Scratchpad Memory Management for Portable Systems with a Memory Management Unit *

Bernhard Egger, Jaejin Lee, and Heonshik Shin
School of Computer Science and Engineering
Seoul National University, Korea
http://aces.snu.ac.kr
{bernhard,jlee}@aces.snu.ac.kr, shinhs@snu.ac.kr

ABSTRACT

In this paper, we present a dynamic scratchpad memory allocation strategy targeting a horizontally partitioned memory subsystem for contemporary embedded processors. The memory subsystem is equipped with a memory management unit (MMU), and physically addressed scratchpad memory (SPM) is mapped into the virtual address space. A small minicache is added to further reduce energy consumption and improve performance. Using the MMU's page fault exception mechanism, we track page accesses and copy frequently executed code sections into the SPM before they are executed. Because the minimal transfer unit between the external memory and the SPM is a single memory page, good code placement is of great importance for the success of our method. Based on profiling information, our postpass optimizer divides the application binary into *pageable*, *cacheable*, and *uncacheable* regions. The latter two are placed at fixed locations in the external memory, and only pageable code is copied on demand to the SPM from the external memory. Pageable code is grouped into sections whose sizes are equal to the physical page size of the MMU. We discuss code grouping techniques and also analyze the effect of the minicache on execution time and energy consumption. We evaluate our SPM allocation strategy with twelve embedded applications, including MPEG-4. Compared to a fully-cached configuration, on average we achieve a 12% improvement in runtime performance and a 33% reduction in energy consumption by the memory system.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Design Studies; D.3.4 [Programming Languages]: Processors–*code generation, compilers, optimization*; D.4.2 [Operating Systems]: Storage Management–*secondary storage, storage hierarchies, virtual memory*

General Terms: Algorithms, Management, Measurement, Performance, Design, Experimentation.

Keywords: Compilers, Postpass optimization, Code placement, Virtual memory, Paging, Heterogeneous memory, Scratchpad, Portable systems.

*This work was supported in part by the Ministry of Education under the Brain Korea 21 Project, by MIC under the IT-SoC Project, and by MIC & IITA through the IT Leading R&D Support Project. ITC at Seoul National University provided research facilities for this study.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

1. INTRODUCTION

Contemporary portable devices are getting more powerful and include an increasing number of features. While just a couple of years ago tech-savvy consumers had to carry around a mobile phone, MP3 player, PDA, and portable TV, these days it is hard to find a phone without a built-in digital camera, MP3 player, personal organizer, and dozens of other features.

Despite this ongoing digital convergence, today's portable devices still do not live up to consumers' expectations concerning battery life. Reducing the energy consumption and thereby increasing the running time of these devices is still an important direction of research.

Most of today's state-of-the-art processors for mobile and embedded systems feature cache and/or scratchpad memories [3, 10, 12, 26]. Both cache and scratchpad memories are made of SRAM cells. Caches are composed of tag and data RAM plus management logic that makes them mostly transparent to the software. Scratchpad memory (SPM), on the other hand, consists of a simple array of SRAM cells and includes neither a tag RAM nor complex comparator logic. Usually the SPM is mapped into the physical address map as a contiguous block of fast memory. Unlike caches, it is the application or operating system's responsibility to determine what parts of the code/data are placed in the SPM. Placing the most frequently accessed parts of the program into the SPM can reduce both the energy consumption and the execution time of an application [4, 23].

Since the SPM is managed by software, placing the correct code/data segments in the SPM requires a careful analysis of the memory access patterns of the application. Allocating frequently accessed code or data to the SPM may benefit both performance and energy consumption. Static SPM allocations select a fixed set of segments to reside in the SPM during the whole execution of the application. Dynamic allocation schemes, on the other hand, modify the contents of the SPM during execution. Such algorithms usually perform loop analysis of the program trace to find the appropriate segments for each execution phase. Dynamic allocation schemes perform better than static ones for programs with several hot-spots during execution. With very few excep-

tions, most of the prior work requires that the SPM size be known at compile time and that the SPM be controlled exclusively by one application.

Today's portable devices usually do not meet these requirements anymore. Today, these devices more frequently run a full-featured operating system with a scheduler, virtual memory, and even a file system. Processes can be created and destroyed on the user's demand and at arbitrary times; the user might want to listen to music while she is composing a text message, or take a snap-shot with the integrated camera while watching her favorite TV program on her PDA.

In this paper, we present a novel scratchpad allocation technique for such systems. We propose a horizontally partitioned memory subsystem with an SPM backed up by a small instruction cache (minicache). Our design does not occupy additional die area compared to an instruction cache with the same (data RAM) size because the SPM requires significantly less die area [4]. The SPM is placed under the control of an SPM manager which itself is part of the runtime system. Processes have their own private virtual address space and may be created or destroyed at arbitrary times. Although our technique can be extended easily to include data and multiple processes, we focus on code and one process only in this paper.

In our approach, a postpass optimizer divides the executable into three different classes based on profiling information: *pageable*, *cacheable*, and *uncacheable*. Whenever a new process is created, the runtime SPM manager sets up the process' virtual-to-physical page table mappings. Main memory cacheable code is placed in cacheable memory regions and uncacheable code is placed in memory regions marked uncacheable. Pageable code resides in the main memory, but it is copied to the SPM before execution. Our postpass optimizer operates on ARM binaries, although our technique is applicable to any processor with an MMU, cache, and scratchpad memory.

The contributions of this paper are as follows. First, our SPM management is truly dynamic since pages are loaded on demand. Thus, our approach is independent of the SPM size.

Second, we propose a new, horizontally partitioned memory architecture composed of a small direct-mapped, physically-addressed minicache and scratchpad memory with 1-cycle access latency. The presence of the minicache enables even SPM-unaware programs to run with reasonable performance.

Third, our technique is implemented in a postpass optimizer. This enables us to process binaries to which the source code is not readily available, including libraries. Unlike any previous work, the SPM-optimized binary image runs unmodified with almost no performance degradation on systems without any SPM at all. We therefore achieve total memory architecture independence: SPM-optimized binaries run on systems with or without SPM, and the proposed memory architecture runs SPM-optimized as well as SPM-unaware binaries.

To the best of our knowledge, this paper presents the first approach to access physically-addressed SPM in a virtual memory environment.

