Recursive Function Data Allocation to Scratch-Pad Memory

Angel Dominguez, Nghi Nguyen and Rajeev Barua
ECE Dept, Univ of Maryland, College Park
{angelod,nghi,barua}@eng.umd.edu

ABSTRACT This paper presents the first automatic scheme to allocate local (stack) data in recursive functions to scratch-pad memory (SPM) in embedded systems. A scratch-pad is a fast directly addressed compiler-managed SRAM memory that replaces the hardware-managed cache. It is motivated by its significantly lower access time, energy consumption, real-time bounds, area and overall runtime. Existing compiler methods for allocating data to scratch-pad are able to place only code, global, heap and non-recursive stack data in scratch-pad memory; stack data for recursive functions is allocated entirely in DRAM, resulting in poor performance.

In this paper we present a dynamic yet compiler-directed allocation method for recursive function stack data that for the first time, is able to place a portion of recursive stack data in scratch-pad. It has almost no software-caching overhead, and is able to move recursive function data back and forth between scratch-pad and DRAM to better track the program's locality characteristics. With our method, all code, global, stack and heap variables can share the same scratch-pad. When compared to placing all recursive function data in DRAM and all other variables in scratch-pad, our results show that our method reduces the average runtime of our benchmarks by 29.3%, and the average power consumption by 31.1%, for the same size of scratch-pad fixed at 5% of total data size. Furthermore, significant savings were observed when comparing our method against cache-based alternatives for SPM allocation. Finally, we show results that analyze the effects of profile variation on our allocation approach and present a modified version of our method which minimizes variation for profile-based allocations.

1 Introduction

In embedded systems, program data is usually stored in one of two kinds of writable 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 for efficient access to frequently used data.

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 study [5] 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 [5]. Even more surprising, the run-time cycle count they measured was 18% better with a scratch pad using a simple static knapsack-based [5] 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: [1, 7, 26, 27, 35]), ahead of caches. Trends in recent embedded designs indicate that the dominance of scratch-pad will likely consolidate further in the future [5, 29], 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. Recent advances have made much progress in compiling *code, global, heap and non-recursive 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 [3, 4, 15, 31, 32] 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 – [4] is the first method to solve this difficulty by partitioning the stack into two stacks, one for scratch-pad and one for DRAM. Second, more recently proposed *dynamic* methods improve upon static methods by allowing variables to be moved at runtime [10, 19, 34, 36, 39]. 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.

Recursive functions Recursive functions are widely used for large classes of computational problems. They are the most natural and efficient way of programming many algorithms, including those that use graphs, trees, and hierarchical databases. Virtually all graph or tree-traversal algorithms are recursive. In addition, some algorithms using arrays, such as quicksort, are also recursive. Recursive functions are common in many embedded domains, include communications, networking, planning, control, and transportation. Indeed, we had no difficulty finding embedded benchmarks with recursive functions. Moreover, in many of these benchmarks, recursive functions dominated the run-time, especially after other optimizations had been done.

Unfortunately, even the most robust of the published SPM allocation schemes lacks support for recursive function stack data. This is not surprising since all existing allocation methods work by finding frequently used variables and placing them into SPM until filled. *The allocator must know both the size of each allocated variable as well as the total amount of SPM available.* Published research has dealt almost entirely with program objects having a fixed size at compile-time and run-time (code, global and stack data), although one recent publication does present a method for heap data as well. *For recursive functions, the stack frame size is fixed at compile-time but not the total number of frames allocated at runtime*. Without knowing their total size, existing SPM allocation methods must leave recursive stack data in main memory.

Allocating recursive stack data to main memory, which is much slower than SRAM, can cause very poor performance. The importance of allocating recursive stack data rises dramatically for programs making significant use of recursive functions, even more so when all other data has been optimized for SPM placement. Also, as embedded platforms become more complex, so will their software, increasing the likelihood that dynamic storage methods such as recursive stack data will be used more heavily. Finally, automatic methods for SPM placement exist only in the form of hardware caches, which have higher power and execution costs when compared against a good compiler-directed SPM allocation scheme, particularly for dynamically allocated data exhibiting randomized accesses.

This paper presents a new approach to handling recursive stack data for allocation purposes. We first take advantage of the fact that a recursive function has a fixed size stack frame known at compile-time. Our approach is to examine the runtime behavior of individual stack frames at different depths corresponding to the discrete stack frames. Using this information, we place the most commonly used depths, as detected by profiling, into SPM and all other depths are left in main memory. Finally, we also present a method to reduce profile-dependence for allocation decisions, crucial when dealing with dynamically allocated program data.

2 Related work

Among existing work, static methods to allocate data to SPM include [3–5,15,28,31,32]. Static methods are those in which the contents of SPM do not change at run-time. Some of these [5, 28, 31] allocate only global variables to SPM, while others [3,4,15,32] can allocate both global and non-recursive stack variables to SPM. Other static methods [33,40] can allocate both code and data 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.

Another approach to SPM allocation are dynamic methods; in such methods the contents of the SPM can change at run-time [10, 12, 24, 25, 33, 34, 36, 38]. The method in [24] can place global and stack arrays while the method in [33] can place global and code. The method in [36] allocates global and stack data to SPM dynamically, with explicit compiler-inserted copying code that copies data between slow memory and SPM when profitable. All dynamic data movement decisions are made at compile-time based on profile information. The method by Verma et. al. in [38] is a dynamic method that allocates code as well as global and stack data. It uses an ILP formulation for deriving an allocation. The work in [34] also allocates code, global and stack data, but using a polynomial-time heuristic. Finally, the method in [10] is the first dynamic SPM allocation method to place a portion of the heap data in the SPM, making it the most complete to date among the SPM allocation schemes by handling code, global, heap and non-recursive stack data.

