Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size

Nghi Nguyen
nghi@eng.umd.edu

Angel Dominguez
angelod@eng.umd.edu

Rajeev Barua
barua@eng.umd.edu

Electrical and Computer Engineering Department
University of Maryland
College Park, MD 20742, USA

ABSTRACT This paper presents the first memory allocation scheme for embedded systems having scratch-pad memory whose size is unknown at compile time. A scratch-pad memory (SPM) is a fast compiler-managed SRAM that replaces the hardware-managed cache. Its uses are motivated by its better real-time guarantees as compared to cache and by its significantly lower overheads in energy consumption, area and access time.

Existing data allocation schemes for SPM all require that the SPM size be known at compile-time. Unfortunately, the resulting executable is tied to that size of SPM and is not portable to processor implementations having a different SPM size. Such portability would be valuable in situations where programs for an embedded system are not burned into the system at the time of manufacture, but rather are downloaded onto it during deployment, either using a network or portable media such as memory sticks. Such post-deployment code updates are common in distributed networks and in personal hand-held devices. The presence of different SPM sizes in different devices is common because of the evolution in VLSI technology across years. The result is that SPM cannot be used in such situations with downloaded code.

To overcome this limitation, this work presents a compiler method whose resulting executable is portable across SPMs of any size. The executable at run-time places frequently used objects in SPM; it considers code, global variables and stack variables for placement in SPM. The allocation is decided by modified loader software before the program is first run and once the SPM size can be discovered. The loader then modifies the program binary based on the decided allocation. To keep the overhead low, much of the pre-processing for the allocation is done at compile-time. Results show that our benchmarks average a 36% speed increase versus an all-DRAM allocation, while the optimal static allocation scheme, which knows the SPM size at compile-time and is thus an un-achievable upper-bound, is only slightly faster (41% faster than all-DRAM). Results also show that the overhead from our embedded loader averages about 1% in both code-size and run-time of our benchmarks.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memories-DRAM, SRAM; B.3.2 [**Memory Structures**]: Design Styles-Cache Memories; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and Embedded Systems; D.3.4 [**Programming Languages**]: Processors-Code Generation, Compilers; E.2 [**Data Storage Representation**]: Linked Representations

General Terms

Performance, Algorithms, Management, Design

Keywords

Memory Allocation, Scratch-Pad, Compiler, Embedded Systems, Downloadable Codes, Embedded Loading, Data Linked List

1. INTRODUCTION

In both desktop and embedded systems, SRAM and DRAM are the two most common writable memory organizations used for program data allocation. 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, a large amount of DRAM is often used to provide low-cost capacity, along with a small-size SRAM to reduce runtime by storing frequently used data. The proper use of SRAM in embedded systems can introduces an average of 2x speedup compared to using DRAM only. This gain 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 [12].

There are two common ways of adding SRAM: either as a hardware-cache or a Scratch Pad Memory (SPM). In desktops systems, caches are the most popular approach. The caching mechanism dynamically stores a subset of the frequently used data in SRAM, satisfying the dynamic behavior of program data. Caches have been a great success for desktops; a trend that is likely to continue in the future. On the other hand, in most embedded systems, the overheads of cache come with serious drawbacks. Cache incurs a significant penalty in area cost, energy, hit latency and real-time guarantees. A detailed recent study [6] compares the tradeoffs of a cache as compared to a SPM. Their results show that a SPM has 34% smaller area and 40% lower power consumption than a cache memory of the same capacity. Further, the runtime with a SPM using a simple static knapsack-based [6] allocation algorithm, was measured to be 18% better as compared to a cache. Thus, defying conventional wisdom, they found absolutely no advantage to using a cache, even in high-end embedded systems where performance is important. Given the power, cost, performance and real time advantages of SPM, it is not surprising that

SPM is the most common form of SRAM in embedded CPUs today. Examples of embedded processor families having SPM include low-end chips such as the Motorola MPC500, Analog Devices ADSP-21XX, Philips LPC2290; mid-grade chips such as the Analog Devices ADSP-21160m, Atmel AT91-C140, ARM 968E-S, Hitachi M32R-32192, Infineon XC166 and high-end chips such as Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, and Motorola Dragonball; there are many others. Trends in recent embedded designs indicate that the dominance of SPM will likely consolidate further in the future [6, 19], for regular as well as network processors.

A great variety of allocation schemes for SPM have been proposed recently [5, 6, 9, 17, 24], but all of them require the SPM size to be known at compile-time. This is because they establish their solutions by reasoning about which data variables and code blocks will fit in SPM at compile-time, which inherently and unavoidably requires knowledge of the SPM size. This has not been a problem for traditional embedded systems where the code is typically fixed at the time of manufacture, usually by burning it into ROM, and is not changed thereafter. There is, however, an emerging and increasing class of embedded systems, where this simple allocation strategy is no longer feasible. These are systems where the code is updated on the embedded system after deployment, through either downloading or portable media, where there is a need for the same executable to run on different implementations of the same ISA. Such a situation is common in networked embedded infrastructure where the amount of SPM is increased every year, due to technological evolutions, as expected by Moore's law. Further, in these systems, code-updates that fix bugs, update security features or enhance functionality are common. Consequently, the downloaded code may not know the SPM size of the processor, and thus is unable to use the SPM properly. This leaves the designers with no choice but to use an all-DRAM allocation or a processor with a cache, in which the well-known advantages of SPMs are lost.

To make code portable across platforms with varying SPM size, one theoretical approach is to recompile the source code separately using all the SPM sizes that exist in practice; and then download all the resulting executables to each embedded node; discover the node's SPM size at run-time; and finally discard all the executables for SPM sizes other than the one actually present. This has several drawbacks. First, many executables need to be broadcast and received, increasing network bandwidth consumption, energy use on any portable devices, and storage requirement on any portable media. Second, the complexity of the system increases and software used to update code becomes significantly larger and more complex. Third, when an unanticipated SPM size (usually larger) is used at a future time, an executable for that size may not be readily available, therefore, may require contacting the vendor who wrote the application source. This is time-consuming at best and impossible if the vendor no longer exists, which is possible since application sources often linger for decades after their first development. It is important to emphasize that this approach is our speculation – we have not found this approach being suggested in the literature, which is not surprising considering its drawbacks listed above. It would be vastly preferable to have a single executable that could run on a system with any SPM size.

Challenges Effectively utilizing memory in SPM-based-embedded systems has always been a challenge. Deriving a memory allocation scheme for such systems when the sizes of SPMs are unknown at compile-time is a greater challenge. Without knowing the size of the SPM at compile-time, it becomes impossible to know which variables could be placed in SPM at runtime. In order to even place a single variable in SPM requires knowing all locations in the binary where that variable is accessed, to be able to update those accesses to the updated SPM address. Extending this to multiple variables with an unknown SPM size becomes challenging. To illustrate, consider a variable A of size 4000 bytes. If the available size of SPM is less than 4000 bytes, this variable A must remain allocated in DRAM at some address, say, 0x8000. Otherwise, A can be effectively allocated to SPM to achieve speedup at some address, say, 0x400. Without the knowledge of the SPM size, the address of A could be either 0x8000 or 0x400, and thus remains unknowable at compile-time. Hence, it becomes difficult to generate an instruction at compile-time that accesses this variable since that requires knowledge of its assigned address. A level of indirection can be introduced in software for each memory access to discover its location first , but that would incur an unacceptably high overhead. An SPM allocation for even a single variable such as A is therefore hard to achieve without knowing the SPM size beforehand.

