Scratch-Pad Memory Allocation without Compiler Support for Java Applications

Nghi Nguyen, Angel Dominguez, and Rajeev Barua
Dept. of Electrical and Computer Engineering, University of Maryland
College Park, Maryland, USA
nghi@eng.umd.edu, angelod@eng.umd.edu, barua@eng.umd.edu

ABSTRACT This paper presents the first scratch-pad memory allocation scheme that requires no compiler support for interpreted-language based applications. 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. Interpreted languages are languages such as Java that are interpreted by a run-time environment instead of being executed directly on hardware.

All existing memory allocation schemes for SPM require compiler analysis to develop the allocation strategy. Specifically, existing allocation schemes for Java-based applications determine the allocations at compile-time. They then annotate the Java bytecodes with these allocation decisions for the Java Virtual Machine (JVM) to implement the actual allocation at runtime. These existing allocation schemes tie the resulting bytecode to specific SPM sizes, therefore preventing the applications from being portable to different SPM sizes. Further, existing methods do not work for unmodified third-party bytecodes produced by compilers other than their specialized compilers.

In this paper, we propose the first ever SPM allocation scheme that is completely implemented inside the JVM. Our method requires no compiler support and works for unmodified bytecodes from any source. Moreover, unlike existing methods, it preserves the portability of bytecode to any SPM size. We investigate our method on the Sun Hotspot JVM on which we achieve a 27.8% improvement on runtime and 21.8% on energy saving versus not using the SPM – the only existing alternative for unmodified bytecodes.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-826-8/07/0009 ...$5.00.

General Terms

Performance, Algorithms, Management, Design

Keywords

Memory Allocation, Scratch-Pad, Compiler, Embedded Systems, JVM, Java, JIT

1. INTRODUCTION

Embedded systems typically employ several memory technologies to combine their advantages. Non-volatile flash memory is often used for storing code. SRAM and DRAM are the two most common writable memories used for program data and for accelerating code access. 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 run-time by storing frequently used data. The proper use of SRAM in embedded systems can result in significant run-time and energy gains compared to using DRAM alone. This gain is likely to increase in the future since the speed of SRAM is increasing by an average of 50% per year at a similar rate to processor speeds [7] versus only 7% a year for DRAM [15].

There are two common ways of adding SRAM: either as a hardware-cache or a Scratch Pad Memory (SPM). In desktop systems, caches are the most popular approach. A cache dynamically stores a subset of the frequently used data or instructions in SRAM. Caches have been a great success for desktops because they are flexible enough to be used by any executable; a trend that is likely to continue in the future. On the other hand, in most embedded systems where code is often tied to particular implementations, the overheads of cache are less justifiable. Cache incurs a significant penalty in area cost, energy, hit latency and real-time guarantees. A detailed 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 run-time 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 like 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. Recent trends indicate that the dominance of SPM will likely consolidate further in the future [6, 30], for regular as well as network processors.

A great variety of allocation schemes for SPM have been proposed in the last decade [5, 6, 8, 29, 39, 41]; more are listed in the related work. They can be categorized into static methods, where the selection of objects in SPM does not change at run time; and dynamic methods, where the selection of memory objects in SPM can change during runtime to fit the program's dynamic behavior. Even in dynamic methods, to avoid allocation overhead at run-time, the changes of SPM allocation across different program portions is usually determined at compile-time in most successful methods, leading to *statically decided dynamic methods*. Most existing methods can place code, global, and stack objects in SPM. Heap objects are more challenging to orchestrate in statically decided methods; however, the first method to successfully place heap objects in SPM has recently appeared [8].

A limitation of all the above existing SPM allocation methods is that they are compile-time methods meant for compiled languages only, such as C and C++. It is not apparent how they can be adapted to interpreted languages, such as Java. Some background is in order. In interpreted languages, a high-level source language program is compiled not into machine code, but into an architecture-independent format. In the case of Java, this format is called Java bytecode. Such bytecode is interpreted by a software called a Java Virtual Machine (JVM). Although interpretation adds overhead, this overhead is reduced by a Just-In-Time (JIT) compiler that translates frequently run code methods into machine code. JIT compilation has the advantage that it can tailor the code to the exact resources available on the client device; for this reason, Java programs now run nearly as fast as their C++ counterparts, and sometimes exceed their performance [21].

Given Java's competitive performance, and because Java bytecode is portable across platforms, Java is becoming increasingly common in embedded systems. For example, a majority of cell phones today are Java-enabled. Indeed many third-party software titles are available for download for mobile phones [11, 12, 14, 19, 37]. Sun microsystem's market-leading Java platform is available in both Standard Edition (J2SE) and Micro-edition (J2ME) versions, for desktops and resource-limited embedded systems, respectively. However, embedded systems have been deployed with the standard edition as well [25].

With this background, we can identify at least three reasons why existing methods for SPM allocation cannot handle Java applications. First, low-level details such as memory allocation purposely are not expressible in java bytecode, which is meant to be platform-independent. Hence if a Java-to-bytecode compiler did SPM allocation, it would have no way to express the allocation results in the bytecode. Consequently, SPM allocation for Java programs must be done in the JVM. Second, even if a method were to be found for expressing memory allocation in bytecode, that bytecode would not be portable to devices with a different size of SPM. This would defeat the key portability advantage of Java. Third, existing SPM allocation methods are have primarily been targeted to C, and deal with code, global, stack and heap data. Java programs have different types of data with different characteristics than C programs, such as the method stack, the operand stack, static class variables, instance-based class variables and method code. It is not immediately apparent how they should be handled by methods for C.

