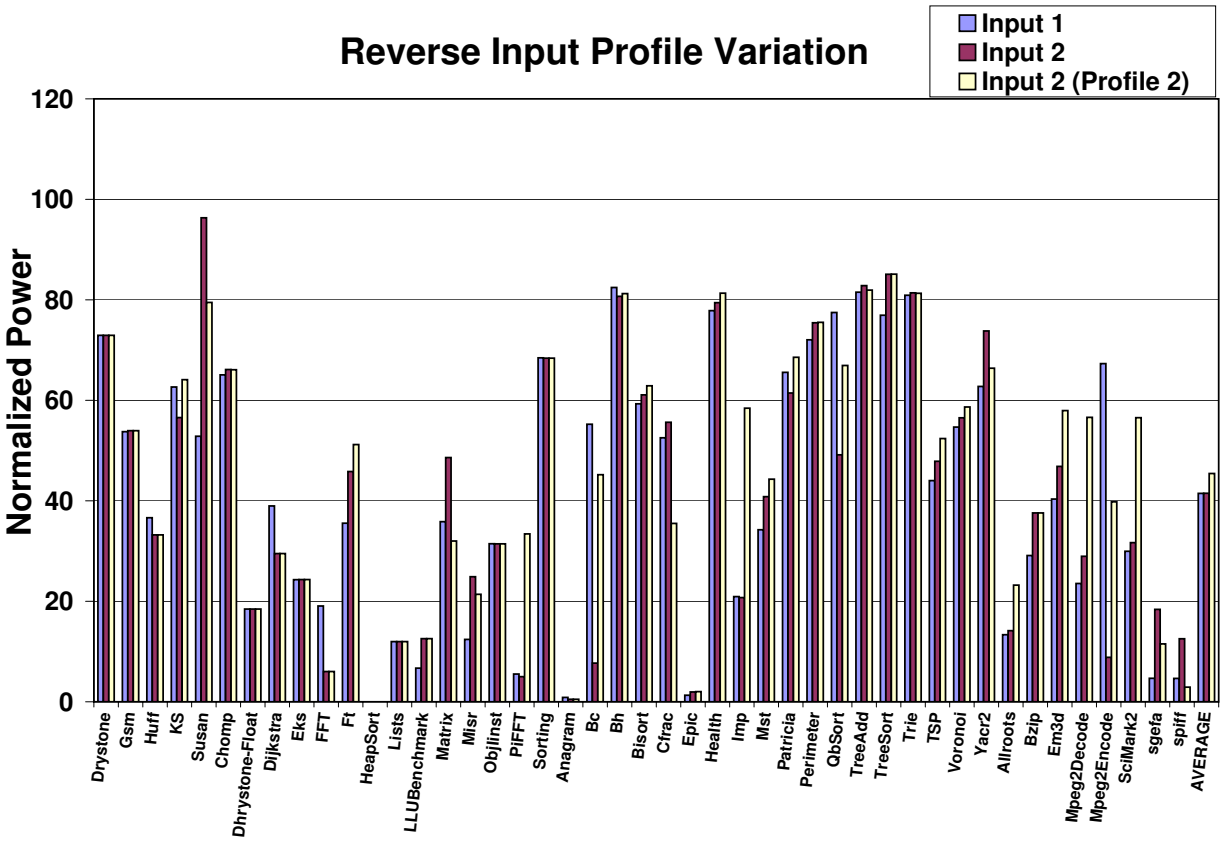


Figure 10.16: Normalized energy usage showing profile input sensitivity, with the inputs applied in reverse order.

profile sensitivity experiments, with input B applied first and then input A.

Bibliography

[1] A. Ignjatovic A. Janapsatya and S. Parameswaran. Adaptive exact-fit storage management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):816–829, 2006.

[2] Javed Absar, Francesco Poletti, Pol Marchal, Francky Catthoor, and Luca Benini. Fast and power-efficient dynamic data-layout with dma-capable memories. In *First International Workshop on Power-Aware Real-Time Computing (PARC)*, 2004.

[3] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratchpad in presence of irregular array access. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1162–1167, Washington, DC, USA, 2005. IEEE Computer Society.

[4] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), August 2002. http://developer.intel.com/technology/itj/2002/volume06issue03/.

[5] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.

[6] *ADSP-21xx 16-bit DSP Family*. Analog Devices, 1996. http://-www.analog.com/processors/processors/ADSP/index.html.