Recursive functions None of the SPM allocation publications in the literature discuss recursive stack handling; instead they leave such data in main memory. However a few techniques aim to convert recursive functions into non-recursive functions. This is best seen in the tail recursion (or tail-end recursion) optimization [6, 8]. In tail-recursive functions, stack frames at different depths can share the same space in memory since they have non-overlapping lifetimes. Since their total stack size is now bounded, scratch-pad allocation is easy. However, most modern compilers such as GCC (also our experimental platform) already implement tail recursion optimizations. Hence all the results present in this paper show benefits beyond those that can be achieved by tail recursive optimizations.

Transformation methods have also been studied that convert limited classes of recursive functions into other forms, such as loops [13]. Another method performs procedure inlining to convert mutual recursion to direct recursion [20]. This allows use of optimization techniques that are most easily applied to directly recursive procedures. A noteworthy publication presents a manual method for transformation of general recursive cycles into iteration [22]. However, their analyses and measurements show that some previously considered optimizations can actually result in slower programs. The fact is that there are many algorithms in computer science which are most elegantly and efficiently expressed as recursive functions and not automatically or beneficially optimizable by existing methods. Indeed, as far as we know, none of the above techniques have been implemented in commercial or widely-used open-source compilers, despite being available for decades.

3 Recursive function stack allocation

In order to consider recursive stack data for allocation to scratch-pad memory (SPM), our approach is to treat each possible recursive function instance called into creation at runtime as an individual variable representing that particular invocation. For this approach, we set the unit size for recursive function stack allocation to be the entire stack frame for that function. The total stack frame size (allocated each time the procedure is called) is fixed at compile-time for functions in written in C. Of course, the trivial case when a recursive function does not consume any stack space does not need to be considered for SPM placement.

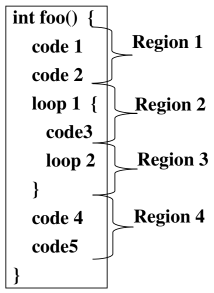
We choose to allocate recursive stack data to SPM at the granularity of an entire stack frame for several reasons. First, some architectures (like ARM) may not allocate the entire frame upon function entry. These platforms instead grow the stack as needed when execution reaches certain points in the function code. For safety, we conservatively bound the size of a recursive function instance to its total maximum frame size and not to any of the possibly smaller sizes seen at runtime during different visits to the function [1]. Second, by not attempting to handle individual variables within a stack frame, we greatly reduce the bookkeeping and code-insertion overheads required by our method to control the allocation of individual instances. Finally, most recursive functions tend to contain a relatively small number of stack variables and consume small amounts of space per frame as compared to non-recursive functions, making the stack frame an attractive abstraction without significant loss of fine-grain control.

To implement our approach to allocating recursive function stack data, we have developed a complete compiler-directed analysis and allocation framework based on the best existing dynamic SPM allocation scheme for heap data from [10]. This publication describes a heuristic approach to dynamic placement of code, global, stack (non-recursive) and heap data into SPM for embedded systems. Our method can be integrated directly to such a scheme using the details provided in the next few paragraphs.

Dynamic SPM allocation framework Since our overall SPM allocation framework is dynamic – in that the contents of SPM are different in different regions of the program – we need to define what our choice of regions is. Our SPM allocation strategy has a fixed allocation inside each region, although the allocation can change at region boundaries. Since our choice of regions is mostly orthogonal to our allocation strategy for recursive frames, we only briefly outline our choice of regions here; the region definition is borrowed from our earlier work in [34]. Regions are defined 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). A region ends when the next one begins. An example of how code is partitioned into regions is in Figure 1. Other choice of regions are possible, but we have not explored them since our experiments have found the above choice to perform well.

For code, global and stack our earlier work has found dynamic allocation to be superior to static allocation [34]. However within recursive functions, for reasons mentioned above, each recursive stack frame goes to a single memory (although different frames go to different regions.) Hence inside recursive functions,

[1]For this same reason, we do not set the granularity down to the individual variable level as some variables may not appear in all invocations at runtime, severely complicating safe dynamic allocation decisions along all possible program paths



```
int foo() {
   code 1
   code 2
   loop 1 {
     code3
     loop 2
   }
   code 4
   code5
}
```

Figure 1: A method *foo()* is divided into code regions

there is little benefit do dividing into regions, and it would also complicate code generation. *For this reason, each recursive function is always considered a single region, regardless of any loops inside it.* Of course, recursive functions are in different regions from any non-recursive functions that call them, or that they call.

Program Profiling When attempting to allocate recursive stack data, it is important to gather both static and dynamic profile information on the program being optimized. Static profile information can be obtained from a compiler by estimating the frequency of code. For example, most register allocators estimate the iteration count of all loops as a fixed value of 10. Nested loops are estimated with a cumulative iteration count of 10^i for loops at nesting depth i. In contrast, dynamic profile information can be obtained through instrumented execution of the program binary using appropriate program inputs. Very simple programs that do not take inputs will have an accurate program representation from only the static profile, but complex programs that are input-dependent will have incomplete program behavior information without a large amount of dynamic profile data. When attempting to optimize dynamically allocated variables such as recursive function stack objects, dynamic profile information becomes much more important to obtain a clear picture of probable program behavior at runtime.

To illustrate dynamic profiling useful for allocating recursive stack data, we present a simple example of a recursive function performing an in-order visitation of data structure nodes forming a graph. Figure 2(a) shows pseudocode which implements the recursive function. This function simply visits each node in a graph before visiting each of its children. Figure 2(b) shows the static profile frequency table (PFT) for the function including extra variable information for this function. Figure 2(c) shows the dynamic PFT for same function for a program input where the function recursed to a maximum depth of three invocations. This last figure shows our approach to dealing with recursive functions and their stack variables. Our method treats each possible runtime depth as a separate variable for profiling and allocation purposes, each with its own

```
preorder(node v) {
  visit(v);
  for each child w of v
     preorder(w); }
```

(a) Code

Name	Size	Accesses
Preorder	20	?