We present the first method in the literature for allocating data in Java programs to SPM that is applicable to unmodified Java bytecode from any source. This is because the method is implemented entirely inside the JVM, and makes no assumptions about the input bytecode. Our method first collects profile data on the access frequencies of various objects inside of the JVM. This is done to discover the most frequently accessed objects, which are the best candidates for SPM allocation. Next, our method discovers the size of the SPM on the particular device the JVM is running on by making a call to the operating system from the JVM. Subsequently, it analyzes the different types of data in the running Java application, and decides their allocation to JVM. Finally it implements this allocation by moving the objects in question to SPM during run-time, and updating their address tables inside the JVM. The result is a java application that is faster and consumes less energy, since it allocates frequent code and data to SPM.

The rest of the paper is organized as follows. Section 2 overviews different related works. Section 3 discusses our method in detail. It dicusses about the characteristics of each memory objects and how we choose them as candidates for SPM allocation from subsection 3.1 through 3.4. Section 4 discuss about how we run the profiling process for each memory-objects candidate inside the JVM. Section 5 describes the SPM allocator. Subsection 5.1 discusses the processing of profile data. Subsection 5.2 discusses the calculation process for allocation bin sizes. Subsection 5.3 discusses the allocation policy implemented inside the JVM. Section 6 presents the experimental environment, benchmarks properties, and our method's results. Section 7 compares our method with systems having caches in various architectures. Section 8 concludes.

2. RELATED WORK

There are many existing methods for SPM allocation that have studied and demonstrated the advantages of SPMs. Most of these methods are for executed code environments, such as those arising from languages such as C or C++, where application binaries are executed directly on the hardware. A few methods are for interpreted languages, such as Java, where application bytecodes are interpreted by a run-time system instead.

Compile-time methods for executed environments In executed environments, SPM allocators are mostly compile-time methods that require toolchain support to make allocation decisions at compile-time. They can be further categorized into static methods and dynamic methods.

Static methods are those whose SPM allocation does not change after the initial allocation [4–6, 16, 29, 32, 33]. Some of these methods [6, 29, 32] are restricted to allocating only global variables to SPM, while others [4, 5, 16, 33] 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 of the other static allocation methods [2, 40] aim to allocate code to SPM rather than data; while others [35, 42] can allocate both code and data.

Dynamic methods are those which can change the SPM allocation during run-time [8, 13, 26, 27, 34, 38, 39, 41]. The method in [26] can place global and stack arrays accessed through affine functions of enclosing loop induction variables in SPM. The method in [38] 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 in [41] 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 [39] also allocates code and data, but using a polynomial-time heuristic. A more complete discussion

of the above schemes can be found in [39]. The method in [8] is the first dynamic SPM allocation method to place a portion of the heap data in the SPM.

Our proposed method is inspired by some of the above methods, most notably [5]. However it adapts those compile-time methods to a Java environment where the profiling and allocation are done at run-time instead, and where the types of data are substantially different than in C programs.

Run-time SPM allocations in executed environment As far as we know, there are no existing successful run-time-only methods for SPM allocation for executed environments. The only run-time-only methods proposed so far have been software caching methods, which have largely not been successful. Software caching [13, 17, 27] emulates a cache in SRAM using software. The tag, data and valid bits are all managed by compiler-inserted code at each memory access. Significant software overhead is incurred to manage these fields, though the compiler already optimizes away some [17, 27]. All these schemes have a fundamental failing – just like a hardware cache, they have poor real-time guarantees, since in any type of cache, cache hits are the common case, but cache misses are the worst case. Cache misses degrade the worst-case execution time significantly compared to the average-case time. This defeats the primary reason for using SPM, which is to improve real-time bounds vs. caches. Moreover, existing software caching schemes have large overheads in managing the cache's fields. Consequently software caching has been widely recognized as a failure, and most recent works have focused on earlier-mentioned compile-time methods.

Unlike software caching, our proposed scheme delivers predictable run-times for any unmodified interpreted codes, and hence good real-time bounds. Further, there is no per-memory-access overhead; instead the overhead is introduced only during the profiling period of the program (which could be off-line) and never afterwards.

SPM utilization in interpreted environment For environments such as Java where codes are interpreted by a run-time system, we found little related work that utilizes the SPM. The method by Tomar, Kim and others in [22, 36] allocates only heap objects to SPM, and requires compiler support. Their Java compiler collects profile information, and then inserts annotations in the Java bytecode containing this information. The JVM is also modified to preferentially place annotated objects to SPM. The method in [22, 36] has at least the following three drawbacks compared to our SPM allocator for Java. First, their method applies only to Java bytecodes produced by their specialized compiler; unlike our method, in which the JVM can optimize unmodified bytecodes produced by *any* compiler. This is an enormous practical advantage in that our method does not require worldwide adoption of a Java compiler technology and annotation standard; instead a user can see benefits on existing bytecodes on day one, independent of adoption by others. Second, their methods can place only heap data in SPM; our methods can additionally place code, static class objects, and stack objects in SPM. This greatly expands the benefits of SPM since code objects in particular tend to be the most frequently used of all objects; others are also often frequently used. Third, their method requires a modified software toolchain comprising a modified compiler and added external instrumentation tools; thus it is no longer transparent to the development environment. Our method leaves the software toolchain unchanged; only the run-time system (JVM) needs modification.

Other related work such as [23] demonstrates the advantages of locality of small objects in Java applications by employing a garbage collector in a design with caches and SPM. In this method, small long-lived heap objects are mapped to SPM to reduce the memory traffic between the cache system and the main memory. This is a very specialized use of SPM and its benefits are mostly orthogonal from our method.