[7] *SHARC ADSP-21160M 32-bit Embedded CPU*. Analog Devices, 2001. http://-www.analog.com/processors/processors/sharc/index.html.

[8] *TigerSharc ADSP-TS201S 32-bit DSP*. Analog Devices, Revised Jan. 2004. http://www.analog.com/processors/processors/tigersharc/index.html.

[9] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pages 318–326. ACM Press, 2003.

[10] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267. ACM Press, 2004.

[11] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, January 1998.

[12] *ARM968E-S 32-bit Embedded Core*. Arm, Revised March 2004. http://-www.arm.com/products/CPUs/ARM968E-S.html.

[13] David Atienza, Stylianos Mamagkakis, Miguel Peon, Francky Catthoor, Jose M. Mendias, and Dimitrios Soudris. Power aware tuning of dynamic memory management for embedded real-time multimedia applications. In *In Proceedings of the XIX Conference on Design of Circuits and Integrated Systems (DCIS '04)*, pages 375–380, 2004.

[14] *Atmel AT91C140 16/32-bit Embedded CPU*. Atmel, Revised May 2004. http://www.atmel.com/dyn/resources/prod_documents/doc6069.pdf.

[15] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the ACM 2nd International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, November 2001. Also at http://www.ece.umd.edu/∼barua.

[16] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS)*, 1(1), September 2002.

[17] R. Banakar, S. Steinke, B-S. Lee, M. Balakrishnan, and P. Marwedel. Scratch-pad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, May 6-8 2002. ACM.

[18] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 1993.

[19] L.A. Belady. A study of replacement algorithms for virtual storage. In *IBM Systems Journal*, pages 5:78–101, 1966.

[20] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[21] E. Berger, B. Zorn, and K. McKinley. Reconsidering custom memory allocation, 2002.

[22] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2001. ACM Press.

[23] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abddsalam Beddaya, and Sulaiman A.Mirdad. Application-level document

caching in the internet. In *Proceedings of the Second Intl. Workshop on Services in Distributed and Networked Environments (SDNE)'95*, pages 125–135, 1990.

[24] R. S. Bird. Notes on recursion elimination. *Commun. ACM*, 20(6):434–439, 1977.

[25] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM Press, 1996.

[26] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37. ACM Press, 1998.

[27] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.

[28] David Brash. *The ARM architecture Version 6 (ARMv6)*. ARM Ltd., January 2002. White Paper.

[29] R. A. Bringmann. *Compiler-Controlled Speculation*. PhD thesis, University of Illinois, Urbana, IL, Department of Computer Science, 1995.

[30] Yun Cao, Hiroyuki Tomiyama, Takanori Okuma, and Hiroto Yasuura. Data memory design considering effective bitwidth for low-energy embedded systems. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 201–206, New York, NY, USA, 2002. ACM Press.

[31] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[32] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[33] G. Chen, I. Kadayif, W. Zhang, M. Kandemir, I. Kolcu, and U. Sezer. Compiler-directed management of instruction accesses. In *DSD '03: Proceedings of the Euromicro Symposium on Digital Systems Design*, page 459, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–75, 2000.

[35] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-Specific Memory Management in Embedded Systems Using Software-Controlled Caches. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[36] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.

[37] Y. Choi and T. Kim. Address assignment combined with scheduling in dsp code generation, 2002.

[38] Yoonseo Choi and Taewhan Kim. Memory layout techniques for variables utilizing efficient dram access modes in embedded system design. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 881–886, New York, NY, USA, 2003. ACM Press.

[39] William D. Clinger. Proper tail recursion and space efficiency. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 174–185, New York, NY, USA, 1998. ACM Press.

[40] *Cacti 3.2*. P. Shivaumar and N.P. Jouppi, Revised 2004. http://-research.compaq.com/wrl/people/jouppi/CACTI.html.

[41] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1998.

[42] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.

[43] H.M. Deitel and P.J. Deitel. *C How To Program*. Prentice Hall, 1994.

[44] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 138–147, New York, NY, USA, 2000. ACM Press.

[45] Document No. ARM DDI 0084D, ARM Ltd. *ARM7TDMI-S Data sheet*, October 1998.

[46] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. *Journal of Embedded Computing(JEC)*, 1:521–540, 2005. IOS Press, Amsterdam, Netherlands.

[47] S. Donahue, M.P. Hampton, R. Cytron, M. Franklin, and K.M. Kavi. Hardware support for fast and bounded time storage allocation. In *Proceedings of the Workshop on Memory Processor Interfaces (WMPI)*, 2001.