We implemented the proposed memory subsystem in our cycle-accurate ARM9E-S core simulator [28]. To evaluate our approach, we used a standard MP3 decoder, an MPEG-4 encoder/decoder, and several applications from MediaBench [16] and MiBench [11]. Compared to a fully-cached configuration with virtually-indexed, physically-tagged (VIPT) caches, on average we achieved a 12% improvement in runtime performance and a 33% reduction in energy consumption by the memory system.

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 describes the memory architecture of our system. Section 4 presents the runtime SPM manager. Section 5 describes in detail the postpass optimizer and our code placement technique. Section 6 explains the evaluation environment, and Section 7 presents the results. Section 8 concludes the paper.

2. RELATED WORK

Existing work on SPM allocation can be divided roughly into two classes: statically allocated and dynamically managed scratchpad memories. In statically allocated SPM allocation, the scratchpad memory is initialized with the designated program parts at load time and its contents do not change during runtime. Dynamically managed SPM, on the other hand, is characterized by the fact that the contents of the SPM change while the program executes. The program points where code and/or data are moved back and forth from the SPM to the main memory are usually predetermined locations immediately before a substantial change in program behavior (e.g., before loops). Both statically-allocated and dynamically-managed SPM can be further classified into techniques that consider only instructions (code), only data, or both.

Static SPM allocation techniques are presented in [1, 2, 4, 9, 22, 31, 32]. Except for [22], all of these approaches require knowledge of the SPM size at compile time. In [1, 2], the authors present SPM allocation schemes that select code blocks which promise the highest energy savings using an algorithm based on Dynamic Programming. While [1] requires special hardware support to split the SPM into several partitions, [2] uses a post-pass optimizer to modify the necessary instructions so that the application runs on a unified SPM. [4] solves the static assignment with a knapsack algorithm, both for code and data blocks. In [32], memory objects are selected based on a cache conflict graph obtained through cache hit/miss statistics. The optimal set of memory objects is selected using an Integer Linear Program (ILP) variant of the knapsack algorithm. In [22], the decision of which blocks should go to the SPM is delayed until the application is loaded, making it independent from the actual scratchpad memory size. Obviously, some profiling information has to be embedded into the application binary, but the authors report only a minimal image size increase. [9] and [31] both aim at multi-tasking systems. While [9] proposes an API that helps the programmer move blocks back and forth between the SPM and the main memory, [31] is an automatic approach. The authors present three sharing strategies: *non-saving*, *saving*, and *hybrid*. In the non-saving approach, the scratchpad memory is completely allocated to the currently active task. The saving approach divides the scratchpad evenly between the applications. The hybrid approach is a mixture of the saving and non-saving method. Since [31] presents a static allocation method, both the SPM size as well as the sharing strategy must be decided at compile time.

Dynamically allocated SPM algorithms are presented in [7, 13, 14, 15, 17, 29, 30]. Both [14] and [15] focus on data arrays accessed from well-structured loop kernels. Arrays are split into so-called *tiles* to allow only parts of an array to be copied to the SPM, which allows arrays bigger than the SPM size to be allocated to the SPM. Also in [17], the authors focus on assigning data arrays to the SPM. To determine the most beneficial set, they first assign registers to so-called *register classes* based on their size. Each register class gets a fixed share of the SPM. Using a conflict graph of live ranges, a graph coloring algorithm determines which array is to be allocated to the SPM at what program points. In [29], code blocks are dynamically copied to the SPM directly before loops. The optimal set is determined using an ILP. [30] focuses on performance optimization and considers local and global data. The program's control-flow graph (CFG) is annotated with timestamps to form a *data program relationship graph (DPRG)*. Using greedy heuristics, the most promising candidates are copied to and from the SPM at well-defined copy points. In [7], loop nests are copied on demand to the SPM. The most beneficial set of loops for a given SPM size is determined by solving an ILP. A dynamic SPM allocation for code is presented in [13]. The allocation is based on the so-called *concomitance* metric, which indicates how correlated in time the execution of various blocks of code are.

Finally, in [6], the authors propose an SPM allocation scheme for heap data. Promising candidates are assigned a fixed-size *bin* that can hold n elements of a dynamically allocated variable. At runtime, only the first n objects are allocated to the SPM. While the bins are fixed in size, their memory location may change during program execution, making this allocation technique a dynamic one.

The horizontal partitioning of memory architectures has been examined recently in [27]. Inspired by the memory architecture of the Intel XScale architecture with a big main data cache and a 2KB minicache, the authors show that, by cleverly allocating the data objects to one of the caches, a substantial amount of energy can be saved.

3. MEMORY SYSTEM

As portable devices get more powerful, they diverge increasingly from traditional embedded systems with a shared physical address space and a set of well-known applications. Portable devices now run full-fledged operating systems with virtual memory management, and tasks are created and destroyed at arbitrary times on the user's demand. Furthermore, cache and SPM configurations vary from device to device even though their processor cores are the same. These factors have limited the application of scratchpad memory allocation algorithms that are tailored to a certain SPM size and assume a fixed set of tasks running in one address space.

Existing memory subsystems of embedded cores that support physically addressed scratchpad memory mapped into a virtual address room are restricted in one of two ways: either the SPM access latency is longer than one cycle, or the SPM is accessed simultaneously with the cache for each request, which wastes energy because only one of the memory structures can contain the requested data.

In the first case, the MMU first translates the virtual address (VA) to a physical address (PA). The PA is then compared to the SPM base register, and the SPM is accessed only if the PA lies within the SPM address range. Unless

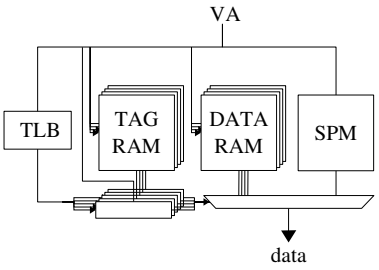


Figure 1: ARM11 level one cache block diagram

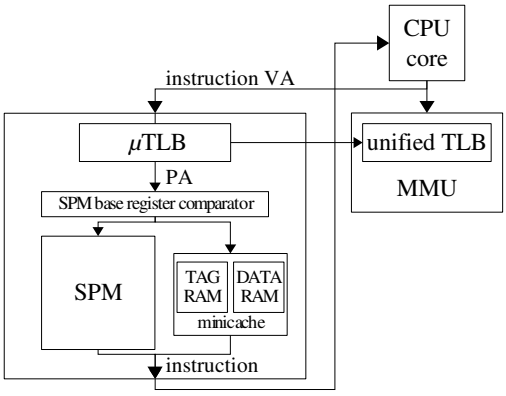


Figure 2: On-chip memory architecture

the translation lookaside buffer (TLB) misses, the VA-to-PA address translation requires one cycle; thus, an SPM access can usually be handled in two cycles. The ARM926EJ-S core [3], for example, implements such a design. Other designs, such as the ARM11 core [3], access the SPM and the cache simultaneously (Figure 1). At the same time, the address translation is performed by a MicroTLB (μTLB) [20], which is basically a fully associative cache with 2 to 16 entries providing fast lookups of recently used page table entries. The usual cache hit signals plus special SPM range hit signals are then used to select the correct data from one of the cache sets or the SPM. While the latency of the cache and the SPM is one cycle, both are active in every memory request, thereby wasting a significant amount of energy. The XScale's horizontally-partitioned, virtually-addressed cache architecture seems to suffer from the same problem [12].