Method Features and Outline In this paper, we introduce a compiler technique for managing SPM-based-embedded systems, which for the first time, does not require the knowledge of the SPM size at compile time. Our method is described as follows. At compile-time, the compiler analyzes the program to identify all the locations in the code segment of the executable that contain the unknowable offsets and addresses. These locations are the load and store instructions that access the program stack, and all locations that store the addresses of global variables. The compiler then stores the addresses of all these locations along with additional information about each variable accessed, such as their Frequency-Per-Byte (FPB), original stack offsets, original global addresses, and variable sizes as part of the executable for use by the embedded loader. This "original" offset or address of a variable is that in an all-DRAM no-SPM allocation. The FPB of a variable denotes the number of times each variable is accessed in the profile data, divided by its size.

At the next step, when the program is loaded into memory at the beginning of run-time, a modified embedded loader is used. The embedded loader is a set of compiler-inserted routines that execute at the beginning of the application's the first execution, but not subsequent executions. The loader routines perform the following three tasks. First, they discover the size of SPM present on the device, either by making an OS or low-level system call if available on that ISA, or by probing addresses in memory using a binary search pattern and observing the latency to find the range of addresses belonging to SPM. Second, the loader routines compute a suitable allocation to the SPM using its just-discovered size and the FPB of variables. Third, the loader implements the allocation by traversing the locations in the code segment of the executable that have unknown fields and replacing them with the SPM stack offsets and global addresses for the run-time-decided allocation. The resulting executable is now tailored for the SPM size on the target device, and can be executed without any further overhead. The executable can be re-run indefinitely, as is common in embedded systems, with no further overhead.

The embedded loader needs a list of all code locations that contain memory address references for stack and global variables that will be optimized. This list could be appended to the executable; but that would increases its code size significantly. To avoid this increase, our approach stores the locations to-be-updated in a linked list *in-place in those locations itself*. In other words, each to-be-modified location stores the displacement in words to the next to-be-modified location in the executable. These displacements are stored in the bits where the still-unknown stack offsets and global addresses will be stored at run-time. At the start of run-time this in-place linked list is traversed, and after the displacement to the next location is read, it is overwritten with the correct offset or address. Since these lists are stored in the unused bits of the executable file, they cause no increase in code size. Only the first element of each linked list needs to be stored in the executable.

Finally at run-time when the SPM size is known, the allocation of code and data to SPM is decided. Our allocation scheme is able to share space for stack variables with non-overlapping lifetimes. The resulting allocation is implemented when the embedded loader modifies the program code at the start of run-time to correctly refer to the addresses of objects in SPM.

Besides data objects, our method can also place frequently executed code blocks in SPM. They are handled much like global variables in deciding their allocation. The implementation of the allocation, however, requires patching the code to insert a branch to the start of the moved block in SPM from its predecessor blocks; as well as a branch to subsequent blocks from the end of the block in SPM. Our method shows how this patching can be done in the embedded loader.

Results and Paper Overview Our method is implemented on GNU tool chain from CodeSourcery [8] for the ARM v5e embedded processor family [3]. Running our allocator for eight benchmarks, we achieve on average a 36% speedup compared to an all-DRAM, no-SPM allocation; this compares to the 41% speedup achieved by the optimal static allocation scheme [5]. The runtime and code size overheads for our embedded loader are around 1.0% each for a single run of the application, and lower still when amortized across multiple runs. These results indicate that with very low runtime and code-size overhead, our method achieves the goal of generating code that is portable across platforms with different sizes of SPM, while obtaining a performance that is close to that of the unachievable optimal upper bound [5].

The rest of the paper is organized as follows. Section 2 describes the scenarios where our method is useful. Section 3 overviews related work. Section 4 discusses the method in [5] whose allocation we aim to reproduce without the knowledge of the SPM size. Section 5 discusses our method in detail from the profiling stage to embedded loading stage. Section 6 discusses the allocation policy used in the embedded loader. Section 7 describes how program codes are allocated into SPM. Section 8 presents the experimental environment, benchmarks properties, and our method's results. Section 9 concludes.

2. SCENARIOS WHERE OUR PROPOSED METHOD IS USEFUL

Our method is useful in the situations when the code is not burned into ROM at the time of manufacture, but is instead downloaded later onto the systems; and moreover, when due to technological evolution, the code may be required to run on multiple processor implementations of the same ISA having differing amounts of SPM.

One situation where this often occurs is in distributed networks such as a network of ATM machines at financial institutions. Such ATM machines may be deployed in different years and therefore have different sizes of SPM. Code-updates are usually issued to these ATM machines over the network, to update their functionality, fix bugs, or install new security features. In current practice, SPMs cannot be used for such code-updates since they do not know the SPM size. We would like to enable such codes to run on any ATM machine with any SPM size. This is made possible by our method.

Another situation where our technology may be useful is in sensor networks. Examples of such networks are the sensors that detect traffic conditions on roads or the ones that monitor environmental conditions over various points in a terrain. In these long-lived sensor networks, nodes may be added over a period of several years. At the pace of technology evolution today, where a new processor implementation is released every few months, this may represent several generations of processors with increasing sizes of SPM that are present simultaneously in the same network. Our method will allow remote code updates, common in such sensor networks, to use the SPM regardless of its size.

A third example is downloadable programs for personal digital assistants (PDAs), mobile phones and other consumer electronics. These applications may be downloaded over a network or from portable media such as flash memory sticks. These programs are designed and provided independently to the configurations of SRAM sizes on the consumer products. Therefore, to efficiently utilize the SPM for such downloadable software, a memory allocation scheme for unknown size SPMs is much needed. There exists a variety of these downloadable programs on the market used for different purposes such as entertainment, education, business, health and fitness, and hobbies. Real-world examples include games such as Pocket DVD Studio [11], FreeCell, and Pack of Solitaires [27]; document managers such as PhatNotes [18], PlanMaker [22], and e-book readers; and other tools such as Pocket Slideshow [7] and Pocket Quicken [15]. In all these situations, our technology would allow these codes to take advantages of the SPM for the first time.

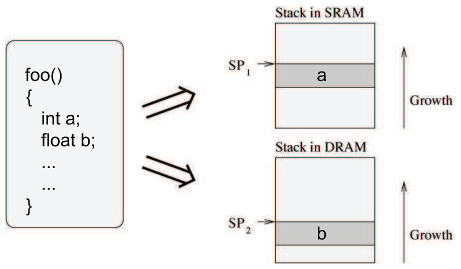
We expect that in the future our technology may eventually even allow desktop systems to use SPM efficiently. One of the primary reasons that caches are popular in desktops is that they deliver good performance for any executable, without requiring it to be customized for any particular cache size. This is in contrast to SPMs, which so far have required customization to a particular SPM size. By freeing programs of this restriction, SPMs can overcome one hurdle to their use in desktops. However, there are still other hurdles to have SPMs become the norm in desktop systems, including that heap data, which our method does not handle, are more common in desktops than in embedded systems. In addition, the inherent advantages of SPM over cache are less important in desktop systems. For this reason we do not consider desktops further in this paper.