(b) Static Profile

Name	Size	Accesses
Preorder depth 1 stack	20	100
Preorder depth 2 stack	20	60
Preorder depth 3 stack	20	40

(c) Dynamic Profile

Figure 2: Example recursive function showing (a) function code, (b) static profile, (c) dynamic profile.

size, access and lifetime information.

Deciding Allocations Before proceeding further, let us consider that what we really would like: we would like to specify exactly the recursive depths for which stack frames should be allocated. For example in figure 2, if we only had space for two recursive stack frames, we would choose those at depths 1 and 2, since those had the greatest dynamic frequency of access. Another example of more frequent access at the root of the recursive tree is presented in Figure 3. This figure shows a tree data structure, and how accesses are more frequent at root nodes. Recursive functions often have this behavior in that the first few frames are more frequent. *However, we have also found programs with the exact opposite behavior, when the last few frames are more frequent; or where some intermediate-depth frames are the most frequent.* To distinguish these cases, we need dynamic profiling as described above.

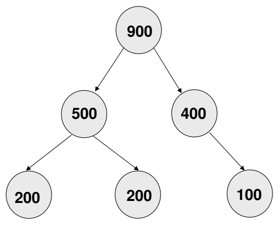


Figure 3: Binary Tree with each node marked by its access frequency for use by allocation analysis.

As an example of the behavior that may result, consider that for some program, depths 3,4 and 5 may be most frequently accessed for some recursive function, and our SPM allocator should preferentially allocate them to SPM. All other depths (1,2,6,7,...) should be allocated to DRAM. Our code generator should be sophisticated enough to generate code to implement this depth-specific behavior. Note that it is incorrect to simply allocate all depths to SPM, since the total SPM size needed for that is unbounded at compile-time.

No existing SPM allocation method is able to distinguish between recursive depths or allocate recursive functions to SPM.

What we need to implement the above desired allocation is a code generation method that can create efficient code to implement the above. At first glance it seems that unwinding the recursive function by repeatedly inlining it might help with code generation, since it provides individual copies corresponding to each invocation depth of the function, essentially cloning the function into separate related functions. A drawback with this approach is that it is hard to determine how many times the function should be inlined in itself since the theoretical maximum is unbounded. Moreover even limited inlining increases code size. For these reasons, our approach never does inlining or cloning. Instead it takes advantage of the behavior observed in applications using recursive functions to logically split them into individual instances which can each be allocated separately and safely.

The key to segmenting a recursive function into its individual invocation instances is to take advantage of its allocation behavior at runtime. Recursive function stack frames must be de-allocated strictly 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. Through judicious code insertion, it is possible for a compiler to treat each possible depth of invocation a recursive function may reach as a logically separate function, allowing individual stack frame control for allocation to a chosen memory area.

Code generation To actually make use of the notion of separate recursive function instances for allocation, proper code generation is essential. Our method begins by inserting a few lines of code at the entry and exit points of each recursive function which are used to increment and decrement a *depth counter*, respectively, for each optimized function body. This counter tracks the current invocation depth of that function at runtime and serves as a way to virtualize a recursive function into separate instances.

The entry point into the function is modified so that the depth counter is checked upon entry and used to decide if the stack pointer should be updated to an SPM location or left at the current program value for main memory. *The most general check that is currently supported is that the depth counter for a certain* *contiguous range of its values should be allocated to SPM.* For example, if depths 3, 4 and 5 are to be allocated to SPM, the check is `(depth_counter` $\geq$ `3) or (depth_counter` $\leq$ `5)`. As another example, if depths 1, 2 and 3 are to be allocated to SPM, the check is `depth_counter` $\leq$ `3`. Although it is possible that the most frequently used depths are not all contiguous, we found in our experiments that is rare, hence our current implementation only supports a single contiguous range. When such a rare case is found, we choose the single range of depth counter values with the greatest cumulative frequency. Checks for multiple contiguous ranges can be supported, but the checking overhead will increase, so it is not clear they would be a good idea.

Each optimized invocation of the function must also reserve enough space to store its stack pointer address for use when swapping the current function in and out of SPM, if dynamic movement of recursive stack frames is desired.

Mutual recursion When a function calls itself, that is direct recursion. Occasionally functions do not call themselves, but are part of a recursive cycle of functions in the program's call graph. For example, function *A()* may call *B()*, which in turn may call *A()*. Recursive cycles which span more than a single function are rare, and our own large benchmark set does not contain any examples of this type of recursion. However, to support the full range of possible programs, we have implemented support for multi-function recursive regions. Our solution is to increment the depth counter at the root function in the recursive cycle – this is the function that is called from non-recursive procedures. Checks are inserted at each function in the recursive cycle, as usual. If there is more than one root function in the recursive cycle, then the depth counter increment can be placed at any one of the roots.

3.1 Profile sensitivity

Profile dependence is a problem inherent to any memory allocation scheme which bases its decisions on program profiles, whether they be compile-time(static) or runtime(dynamic) profiles. As individual programs become more complex, they also tend to exhibit a much higher degree of input profile dependence in terms of execution and data access patterns. This is particularly true for dynamically allocated data such as heap or recursive stack objects. For example, the most frequently accessed recursive depths could be different for a certain input data set compared to the representative input data set used for profiling. For such profile sensitive applications, this dependence can cause poor results with bad allocation predictions.

