# Scratchpad Memory Management in a Multitasking Environment [*]

Bernhard Egger
Samsung Advanced Institute of Technology
Giheung-gu, Yongin-si
Gyeonggi-do 446-712, Korea
bernhard@aces.snu.ac.kr

Jaejin Lee
School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
jlee@cse.snu.ac.kr

Heonshik Shin
School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
shinhs@snu.ac.kr

## ABSTRACT

This paper presents a dynamic scratchpad memory (SPM) code allocation technique for embedded systems running an operating system with preemptive multitasking. Existing SPM allocation schemes do not support multiple tasks or only a fixed number of processes that are known at compile time. These schemes rely on algorithms that select code depending on the size of the SPM. In contemporary portable devices, however, processes are created and terminated on demand and the SPM is shared among them.

We introduce a dynamic scratchpad memory code allocation technique for code that supports dynamically created processes. At runtime, an SPM manager (SPMM) loads code pages of the running applications into the SPM on demand. It supports different sharing strategies that determine how the SPM is distributed among the running processes. We analyze several sharing strategies with regard to several preferable properties of multiprocess SPM allocation schemes.

We evaluate the proposed multiprocess SPM allocation techniques and compare them to a fully-cached reference system by running several multiprocess benchmarks. The benchmarks comprise of multiple embedded applications such as H.264, MP3, MPEG-4, and PGP. On average, we achieve a 47% improvement in throughput and a 32% reduction in energy consumption. A comparison with the unachievable lower bound shows that the best SPM sharing strategy exploits 87% of the runtime improvements and 89% of the energy savings possible.

**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, Multitasking, Virtual memory, Paging, Heterogeneous memory, Scratchpad, Portable systems

## 1. INTRODUCTION

Today's portable devices perform an ever increasing number of diverse functions that were traditionally handled by several devices specializing on one task. This ongoing convergence has several effects: first, portable devices contain more and more general purpose CPUs and move away from traditional embedded systems with specialized processors. Second, CPUs in portable devices continue to get more powerful to provide the computing power necessary to run several complex tasks in a multitasking environment.

Reducing the energy consumption for such devices is a major concern. Studies have shown that the memory system is responsible for a large portion of the total energy consumption [12, 20]. To increase performance and reduce the energy consumption, system designers make use of memory hierarchies to reduce off-chip memory accesses either by using hardware caches, scratchpad memories, or both.

Many studies have discussed code and/or data SPM allocation techniques [1, 4, 6, 11, 12, 19, 26, 27]. Significant energy savings and performance improvements can be achieved by placing frequently accessed code/data blocks in the SPM. However, most studies require the size of the SPM to be known at compile time. While a few consider multiple running tasks [1, 26], none of the proposed techniques can handle dynamically created tasks.

Contemporary portable devices often run an operating system with a scheduler, virtual memory, and even a file system. Processes are created and destroyed on the user's demand and at arbitrary times. Furthermore, the varying hardware configurations of the devices make it impractical for applications to be tailored to one specific SPM size.

In this paper we introduce a dynamic SPM code allocation technique for systems running an operating system with virtual memory and preemptive multitasking. The SPM is managed by an SPM manager (SPMM) that adapts as processes join and leave. It supports various SPM sharing strategies that define how the SPM is shared among the running processes. We propose three strategies and provide an analysis of their runtime overhead and performance.

[*] This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software] and by the Ministry of Education, Science and Technology under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

*EMSOFT'08,* October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-468-3/08/10 ...$5.00.

Our method is outlined as follows: Based on profiling information, a postpass optimizer sorts the code of an application based on the access frequency. Temporally local code is then clustered together into pages the size of an MMU page. Local data, i.e. constant pools are extracted from the code and placed in separate data pages. For each page, the postpass optimizer adds information to the SPM-optimized application binary about the page's access frequency and whether it belongs to a loop. Which code pages are loaded into the SPM is decided at runtime, i.e., the generated binaries are independent of the size of the available SPM.

At runtime, the SPM manager is notified by the operating system whenever a new process is created, destroyed, scheduled, or changes its ready-to-run status. It allocates the SPM to the running processes depending on the currently active sharing strategy and copies code pages into the SPM on-demand by intercepting MMU page fault exceptions.