To eliminate these difficulties, we propose the following horizontally-partitioned, on-chip memory subsystem for the instruction side of a Harvard architecture (see Figure 2). The subsystem consists of a μTLB, a scratchpad memory, and a direct-mapped minicache. The μTLB first translates the VA of the instruction fetch issued by the core. The resulting PA is checked against the SPM base region register to determine whether the request should be routed to the SPM or the mini cache. If the address lies outside the SPM range, the physically addressed minicache is accessed.

The purpose of the minicache is to cover the external memory access cost caused by those instructions that are not executed from the SPM. As we will show in Section 5, under certain conditions, it might not be worth executing instructions from the SPM because the additional energy consumed by copying them to the SPM is higher than the energy consumed by running them from the minicache.

16-entry μTLB	1.14ns
16KB SPM	0.81ns
1K direct-mapped cache	0.81ns
Total latency (μTLB $\rightarrow$ SPM or cache)	1.95ns

Table 1: Access Latencies

Size [KB]	Cache (4-way, 32B line)		SPM		SPM + 1KB minicache
	energy [nJ]	area [mm^2]	energy [nJ]	area [mm^2]	area [mm^2]
4	0.55	0.53	0.14	0.29	0.45
6			0.16	0.39	0.55
8	0.56	0.73	0.18	0.49	0.65
12			0.19	0.71	0.87
16	0.61	1.19	0.21	0.94	1.10

Table 2: Comparison of energy per access and die area for cache and SPM (0.13μm technology)

The data side of the first level memory hierarchy remains unchanged and implements a standard virtually-indexed, physically-tagged cache.

The disadvantage of our design is that serializing the VA-to-PA address translation with the SPM/cache access increases the latency of an instruction fetch by approximately the access time of the μTLB. With the current 0.13μm manufacturing process, however, core clocks of up to 500MHz can be supported with a 1-cycle latency. Table 1 shows access latencies for both μTLB, cache, and scratchpad memories. The numbers were obtained from CACTI [33].

For clock frequencies above 500MHz, a possible solution is to split the instruction fetch pipeline stage into two separate parts, effectively increasing the pipeline depth by one.

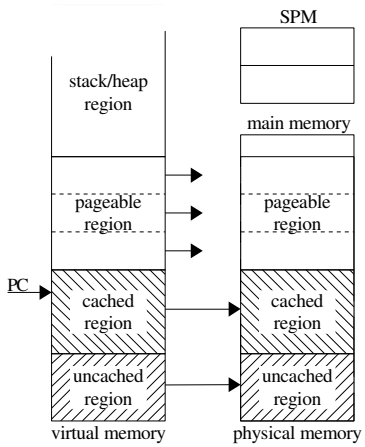
The SPM is more efficient than a cache of the same size in terms of both energy and area. Table 2 shows energy and area parameters for various cache and SPM sizes. For example, a 4KB, 4-way set associative minicache with a line size of 32 bytes occupies a die area of $0.53mm^2$. A 4KB SPM ($0.29mm^2$) plus a 1KB, direct-mapped cache with a line size of 32 bytes ($0.16mm^2$) requires only 85% of the cache area. Even a 6KB SPM ($0.38mm^2$) together with the aforementioned minicache increases the required die area by only about 5%.

Unlike a cache, a scratchpad memory is not managed by hardware. Instead, the runtime system is responsible for its contents and efficient utilization. This is the subject of the next section.

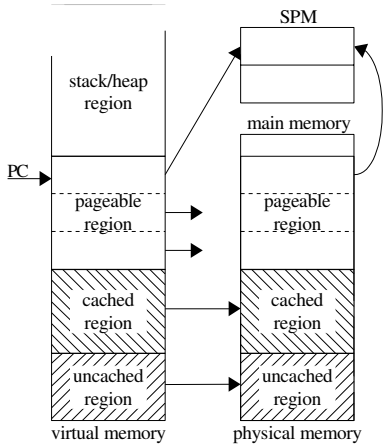
4. SPM MANAGEMENT

In this section, we describe the runtime support that is necessary to run a specially prepared application binary efficiently on the proposed memory architecture. While on cached cores, binaries run without further software support, the SPM has to be explicitly managed either by the application itself or by a dedicated SPM manager as a part of the runtime environment.

As mentioned before, we believe that techniques that allow the SPM to be controlled entirely by the running application are particularly well-suited for embedded systems where the set of running tasks and the size of the SPM are known in advance. However, the optimized binaries are tailored to one configuration, and might not (or only inefficiently) run on different configurations.



(a) Memory layout after loading the application binary. Pageable code is not mapped to generate a prefetch abort exception when it is entered. The PC is set to the image entry point.



(b) Memory layout after the PC has entered pageable code. The memory page that caused the fault is loaded into SPM, the PTE entry is changed accordingly, and execution continues from the SPM.

Figure 3: Operation of the SPM Manager

The SPM management technique described in this section depends neither on a certain size of the SPM nor on a predetermined set of running applications. The technique is fully integrated into the runtime environment. Our runtime scratchpad memory manager (SPMM) manages the SPM as a global resource. The SPMM uses demand-paging techniques similar to those in virtual memory systems [8] to track the course of the application.

Pageable code (that is, code that has been identified to be executed from SPM) is allocated to pages whose size corresponds to that of a virtual memory page (see Section 5). When an application binary is loaded, the runtime environment sets up its virtual-to-physical memory mappings by allocating MMU page tables that map the expected virtual memory layout to real physical pages.