3. RELATED WORK

Static methods to allocate data to SPM include [4–6, 13, 17, 20, 21]. Static methods are those whose SPM allocation does not change at run-time. Some of these methods [6,17,20] are restricted to allocating only global variables to SPM, while others [4,5,13,21] can allocate both global and stack variables to SPM. These static allocation methods either use greedy strategies to find an efficient solution, or model the problem as a knapsack problem or an integer-linear programming problem (ILP) to find an optimal solution.

Some static allocation methods [2, 25] aim to allocate code to SPM rather than data. Other static methods [23, 26] can allocate both code and data to SPM. Their data allocation is still restricted to global and stack data only. The goal of the work in [1] is yet another: to map the data in the scratch-pad among its different banks in multi-banked scratch-pads; and then to turn off (or send to a lower energy state) the banks that are not being actively accessed.

Dynamic methods are those which can change the SPM allocation during run-time [9, 16, 24]. The method in [16] 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. The method in [24] is a fully general dynamic method that can place all kinds of global



```
foo()
{
  int a;
  float b;
  ...
  ...
}
```

Figure 1: Example of stack split into two separate memory units. Variables *a* and *b* are placed on SRAM and DRAM respectively. A call to *foo()* requires the stack pointers in both memories to be incremented.

and stack variables in SPM. It uses a whole-program analysis that aims to consider the interactions between neighboring code regions to minimize the transfer of data between SPM and DRAM while maximizing the fraction of data found in SPM. The method in [9] is a dynamic method that is the first SPM allocation method to place a portion of the heap data in the SPM.

All the existing methods discussed above require the compiler to know the size of the SPM. Moreover, the resulting executable is meant only for processors with that size of SPM. Our method is the first to produce an executable that makes no assumptions about SPM size and thus is portable to any possible size.

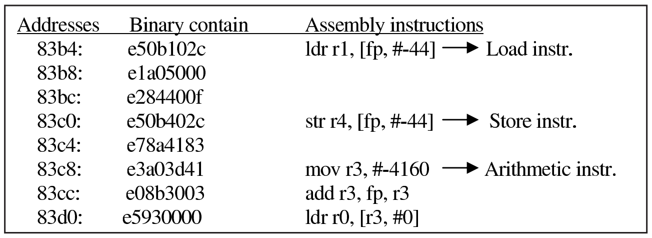
4. BACKGROUND

The allocation strategy used by the loader in our method aims to produce an allocation that is as similar as possible to the optimal static allocation method presented in [5] for global and stack variables. This section outlines their method since our method builds upon its foundation.

The allocation in [5] is as follows. In effect, for global variables, the ones with highest FPB are placed in SPM. For stack variables, to allow variables in the same stack frame to be allocated to different memories (SPM vs DRAM), a *distributed stack* is used. Here the stack is partitioned into two stacks for the same application: one for SPM and the other for DRAM. Each stack frame is partitioned, and two stack pointers are maintained, one pointing to the top of the stack in each memory. An example of how a single stack frame is distributed is shown in figure 4. The allocator places the frequently used stack variables in the SPM stack, and the rest are in the DRAM stack. In this way, at run-time, only the frequently-used stack variables (such as variable *a* in figure 4) appear in SPM.

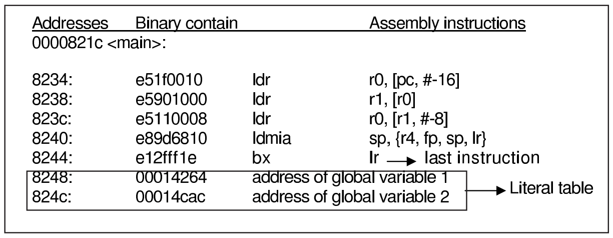
The method in [5] formulates the problem of searching the space of possible allocations with an objective function and a set of constraints. The objective function to be minimized is the expected run-time with the allocation, expressed in terms of the proposed allocation and the profile-discovered frequency-per-bytes (FPBs) of the variables. The constraints are that for each path through the call graph of the program, the size of the SPM stack fits within the SPM's size. This constraint automatically takes advantage of the limited lifetime of stack variables: if *main()* calls *f1()* and *f2()*, then the variables in *f1()* and *f2()* share the same space in SPM, and the constraint correctly estimates the stack height in each path. As we shall see later, our method also takes advantage of the limited lifetime of stack variables.

This search problem is solved in two ways: using a greedy search and a provably optimal search based on Integer-Linear Programming (ILP). Results show that both approaches produce good results, with the ILP solution doing only marginally better. Therefore the greedy solver is also near-optimal. For this reason, and since the greedy solver is much more practical in real compilers, in our



```
Addresses    Binary contain        Assembly instructions
  83b4:       e50b102c             ldr r1, [fp, #-44] ⟶ Load instr.
  83b8:       e1a05000
  83bc:       e284400f
  83c0:       e50b402c             str r4, [fp, #-44] ⟶ Store instr.
  83c4:       e78a4183
  83c8:       e3a03d41             mov r3, #-4160  ⟶ Arithmetic instr.
  83cc:       e08b3003             add r3, fp, r3
  83d0:       e5930000             ldr r0, [r3, #0]
```

Figure 2: Stack Variable access locations in the executable file. Instruction "ldr rx, [fp, c]" $\equiv$ rx $\leftarrow$ mem[fp+c] and "str rx, [fp, c]" $\equiv$ mem[fp+c] $\leftarrow$ rx



```
Addresses   Binary contain                     Assembly instructions
0000821c <main>:

8234:       e51f0010      ldr                  r0, [pc, #-16]
8238:       e5901000      ldr                  r1, [r0]
823c:       e5110008      ldr                  r0, [r1, #-8]
8240:       e89d6810      ldmia                sp, {r4, fp, sp, lr}
8244:       e12fff1e      bx                   lr ⟶ last instruction
8248:       00014264      address of global variable 1
824c:       00014cac      address of global variable 2  ⟶ Literal table
```

Figure 3: Global Variable Address Locations. Instruction "ldmia rx,[ry,rz,rw]' ' $\equiv$ ry $\leftarrow$ mem[rx], rz $\leftarrow$ mem[rx+4], rw $\leftarrow$ mem[rx+8]

evaluation we use the greedy solver for both the method in [5] and in the off-line component of our method, although either could be used.

5. METHOD

Our method's goal is to achieve as close an allocation as possible to that in Avissar et al. [5]. In particular, we will emulate the run-time behavior of [5] to the extent possible including its use of a distributed stack. However, our mechanisms at compile-time and loading-time will be significantly different from their method, and more complicated, because of the difficulties introduced by not knowing the SPM's size at compile-time.