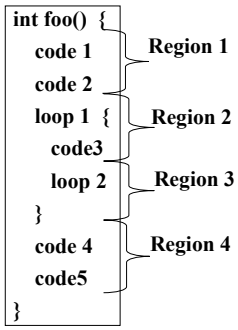

Figure 1: A large method *foo()* is divided into code regions

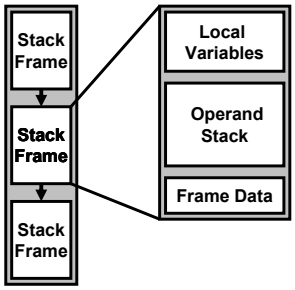

Figure 2: Java Stack and Frame Structure

3. JAVA CODE AND DATA OBJECTS, AND THEIR SPM ALLOCATION

In this paper, we propose a SPM allocation method for Java-based applications that is the first such method to be completely implemented inside the JVM. As a result, it is the first method that requires no compile-time support and works on unmodified bytecodes from any source.

The JVM is customized to implement our method as follows. The JVM first collects profiling information about the frequency of access of targeted memory objects in the program. These memory objects can be categorized into the program code regions, static class variables, stack areas, and heap objects of the Java applications. The profiling is run for a period of time at the beginning of the JVM's run-time. At the end of this period, the profiling data is provided to our allocator, which statically computes a performance-efficient SPM allocation using a greedy heuristic. From this point on, the program takes advantage of the SPM until the end of run-time.

In the rest of this section, details of the memory objects in Java programs that our method targets are presented. Thereafter section 4 will present our profiler and section 5 will present our SPM allocator.

3.1 Program Code Regions

In JVMs, the Java application's code is stored in the method section of the JVM's internal data structures. The JVM's method section contains information about the application classes, and the code for the *methods* (code routines) in each class. This section is similar to the storage area or the TEXT segment for compiled code in executed environments. The starting address, size, and contents of each class's methods are available and accessible by the JVM.

Our method partitions the code of the Java application into regions. As a first cut, the code regions are chosen as the java methods in the application. Each java method becomes a candidate for SPM allocation. Profile data for the access frequency of each method is collected. Next, the profile data is sent to the allocator along with the information about each method's size, address, and contents. The profiler and allocator will be discussed in later sections.

As an alternative to the above method-level code partition, more fine-grained code allocation can be achieved by partitioning large methods into smaller regions. Some criteria for a good choice of

region are (i) the regions should not be too big thus allowing a fine-grained consideration of placement of code in SPM; (ii) the regions should not be too small to avoid a very large allocation 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, a more fine-grained region can be 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. This choice of regions is similar to the choice of regions in [39]; that paper has more examples and discussion about this choice of regions.

In SPM allocators for executable environments, patching code is needed at the start and end of code regions allocated to SPM to maintain correct control flow. At the start of the SPM region, predecessor regions must branch to the region using patching code since it is no longer contiguous in memory; a similar branch to successor regions is needed at the end. Such patching code is also needed in our method for regions that are parts of a java method. However for regions which are entire methods, the code does not need to be patched; instead only the method's address stored in the method dispatch table in the JVM is updated. The reason is that java bytecode does not contain directly addressed calls to other methods – all calls are to symbolic addresses maintained in the method dispatch table.

In typical JVMs, hot (frequently executed) methods are compiled by a just-in-time (JIT) compiler into native code to reduce their run-time. This process is unaffected by our method. Indeed our method can store either cold (bytecode) methods or hot (native) methods in the SPM. Usually since SPM allocators place frequently executed code in SPM, one would expect all SPM methods to be hot; however this is only approximately true since the criteria for what is hot for SPM and for the JIT compiler can be different.

3.2 Static Class Variables

Objects in object-oriented languages in Java are often instances of classes, which are data types containing internal class variables (which could be public or private), and associated methods (code routines). Class variables can also be orthogonally classified as static or instance-based variables. *Static variables* of a class are allocated once and shared among all instances of that class; whereas *instance-based class variables* are allocated individually for each instance of the class. By default, variables in Java are instance-based, unless their declaration is preceded by the keyword `static` in the Java source code.

Static class variables in Java applications have similar characteristics to global variables in executed languages, in that their lifetime spans the entire program, and their total space is compile-time known. In contrast, instance-based class variables are allocated in the heap data area, since their lifetimes begin only when a *new()* method for a class is called, and end when either the program or the garbage collector frees the object. Moreover their total size depends on how many times *new()* is invoked, much like how many times *malloc()* is invoked in C.

Our SPM allocator handles static class variables much like global variables are allocated in executed languages. In particular their allocation is straightforward since their size in SPM is known. They are simply allocated in SPM along with other objects in decreasing order of frequency of access until the SPM is full (details in section 5). In contrast, instance-based variables are handled like heap objects, whose handling is more complex since their size in SPM is not known. Section 3.4 describes the allocation of heap objects in SPM.

3.3 Stack Objects

In standard JVMs, each JVM thread has a private Java stack, which is equivalent to the local stack of executed environments. The Java stack is composed of multiple stack frames, where a stack frame is created each time a method is invoked in the application. Each frame contains the local data of a class method. A frame is further divided into three components [24]: (i) local variables declared in the method; (ii) an operand stack for pushed and popped values in the Java virtual machine abstraction; and (iii) a set of frame data such as references to the runtime constant pool. An illustration of the Java stack's structure is presented in figure 2.

In our proposed method, each of the three components of a Java stack frame is considered separately as candidates for SPM allocation. This means that each component is contiguously allocated to a single memory (DRAM or SPM), but different components may be allocated to different banks. The reason that we choose this coarse-grained allocation is because the local variables in each stack frame are scalar variables (local array variables in Java are always heap-allocated); therefore each local variable is often too small to be a good candidate just on itself for SPM consideration. Moreover this choice is efficient to implement in the JVM since the address of each component is looked up in a table whose contents can be changed by the SPM allocator. In contrast, a finer-grained allocation would require additional overhead since individual fields of a component are not independently addressed; instead they are accessed using address arithmetic which would no longer be correct with successive fields being accessed non-contiguously. This would require an additional level of indirection; hence additional overhead.