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 data allocated and accessed at runtime. Methods for optimizing dynamically allocated data must rely on dynamic profiling of the program using typical inputs. When basing general decisions on a limited set of profiles, it is vitally important to reduce the sensitivity of the program allocation scheme program profiles used. Comprehensive analysis methods greatly increase the chance that a chosen allocation will work well for the majority of expected program inputs, particularly when dynamically allocated memory accounts for a sizable percentage of total data accesses. Our approach to reducing profile sensitivity lies in accounting for both the static program profile and as many dynamic profiles as can be obtained from representative inputs.

We take the following steps to improve the robustness of performance improvements across data sets. First we create the profile frequency tables (PFTs) for each region for each program input, containing the code frequency of that region, as well as the profile frequency of variables accessed in that region. For recursive functions (which always contain only one region), the PFT should contain one row per stack depth. An example of the complete PFT for the recursive function region in figure 2(a) is in figure 4. This is an extended version of the PFT in figure 2(c), but containing extra rows for the code block, and for a global variable G accessed in the region (not shown in the code). The second-to-last column of figure 4 shows the access frequencies for an extra data set (numbered input data set 2). The last column shows the average access frequency across data sets 1 and 2 for each variable. This average is used in the calculation of the depths to be allocated to SPM, instead of the frequencies from only one data set. This averaging mechanism is a good way to prevent the profile data from being misled by extremes in input data sets. We have found that this averaging improves the robustness of the performance gain across data sets by avoiding over-specialization for any one data set.

Variable Name	Size (bytes)	Input data set 1	Input data set 2	Average Frequency
Preorder Code	28	200	180	190
Preorder depth 1 stack	20	100	80	90
Preorder depth 2 stack	20	60	66	63
Preorder depth 3 stack	20	40	34	37
Global G	4	20	18	19

Figure 4: Dynamic profile frequency table for the recursive program region in figure 2(a). This is a more complete version of figure 2(c) with additional variables and an additional data set 2.

4 Results

This section presents the results obtained by comparing our allocation method for recursive function stack data against the usual practice of placing such data in DRAM, for a variety of compiler and architecture configurations. For comparison, since there exists no other automatic compiler methods to handle recursive function stack data, we use the most general existing compiler-directed SPM allocation scheme for code, global, heap and non-recursive stack data from [10]. The scheme in [10] is chosen for comparison since it is one of the only schemes in the literature that can also handle heap data. Its handling of code, global and non-recursive stack data is based on the dynamic method in [34]. Hence the comparison method represents the state-of-the-art method for SPM allocation today.

Our method for allocation of recursive stack data is built on top of the comparison scheme and augments its capabilities. Since our method and the comparison are implemented in the same compiler and simulation environment, the comparison is fair. All applications are compiled automatically using full optimization levels without requiring the user to specify anything other than the SPM space available on the target platform. An external DRAM with 20-cycle latency, an external Flash memory of 20-cycle latency, and an internal SPM (SRAM) with 1-cycle latency is simulated in the default configuration. Flash is used to store code; DRAM and SPM store program data. The default configuration has an SPM size which is 5% of the total data size of 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.

Methodology Details Our compiler-based allocation method is implemented on the GNU Compiler Collection(GCC) v4.1 cross-compiler [11] released by CodeSourcery and targeting the ARM v5e embedded

	Anagram	Bh	Bisort	Cfrac	Epic	Health	Mst	Patricia	Perimeter	Qbsort	Treeadd	Treesort	Trie	Tsp	Voronoi	Yacr2
Benchmark Source	PtrDist	Olden	Olden	MallocBench	MediaBench	Olden	Olden	MiBench	Olden	McCat Suite	Olden	LLVM Suite	McCat Suite	Olden	Olden	PtrDist
% Data that is Rec. Stack	0.08	0.93	0.57	0.20	0.03	0.04	0	0.02	0.06	0.04	0.03	0.02	0.18	0.06	0.05	0.03
% Data Access by Rec.Stack	0.07	58.20	66.08	1.02	0.15	74.14	1.07	2.71	78.18	15.64	97.70	56.34	68.79	50.89	8.86	12.48
% Rec. Access to SPM	100	96.78	41.87	0	100	99.75	100	97.03	99.54	0	100	100	92.21	28.68	83.58	98.90
# Unique Rec. Stack Depths	1	18	22	5	32	11	1	6	24	13	21	15	106	14	11	45

Figure 5: Benchmark Statistics Table.

processor family [16]. Execution results are obtained from an ARM simulator included as part of the Gnu Debugger (GDB) v6.2 software, augmented to accurately model its execution and power characteristics. The energy consumed by programs is estimated using the instruction-level power model proposed in [21] using ARM specific information from [30]. To model SPM, we have adopted an approach similar to those in [5], [33] and [18], in which we simplify the CACTI [9] estimation model to match an SRAM memory module. We have also incorporated the DRAM power estimation model provided by MICRON [17] for their external DDR Synchronous DRAM chip [23]. All devices were simulated at 200MHz with an operating voltage of 1.5V.

Benchmark Suite For our experiments, we gathered a large number of freely available applications suited to the embedded domain. All applications make use of code, global, stack, heap and recursive stack data and have not been modified. Figure 5 shows important statistics from each program. Each program contains between 1-100 unique recursive stack instances. For the set, recursive stack data makes up less than 1% of the data size, yet recursive stack accesses make up more than 50% of total accesses for fully half of the programs. At only 5% SPM, our method is able to place more than 90% of all recursive stack accesses into SPM for 70% of the benchmarks. As with any allocation method, improvement is proportionate to the contribution of the optimized variable to total program runtime.