At the beginning, the mappings of pageable code are disabled by clearing the respective bits in the page tables (Figure 3 (a)). After the page table mappings have been set up, the application starts to run by setting the program counter (PC) to the designated entry point. As soon as the PC

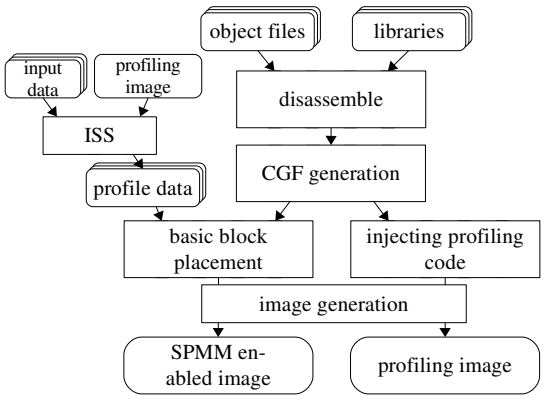


Figure 4: The post-pass optimizer

reaches pageable code, the MMU will generate a *prefetch abort exception* due to the disabled page table entry (PTE) in the page tables of the MMU.

The runtime environment forwards the prefetch abort exception to the SPMM. The SPMM loads the page into the SPM and modifies the corresponding PTE to map the virtual address range of the page in question to the SPM. Then, it enables the PTE and restarts the aborted instruction fetch (Figure 3 (b)). The code now runs without further interruption until it encounters another disabled PTE, and the whole page loading mechanism is invoked again. For pages residing in the SPM, no additional cost occurs when the execution of the application reaches such a page.

Depending on the size of the pageable code and the number of SPM pages available, pages may need to be evicted before a new page can be loaded into the SPM. Since code pages are only read from but never written to, the SPMM does not need to copy the page back to main memory; it simply overwrites the old page with the contents of the new one. The PTE of the evicted page has to be invalidated, however, to trigger invocation of the SPMM as soon as that page is entered again. For this purpose, the SPMM keeps track of which pages in the SPM are occupied and which pages are free. The SPMM also stores the original PTE of each code page currently residing in the SPM. Currently, the SPMM uses a round-robin policy for page replacement.

Our demand paging for SPM is independent of the SPM size and directly applicable to multiple tasks. Furthermore, since all addresses are virtual addresses, no code patching is necessary. An SPMM-enabled application runs just as well as an unmodified binary on a cached-only architecture. The opposite, of course, is not the case.

As should be clear now, the pageable code region should contain only code sections that are worth being copied to the SPM before execution. In an unmodified application binary, frequently executed code is randomly spread over the whole binary image. The next section discusses in more detail our postpass optimizer and how it modifies the binary to achieve optimal results in conjunction with the SPM manager.

5. THE POST-PASS OPTIMIZER

5.1 Overview

Figure 4 shows the organization of our postpass optimizer called SNACK-pop that is part of our **S**eoul **N**ational University **A**dvanced **C**ompiler tool **K**it [24]. SNACK-pop operates on the ARM/Thumb instruction set, including the DSP extensions. ARM floating point instructions are not supported. Using a postpass optimizer has several advantages. First, any binary can be optimized for our scratchpad allocation technique without requiring access to the source code and recompiling the application. Second, a postpass optimizer enables whole program optimization, including libraries, which is impossible at the source level. Finally, since optimizations concerning code layout are rather low-level in nature, postpass code arrangement is well-suited for this purpose.

```
foo()       ...
    0x14    add     r1, pc, #14         ;=#0x30
    0x18    bl      strcmp
            ...
    0x30    dcb     "imahugabimbo", 0
```

(a)

```
foo()       ...
    0x14    ldr     r1, [pc, #14]       ;=#0x30
    0x18    bl      strcmp
            ...
    0x30    dcd     foo_string          ;=#0x5c

global data
    foo_string
    0x5c    dcb     "imahugabimbo",0
```

(b)

Figure 5: Constant data extraction from local constant pools

The inputs to our post-pass optimizer are application binaries and libraries in the ARM ELF file format. SNACK-pop disassembles the object files into code and data segments, and resolves all undefined symbols. In the next step, the code blocks are further divided into functions composed of basic blocks. Branches with hard-coded offsets are resolved and replaced by relocation information to enable SNACK-pop to freely relocate code.

Constant data residing in local data pools requires special attention. Most ARM compilers place constant data used in a function into the function's constant pool. Consider, for example, a function *foo()* that contains a call to the *strcmp()* string comparison function with one of the strings hardcoded in the source. The string is placed into *foo*'s constant pool which is included in *foo*'s code segment (Figure 5 (a)). One of the arguments to *strcmp()* is a pointer to the constant string and is generated by adding the offset from the current PC to the first character of the embedded string. Now imagine that both *foo()* and *strcmp()* are located in the pageable code region, but not in the same page. If the SPM is full, and the call to *strcmp()* causes the SPMM to load the page containing *strcmp()* into the SPM, *foo*'s page might get evicted. Since *strcmp()* contains a reference to *foo*'s constant pool, the access of this data would be aborted immediately, causing *foo*'s page to be reloaded. Excessive thrashing could result in extreme cases when only few SPM pages are available.

To avoid such scenarios, whenever SNACK-pop detects the passing of pointers that point to a function's constant pool, it extracts the referenced constant data from the func-

tion and places it in a global data region that is not pageable. The call is then modified to pass a pointer to the global data. (Figure 5 (b)).

To generate an SPMM-optimized binary, SNACK-pop first inserts instrumentation code into each function and basic blocks to gather profiling information. The profiling image is run on an instruction set simulator (ISS) to get the call/return profile. The unmodified reference image is also run to obtain a trace of all instruction fetches during the execution of the application. Several profiles and instruction traces are generated with different training data sets.

In a second step, those profiles are again fed into SNACK-pop to generate an SPMM-enabled binary. SNACK-pop computes the average number of reads for each basic block. For each block, the energy model described in the following subsection is solved and the basic block is assigned to one of the *paged*, *cached*, or *uncached* code regions. This process is described in greater detail in Subsection 5.3. Once the code placement is done, SNACK-pop generates a new ELF binary and inserts six new symbols that contain the location and size of each of the three code regions *paged*, *cached*, and *uncached*. When loading an ELF binary, the SPMM manager searches the image for these six symbols. If they are present, the memory mappings are setup accordingly. If not, the image is an unoptimized image and will use only the minicache.