3.4 Heap Data

Heap objects in Java programs arise primarily from two sources. The first is objects allocated as instances of classes. These objects contain only the instance-based class variables of those classes on the heap; the static variables are treated as globals. The second source of heap objects are array variables in methods, which in Java are always heap-allocated. This is unlike in executed languages where array variables in procedures are stack-allocated.

Each JVM designates one heap data area that is shared among all Java threads. The JVM maintains the information about each heap object, such as it's size and address. Heap objects will be considered by a straightforward adaptation of our previous work on heap allocation to SPM [8], adapted to JVMs instead of binaries.

In Java programs, although the number of references to code are the greatest, the number of references to heap are typically more than to other types of program data. Since the number of heap objects is large and unbounded at compile-time, considering each heap object in the SPM allocation process would be very inefficient and incur high overhead. For this reason, similar to the method in [8], we divide all heap data objects into groups for SPM allocation.

Different from the method in [8], we gather all the heap objects of the same class, including instances and array variables of that class, into one group. A problem for SPM allocation is that the total size of objects per group is not known at allocation time, and hence the group cannot be guaranteed to fit in SPM. This difficulty is handled by allocating only a fixed number of objects per group

```
void        Bin_Size_Calculation() {
1.   SRAM_remaining = MAX_SRAM_SIZE
2.   for(each memory-object candidate v sorted in decreasing order of LFPB) do {
3.       if(v is not a heap-object candidate)
4.           SRAM_remaining = SRAM_remaining - size(v)
5.       else { /* v is a heap class */
6.           if(LFPB(v) > 1) {
7.               bin_size(v) = SRAM_remaining × LFPB(v) / (Σ all candidates U_i such that LFPB(U_i)>1  LFPB(U_i))
8.               bin_size(v) = MIN(bin_size(v), size(v) in profile data)
                 /*size of v in profile data is the total size of all heap object belongs to class v*/
9.               bin_size(v) = next-higher-multiple-of-heap-objects-size(bin_size(v))
10.          }else
11.              bin_size(v) = 0
12.          SRAM_remaining = SRAM_remaining - bin_size(v)
13.      }
14.  }
15.  return;}
```

Figure 3: Algorithm for finding the allocated size in SPM for each candidate.

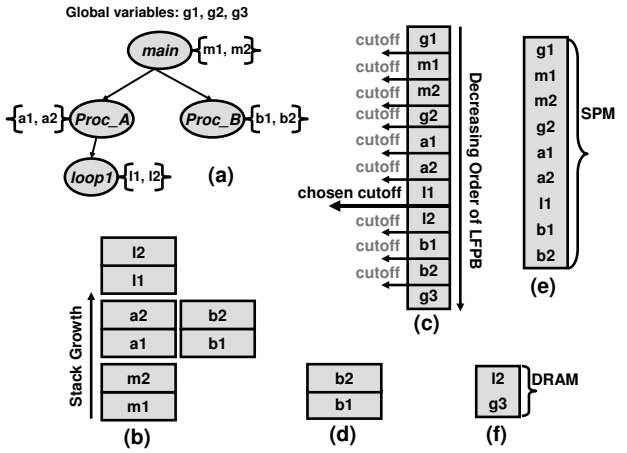

Figure 4: Allocation example showing (a) program call graph with global and local variables lists; (b) the stack of this application; (c) sorted list of all program variables; (d) bonus set; (e) SPM variable list; and (f) DRAM variable list.

in SPM in a fixed-size data structure called a *bin* in SPM. When the bin is full, any remaining objects in the group are allocated to DRAM. Intuitively, more frequently accessed groups (as revealed by profiling) are allocated larger bins to maximize the latency and energy gain from SPM allocation. The exact formula for bin-size computation will be described in section 5.2.

Since explicit de-allocations are not required in Java, a garbage collector (GC) is used to recover unreachable objects. SPM allocation does not affect GC since objects can be collected from anywhere in the memory address space, including the SPM address range. However since the free lists for the DRAM and SPM heaps are maintained separately by our SPM allocator, the GC needs a minor modification to return freed SPM objects to the SPM free list, rather than to the DRAM free list.[1]

4. PROFILING

This section discusses the profiling period of our method. The profiling period is the very first step of our method and runs at the beginning of the application's run. It measures the access frequency of each memory object class. To collect access frequency, our method keeps track of the number of time each SPM candidate is accessed during the profiling period. For each SPM candidate, a counter is maintained in the JVM that is incremented by one every time that candidate is accessed. For code-region candidate, the access frequency is counted as the total number of times any instruction in that code region is accessed during the profiling period. For static class variables, and the stack-object candidates, the access frequency's count is incremented every time these memory objects are accessed in the program. For heap objects, the access frequency of a class is incremented by one each time a variable of that class is accessed.

Profiling is run only for a certain amount of time at the beginning of an application's execution. Since it adds overhead, it is turned off after some time. The profiling time in our method is designed such that it is not too long to incur significant runtime overhead, nor too short to be inaccurate in detecting access frequency trends. With these criteria, we choose to run the profiling period from the start of JVM's run until the first time a code method is detected as a hot method by the hotspot detection mechanism. Experimentally we have found this heuristic to work well – other choices are possible, but have not been explored. More details about hotspot detection for Sun's industry-leading JVM can be found in [18].

After the profiling time, profiling code is not executed by inserting a run-time check on a *profiling-done* boolean variable. This is check incurs some overhead of its own, but much less than the full profiling code. In future work, we will explore how the overhead of profiling can be reduced by moving its code to outside larger portions of the JVM code, thereby reducing the frequency of profiling events. The results section demonstrates that the gain from our method far outweighs the costs of profiling even without optimizations. To be fair, all the overall gain numbers in this paper have subtracted the overhead of profiling, and present the net gain.

5. SPM ALLOCATION METHOD

Once profiling is complete, the actual SPM allocation is performed. This section describes the allocator.