Runtime and energy gain Figure 6 compares the normalized runtime from our method versus from the existing practice of placing all recursive stack data in DRAM. Without our method, this SRAM is used only by code, global, heap and non-recursive stack data; with our method the SRAM is shared by all types of variables. *The figure shows that the average runtime reduces by 29.3% by using our method for the exact same architecture*. The large average improvement shows the potential of our method to reduce runtime of recursive applications beyond today's state-of-the-art.

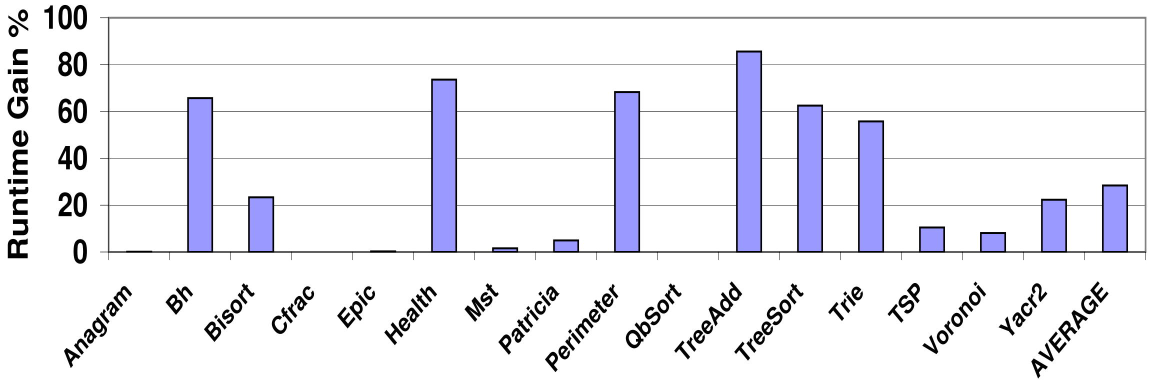


Figure 6: Normalized runtime from using our method versus allocating heap data in DRAM.

In general, runtime and energy improvements from allocation of recursive stack data to SPM are proportional to the percentage of data accesses made to recursive data, and inversely proportional to the percentage of the program data size consumed by recursive stack variables. Applications with a small percentage of accesses going to recursive data will not benefit greatly from our scheme, and can be seen in the performance of Anagram, Cfrac, Epic, MST and Patricia. Other applications in our set are almost dominated by recursive functions and more than half of all data accesses are made to recursive stack frame. This is the case in Bh, Bisort, Health, Perimeter, Treeadd, Treesort, Trie and Tsp. Some applications may have high percentages of recursive stack accesses, but cannot be easily placed at 5% spm due to allocation pressure from more important code and other variables at limited SPM sizes. This is the case in Qbsort, where most recursive data is placed at 10% SPM and larger sizes. Yacr is also interesting in that the allocation of recursive data is done statically and reduces the transfer costs incurred by non-recursive functions allocated by the baseline method. Other programs with modest amounts of recursive stack accesses will show more modest improvements in runtime, such as Voronoi.

Energy gain Figure 7 compares the energy consumption of programs using our method versus placing recursive stack data in DRAM. The figure shows *an average reduction of 31.1% in energy consumption* using our method. This result demonstrates that our approach has the potential to not only significantly improve runtime, but also energy consumption. While our method primarily seeks to reduce runtime, this corresponds with a proportionate reduction in the energy consumption of the system for applications in our experiments. The energy reduction from SPM allocation for two reasons: because SRAM cells take less energy to access than DRAM cells; but much more importantly, the latency saving with SRAM means the

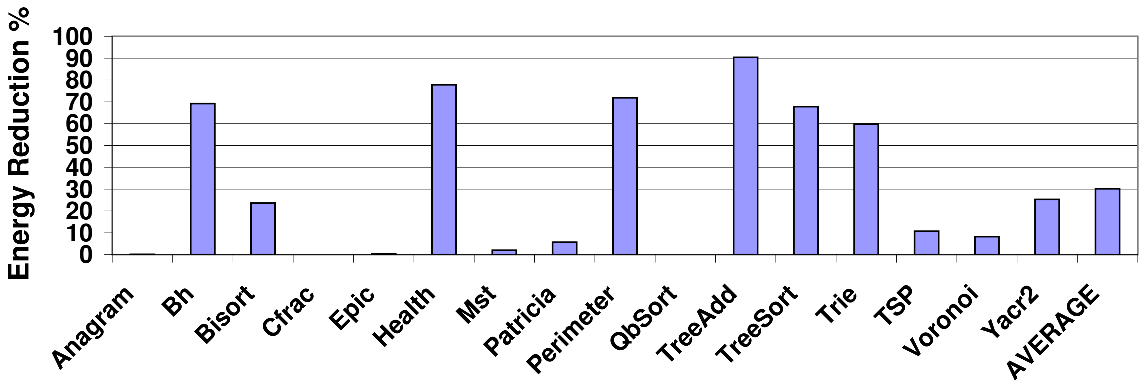


Figure 7: Normalized energy consumption comparing our method against the baseline.

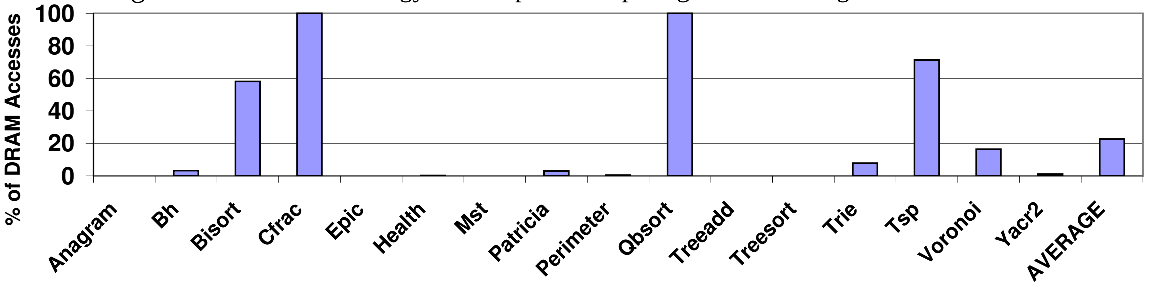


Figure 8: Percentage of recursive stack accesses made to DRAM.

processor pipeline is stalled for less time, saving on processor idle-cycle energy, which is very significant.