Our method introduces modifications to the profiling, compilation, linking and loading stages of code development. The tasks at each stage are described below.

Profiling Stage The application is run multiple times with different inputs to collect the number of accesses for each variable in the program for each input; and an average is taken. These numbers of accesses represent how frequently a variable is accessed in the application. Next, this frequency of each variable is divided by its size in bytes to yield its FPB. Intuitively, variables with higher FPB should have higher priority for placement in SPM. Thereafter, a list of variables sorted in decreasing order of their FPBs is created. This list is stored in the output of the profiling stage; along with each variable is also stored its FPB and its size.

Compiling Stage Since the SPM size is unknown, we do not fix the allocation at compile-time, but delay the assignments of variable addresses and offsets until runtime. Various types of preprocessing are done in the compiler to reduce the embedded loader overhead. These are described next.

As the first step, the compiler analyzes the program to identify all the code locations that contain data which is unknown due to not knowing the SPM's size at compile time. These locations are the load and store instructions that access program stack variables,

and all locations that store the addresses of global variables.

Addresses	Binary contain	Assembly instructions	
83b4:	e50b1*02c*	ldr r1, [fp, #-44]	16
83b8:	e1a05000		
83bc:	e284400f		
83c0:	e50b4*02c*	str r4, [fp, #-44]	8
83c4:	e78a4183		
83c8:	e51b3*02c*	ldr r3, [fp, #-44]	

Figure 4: Before Modification

Addresses	Binary contain	Assembly instructions
83b4:	e50b1*010*	ldr r1, [fp, #16]
83b8:	e1a05000	...
83bc:	e284400f	
83c0:	e50b4*008*	str r4, [fp, #8]
83c4:	e78a4183	
83c8:	e51b3*008*	ldr r3, [fp, #8]

Figure 5: After Modification

These locations are identified by their addresses in the executable file.

Let us consider how stack accesses are handled. For the ARM architecture on which we performed our experiments, the locations in the executable file that affect the stack offset assignments are the load and store instructions that access the stack variables, and the arithmetic instructions that calculate their offsets. In the usual case, when the stack offset value is small enough to fit into the immediate field of the load/store instruction, these load and store instructions are the only ones that affect the stack offset assignments. The first **ldr** and the subsequent **str** instructions in Figure 2 illustrate two accesses of this type, where the stack offset value of -44 from the frame pointer (*fp*) fits in the 12-bit immediate field of the load/store instructions in ARM.

In some rare cases, when the stack offset value is larger than the range of the immediate field of load/store instruction, additional arithmetic instructions are needed to calculate the correct offset of a stack variable. Such cases arise for procedures with frame sizes that are larger than the span of the immediate field of the load/store instructions. In ARM, this translates to stack offsets larger than $2^{12} = 4096$ bytes. In these rare cases, the stack offset is first moved to a register and then added to the frame pointer. An example is seen in the three-instruction sequence (**mov**, **add**, **ldr**) at the bottom of figure 2. Since the **mov** instruction allows 16-bit immediates, stack offsets of up to 64Kbytes are allowed in this addressing mode. So the offset shown (-4160) fits in the immediate field of the **mov** instruction. In this case, only the **mov** instruction needs to be added to the linked list of locations with unknown immediate fields that we maintain, since only its field needs to be changed by the embedded loader.

For global variables in ARM, the addresses are stored in the literal tables. Literal tables are locations in the code segment which contain the full 32-bit addresses of global variables in the program. In ARM, they reside just after each procedure in the executable file. In a rare situation, the literal tables can also appear in the middle of the code segment of a function with a branch instruction jumping around it for the sake of program correctness. This situation occurs only when the code length of a function is larger than the range of the load immediates used in the code to access the literal tables. An example of a literal table is presented in the figure 3.

After identifying the locations that need to be modified by the embedded loader – those are locations containing stack offsets and global addresses – the compiler creates a linked-list of such locations for each variable for use in the linking stage. This compiler linked-list is not yet the in-place linked list stored in the instructions themselves. It is, however, used later to establish the actual in-place linked-list at linking time, when the exact displacements of the to-be-modified locations are known.

The compiler also analyzes the limited lifetimes of the stack variables to determine the additional sets of variables for allocating into SPM for each cut-off point. Details of the allocation policy and lifetime analysis are presented in section 6. Finally the compiler inserts the embedded loader routines into the code. A part of the loader is code that will find the SPM vs. DRAM allocation at run-time.

Linking Stage At the end of the linking stage, to avoid significant code-size overhead, we store the linked-list of all locations in code segment with unknown immediate values in-place in the locations themselves. This is possible since the locations will be overwritten with the correct immediate values only at the start of run-time, and until then, they can be used to store the displacement to the next element in the list, expressed in words. To achieve this, at the end of the linker stage, the linker traverses the compiler-generated linked lists, and converts them to the in-place format. This is possible in the linker stage since the exact displacements in the executable of all locations is now known. The addresses of the first locations in the linked-lists are also stored elsewhere in a table in the executable to be used at run-time as the starting addresses of the linked-lists. With this technique, the linked lists can be easily traversed at the start of run-time.

An example of the code conversion in the linker is shown in figures 4 and 5. Figure 4 shows the output code from the compiler with the stack offsets assuming an all-DRAM allocation. Figure 5 shows the same code after the linker converts the separately stored linked lists to in-place linked lists in the code. Each instruction now stores the displacement to the next address.

The in-place linked list representation is possible because in most cases the bit-width of the immediate fields is sufficient to store the displacement to the next access of a variable. For example, for stack accesses in ARM, the immediates are either 12 or 16-bit as described earlier, which yield an allowed displacement to the next instruction of 4096 or 64K words. The presence of these multiple widths in the same linked list causes no problem since the loader will look at the instruction opcode to know which is used (**ldr**/**str** $\Rightarrow$ 12 bits; **mov** $\Rightarrow$ 16 bits). In most cases, the next use of a variable is close to the current use, so this displacement is adequate. In the rare case it is not, a new linked list is created for this same variable and handled identically thereafter.

For global variables, the literal table entries are 32-bits wide. In a 32-bit address space, this is wide enough to store all possible displacements, so a single linked list is always adequate for each global variable.

Although our method above is illustrated with the example of the ARM ISA, it is applicable to most embedded ISAs. To apply our method for any other ISA, the locations in the program code that store the immediates for stack offsets and global addresses must be identified and stored in the linked lists. The exact widths of the immediate fields may differ from ARM, leading to more or fewer linked lists than in ARM. However because accesses to the same variable are often close together in the code, the number of linked lists is expected to remain small.