5.1 Processing of profile data

As a first step, the allocator aggregates the profile data into metrics that are useful for SPM allocation. The primary metric we define is LFPB, defined as the follows. The most meaningful measure of the gain from placing a candidate in SPM is its *LFPB* (the product of Latency and Frequency Per Byte). The frequency per byte (FPB) is the raw frequency count of that candidate divided by the total size of all the objects in that candidate, measured in bytes. The FPB represents how often each byte is ac-

[1]Our current implementation does not implement this freed SPM object recovery. When it is implemented, the results can only improve from those presented.

Application	Source	Description	Line of Code
Crypto	Boucycastle	Cryptography	1493
LU	Scimark Benchmark	LU matric factorization	290
Calculator	Javaboutique	Calculator with advanced functions	630
FFT	Scimark Benchmark	FFT of complex, double precision data	460
Jacobi SOR	Scimark Benchmark	Jacobi Successive Over-relaxation algorithm	81
MonteCarlo	Scimark Benchmark	Integration method for approximating the value of Pi	335

Table 1: Application Characteristics

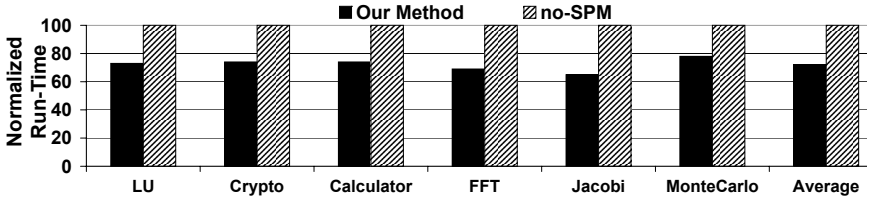

Figure 5: Runtime speedup compared to no-SPM method

cessed in one run; candidates with higher FPB result in greater gain from SPM allocation. Many embedded systems have more than one type of slow memory (for example, DRAM for data and Flash memory for code). To account for their differing latencies, the FPB is multiplied by the excess latency of the slow memory $(Latency_{slow-memory} - Latency_{SPM})$, yielding the LFPB, which directly represents the gain in cycles from placing the object in SPM.

A sorted list of memory-object candidates is also created in the decreasing order of their LFPB products. This list is later used by our allocator in a greedy heuristic to allocate memory objects onto SPM based on their LFPB value. The candidate with higher LFPB has more priority for SPM space. The LFPB value is used in this heuristic since it roughly equals to the gain in run-time of placing a candidate in SPM.

5.2 Bin Size Calculation

Next, the allocator calculates the effective SPM space to allocate to each SPM candidate. This step is mainly for heap data since for all other candidate types, it is logical to allocate a SPM space that equals to the size of the candidate. However heap data candidates, which are Java classes in our method, may contains many heap objects as described in section 3.4. Since the total number of heap objects per class is data-dependent and impossible to predict at allocation time, it is not possible to allocate the entire class to SPM. Instead section 3.4 shows how fixed size *bins* are allocated for each class. Intuitively classes with more frequently accessed objects on average should be allocated larger bins.

The steps for calculating the bin sizes for classes are presented in the algorithm in figure 3. The available SPM space is distributed to memory-object candidates in the decreasing order of their LFPB values. For non-heap objects, the candidates are allocated a SPM space equivalent to their sizes. For heap allocation bins, the SPM space is assigned according to the calculation from line 6 to line 12. Line 6 of the algorithm is to ensure that SPM space is only allocated to memory-objects that are reused in the application, meaning all portions of these objects are accessed more than once. Line 7 calculates the SPM size for an allocation bin as a fraction of the currently available SPM space that is proportional to its LFPB. Line 8 is to make sure SPM space is not allocated to a bin more than its actual size in the profile data, which is the total size of all heap objects belong to that bin. Line 10 is to ensure the allocated space is a multiple of the the size of allocated objects in that Java class, so that no space is wasted inside the bin for that class.

5.3 Allocation computation

At run-time, computationally expensive solvers such as ILP solvers will severely slow down the rewriting process, and require the embedded device to contain the solver's code. Thus ILP solvers are largely infeasible to implement inside the JVM, where a simple-to-implement low-overhead method is needed. We propose to use the greedy method in our work that will appear in [28]. That work shows that for executed environments, the greedy solver achieves over 90% of the application speedup from the optimal solver, and it has very low run-time overhead.

Our greedy solver works as follows. First, it creates a table of all SPM object candidates in the decreasing order of their LFPB. Figure 4(c) shows this table for the program in Figure 4(a). Next, the solver uses a simple greedy heuristic – objects are allocated to SPM in the order in the LFPB-sorted table, starting from the object with the highest LFPB, down to objects with lower LFPBs, until the SPM is full. The remaining objects are placed in DRAM. This intuitively makes sense since, as mentioned earlier, the benefit of placing a object in SPM is proportional to its LFPB. The boundary between SPM and DRAM objects in the table is called the cutoff point. Figure 4(c) shows all possible cutoff points for that application. It also shows the chosen cutoff for a certain SPM size. With that size, all the objects above the chosen cutoff will fit in SPM while the rest go to DRAM.

It is possible to improve this heuristic further. More candidates can fit in SPM than are chosen by the above greedy search by observing that stack objects may have disjoint lifetimes. For example in Figure 4(a), stack objects (b1,b2) have disjoint lifetimes both with stack objects (a1,a2) and with (l1,l2) – this is depicted in Figure 4(b). Thus, although objects b1 and b2 are not chosen at the cut-off point for SPM, they can nevertheless fit in SPM by sharing space with objects a1 and a2, since (b1,b2) are never live at the same time as (a1,a2). (This assumes that (a1,a2) are at least as large as (b1,b2).) These extra objects which fit in SPM (here (b1,b2)) are called *limited-lifetime bonus objects*. These are computed by analyzing the call-graph; details are in [28]. Figure 4(d) shows the bonus set for that example. Figures 4(e) and (f) show the final set of objects in SPM and DRAM, respectively.