5.2 Code Classification

Based on the trace profile, SNACK-pop determines the code region for each block b_i:

$$Loc_i = \begin{cases} uncached & \text{if the code is executed less than once} \\ cached & \text{if } E_{cached}(b_i) < E_{paged}(b_i) \\ paged & \text{otherwise} \end{cases}$$

Code that is executed less than once on average is assigned to an *uncached* code region. Such code would generate only cache misses and no hits. Due to the spatial locality of a cache line, it might run slightly faster from cache, but since it is executed once at most, the performance penalty is negligible. For blocks executed more frequently, the following energy model is used:

$$E_{paged}(b_i) = A_i E_{spm} + MS_i(E_{ext} + E_{spm}) \quad (1)$$

$$E_{cached}(b_i) = A_i(E_{cache} + m_{cache} E_{miss}) \quad (2)$$

where

A_i	number of instructions fetched from block b_i
S_i	size of block b_i in words
M	average number of page misses
m_{cache}	cache miss ratio
E_{spm}	SPM access energy
E_{ext}	external memory access energy
E_{cache}	cache access energy
E_{miss}	cache miss energy

The first term in (1), $A_i E_{spm}$, represents the energy required to execute block b_i from the SPM. The second term, $MS_i(E_{ext} + E_{spm})$, computes the cost of copying block b_i from main memory to the SPM. The empirical factor M is used to consider the fact that a block might get copied to the SPM several times. Note that we consider the pure copy cost only and not the additional overhead of the SPMM. Basic blocks are usually much smaller than one memory page so that the SPMM cost does not have a big impact on a single basic block. The energy consumed when executing block b_i from the cache is computed by (2).

This computation is performed on the function level for each basic block. We then perform function splitting similar to the one described in [25]: first, the basic blocks are reordered according to their intended storage location. The reordering might invalidate some fall-through edges in the control flow graph. Additional branch instructions are inserted as needed to restore the correct control flow. Next, the function f_k is divided into up to three parts f_k^{paged}, f_k^{cached}, and $f_k^{uncached}$ in such a way that basic blocks that should be executed from SPM are located in f_k^{paged}, and accordingly for *cached* and *uncached*.

The cached and uncached functions f^{cached} and $f^{uncached}$ are then allocated to the cached or the uncached code segment, respectively. For the paged functions f^{paged}, we apply a clustering heuristics as described in the next subsection.

5.3 Pageable Code Arrangement

Intuitively, an optimal page placement must

- allocate the pageable code to as few pages as possible
- cluster temporally local code together in as few pages as possible

The optimal page placement problem is harder than Knapsack because the code must not only be allocated to as few pages as possible, but we also have to consider the temporal relationship between all candidates.

We, therefore, have developed the following heuristics that work reasonably well for a wide range of benchmarks. Figure 6 illustrates the steps of the heuristics on a running example. We assume a page size of 128 bytes.

In the first step, we detect loops by looking at the dynamic call graph (DCG). What we call *loop* is not a loop in the traditional sense, but its effect is the same as that of a loop at the source level. Figure 6 (a) shows an example of a DCG depicting eight functions and their size.

Let $\mid a \rightarrow b \mid$ denote the weight of the edge $a \rightarrow b$ in the DCG (that is, the number of calls from a to b). Also, $\mid * \rightarrow b \mid$ denotes the sum of the weights of all incoming edges to b. We consider a function f to be a *loop header* if

$$\frac{\mid g \rightarrow f \mid}{\mid * \rightarrow g \mid} \geq threshold$$

(i.e.,the number of calls $g \rightarrow f$ divided by the number of incoming calls to g exceeds a certain threshold value). In Figure 6 (b), the functions c and f have been identified as loop headers.

For each loop header hd, the members in the loop with loop header hd are identified by computing the *closure* of the loop, $closure(hd, hd)$. The function *closure* is recursively defined by

$$closure(hd, f) = \bigcup closure(hd, h), \forall h, \frac{\mid f \rightarrow h \mid}{\mid * \rightarrow hd \mid} \geq 1$$

(i.e., the loop consists of all functions h that are called at least as many times as the loop header itself). In Figure 6 (b), $closure(c, c) = \{c, e, f, g, h\}$ and $closure(f, f) = \{f, h\}$. Note that function g is *not* a member of $closure(f, f)$ because the number of calls to f, $\mid * \rightarrow f \mid$ is bigger than the number of calls from f to g. Function g is, however, a member of $closure(c, c)$.

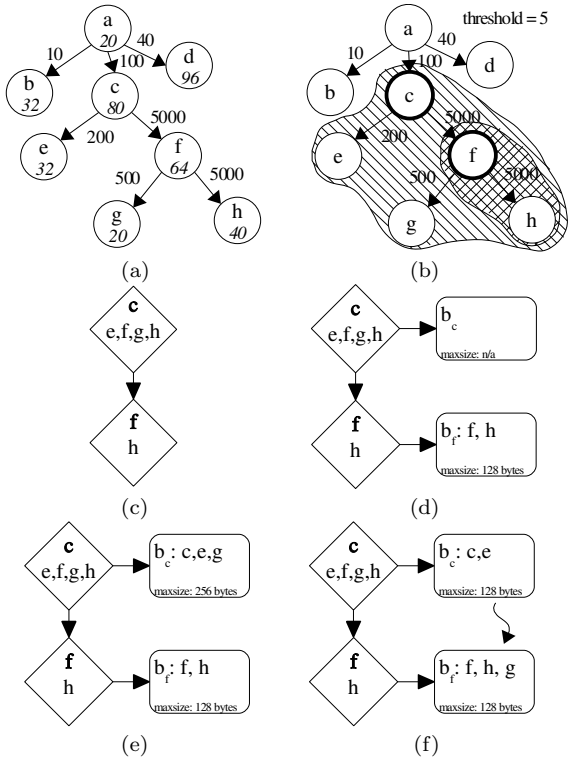


Figure 6: Code placement example

After detecting all the loops in the DCG, we build the loop call graph (LCG), which is simply a directed graph with the loops as nodes and an edge between loop l_1 and l_2 if loop l_2 is an inner loop of l_1 (Figure 6 (c)).

The LCG is then traversed in a depth-first manner (i.e. the innermost loops are processed first). For each loop l_i a bin b_{l_i} is allocated. We insert all functions f^{paged} to bin b_{l_i} that are contained in l_i and have not yet been allocated to any other bin. After all nodes in the LCG have been processed, the maximum size of a bin is set to

$$size_{max}(b_{l_i}) = \lceil \frac{\sum_{f \in b_{l_i}} size(f)}{pagesize} \rceil \cdot pagesize \qquad (3)$$

where *pagesize* denotes the size of one memory page. Figures 6 (d) and (e) show the state of the LCG and the associated bins after processing f and c, respectively. Note that even though loop c contains the functions c, e, f, g and h, only c, e, and g are assigned to bin b_c because f and h have already been placed in bin b_f.