Embedded Loader The embedded loader is implemented in a set of compiler-inserted codes that are executed just after the program

```
Define:
A: is the list of all global and stack variables in decreasing FPB order
Greedy_Set: is the set of variables allocated greedily to SPM
Limited_Lifetime_Bonus_Set: is the limited-lifetime-bonus-set of variables SPM
GREEDY_SIZE: is the cumulated size of greedily allocated variables to SPM at each cutoff point
BONUS_SIZE: is the cumulated size of variables in limited-lifetime-bonus-set
MAX_HEIGHT_SPM_STACK: the maximum height of the SPM stack during lifetime of current variable

void            Find_allocation(A) { /* Run at compile-time */
1.    for (i = beginning to end of FPB list A) {
2.        GREEDY_SIZE ← 0; BONUS_SIZE ← 0;
3.        Greedy_Set ← NULL; Limited_Lifetime_Bonus_Set ← NULL;
4.        for (j = 0 to i) {
5.            GREEDY_SIZE ← GREEDY_SIZE + size of A[j]; /* j^th variable in FPB list */
6.            Add A[j] to the Greedy_Set;
7.        }
8.        Call Find_limited_lifetime_bonus_set(i, GREEDY_SIZE);
9.        Save Limited_Lifetime_Bonus_set for cut-off at variable A[i] in executable;
10.   }
11.   return;}

void            Find_limited_lifetime_bonus_set(cut-off-point, GREEDY_SIZE) {
12.   for (k = cut-off-point to end of FPB list A) {
13.       Add stack variables in Greedy_Set ∪ Limited_Lifetime_Bonus_Set to SPM stack;
14.       if (A[k] is a stack variable) {
15.           Find MAX_HEIGHT_SPM_STACK among all call-graph paths from main() to leaf procedures that go through procedure containing A[k];
16.       } else {          /* A[k] is global variable */
17.           Find MAX_HEIGHT_SPM_STACK among all call-graph paths from main() to leaf procedures;
18.       }
19.       ACTUAL_SPM_FOOTPRINT ← (Size of globals in Greedy_Set ∪ Limited_Lifetime_Bonus_Set) + MAX_HEIGHT_SPM_STACK;
20.       if (GREEDY_SIZE - ACTUAL_SPM_FOOTPRINT ≥ size of A[k]) {        /* L.H.S. is over-estimate amount */
21.           add A[k] into the Limited_Lifetime_Bonus_Set
22.           BONUS_SIZE ← BONUS_SIZE + size of A[k];
23.       }
24.   }
25.   return;}
```

Figure 6: Compiler pre-processing pseudo-code that finds Limited_Lifetime_Bonus_Set at each cut-off

is loaded in memory. As a part of the executable, it is executed by an OS call to a loader routine in the executable, or by an application call at the start of main(). It is executed only before the first time the program is run, and not before subsequent runs; these can be differentiated by a persistent *is-first-time* boolean variable in the loader routine. In this way, the overhead of the embedded loader is encountered only once even if the program is re-run indefinitely, as is common in embedded systems.

The loader routines perform the following three tasks. First, they discover the size of SPM present on the device, either by making an OS system call if available on that ISA, or by probing addresses in memory using a binary search and observing the latency to find the range of addresses in SPM. Second, the loader routines compute a suitable allocation to the SPM using its just-discovered size and the frequency-per-byte of variables. The details of the allocation are described in section 6. Third, the loader implements the allocation by traversing the locations in the code that have unknown fields and replacing them with the stack offsets and global addresses for the run-time-decided allocation. The resulting executable is now tailored for the SPM size on that device, and can be executed without any further overhead.

Although the embedded loader name implies that it is part of the already-provided default system loader, it does not have to be. Indeed the system loader need not be modified at all in our framework. Instead the embedded loader is usually implemented as a set of routines that are executed from inside the application at its beginning; the routines themselves can be stored as part of the application or in a library. Nevertheless since its functionality is closer in spirit to a loader, we feel the "loader" name is appropriate.

6. ALLOCATION POLICY IN EMBEDDED LOADER

The SPM-DRAM allocation is decided by the embedded loader using the run-time discovered SPM-size, the frequency-per-byte (FPB) of each variable and additional pre-processing information about limited lifetimes of stack variables that the compiler provides. The greedy profile-driven cost allocation in the loader is as follows. The embedded loader traverses the list of all global and stacks variables stored by the compiler in a decreasing order of their FPBs, placing variables into SPM, until the cumulative size of the variables allocated so far exceeds the SPM size. This point in the list is called its *cut-off* point.

We observe, however, that the SPM may not actually be full on each call graph path at the cut-off point because of the limited lifetimes of stack variables. For example, if *main()* calls *f1()* and *f2()*, then the variables in *f1()* and *f2()* can share the same space in SPM since they have non-overlapping lifetimes, and simply cumulating their sizes over-estimates the maximum height of the SPM stack. Thus the greedy allocation under-utilizes the SPM.

Our method uses this opportunity to allocate an additional set of stack variables in SPM to utilize the remaining SPM space. We call this the *limited-lifetime-bonus-set* of variables to place in SPM. To avoid an expensive search at loading time, this set is computed off-line by the compiler and stored in the executable for each possible cut-off point in the FPB-sorted list. Since the greedy search can cut-off at any variable, a bonus set must be pre-computed for each variable in the program. Once this list is available to our embedded loader at the start of run-time, it implements its allocations in the same way as for other variables. In this way, our method takes advantage of the limited lifetimes of stack variables.

The compiler algorithm to compute the *limited-lifetime-bonus-set* of variables at each cut-off point in the FPB list is presented in figure 6. Lines 1-11 show the main loop traversing the FPB-sorted list in decreasing order of FPB. Lines 4-7 find the greedy allocation for the cut-off point at variable i. Line 8 makes the call to a rou-

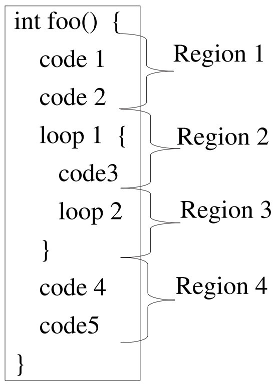


Figure 7: Program code is divided into code regions

tine to find the *limited-lifetime-bonus-set* at this cut-off point; the routine is in lines 12-25. Then the unutilized space in SPM is computed as the difference of the greedily-estimated size and the actual memory footprint (line 20), which may be lower because of limited lifetimes. Additional variables are then found to fill this space in decreasing order of FPB among the remaining variables. This search of bonus variables considers the stack allocation only along paths through the current variable's procedure if it is a stack variable (line 15); therefore it does not itself over-estimate the memory footprint.

Two factors reduce the code-size increase from storing the bonus sets at each cut-off. First, the bonus sets are stored in bit-vector representation on the set of variables, and so are extremely compact. Second, in a simple optimization, instead of defining cut-offs at each variable, a cut-off is defined at a variable only if the cumulative size of variables from the previous cut-off exceeds CUT_OFF_THRESHOLD, a small constant currently set at 10 words. This avoids defining a new cut-off every time a single scalar variable is considered; instead groups of adjacent scalars with similar FPBs to be considered together for purposes of computing a bonus set. This can reduce the code space increase by up to a factor of 10, with only a small cost in SPM utilization.