*The contributions of this work are as follows:* (1) We introduce an SPM management technique for multitasking systems with dynamic process creation and destruction. The proposed technique is independent of both the number of processes and the concrete hardware configuration. The SPMM is built as a module to facilitate integration into existing operating systems. (2) We develop, implement, and analyze three different SPM sharing strategies for multitasking systems. In order to outperform a cached system in terms of both energy and performance, the overhead introduced by the SPM management must be kept to a bare minimum. The proposed sharing strategies therefore keep the computational complexity as low as possible. (3) Lacking a benchmark suite for multitasking systems, we propose representative multiprocess benchmarks for portable systems.

We evaluate the proposed SPM management technique on a cycle-accurate ARM9E-S simulator [14] with a combination of concurrently running applications. Compared to a fully-cached reference system, on average, we achieve a 47% improvement in throughput and a 32% reduction in energy consumption. Compared with the (in practice unachievable) lower bound, in which all executed code is located in SPM, the global SPM sharing strategy exploits 87% of the runtime improvements and 89% of the energy savings possible.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the runtime environment, the SPM manager, and various SPM sharing strategies. In Section 4, we describe the postpass optimizer framework. Section 5 explains simulation environment, the runtime environment, and the experimental setup. Section 6 presents the results. The paper ends with a conclusion and future work in Section 7.

## 2. RELATED WORK

Existing work on SPM allocation can be roughly divided into static and dynamic schemes. Static techniques initialize the SPM with the designated program parts at load time, and the contents of the SPM do not change at runtime. In dynamic SPM allocation techniques the contents of the SPM change while the program executes.

Static SPM allocation techniques are presented in [1, 3, 19, 27]. Except for Nguyen's work [19], all of these techniques require knowledge of the SPM size at compile time. Angiolini *et al.* [1] present an SPM allocation scheme that selects code blocks which promise the highest energy savings using an algorithm based on Dynamic Programming. Banakar *et al.* [3] solve the static assignment with a knapsack algorithm, both for code and data blocks. Verma *et al.* [27] select memory objects based on a cache conflict graph obtained through cache hit/miss statistics. The optimal set of memory objects is obtained by solving an Integer Linear Program (ILP) variant of the knapsack algorithm. In [19], Ngyuen *et al.* delay the decision which blocks should go to the SPM until the application is loaded, making their approach independent from the scratchpad memory size.

Dynamically allocated SPM algorithms are presented in [11, 12, 15, 24, 25]. Kandemir *et al.* [12] focus on data arrays accessed from well-structured loop kernels. Arrays are split into *tiles* that are transferred to the SPM independently. Also Li *et al.* [15] assign parts of data arrays to the SPM based on a graph coloring algorithm. Steinke *et al.* [24] present a technique that dynamically copies the optimal set of code blocks to the SPM based on an ILP. Udayakumaran *et al.* [25] construct a *data program relationship graph (DPRG)*, which then guides a greedy heuristical algorithm to determine the most promising candidates. Janapsatya *et al.* [11] introduce the so-called *concomitance* metric which measures the temporal locality of code block. At runtime, blocks with a strong correlation are copied together to the SPM. Dominguez *et al.* [5] propose an SPM allocation scheme for heap data. Promising candidates are assigned to a fixed-size *bin* that can hold up to $n$ elements of a dynamically allocated variable. At runtime, a heap manager allocates an object to the SPM only if there is free space in its predetermined bin.

The horizontal partitioning of memory hierarchies has been examined in [4, 7, 23]. Inspired by the memory architecture of the Intel XScale with a big main data cache and a 2KB minicache [10], Shrivastava *et al.* [23] show that by cleverly allocating the data objects to one of the caches a substantial amount of energy can be saved. Egger *et al.* [7] present a dynamic SPM allocation technique for a horizontally partitioned memory system consisting of an SPM and a small cache. At runtime, an SPM manager intercepts MMU page faults to load frequently executed code into the SPM on demand. Their approach is similar to the one taken in this paper, but only works for single tasks. Cho *et al.* [4] propose a similar technique for the data side.