6. RESULTS

This section presents experimental results by comparing our proposed scheme against a no-SPM allocation method, which is the only other option for Java programs on processors with SPM. Since our scheme is the first SPM allocation scheme for Java applications

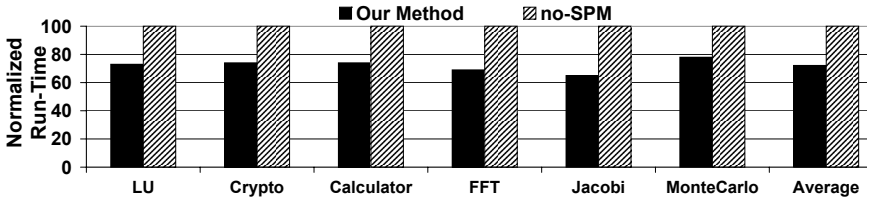

Figure 6: Breakup of runtime contributions to speedup of different object types

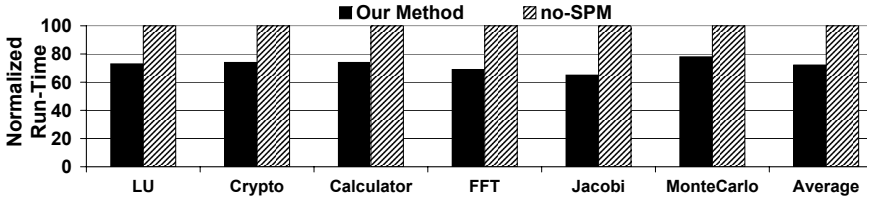

Figure 7: Energy consumption compared to no-SPM method

without compiler support, there is no other comparison applied. Future work may include further comparisons with related work in executed environments to evaluate the differences in advantages and disadvantages of the two types.

Experimental Environment In our initial experiments we have modified the industry-leading Sun HotSpot JVM to implement our methods. The JVM is then simulated by GDB for ARM v5e embedded processor family [3] to collect statistics. An external DRAM with 20-cycle latency, Flash memory with 10-cycle latency, and an internal SRAM (SPM) with 1-cycle latency are simulated. The memory latencies assumed for Flash and DRAM are representative of those in modern embedded systems [10, 20]. 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 greedily allocated to SPM by our compiler. The SRAM size is configured to be 20% of the total data size in the program. This percentage is varied in additional experiments. The benchmarks' characteristics are shown in table 1.

Runtime Speedup The run-time for each benchmark is presented in figure 5 for our method vs. the no-SPM allocation configuration. Averaging across all benchmarks, our full method (the first bar) achieves a 27.8% speedup compared to the no-SPM allocation method (the second bar). The speedup obtained here already takes into account the overhead of our method which is reported below. Ignoring the overhead, our gain in runtime would be higher at 37% across all benchmarks.

To further understand the speedup performance of our method, the figure 6 shows the individual contribution to speedup from each object type. Code-region candidates contribute the most speedup which is 14.2% on average, heap data contribute at second with 7.3%, stack objects with 4.2%, while static class variables contribute the least at approximate 2.1% across all benchmarks. This is reasonable since code and heap data are usually the two object types that are most frequently accessed in Java applications.

Energy Saving Figure 7 compares the energy consumption of our method against the no-SPM allocation method for our benchmarks. The experimental setup to measure energy use is the same as in [28]. Our method is able to achieve a 21.8% gain in energy consumption compared to the no-SPM allocation scheme. Similarly to the speedup results, this gain in energy saving already takes into account the overhead from our method. Ignoring the overhead, the gain from our SPM allocation would have been higher.

Run Time Overhead Figure 8 shows the increase in the run-time of the JVM from our implementation as percentage of the run-time of one execution of the application. The figure shows that the run-time overhead averages 9.2% across the benchmarks. A majority of the overhead is from profiling stage, where the access frequencies of memory objects are collected. The other contribution of the overhead is from the allocation stage, where the greedy heuristic is run. The overhead of our method is relatively high because the profiling stage is run completely at the runtime of the JVM. In future work, we will investigate methods to cut down this cost. We believe this overhead may be significantly reduced by combining profile events in intelligent ways.

Other overhead from our method is the JVM's code size increased due to the insertion of our implementation. This overhead is small and within 1% of the original JVM size. Besides the above overheads, there is no other overhead since our method requires no compile-time support or bytecode modifications.

Runtime vs. SPM size Figure 9 shows the variation of run-time for the LU 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, our method 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 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 objects that do not fit are not frequently used. The point of this example is to illustrate the effects of SPM sizes on our method's performances. As studied in previous work [28, 39], this relationship between SPM sizes and performance applies in general for different SPM allocation environments.

7. COMPARISON WITH CACHES

The key advantage of our method over all existing SPM allocation schemes for Java programs is that we are able to deliver good performance while not requiring any compiler support or bytecode modifications. In cache-based embedded systems, frequently used data and code are moved in and out of SRAM dynamically by hardware at run-time; therefore caches are also able to deliver good results without the compile-time knowledge of SRAM sizes. For this reason, it is insightful to evaluate our performance versus cache-based systems. In this section, we discuss several comparisons in term of performance of our method for SPM versus alternative architectures, using either cache alone or cache and SPM together.