7. CODE ALLOCATION

Our method allows program code to be allocated to SPM in similar manner to data. Code is considered for placement in SPM at the granularity of *regions*. For this reason, the program code is partitioned into regions. Some criteria for a good choice of regions are (i) the regions should not too big to allow fine-grained consideration of placement of code in SPM; (ii) the regions should not be too small to make for a very large search problem and excessive patching of code; (iii) the regions should correspond to significant changes in frequency of access, so that regions are not forced to allocate infrequent code in them just to bring their frequent parts in SPM; and (iv) except in nested loops, the regions should contain entire loops in them so that the patching at the start and end of the region is not inside a loop, and therefore has low overhead. With these considerations, we define a new region to begin at (i) the start of each procedure; and (ii) just before the start, and at the end, of every loop (even inner loops of nested loops). Other choices are possible, but we have found this heuristic choice to work well in practice.

An example of how code is partitioned into regions is in Figure 7. As the following step, each region's profiled data such as size, FPB, start and end addresses are collected at the profiling stage along with profiled data for program variables.

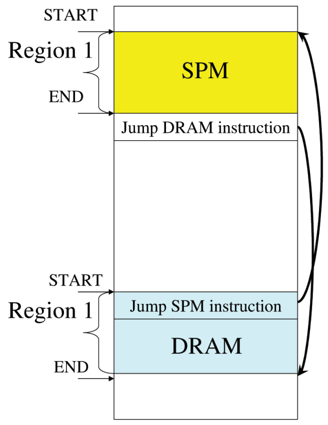


Figure 8: Jump instruction is inserted to redirect control flow between SPM and DRAM

Since code regions and global variables have the same lifetime characteristics, code allocation, therefore region allocation, is decided at embedded loading time using the same allocation policy as global variables. The greedy profile-driven cost allocation in the embedded loader is modified to include code regions as follows. The embedded loader traverses the list of all global variables, stacks variables, and code regions stored by the compiler in a decreasing order of their FPBs, placing variables and transferring code regions into SPM, until the cumulative size of the variables and code allocated so far exceeds the SPM size. At this cut-off point, an additional set of variables and code regions, which are established at compile time by the limited-lifetime-bonus-set algorithm for both data variables and code regions, are also allocated to SPM. The limited-lifetime-bonus-set algorithm is modified to include code regions, which are treated as additional global variables.

Since relocating a region of program code into SPM can break the control flow of the program, code-patching is needed at several places to ensure that the code with SPM allocation is functionally correct. Figure 8 shows the patching needed. At embedded loading time, for each code region that is transferred to SPM, our method inserts a jump instruction at the original DRAM address of the start of this region. The copy of this region in DRAM becomes unused DRAM space[1]. Upon reaching this loading-time inserted instruction, execution will jump to the SPM address this region is assigned, thereby, redirecting all incoming execution paths of this regions to the correct address in SPM.

Similarly, we also insert a patching instruction as the last instruction of the SPM allocated code region, which redirects program flow back to DRAM. The distance from the original DRAM space to the newly allocated SPM space of the transferring region usually fits into the immediate field of the jump instructions. In the ARM architecture, which we use for evaluation, jump instructions have a 24-bit offset which is large enough in most cases. In the rare cases that the offset is too large to fit in the space available in the jump instruction, a longer sequence of instructions is needed for the jump; this sequence first places the offset into a register and then jumps to the contents of the register.

Besides incoming and outgoing paths, side entries and side exits of the optimized regions also need modification to ensure correct

[1]We do not attempt to recover this space since it will require patching code even when it is not moved to SPM, unlike in our current scheme. Moreover since the SPM is usually a small fraction of the DRAM space, the space recovered in DRAM will be small.

Application	Source	Description	Data Size(Bytes)	Lines of Code	# of assembly instr.
StringSearch	MIBench	A Pratt-Boyer-Moore String Search	12820	3037	4433
CRC	MIBench	32 BIT ANSI X3.66 CRC checksum	1068	187	504
Dijkstra	MIBench	Shortest path Algorithm	4097	174	501
EdgeDetect	UTDSP	Edge Detection in an image	196848	297	701
FFT	UTDSP	Fast Fourier Transform	16568	189	478
KS	PtrDist	Minimum Spanning Tree for Graphs	27702	408	1327
MMULT	UTDSP	Matrix Multiplication	120204	164	416
Qsort	MIBench	Quick Sort Algorithm	7680000	45	116

Table 1: Application Characteristics

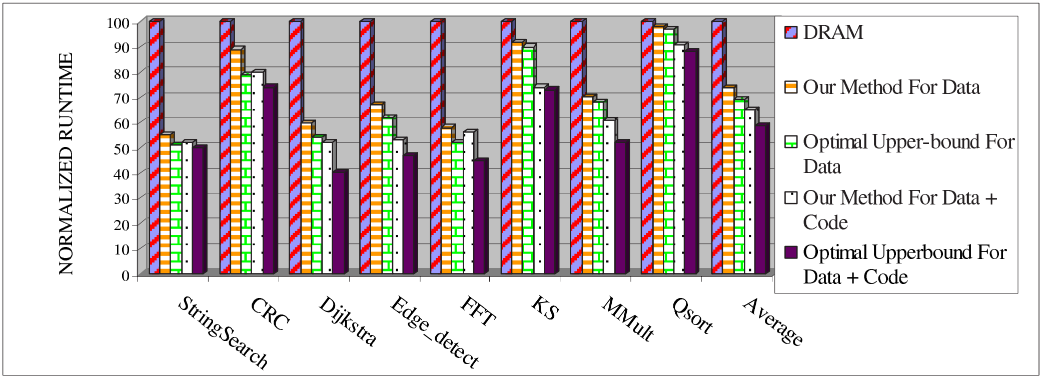


Figure 9: Runtime speedup compared to all-DRAM method and Static Optimal Method

control flow. With our definition of regions, side entries are usually caused by unstructured control flow from programming statements such as "goto", which are rare in applications. Our method does not consider regions which are the target of unstructured control flow for SPM allocation; therefore, no further modification is needed for side entries of SPM-allocated regions.

On the other hand, side exits such as procedure calls from our code regions are common. The returns from procedures do not need patching since their target address is computed at run-time. However side exits such as the calls to procedures are patched as follows. For each SPM-allocated code region, the branch offsets of all control transfer instructions that branch to outside of the region they belong to, are adjusted to the new corrected offsets. These new corrected branch offsets are calculated by adding the original branch offsets to the distance between DRAM and SPM starting addresses of the transferring regions.

The final step in the patching required for code allocation is the modification of load-address instructions of global variables, which are accessed in the SPM-allocated regions. The load-address instruction of a global variable is obtained from a PC-relative load that loads the address of the global variables from the literal table also in the code. Allocating code regions with such load-address instructions into SPM will make the original relative offsets invalid. Besides, for the ARM architecture, the relative offsets of the load-address instructions are 12-bit. Thus, it is quite likely that the distance from the load-address instructions in SPM to the literal tables in DRAM is too large to fit into those 12-bit relative offsets. To solve these two problems, our method generates a second set of literal tables which reside in SPM. During when code objects are being placed in SPM one-after-another, a literal table is generated at a point in the SPM layout if code about to be placed cannot refer to the previously generated literal table in SPM since it is out of range. This leads to (roughly) one literal table per $2^{12} = 4096$ bytes of code in SPM. These secondary SPM literal tables contain the addresses of only those global variables that refer to it. Afterward, the relative offsets of these load-address instructions are adjusted to the corrected offsets, which are calculated by the distance from the load-address instructions to the SPM literal table.