Next, we consider all non-leaf nodes of the LCG (i.e., loops containing inner loops). Using the *bestfit* algorithm [5], we push functions allocated to the outer loop bin into the bins of its inner loops as long as the size of the inner loop's bin b_{li} does not exceed $size_{max}(b_{li})$. The idea here is to pack functions together that have a close temporal relationship to achieve a better utilization of the bins and to reduce their internal fragmentation. Here it also becomes clear why we have calculated a maximum bin size. If a bin could grow without any constraint, all outer loop functions would simply get pushed down to the bin of the innermost loop. This would result in huge bins containing a lot of functions that do not necessarily have a close temporal relationship.

Memory	Read	Write
cache	1	1
SPM	1	1
SDRAM		
non-sequential	27	27
sequential	24	24

Table 3: Access latencies in CPU cycles

After no more functions can be pushed to inner loops' bins, the maximum size of the outer loop bin is recalculated according to (3). In Figure 6 (f), function g is pushed to b_f and the maximum size of bin b_c is reduced to 128 bytes.

Functions that are to be paged, but are not part of any loop, are placed last. For each unplaced function f_k^{paged}, we follow the DCG up towards the root. For each caller encountered on the way up, we compute a fictitious loop with a threshold of one. If that loop contains both f_k^{paged} and another loop l, we try to include f_k^{paged} in the bin b_l. Any remaining functions are allocated to an extra bin.

6. EVALUATION ENVIRONMENT

We have evaluated the effectiveness of our approach using SNACK-armsim [28]. SNACK-armsim is a cycle-accurate architecture simulator that models the ARM9E-S core. It supports the ARMv5TE instruction set and includes timing models for the pipelined ARM9E-S core, the MMU including the unified TLB, caches including μTLBs, the AMBA AHB bus [3], and external memory. For this work, we also have modeled the on-chip memory architecture presented in Section 3. The processor core clock in SNACK-armsim was set to 200MHz and the caches were virtually-indexed and physically-tagged. The cache, SPM, and external memory (SDRAM) latencies are shown in Table 3.

We use the total execution time as the performance metric and the total energy consumed in the memory subsystem as the energy metric. The components of the memory subsystem include the on-chip instruction and data caches, the SPM, the off-chip bus, and the main memory (SDRAM). The energy consumption by the memory subsystem is computed by

$$\begin{aligned} T_{total} &= \frac{core\ clocks}{core\ frequency} \\ E_{total} &= E_{SPM} + E_{icache} + E_{dcache} + \\ &\quad E_{ext\_static} + E_{ext\_dynamic} \end{aligned}$$

The energies for cache and SPM are computed by

$$\begin{aligned} E_{cache} &= e_{cache}(hit + \frac{linesize}{4} \cdot miss) \\ E_{SPM} &= e_{SPM}(read + write) \end{aligned}$$

The SDRAM energy is composed of static and dynamic energy [19]. We have modeled the low power 128MB Micron MT48H8M16LF [18] SDRAM with a memory bus frequency f_{mem} of 100MHz and a supply voltage V_{dd} of $1.8V$. The static energy consumption includes the standby power and the power to periodically refresh the SDRAM cells. The energy of off-chip memory accesses E_{ext_static} is the SDRAM's static energy and is given by

$$E_{ext\_static} = P_{standby} \cdot T_{total}$$

$P_{standby}$ is the static power consumption of the SDRAM and is computed from [18]. $E_{ext\_dynamic}$ includes both the bus

cache	nJ	SDRAM (dynamic)	nJ
1KB, 1-way	0.20	random read	11.75
4KB, 4-way	0.55	burst read	26.98
8KB, 4-way	0.56	random write	10.40
16KB, 4-way	0.61	burst write	13.27
SPM	nJ	SDRAM	mW
4KB	0.15	static power	57.30
6KB	0.16		
8KB	0.18		
12KB	0.19		

Table 4: Access energy and power parameters. The bus energy is included in the SDRAM dynamic energy. The burst length is 8 words.

Application	Code Size [KB]	Reference ICache
Combine	12	4KB, 4-way
FFT	14	4KB, 4-way
Epic	20	4KB, 4-way
Unepic	20	4KB, 4-way
MP3-dec	24	8KB, 4-way
MPEG4-dec	44	8KB, 4-way
MPEG4-enc	49	8KB, 4-way

Table 5: Benchmarks

and SDRAM dynamic energy. The dynamic energy of the SDRAM is also computed from [18]. The per-access bus energy is taken from [27]. Table 4 shows the parameters used in our energy calculations.

We use 12 embedded applications to evaluate our work. These include nine benchmarks from MiBench [11] and MediaBench [16], the official ISO MP3 decoder [21], and MPEG-4 XviD encoding/decoding [34]. We put the benchmarks *Quicksort, Dijkstra, SHA, ADPCM-enc, ADPCM-dec,* and *Bitcount* together into one benchmark called *Combine*. Each of the smaller benchmarks is executed once in *Combine*. *Combine* represents an embedded application with multiple phases. Table 5 summarizes the characteristics of each benchmark. We set $M = 2$ and $threshold = 4$ for the clustering algorithm (Section 5) for all benchmarks.

The reference cases were obtained using a fully cached system with 4-way associative virtually-indexed, physically-tagged instruction and data caches. For the smaller benchmarks (*Combine, Epic, FFT,* and *Unepic*), the instruction cache has a size of 4KB, and 8 kilobyte for the bigger ones (*MP3, MPEG4-dec*, and *MPEG4-enc*). The data cache has a fixed size of 16KB.

7. EXPERIMENTAL RESULTS

7.1 Code Placement

Table 6 shows the results of our code placement algorithm with 1KB page size, as described in Section 5.3. On average, each memory page of the *pageable* region is 85% filled (i.e. 15% of each page are unallocated). While the SPMM is aware of the actual page size and loads only as many bytes into the SPM as needed, the unallocated space cannot be used to hold other code or data because the minimal allocation unit is one memory page. Our code placement strategy does not consider whether the total amount of pageable code can be easily allocated to 1KB pages. It might, for example, allocate 2052B of code to the pageable region that will inevitably require three pages (1024B + 1024B + 4B). In addition, the clustering algorithm groups code at the function level, which also leads to internal fragmentation.