It is, however, important to note that our method is useful regardless of the results of the comparisons with caches. This is because there are a great number of embedded architectures which have a

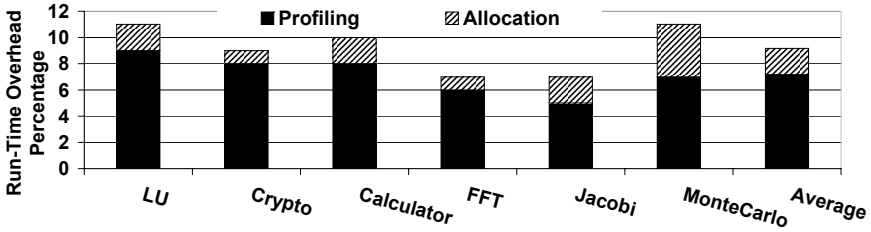

Figure 8: Runtime Overhead

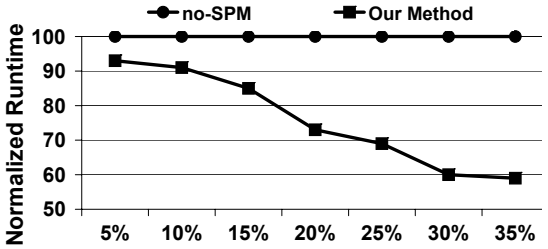

Figure 9: Runtime Speedup with varying SPM Sizes for LU Benchmark

SPM and DRAM directly accessed by the CPU, but have no cache. A detailed list of examples can be found in [28]. We have found at least 80 such embedded processors with no caches but with SRAM and external memory (usually DRAM) in our search. These architectures are popular because SPMs provide better real- time guarantees [43], power consumption, access time and area cost [2,6,35,40] compared to caches.

Nevertheless, it is interesting to see how our method compares against processors containing caches. We compare three architectures (i) an SPM-only architecture; (ii) a cache-only architecture; and (iii) an architecture with both SPM and cache of equal area. To ensure a fair comparison the total silicon area of fast memory (SPM or cache) is equal in all three architectures. 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 [31, 45]. The cache area available is split in the ratio of 1:2 among the I-cache and D-cache. This ratio is selected since it yielded the best performance in our setup compared to other ratios we tried. The caches simulated are direct-mapped (this is varied later), have a line size of 8 bytes, and are in 0.5 micron technology. The SPM is of the same technology except we remove the tag memory array, tag column multiplexers, tag sense amplifiers and tag output drivers in Cacti since they are not needed for SPM. The Dinero cache simulator [9] is used to obtain run-time results; it is combined with Cacti's energy estimates per access to yield the energy results.

Figure 10 compares the run-times of different architectures, normalized with respect to a no-SPM allocation (=100). The first bar shows the run-time with our method in a SPM-only system. The second bar shows the run-time of a pure cache-based system. The third bar shows the run-time of our method in a cache and SPM design. In this cache and SPM design, all less-frequently accessed objects that our method presumes to be in DRAM is placed in cached-DRAM address space instead; thus the slow memory transfers are accelerated. By comparing the first and second bar, we see that cache-based systems delivers better performance in absolute run-time than that of our method. However, when applying our method together with cache-based systems as showed in the third bar, it delivers the best run-time performance.

Figure 11 shows the normalized energy consumption, normalized with respect to the no-SPM allocation scheme (=100) for the same configurations as in figure 10. Our energy results are collected as the total system-wide energy consumption of application programs, including the energy use of the DRAM, SRAM, FLASH, and the main processor. By comparing the first and second bar of figure 11, we see that the SPM-only method saves less energy than the cache-only architecture. Further, the SPM + Cache combination delivers the lowest energy use.

Figures 12 and 13 measure the impact of varying cache associativity on the run-time and energy usage, respectively, on the cache-only architecture. The two figures show that with increasing associativity the run-time is relatively unchanged; for this reason a direct-mapped cache is used in the earlier experiments in this section.

In conclusion, the results show that our method for SPM delivers less improvement as compared to a cache-only architecture and that a SPM + cache architecture provides the best energy and run-time. Although, cache systems can deliver better average-case performance than our method, our method still delivers a very good performance as compared to the no-SPM allocation. Moreover, our method still has merit because of two other advantages of SPMs over caches not apparent from the results above. First, it is widely known that for code, global data and stack data, SPMs have significantly better real-time guarantees than caches [5, 35, 43].[2] Second, as described above, there are a great number of important embedded architectures which have SPM and DRAM but no caches of any type; we have counted over 80 such embedded processor types, spanning processors of all performance levels.

8. CONCLUSION

As Java is becoming more and more widespread in the embedded systems, especially in mobile devices, SPM allocation for Java-based programs is becoming increasingly important. In this paper, we introduce the first SPM allocation technique for Java applications that requires no compile-time support or bytecode modifications. This means that our method can improve third-party applications compiled with any compiler, rather than only applications compiled with specialized compilers suggested by existing methods in research. Our method is able to consider both code objects and data objects, including heap, stack, and static class objects for SPM allocation. Our proposed method is also able to share memory between stack objects that have mutually disjoint life-times.

[2]Caches improve average-case performance, but for most memory accesses, the worst case is a miss – this severely degrades real-time bounds. SPM allocation strategies for code, globals and stack are usually predictable in that the contents of SPM are precisely known at any program point; thus average case and worst case numbers are the same for SPM. This is not true for Heap data since the precise objects actually in SPM are not predictable at compile-time in our method; however most of our gain is for non-heap data. The careful quantitative study in [44] found that despite using the state-of-the-art AbsInt tool [1] for cache worst-case prediction, widely recognized as the best available, for their benchmarks, *the real-time bounds with cache were up to 3-4X worse than with SPM.*