8. RESULTS

This section presents our results by comparing the proposed allocation scheme for embedded systems with unknown size SPM against an all-DRAM-allocation and against Avissar et. al's method in [5]. We compare our method to the all-DRAM-allocation method since all existing methods are inapplicable in our target systems where the SPM size is unknown at compile-time; thus, they have to force all data and code allocation to DRAM. We also compare our scheme to [5] to show that our scheme obtains a performance which is close to the un-achievable optimal upper-bound.

Experimental Environment Our method is implemented on the GNU tool chain from CodeSourcery [8] that produces code for the ARM v5e embedded processor family [3]. The process of identifying variable accesses and addresses, analysis of variable limited lifetime, and embedded loader codes generation are implemented in the GCC v3.4 cross-compiler. The modifications of executable file are done in the linker of the same tool chain. The memory characteristics are as follows. An external DRAM with 20-cycle latency, Flash memory with 10-cycle latency, and an internal SRAM (SPM) with 1-cycle latency are simulated. Data is placed in DRAM and code in Flash memory. Code is most commonly placed in Flash memory today when it needs to be downloaded. A set of most frequently used data and code is intelligently allocated to SPM by our compiler. The memory latencies assumed for Flash and DRAM are representative of those in modern embedded systems [10, 14]. The SRAM size is configured to be 20% of the total data size in the program[2]. The benchmarks' characteristics are shown in table 1.

Runtime Speedup The run-time for each benchmark is presented in figure 9 for five configurations: all-DRAM, our method for data

[2]We could have also chosen a second SRAM size to be 20% of total code + data size, when evaluating the methods for both code and data. However, to make comparisions between methods for data only and for both code and data, we had to choose one SRAM size so that the comparison is fair.

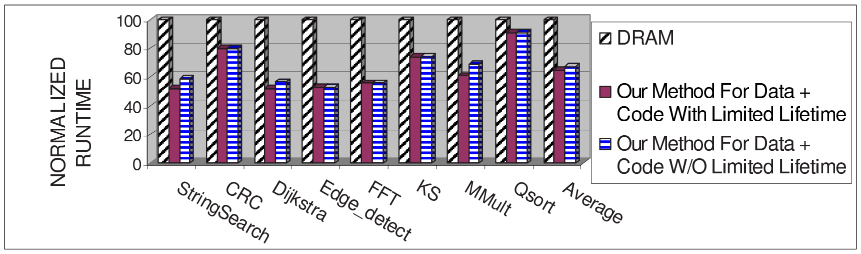


Figure 10: Runtime speedup of our method with and without limited lifetime compared to all-DRAM method

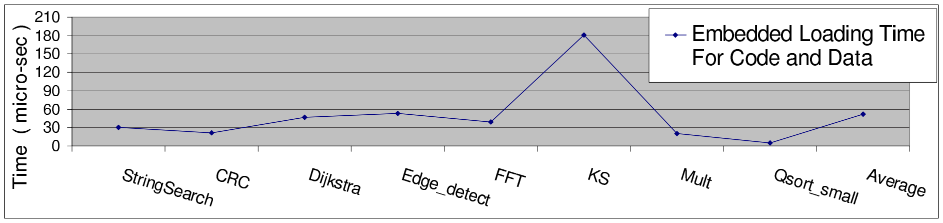


Figure 11: Variation of embedded loading time across benchmarks

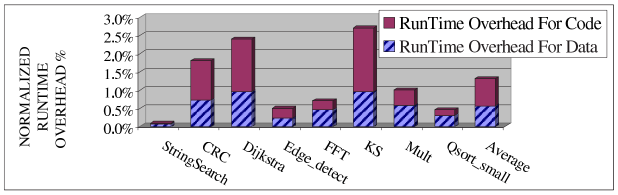


Figure 12: Runtime Overhead

allocation only, optimal upper bound obtained by using [5] for data allocation only, our method enhanced for both code and data allocations, and the optimal upper bound obtained by using [5] for both code and data allocations. Averaging across the eight benchmarks, our full method (the fourth bar) achieves a 36% speedup compared to all-DRAM allocation (the first bar). The provable optimal static allocation method [5], extended for code in addition to data, achieves a speedup of 41% on the same set of benchmarks (the fifth bar). This small difference indicates that we can obtain a performance that is close to that in [5] without requiring the knowledge of SPM's size at compile time.

The figure also shows that when only data is considered for allocation to SPM, a smaller run-time gain of 27% is observed versus an upper bound of 31% for the optimal static allocation. This shows that considering code for SPM placement rather than just data yields an additional 36%-27%=9% improvement in run-time for the same size of SPM.

The performance of the limited-lifetime algorithm is showed in figure 10. The difference between the second and third bars in figure 10 gives the improvement using our limited lifetime analysis, as compared to a greedy allocation, for each benchmark. Although the average benefit is small (4% on average), for certain benchmarks (for example, StringSearch and Dijkstra), the benefit is greater. This shows that the limited lifetime enhancement is worth doing but is not critical.

Loading Time Overhead Figure 12 shows the increase in the run-time from the embedded loader as a percentage of the run-time of one execution of the application. The figure shows that this run-time overhead from the loader averages only 1.3% across the benchmarks. A majority of the overhead is from code allocation including the latency of copying code from DRAM to SRAM at embedded loading time. The overhead is an even smaller percentage when amortized over several runs of the application; re-runs are common in embedded systems. The reason why the runtime overhead is small is explained as follows. The embedded loading time is proportional to the total number of appearances in the executable file of load and store instructions that accesses the program stack, and the locations that store global variables addresses. These numbers are in-turn upper-bounded by the number of static instructions in the code. On the other hand, the run-time of the application is proportional to the number of dynamic instructions executed, which usually far exceeds the number of static instructions because of loops and repeated calls to procedures. Consequently the overhead of the loader is small as a percentage of the run-time of the application.

Another metric is the absolute time taken by the embedded loader. This is the waiting time between when the application has finished downloading and is ready to run after the loader has completed its work. For a good response time, this number should be low. Figure 11 shows that this waiting time is very low, and averages 50 micro-seconds across the eight benchmarks. It will be larger for larger benchmarks, and is expected to grow roughly linearly in the size of the benchmark.

Code Size Overhead Figure 13 shows the code size overhead of our method for each benchmark. The code-size increase from our method compared to the unmodified executable that does not use the SPM averages 1.0% across the benchmarks. The code-size overhead is small because of our technique of reusing the un-used bit-fields in the executable file to store the linked lists containing locations with unknown stack offsets and global addresses. In addition, the figure shows the code size overhead from its constituent two parts: the embedded loader codes and the additional information about each variable and code region in the program which is stored until runtime. These additional information are the starting addresses of the location linked-lists, regions sizes, regions start and end addresses, variables sizes, original stack offset and global variable addresses.