Reduction in recursive stack DRAM accesses Figure 8 shows the percentage of memory accesses to recursive stack data going to DRAM after applying our method. In this figure, any applications without a bar indicate that our method was able to allocate all accessed recursive stack variables into an SPM of 5% data size. The number of DRAM accesses is sometimes increased by the DRAM-to-SPM copying code at the beginning of dynamic regions, but is reduced much more by the increased locality afforded by SPM. Considering both effects, the average net reduction across benchmarks is a significant 77.4% reduction in DRAM accesses for an SPM size that is only 5% of the total data size. Analyzing the results shows that our method was able to place many important recursive stack variables into SRAM without involving transfers, explaining the high reduction in DRAM accesses for many benchmarks. This was somtimes correlated with a small increase in transfers for less important variables, which were evicted to make room for the more frequently accessed recursive stack variables.

Effect of varying SPM size Figure 9 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 average runtime gain from our method varies from 29.3% to 39.5%, when the scratch-

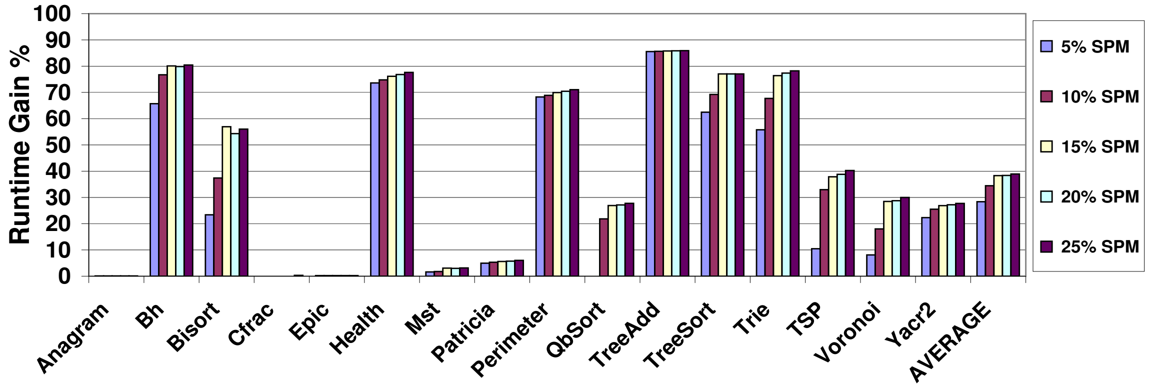


Figure 9: Effect of varying SPM size on runtime improvement.

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 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 [14]. We also observed a reduction in energy consumption ranging from 31.1% to 43.5% when spm size is varied (not shown). The results from this experiment reinforce the effectiveness of our technique for a range of SPM sizes in embedded platforms.

5 Comparison with caches

This section compares the performance of our method for scratch-pad memories (SPM) versus alternative architectures using either cache memory alone, SPM alone or cache and SPM together. It is important to note that our method is useful regardless of this comparison because there are a great number of embedded architectures which have SPM and DRAM, but have no data cache. These architectures are popular because SPMs are simple to design and verify, and provide better real-time guarantees for global and stack data [41], power consumption, and cost [2,5,33,38] compared to caches. Nevertheless, it is interesting to see how our method compares against processors containing 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 variables by allocating them more space in SPM. One 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 recursive stack variables in SRAM, while our method retains a fixed subset.

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 our main results section (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 [9, 42]. The unified cache simulated is direct-mapped (better hit rate for very small cache sizes), has a line size of 8 bytes (minimum supported by Cacti), 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 [37] is used to obtain run-time results; it is combined with Cacti's energy estimates per access to yield the energy results.

Figure 10 shows the normalized run-times for different architecture and compiler pairs, obtained for all benchmarks. The first bar is without our recursive stack allocation methods 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 also apply our recursive stack allocation method. The third and fourth bars are similar to the first and second, except that for these two we have a SPM and cache available on the same platform. The third bar shows the results when we allocate code, global, heap and non-recursive stack objects to SPM and let the cache handle all recursive stack accesses. 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 is accessed through the cache only.

From the results shown in figure 10, we see that the cache-only approach performs significantly worse

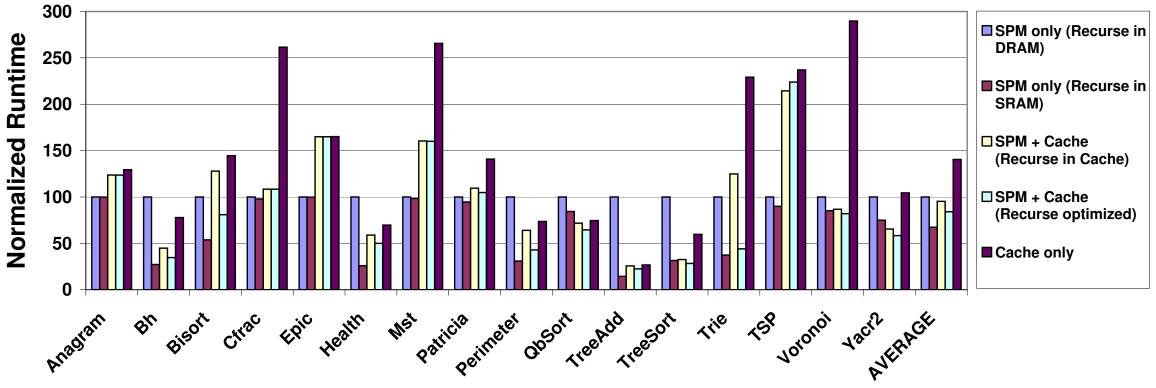


Figure 10: Normalized run-times for architectures containing different combinations of SPM and cache.