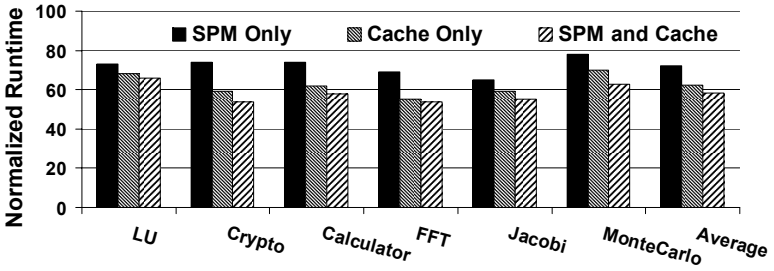

Figure 10: Comparisons of Normalized Runtime with Different Configurations

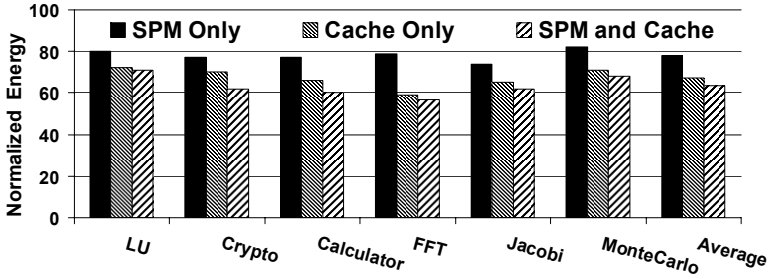

Figure 11: Comparisons of Normalized Energy Consumption with Different Configurations

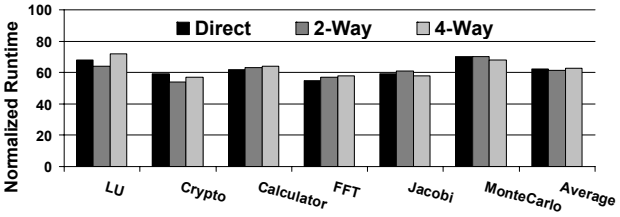

Figure 12: Normalized run-time for different set associativities for a cache-only configuration

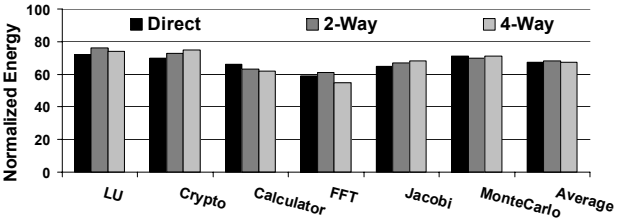

Figure 13: Normalized energy usage for different set associativities for a cache-only configuration

Our results indicate that on average, the proposed method achieves a 27.8% net speedup compared to a no-SPM allocation. This speedup is after a 9.2% runtime overhead. The overhead is mainly due the profiling stage at runtime. In future work, we plan to study different methods for reducing this overhead.

9. REFERENCES

[1] aiT: Worst Case Execution Time Analyzers. AbsInt Angewandte Informatik GmbH. 2004. `http://www.absint.com/ait`.

[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, Rev. 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] M. BOHR, B. Doyle, J. Kavalieros, D. Barlage, A. Murthy, M. Doczy, R. Rios, T. Linton, R. Arghavani, B. Jin, S. Datta, and S. Hareland. Intelt's 90 nm technology: Mooret's law and more, September 2002. Document Number: [IR-TR-2002-10].

[8] A. Dominguez, S. Udayakumaran, and R. Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. *In Journal of Embedded Computing(JEC)*, 1(4), 2005. IOS Press, Amsterdam, Netherlands.

[9] J. Edler and M. Hill. Dineroiv cache simulator. Revised 2004. http://www.cs.wisc.edu/ markhill/DineroIV/.

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

[11] GetJar. Java nokia series 60 software. http://www.getjar.com/software/Java/Nokia_Series_60.

[12] Google. Google mobile gmail. http://www.google.com/mobile/mail/index.html.

[13] 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.

[14] Handango. Java application for cell phone. http://www.handango.com/.

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

[16] 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.

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

[18] S. M. Inc. Java SE HotSpot at a Glance. `http://java.sun.com/javase/technologies/hotspot/`.

[19] Informit. Java on pocket pc devices. http://www.informit.com/articles/article.asp?p=344816&rl=1.

[20] 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.

[21] J.P.Lewis and U. Neumann. *Performance of Java versus C++*. Computer Graphics and Immersive Technology Lab, University of Southern California, Jan 2003. Updated 2004. `http://www.idiom.com/~zilla/Computer/javaCbenchmark.html`.

[22] S. Kim, S. Tomar, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Energy-efficient java execution using local memory and object co-location. In *Proceedings of Computers and Digital Techniques*, pages 33–42. IEEE, 2004.

[23] C. Lebsack and J. Chang. Using scratchpad to exploit object locality in java. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors (ICCD)*, pages 381– 386. IEEE, 2005.

[24] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. 1999. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[25] S. Microsystems. Java se embedded overview. http://java.sun.com/javase/embedded/overview.jsp.

[26] 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.

[27] 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.

[28] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *To appear in the ACM Transactions on Embedded Computing Systems (TECS)*, 2007. `http://www.ece.umd.edu/~barua/nguyen-TECS-2007.pdf`.

[29] 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.

[30] 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/.

[31] P. Shivakumar and N. Jouppi. Cacti 3.2. Revised 2004. http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[32] 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.

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

[34] S. Steinke, N. Grunwal, L. Wehmeyer, R. Banakar, M. Balakrishnan, , and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS) (Kyoto, Japan)*. ACM, 2002.

[35] 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.

[36] S. Tomar, S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Use of local memory for efficient java execution. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors (ICCD)*. IEEE, 2001.

[37] TuxMobil. Free java applications for (linux) pdas and mobile cell phones. http://tuxmobil.org/pda_linux_apps_java.html.

[38] 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.

[39] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic Allocation for Scratch-Pad Memory using Compile-Time Decisions. *To appear in the ACM Transactions on Embedded Computing Systems (TECS)*, 5(2), 2006. `http://www.ece.umd.edu/~barua/udayakumaran-TECS-2006.pdf`.

[40] 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.

[41] 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+ISSS)*. ACM, 2004.

[42] 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.

[43] 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.

[44] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.

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