[48] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.

[49] N. Dutt. Memory organization and exploration for embedded systems-on-silicon, 1997.

[50] Y. Feng and E. Berger. A locality-improving dynamic memory allocator, 2005.

[51] Poletti Francesco, Paul Marchal, David Atienza, Francky Catthoor Luca Benini, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *In Proceedings of the Design Automation Conference*, pages 238–243. ACM Press, June,2004.

[52] Bill Gatliff. Embedding with gnu: Newlib. *Embedded Systems Programming*, 15(1), 2002.

[53] GNU. *GNU Compiler Collection.* Cambridge, Massachusetts, USA, http://gcc.gnu.org/, 2006. Also available at http://gcc.gnu.org/.

[54] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. In *Software-Practice and Experience*, pages 929–965, 1996.

[55] Peter Grun, Nikil Dutt, and Alex Nicolau. Memory aware compilation through accurate timing extraction. In ACM, editor, *Proceedings 2000: Design Automation Conference, 37th, Los Angeles Convention Center, Los Angeles, CA, June 5–9, 2000*, pages 316–321, New York, NY, USA, 2000. ACM Press.

[56] Peter Grun, Nikil Dutt, and Alex Nicolau. Apex: access pattern based memory architecture exploration. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 25–32, New York, NY, USA, 2001. ACM Press.

[57] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 177–186, New York, NY, USA, 1993. ACM Press.

[58] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[59] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 141–152. ACM Press, 2002.

[60] G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, June 2000.

[61] Pu Hanlai, Ling Ming, and Jin Jing. Extended control flow graph based performance optimization using scratch-pad memory. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 828–829, Washington, DC, USA, 2005. IEEE Computer Society.

[62] Peter Harrison and Hessam Khoshnevisan. Efficient compilation of linear recursive functions into object level loops. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 207–218, New York, NY, USA, 1986. ACM Press.

[63] Laurie J. Hendren, Chris Donawa, and Maryam Emami et all. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, LNCS 757, 1993.

[64] John Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, third edition, 2002.

[65] Vesa Hirvisalo and Sami Kiminki. Predictable timing behavior by using compiler controlled operation. In *4th Intl WORKSHOP ON WORST-CASE EXECUTION TIME (WCET) ANALYSIS*, 2004.

[66] Jason D. Hiser and Jack W. Davidson. Embarc: an efficient memory bank assignment algorithm for retargetable compilers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191. ACM Press, 2004.

[67] *M32R-32192 32-bit Embedded CPU*. Hitachi/Renesas, Revised July 2004. http://documentation.renesas.com/eng/products/mpumcu/rej03b0019_-32192ds.pdf.

[68] *SH7050 32-bit CPU*. Hitachi/Renesas, Revised Sep. 1999. http://-documentation.renesas.com/eng/products/mpumcu/e602121_sh7050.pdf.

[69] C. Huneycutt and K. Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *Proceedings of the International Conference on Parallel Processing*, pages 621–630, 2002.

[70] *The PowerPC 405 Embedded Processor Family.* IBM Inc. Microelectronics, 2002. http://www-306.ibm.com/chips/products/powerpc/processors/.

[71] *The PowerPC 440 Embedded Processor Family.* IBM Inc. Microelectronics, 2002. http://www-306.ibm.com/chips/products/powerpc/processors/.

[72] *XC-166 16-bit Embedded Family.* Infineon, Revised Jan. 2001. http://-www.infineon.com/cmc_upload/documents/036/812/c166sv2um.pdf.

[73] Intel. *Intel StrongARM SA1110 Embedded Procesor*, 2000. http://developer.intel.com/design/pca/applicationsprocessors/1110_brf.htm.

[74] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Data reuse analysis technique for software-controlled memory hierarchies. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10202, Washington, DC, USA, 2004. IEEE Computer Society.

[75] Ilya Issenin and Nikil Dutt. Foray-gen: Automatic generation of affine functions for memory optimizations. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 808–813, Washington, DC, USA, 2005. IEEE Computer Society.

[76] Arun Iyengar. Design and performance of a general-purpose software cache. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[77] Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. Software-assisted cache replacement mechanisms for embedded systems. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 119–126, Piscataway, NJ, USA, 2001. IEEE Press.

[78] Jeff Janzen. Calculating Memory System Power for DDR SDRAM. In *DesignLine Journal*, volume 10(2). Micron Technology Inc., 2001. http://-www.micron.com/publications/designline.html.