Memory Access Distribution Figure 14 show the distribution of memory accesses between SRAM and DRAM. This is another view of our method's performance in terms of how many percentages of the memory accesses our method is able to direct to SRAM instead of allocating them in DRAM. Figure 14 indicates that on average, 53% of memory accesses are to SRAM for our method; vs. 59% for Avissar et. al's method in [5].

Runtime vs. SPM size Figure 15 shows the variation of runtime

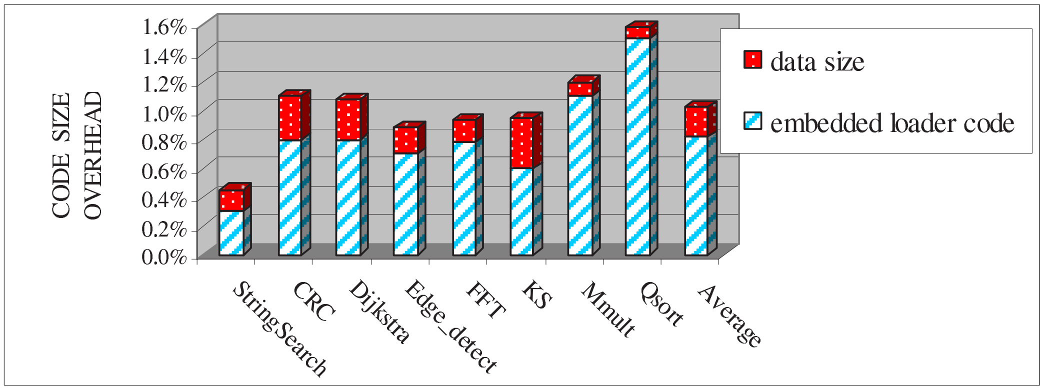


Figure 13: Variation of code size overhead across benchmarks

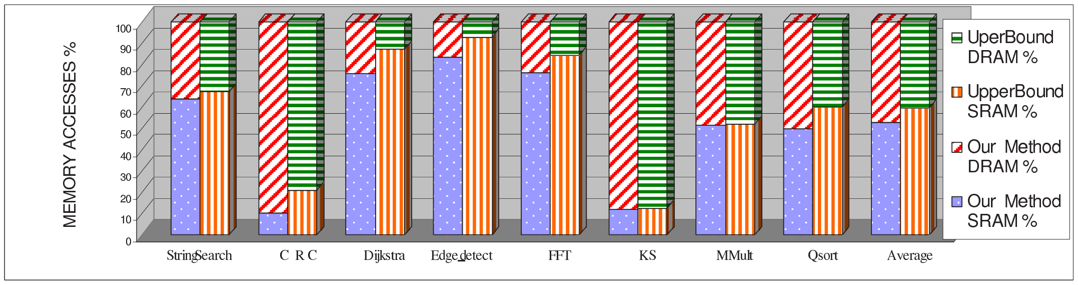


Figure 14: Comparison of memory access distribution against Static Optimal Method

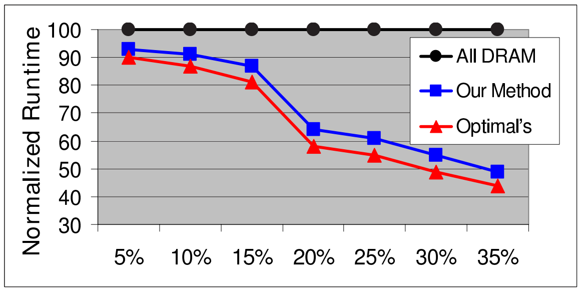


Figure 15: Runtime Speedup with varying SPM Sizes for Dijkstra Benchmark

for the Dijkstra benchmark with different SPM size configurations ranging from 5% to 35% of the data size. When the SPM size is set to lower than 15% of the data size, both our method and the optimal solution in [5] do not gain much speedup for this particular benchmark. Our method starts achieving good performance when the SPM size is more than 15% of the data size since at that point the more significant data structures in the benchmark start to fit in the SPM. When the SPM size exceeds 30% of the data set, a point of diminishing returns is reached in that the variables that do not fit are not frequently used. The point of this example is not to so much to illustrate the absolute performance of the methods. Rather it is to demonstrate that our method is able to closely track the performance of the optimal static allocation in robust manner across the different sizes by using the exact same executable. In contrast the optimal static allocation uses different executables for each size.

9. CONCLUSION

In this paper, we introduce a compiler technique that, for the first time, is able to generate code that is portable across different SPM sizes. With technology evolution every year leading to different SPM sizes for the same ISA's processor implementations, there is a need for a method that can generate such portable code. Our method is also able to share memory between stack variables that have mutually disjoint lifetimes. Our results indicate that on average, the proposed method achieves 36% speedup compared to all-DRAM allocation without knowing the size of the SPM at compile-time. The speedup is only slightly higher (41% vs all-DRAM) with an unattainable optimal upper-bound allocation that requires knowing the SPM size [5].

Our method currently only considers program code, global variables and stack variables for static SPM allocation. Dynamic allocation schemes and heap allocation schemes may also be investigated in the future.

10. REFERENCES

[1] F. Angiolini, L. Benini, and A. 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.

[2] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. 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.

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

[4] O. Avissar, R. Barua, and D. 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.

[5] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory

Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS)*, 1(1), September 2002.

[6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad 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.

[7] Cnetx. Downloadable software. http://www.cnetx.com/slideshow/.

[8] CodeSourcery. http://www.codesourcery.com/.

[9] A. Dominguez, S. Udayakumaran, and R. Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. *Journal of Embedded Computing(JEC)*, 2005. Cambridge International Science Publishing. *To appear August 2005*.

[10] *Intel wireless flash memory (W30)*. Intel Corporation. http://www.intel.com/design/flcomp/datashts/290702.htm.

[11] Handango. Downloadable software. http://www.handango.com/.

[12] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[13] J. D. Hiser and J. 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.

[14] J. 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.

[15] Landware. Downloadable software. http://www.landware.com/pocketquicken/.

[16] M.Kandemir, J.Ramanujam, M.J.Irwin, N.Vijaykrishnan, I.Kadayif, and A.Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Design Automation Conference*, pages 690–695, 2001.

[17] 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.

[18] Phatware. Downloadable software. http://www.phatware.com/phatnotes/.

[19] 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/.

[20] J. Sjodin, B. Froderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.

[21] J. Sjodin and C. V. Platen. Storage Allocation for Embedded Processors. *Compiler and Architecture Support for Embedded Computing Systems*, November 2001.

[22] Softmaker. Downloadable software. http://www.softmaker.de.

[23] 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.

[24] S. Udayakumaran and R. 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.

[25] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, automation and test in Europe*, page 21264. IEEE Computer Society, 2004.

[26] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004)*, 2004.

[27] Xi-art. Downloadable software. http://www.xi-art.com/.