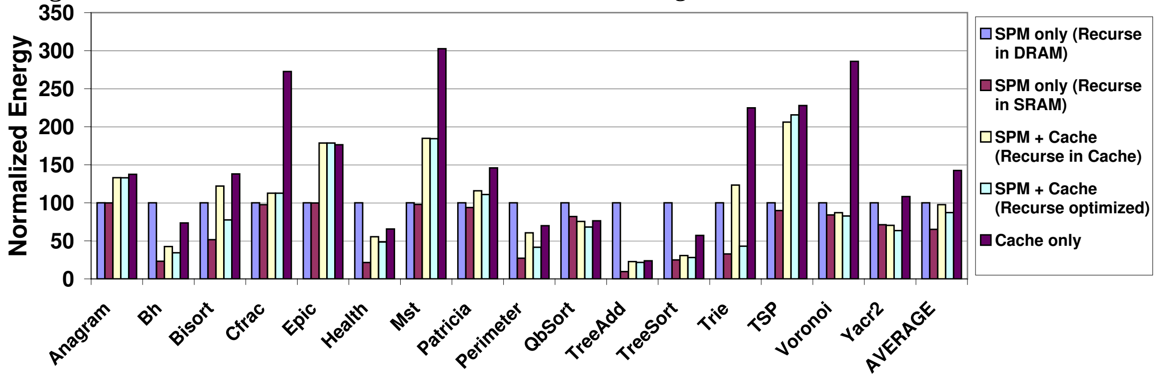


Figure 11: Normalized energy usage averaged across all benchmarks for different architecture/compiler pairs.

than any of the other methods on average. This correlates with our results from previous work on heap data allocation, where we found that very small caches perform very poorly for programs that make heavy use of dynamically allocated data, usually causing ”‘cache thrashing”’ at runtime. The scenarios where our method was applied on the Cache + SPM platform performed better than the cache-only scenario, but failed to reach the performance of the SPM-only hardware platform using a compiler-directed dynamic allocation scheme. Finally, the scenario where our allocation scheme is applied to an SPM-only platform performed the best with a 32.6% improvement in runtime compared to the baseline, and a remarkable 72.1% improvement on average over the cache-only architecture, which itself performed 40.45% worse than the baseline. Figure 11 shows the normalized energy consumption for the same configurations as in figure 10, and tracks the execution results.

It is interesting to analyze the strengths and weaknesses of our method versus caches in the light of these results. From careful analysis of individual benchmark results, we have found that in many cases, caches simply do not perform well for dynamically allocated program data, particularly at small 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 recursive stack data (and heap) than with non-recursive stack and global data. The most common use of recursive functions in applications is for processing of dynamic data structures such as lists, trees and graphs. Dynamic and recursive traversal of such data structures is often unlocalized with pointer reference chains tending to access non-sequential memory locations; both are 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 variables too large to place in SPM. The cache scenarios also tended to make useless cache transfers for data which our method left in main memory. We found that a great deal of program data should always remain in main memory, as transferring it in and out for only a few accesses is seriously detrimental to efficiency.

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-based systems.

6 Profile sensitivity

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 nonprofiled input set. We would also like to evaluate the performance of our profile averaging pass for reducing profile sensitivity. Because we are dealing with two very different program inputs, each with its own data size and runtime characteristics, for these experiments we fix the SPM size to be 5% of the larger data size for a fair comparison. All other experiments in this paper are based on input A. Other experiments (not shown) based on input B showed some fluctuations in results for the applications, but on average achieved runtime and energy savings within 2% of those from input A.

Figure 12 shows the runtime gain comparison results for our profile sensitivity experiments. The first

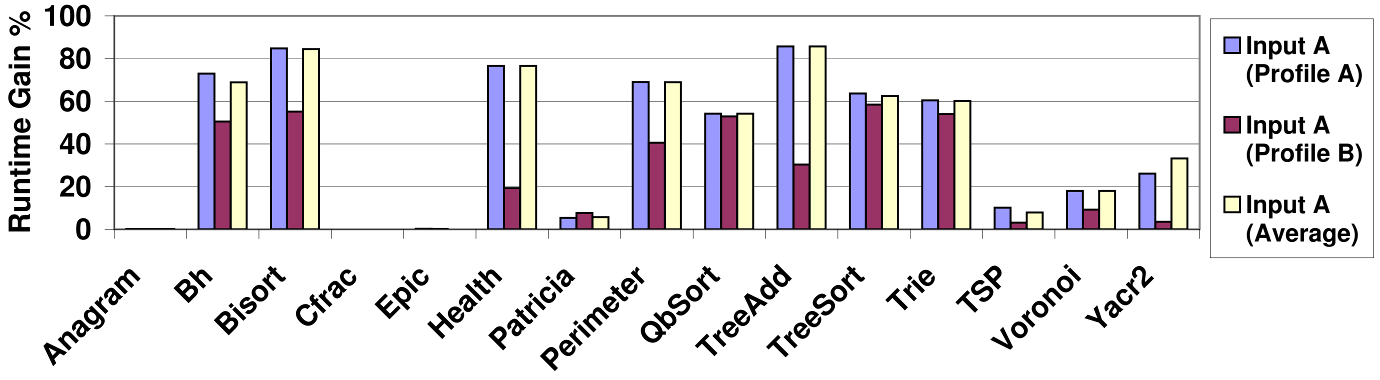


Figure 12: Runtime improvement results illustrating profile input sensitivity. (Input A)