Application	pageable	cacheable	uncacheable	pages
combine				
codesize	5.4KB	2.9KB	3.7KB	7
# of instr.	191.1M	5.7K	357	
fft				
codesize	6.5KB	3.0KB	4.5KB	8
# of instr.	91.4M	3.7K	133	
epic				
codesize	7.6KB	5.5KB	6.9KB	8
# of instr.	329.9M	15.0K	635	
unepic				
codesize	5.8KB	7.4KB	6.8KB	7
# of instr.	30.6M	24.2K	499	
mp3-dec				
codesize	17.6KB	0.8KB	5.6KB	20
# of instr.	82.1M	2.8K	147	
mpeg4-dec				
codesize	16.3KB	7.4KB	20.3KB	19
# of instr.	54.4M	16.7K	1.6K	
mpeg4-enc				
codesize	14.8KB	7.7KB	22.5KB	19
# of instr.	35.6M	18.6K	2.3K	

Table 6: Code placement for memory pages of 1KB size. The fourth column shows the number of pages allocated to pageable code.

7.2 Execution time and Energy Consumption

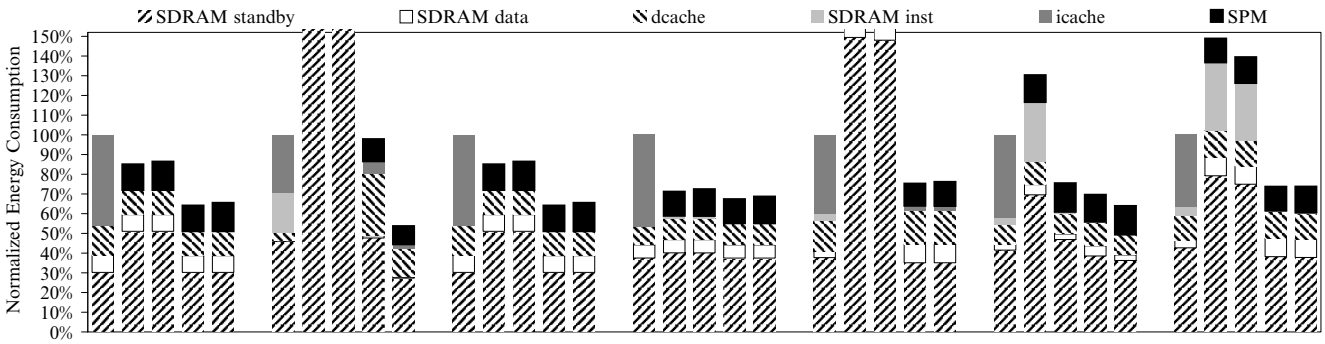
Figure 8 (a) shows the energy consumption and execution time of our SPM allocation technique for different configurations. Reference is the reference case with instruction and data cache, but no SPM. For each application, the size of the instruction cache is set to 20-30% of the executed code size (Table 5). The energy consumption of the different configurations is normalized to reference. Table 7 shows the number of page faults in addition to the energy consumption and execution time of our technique. Figure 8 (b) displays the total number and distribution of instruction fetches from the different memory components. For most applications, the instruction cache hits of reference are completely absorbed by the SPM. This shows that the code classification is very effective. A significant number of instructions are fetched from the minicache/SDRAM only for applications with a large number of page faults (fft and mp3-dec) because the SPM manager itself is never executed from SPM.

Without the minicache. As shown in Table 7, without a minicache, the average execution time is 231% when the SPM size equals the instruction cache size of the reference case, or 188% for 1.5 times the cache size. The memory subsystem energy consumption is also higher than reference with 165% and 132%, respectively. This is somewhat surprising since Table 6 shows that practically all instruction fetches are allocated to pageable code regions. A closer look at the results in Table 6 reveals that the number of page faults has a direct influence on both the execution time and energy consumption. Moreover, more than 99% of all instruction fetches from SDRAM is caused by the SPMM. This effect is shown in Figure 8 (b) with bigger SDRAM inst sections in the bars without the minicache for *fft*, *mp3-dec*, *mpeg4-dec*, and *mpeg4-enc*.

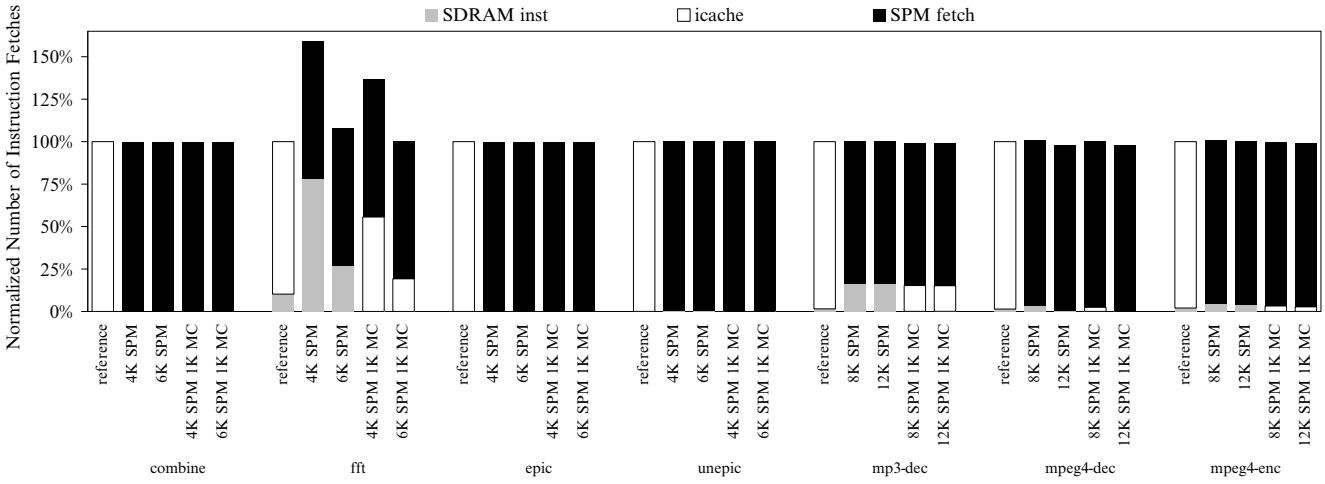
With the minicache. With a 1KB direct-mapped minicache, the average execution time and energy consumption are 97% and 72% for an SPM size equal to the instruction cache in reference, and 88% and 67% for an SPM size that is 1.5x the size of the cache. If the SPM is saturated, increasing the SPM size from 1.0x to 1.5x will slightly increase the energy consumption of *combine*, *epic*, *unepic*, and