To this day, the use of SPM in multitasking environments has not been widely studied. Verma *et al.* [26] present a static approach for a known set of processes. They propose three sharing strategies. The non-saving strategy divides the scratchpad evenly between the applications. In the saving approach, the whole scratchpad memory is given to the currently active task and saved and restored at each task switch. The hybrid approach is a mixture of the non-saving and the saving method: parts of the SPM are assigned exclusively to the processes and a common area is shared and needs to be saved/restored at each task switch. Since the SPM is statically allocated, their approach requires both the size of the SPM and the number of running processes to be known at compile time, which makes it unsuitable for a multitasking environment where processes are created and destroyed dynamically. In our work, the application binaries are not tailored to a specific SPM size and the runtime SPM manager redistributes the available SPM to the running processes on demand.

Machanick *et al.* [16] present an approach similar to this work by replacing the unified second-level cache by a software-managed SRAM. The main memory serves as a paging device for the SRAM. Pages are loaded into the SRAM whenever one of the first level caches misses. The main difference to our work is that we use the SPM in the first-level memory hierarchy as a software-managed cache, while [16] focuses on the second-level cache. It is possible to apply their concept to the first-level caches as well, however, since the latency of the L1 cache is much more critical to performance and energy consumption, unmodified binaries will not run at satisfactory performance. In our approach, temporally-local code is clustered together at compile time, and the decision whether a block is loaded into the SPM or mapped to the cache is made at runtime based on the number of running processes and the number of page faults (i.e., misses).

# 3. MULTIPROCESS SPM MANAGEMENT

## 3.1 The SPM Manager

The SPM manager (SPMM) is designed to run on a horizontally partitioned memory system with a cache and an SPM on the first level memory hierarchy as found in many contemporary embedded processors such as the ARM11 processor [2]. The SPMM manages the SPM as an operating-system controlled cache. It distributes the SPM to the running processes and decides which code pages should be loaded into the SPM and which are assigned to the hardware cache. It supports both SPM-optimized and SPM-unaware applications. Only SPM-optimized binaries will make use of the SPM, while SPM-unoptimized binaries are mapped to the hardware cache.

The SPMM needs to be notified for the following five events: (1) a new process is created, (2) a running process exits, (3) a process changes its ready-to-run status, (4) a process is scheduled, and (5) an MMU page fault occurs. Whenever an new process is created, the SPMM checks whether the application is SPM-optimized, i.e., has been processed by our postpass optimizer (Section 4). SPM-optimized binaries contain a map of all code blocks listing their access frequencies and loop affiliations. Every time the number of ready-to-run processes changes, the SPMM redistributes the SPM to all SPM-optimized processes that are ready to run. If necessary, it also modifies the processes' virtual memory mappings based on the active SPM sharing strategy (Section 3.2), the number of available SPM pages, and the information contained in each process' code block map. Code pages that are rarely accessed get mapped to the cache in order not to pollute the SPM. The memory mappings of pages that are to be loaded into the SPM before execution are marked invalid so that accesses trigger an MMU page fault. Whenever the control flow reaches unmapped code, the MMU generates a pagefault exception. The exception is handled by the RTE and forwarded to the SPMM which then loads the requested page into the SPM and enables its page table entry before the aborted instruction is restarted. If there is no free page in the SPM, the SPMM chooses victim and disables its memory mappings so that future accesses to the evicted page will again trigger a page fault.

Currently, the SPMM employs a round-robin replacement strategy because round robin does not depend on additional hardware support in the form of page reference or aging bits and its computational complexity is very low.

The decision which blocks are mapped to the cache and which are to be loaded into the SPM prior to execution is based on the number of available pages to the process. The information embedded in the code block map not only contains each block's access frequency, but also its loop affiliations and the loop hierarchy. If the size of the current working set exceeds the number of available pages, thrashing will occur. In that case, the SPMM maps the least frequently accessed pages of the working set to the cache.