[79] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguad. An integer linear programming approach for optimizing cache locality. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 500–509, New York, NY, USA, 1999. ACM Press.

[80] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 628–633, New York, NY, USA, 2002. ACM Press.

[81] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A graph based framework to detect optimal memory layouts for improving data locality, 1999.

[82] Mahmut Kandemir and Ismail Kadayif. Compiler-directed selection of dynamic memory layouts. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 219–224, New York, NY, USA, 2001. ACM Press.

[83] Mahmut T. Kandemir, Ismail Kadayif, and Ugur Sezer. Exploiting scratch-pad memory using presburger formulas. In *ISSS*, pages 7–12, 2001.

[84] Mahmut T. Kandemir, J. Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Design Automation Conference*, pages 690–695, 2001.

[85] Mahmut Taylan Kandemir. A compiler technique for improving whole-program locality. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–192, New York, NY, USA, 2001. ACM Press.

[86] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.*, 2(1-4):151–164, 1993.

[87] Eric Larson and Todd Austin. Compiler controlled value prediction using branch predictor based confidence. In *Proceedings of the 33th Annual International Symposium on Microarchitecture (MICRO-33)*. IEEE Computer Society, December 2000.

[88] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[89] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[90] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 267–269, New York, NY, USA, 1999. ACM Press.

[91] Lin Gao Lian Li and Jingling Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society.

[92] Chi-Keung Luk and Todd C. Mowry. Cooperative instruction prefetching in modern processors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 182–194, November 30-December 2 1998.

[93] V. De La Luz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *DAC '02:*

Proceedings of the 39th conference on Design automation, pages 213–218, New York, NY, USA, 2002. ACM Press.

[94] Mahesh Mamidipaka and Nikil Dutt. On-chip stack based memory organization for low power embedded architectures. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 1082– 1087, 2003.

[95] Peter Marwedel Manish Verma, Lars Wehmeyer. Efficient scratchpad allocation algorithms for energy constrained embedded systems. In *Power-Aware Computer Systems(PACS)*, pages 41–56, 2003.

[96] Peter Marwedel Manish Verma, Stefan Steinke. Data parititioning for maximal scratchpad usage. In *Asia South Pasific Design Automation Conference (ASPDAC)*, 2003.

[97] Peter Marwedel, Lars Wehmeyer, Manish Verma, Stefan Steinke, and Urs Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 4–11, Piscataway, NJ, USA, 2004. IEEE Press.

[98] Lewis J.A. Black B. Lipasti M.H. Avoiding initialization misses to the heap. In *International Symposium on Computer Architecture(ISCA)*, pages 183–194, 2002.

[99] *128Mb DDR SDRAM data sheet*. (Dual data-rate synchronous DRAM) Micron Technology Inc., 2003. http://www.micron.com/products/dram/ddrsdram/.

[100] Csaba Andras Moritz, Matthew Frank, and Saman Amarasinghe. FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *The 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 12 2000.

[101] *CPU12 Reference Manual*. Motorola Corporation, 2000. (A 16-bit processor). http://e-www.motorola.com/brdata/PDFDB/MICROCONTROLLERS/-16_BIT/68HC12_FAMILY/REF_MAT/CPU12RM.pdf.

[102] *M-CORE - MMC2001 Reference Manual*. Motorola Corporation, 1998. (A 32-bit processor). http://www.motorola.com/SPS/MCORE/-info_documentation.htm.

[103] *Coldfire MCF5206E 32-bit CPU*. Motorola/Freescale, Revised 2002. http://-www.freescale.com/files/dsp/doc/fact_sheet/CFPRODFACT.pdf.

[104] *Dragonball MC68SZ328 32-bit Embedded CPU*. Motorola/Freescale, Revised April 2003. http://www.freescale.com/files/32bit/doc/fact_sheet/-MC68SZ328FS.pdf.

[105] *MPC500 32-bit MCU Family*. Motorola/Freescale, Revised July 2002. http://-www.freescale.com/files/microcontrollers/doc/fact_sheet/MPC500FACT.pdf.

[106] O. Ozturk, M. Kandemir, I. Demirkiran, G. Chen, and M. J. Irwin. Data compression for improving spm behavior. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 401–406, New York, NY, USA, 2004. ACM Press.

[107] Krishna V. Palem, Rodric M. Rabbah, III Vincent J. Mooney, Pinar Korkmaz, and Kiran Puttaswamy. Design space optimization of embedded memory systems via data remapping. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 28–37, New York, NY, USA, 2002. ACM Press.

[108] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, 2001.