Configuration:	no minicache						1KB, direct mapped minicache					
SPM size:	1.0x reference cache size			1.5x reference cache size			1.0x reference cache size			1.5x reference cache size		
Application	time	energy	pagefit	time	energy	pagefit	time	energy	pagefit	time	energy	pagefit
combine	169%	85%	8	169%	87%	3	100%	64%	8	100%	66%	3
fft	929%	1017%	276135	345%	372%	96320	103%	98%	276135	59%	54%	96320
epic	169%	85%	134	169%	87%	8	100%	64%	134	100%	66%	8
unepic	107%	71%	8	107%	73%	5	100%	67%	8	100%	69%	5
mp3-dec	396%	325%	7220	393%	323%	6590	93%	75%	7220	93%	76%	6590
mpeg4-dec	168%	130%	6088	113%	76%	12	93%	70%	6088	87%	64%	12
mpeg4-enc	186%	149%	6389	176%	140%	5272	89%	74%	6398	89%	74%	5272
Geom. mean	231%	165%		188%	132%		97%	72%		88%	67%	

Figure 7: Execution time, memory subsystem energy consumption, and the number of page faults for different on-chip memory subsystem configurations



(a) Energy consumption of the memory subsystem for different configurations



(b) Total number and distribution of instruction fetches for different configurations

Figure 8: Results

mp3-dec (Figure 8 (a)) due to the greater SPM access energy. Even though *mp3-dec* in the case of 1.5x produces 650, or 9% fewer page faults than the 1.0x case, the overall energy consumption increases by 2% due to the higher access cost (per-access energy) of a larger SPM.

If the SPM is not saturated, increasing the size will significantly improve both performance and energy consumption. While *fft* is thrashing with 4KB of SPM, increasing the size by two pages (2KB) reduces the number of page faults by 179,815, or 65%.

The cost of the SPMM. On average, one page fault costs 190 instructions, 270 loads from, and 29 stores to SDRAM. The SDRAM-to-SPM block copy routine alone is responsible for 124 instructions, 232 loads, and 8 stores. Figure 8 (b) shows that the cost of the SPMM is not noticeable for most cases with the minicache. For *fft*, the 4KB SPM is clearly too small and thrashing occurs, thereby resulting in a large number of SDRAM accesses or minicache accesses, depending on the configuration. The evidence is an extremely high number of page faults, and a significantly larger number of instruction fetches.

Overall, our instruction SPM management technique with a small minicache is very effective. It achieves a 12% improvement in performance and a 33% reduction in memory subsystem energy consumption in comparison to a conventional instruction cache with similar die area requirements.

8. CONCLUSION AND FUTURE WORK

This paper proposes a horizontally split memory architecture with a big SPM and a small minicache instead of an instruction cache.

To manage the SPM in a virtual memory environment, we introduce an SPM allocation scheme that loads frequently executed code sections into the SPM on demand. We exploit the hardware's page fault exception mechanism to automatically trigger the invocation of our runtime SPM manager. Since the minimal transfer unit is one memory page, the code placement is of great importance to the success of our technique. We have implemented code placement heuristics in our post-pass optimizer to generate SPM-optimized binaries from ordinary ARM ELF files. SPM-optimized binaries run without modification on different SPM sizes and even on cached-only cores. Also, SPM-unaware binaries run with satisfactory performance on our on-chip memory subsystem.

To support 1-cycle latency accesses to both the SPM and the cache, even when the MMU is active, we serialize the address translation and the SPM/cache access. We show that, with the current 0.13μ manufacturing technology, such a design can support clock frequencies of up to 500MHz.

Compared to a conventional fully-cached configuration, on average we achieve a 12% improvement in runtime performance and a 33% reduction in energy consumption by the memory system, with no increase in the required die area.

Our future work includes exploring better code placement strategies to more effectively utilize the available SPM space. We also intend to extend our work to multiple running tasks.

9. ACKNOWLEDGMENTS

We would like to thank Yoonsung Nam for his work on the simulator, and the anonymous reviewers for their helpful feedback.

10. REFERENCES

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 318–326, 2003.

[2] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, 2004.

[3] ARM Ltd,. http://www.arm.com.

[4] 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 *Proc. of the 10th International Symposium on Hardware/Software Codesign (CODES)*, May 2002.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4), 2005.

[7] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, 2006.

[8] J. Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Commun. ACM*, 4(10):435–436, 1961.

[9] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 238–243, 2004.

[10] Freescale Semiconductor. http://www.freescale.com.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, December 1998.

[12] Intel Corp. http://www.intel.com.

[13] A. Janapsatya, A. Ignjatovic;, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 612–617, 2006.

[14] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 628–633, 2002.

[15] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 690–695, 2001.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[17] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 329–338, 2005.

[18] Micron Technology, Inc. MT48H8M16LF Mobile SDRAM. 2003.

[19] Micron Technology, Inc. Mobile SDRAM Power Calc 10. 2004.

[20] J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, S. Przybylski, T. Riordan, and C. Rowen. A cmos risc processor with integrated system functions. In *COMPCON*, March 1986.

[21] ISO MP3 Reference Decoder. http://www.mp3-tech.org/programmer/sources/dist10.tgz.

[22] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 115–125, 2005.

[23] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design Automation and Test Conference*, March 1997.

[24] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT'04: The ACM Conference on Embedded Software*, September 2004.

[25] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, 1990.

[26] Philips LPC3180 microcontroller. http://www.standardics.philips.com/.

[27] A. Shrivastava, I. Issenin, and N. Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 90–96, 2005.

[28] Seoul National university Advanced Compiler tool Kit. http://aces.snu.ac.kr/snack.html.

[29] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, 2002.

[30] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, 2003.

[31] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marvedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *3rd Workshop on Embedded Systems for Real-Time Multimedia*, September 2005.

[32] M. Verma, L. Wehmeyer, and P. Marvedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of International Conference on Design, Automation and Test in Europe (DATE)*, February 2004.

[33] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid State Circuits*, 31(5):677–688, 1996.

[34] Xvid MPEG-4 Video Codec. http://www.xvid.org.