bar shows the scenario we use profile information from input A to optimize and gather results. The second bar shows the case where we optimize based on input A's profile, but gather results using input B. The third bar shows the results from using input A when we combine both inputs profiles using our averaging pass. By examining the difference between the first and second bars, we may observe which applications are profile dependant in terms of their recursive stack allocations. We present similar results in figure 13 except these are based on input B as the primary input instead of A. In general, we saw more sensitivity when basing an allocation on a small program input profile and then using a much larger and more complex input on the optimized binary. This is reflected in the generally better performance in figure 13, where the input B set was generally more complex and consumed a much larger amount of runtime and energy than input A.

Looking at both sets of results, we find that our averaging optimization is able to greatly reduce the profile sensitivity from our allocation approach. This can be seen by comparing the three bars in each figure. In most cases, the average profile results are the same or only slightly worse when using an averaged profile versus the original profile for each input. We find that as little as two different profile inputs can significantly reduce the sensitivity of recursive stack data allocations to the input profile used. These results serve to reinforce the fact that programs making heavy use of dynamically allocated data are much more prone to input profile dependance for allocation schemes decided at compile-time.

7 Conclusion

This paper presents the first automatic scheme to allocate local (stack) data in recursive functions to scratch-pad memory (SPM) in embedded systems. With our method, all code, global, stack and heap variables can share the same scratch-pad dynamically at runtime. Our method is shown to significantly reduce

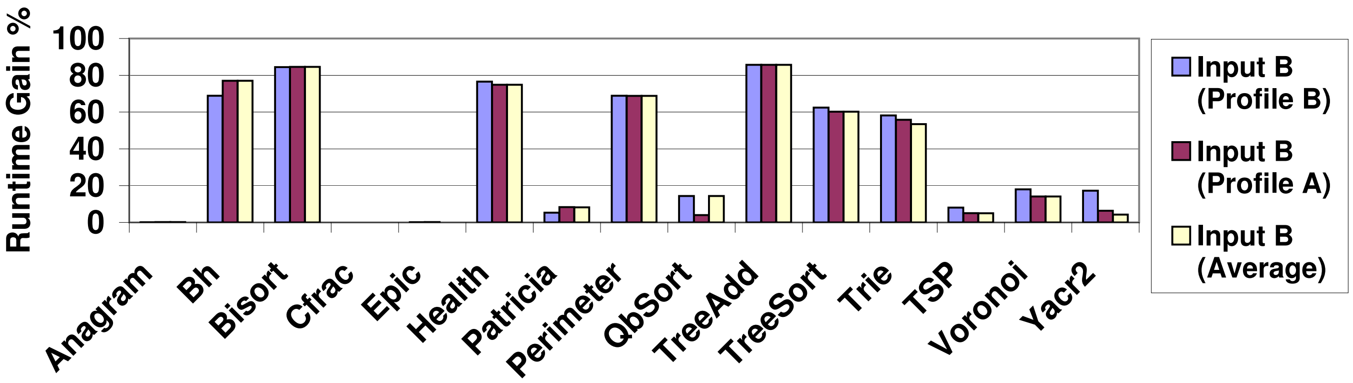


Figure 13: Runtime improvement results illustrating profile input sensitivity (Input B).

runtime and energy for applications making heavy use of recursive stack data and also outperforms cache-based schemes. Finally, we present an evaluation and solution to the input dependence problem common to profile-based allocation schemes which most commonly afflicts dynamically allocated data such as heap and recursive stack data.

References

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

[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] 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.

[4] 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.

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

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

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

[8] W. 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.

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

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

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

[12] 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.

[13] P. Harrison and H. 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.

[14] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, third edition, 2002.

[15] 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.

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

[17] 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.

[18] M. T. Kandemir, I. Kadayif, and U. Sezer. Exploiting scratch-pad memory using presburger formulas. In *ISSS*, pages 7–12, 2001.

[19] M. T. 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.

[20] O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.*, 2(1-4):151–164, 1993.

[21] T.-C. Lee, V. Tiwari, S. Malik, and M.Fujita. Power Analysis and Minimization Techniques for Embedded DSP Software. *IEEE Transactions on VLSI Systems*, Mar. 1997.

[22] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 73–82, New York, NY, USA, 1999. ACM Press.

[23] *128Mb DDR SDRAM data sheet*. (Dual data-rate synchronous DRAM) Micron Technology Inc., 2003. http://www.micron.com/-products/dram/ddrsdram/.

[24] 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.

[25] C. A. Moritz, M. Frank, and S. Amarasinghe. FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *The 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 12 2000.

[26] *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.

[27] *M-CORE - MMC2001 Reference Manual*. Motorola Corporation, 1998. (A 32-bit processor). http://www.motorola.com/SPS/-MCORE/info_documentation.htm.

[28] 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.

[29] 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/.

[30] A. Sinha and A. Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*, pages 220–225, 2001.

[31] 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.

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

[33] 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.

[34] A. D. Sumesh Udayakumaran and R. Barua. Dynamic Allocation for Scratch-Pad Memory using Compile-time Decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006.

[35] *TMS370Cx7x 8-bit microcontroller*. Texas Instruments, Revised Feb. 1997. http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.

[36] 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.

[37] *DineroIV Cache simulator*. J. Edler and M.D. Hill, Revised 2004. http://www.cs.wisc.edu/ markhill/DineroIV/.

[38] 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.

[39] 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.

[40] 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.

[41] L. Wehmeyer and P. Marwedel. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[42] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, 1996.