[109] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proc Eur. Design Test Conf*, pages 7–11, 1997.

[110] P. R. Panda, N. D. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. In *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 3–13, 1999.

[111] P. R. Panda, N. D. Dutt, and A. Nicolau. On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3), July 2000.

[112] Sri Parameswaran and Jrg Henkel. I-copes: fast instruction code placement for embedded systems to improve performance and energy efficiency. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 635–641, Piscataway, NJ, USA, 2001. IEEE Press.

[113] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 116–127. ACM Press, 1992.

[114] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, New York, NY, USA, 2002. ACM Press.

[115] *LPC2290 16/32-bit Embedded CPU.* Philips, Revised Feb. 2004. http://-www.semiconductors.philips.com/acrobat_download/datasheets/LPC2290-01.pdf.

[116] Anand Ramachandran and Margarida F. Jacome. Xtream-fit: an energy-delay efficient data memory subsystem for embedded media processing. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 137–142, New York, NY, USA, 2003. ACM Press.

[117] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 179–190, Washington, DC, USA, 2005. IEEE Computer Society.

[118] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153, New York, NY, USA, 2002. ACM Press.

[119] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R.Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[120] Compilation Challenges for Network Processors. *Industrial Panel, ACM Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, June 2003. Slides at http://www.cs.purdue.edu/s3/LCTES03/.

[121] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[122] Matthew L. Seidl and Benjamin Zorn. Low cost methods for predicting heap object behavior. In *Second Workshop on Feedback Directed Optimization*, pages 83–90, Haifa, Israel, 1999.

[123] Matthew L. Seidl and Benjamin G. Zorn. Implementing heap-object behavior prediction efficiently and effectively. *Software Practice and Experience*, 31(9):869–892, 2001.

[124] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Symposium on Principles of Programming Languages*, pages 295–306, 2002.

[125] Amit Sinha and Anantha Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*, pages 220–225, 2001.

[126] Jan Sjodin, Bo Froderberg, and Thomas Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.

[127] Jan Sjodin and Carl Von Platen. Storage Allocation for Embedded Processors. *Compiler and Architecture Support for Embedded Computing Systems*, November 2001.

[128] R.M. Stallman. *GNU Compiler Collection Internals.* Cambridge, Massachusetts, USA, http://gcc.gnu.org/onlinedocs/gccint, 2002. Also available at http://gcc.gnu.org/onlinedocs/gccint.

[129] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, FL, January 1996.

[130] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*. ACM, 2002.

[131] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, page 409. IEEE Computer Society, 2002.

[132] Angel Dominguez Sumesh Udayakumaran and Rajeev Barua. Dynamic Allocation for Scratch-Pad Memory using Compile-time Decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006.

[133] Andrew S. Tanenbaum. *Structured Computer Organization (4th Edition).* Prentice Hall, October 1998.

[134] Peiyi Tang. Complete inlining of recursive calls: beyond tail-recursion elimination. In *ACM-SE 44: Proceedings of the 44th annual southeast regional conference*, pages 579–584, New York, NY, USA, 2006. ACM Press.

[135] *TMS370Cx7x 8-bit microcontroller.* Texas Instruments, Revised Feb. 1997. http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.

[136] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems (CASES)*, pages 276–286. ACM Press, 2003.

[137] *DineroIV Cache simulator.* J. Edler and M.D. Hill, Revised 2004. http://www.cs.wisc.edu/ markhill/DineroIV/.

[138] Osman S. Unsal, Rakshit Ashok, Israel Koren, C. Manik Krishna, and Csaba Andras Moritz. Cool-cache for hot multimedia. In *Proceedings of the International Symposium on Microarchitecture*, pages 274–283, 1990.

[139] Osman S. Unsal, Raksit Ashok, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. Cool-cache: A compiler-enabled energy efficient data caching framework for embedded/multimedia processors. *Trans. on Embedded Computing Sys.*, 2(3):373–392, 2003.

[140] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *International conference on Hardware/Software Codesign and System Synthesis(CODES+ISIS)*. ACM, 2004.

[141] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, automation and test in Europe*, page 21264. IEEE Computer Society, 2004.

[142] Frdric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2001.

[143] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 26(3):357–374, 1996.

[144] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004)*, 2004.

[145] Lars Wehmeyer and Peter Marwedel. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[146] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

[147] S.J.E. Wilton and N.P. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, 1996.

[148] Qing Yi, Vikram Adve, and Ken Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 169–181, New York, NY, USA, 2000. ACM Press.