**Integration into Existing Operating Systems.** In order to facilitate easy integration into existing operating systems, the SPMM is built as a module. It needs to be invoked by the RTE for the five events listed in the previous section. In our implementation, we define a global *SPMM* record that contains five function pointers corresponding to the five events:

```
SPMM = record
   on_create_process
   on_destroy_process
   on_change_status
   on_schedule_process
   on_page_fault
end
```

The necessary changes to the existing code of the RTE are thus reduced to a minimum. The integration of the SPMM into the scheduler, for example, is outlined by the **if** statement in the following pseudocode ($p_{current}$ denotes the currently running process):

```
procedure schedule(register_context *context)
var $p_{old}$: process
begin
  $p_{old} = p_{current}$
  $p_{old}$.context = context
  $p_{current}$ = select_next_process()
  context = $p_{current}$.context

  if (spmm.on_schedule_process != NULL) begin
    spmm.on_schedule_process($p_{old}$, $p_{current}$)
  end
end
```

This implementation also enables us to freely switch the SPM sharing strategy at runtime, even while processes are running. This is possible because processes are not aware of the SPM sharing strategy.

**Effect on Virtual Memory Systems** The SPM can be regarded as an additional layer in the memory hierarchy of a virtual memory system with paging and thus does not hinder paging of memory pages to an external storage medium. Minor modifications to the pagefault exception handler are necessary to redirect pagefault exceptions that are caused by pages to be run from the SPM to the SPMM.

## 3.2 SPM Sharing Strategies

Where a page is placed in the SPM and which page is evicted from it when a page fault exception occurs is determined by the SPM sharing strategy. We propose three different strategies for multiprocess SPM allocation: *shared*, *dedicated*, and *dedicated with pool*. In the *shared* SPM sharing strategy, the SPM is treated as a global resource and shared among all running processes. The *dedicated* SPM sharing strategy divides the SPM into $n$ disjoint regions and assigns one region to each SPM-optimized process. The *dedicated with pool* SPM sharing strategy is a combination of the *shared* and the *dedicated* strategies: part of the SPM is divided into $n$ disjoint regions while the remaining portion is shared among all processes.

We have designed all three strategies with the following goals in mind: easy adaptation, maximum preservation, computational complexity, small task-switching overhead, and fairness. Adaptation is important because processes are created and destroyed at arbitrary times. An SPM sharing strategy must be able to easily adapt at runtime to a varying number of SPM-optimized processes. Whenever a process joins or leaves, the SPM is redistributed among all active processes. Ideally, the newly computed allocation preserves as much of the old state as possible. For example, a recently allocated page should not be evicted from the SPM before all the older pages have been replaced. Similarly, if a process has to release a few of its pages because a new process joins, then these pages should be the ones that would be replaced next because these are the oldest pages. Another case arises when a process $p$ has to surrender pages to another process $q$. The pages moved from $p$ to $q$ may still contain code of $p$, even though they now belong to $q$. An SPM sharing strategy with good preservation will not evict that code from the SPM until process $q$ actually requires the pages.

Since a new SPM allocation needs to be computed at runtime whenever a process joins or leaves, it is important that the allocation algorithms do not require complex calculations. Similarly, the task-switching overhead introduced by the SPM sharing strategy should be kept as small as possible. Finally, an SPM sharing strategy should be fair, that is, a single process should not be able to cling to the whole or an over-proportional part of the SPM and thereby discriminating other processes.

### 3.2.1 The Shared Strategy

In the *shared* SPM sharing strategy, the SPM is shared among all processes (Figure 1 (a)). The SPMM maintains a single round-robin pointer, *next*, that points to the next block to be replaced. Whenever a pagefault occurs, the requested page is loaded into the designated SPM page, and the *next* pointer is moved to the following page in the list. If *next* does not point to a free page, that page is evicted from the SPM as soon as a new page needs to be loaded. On other words, the *shared* strategy manages the SPM as a fully-associative, software-managed cache with round-robin replacement.

The *shared* SPM sharing strategy satisfies all but one goal. It easily adapts to joining or leaving processes and preserves the existing SPM allocation. It does not require any computation when the number of running task changes, and there is no task switching overhead introduced by the *shared* strategy. It is, however, not particularly fair because a process with a big working set can request as many pages as needed, thereby evicting most or all pages of the other processes.

### 3.2.2 The Dedicated Strategy

For the *dedicated* SPM sharing strategy, the SPMM splits the SPM into $n$ disjoint regions where $n$ is the number of active SPM-optimized processes. These regions are *dedicated* to the process they are assigned to, i.e., other processes cannot load pages into another process' region. The size of each dedicated region is determined by the *SPM division policy.* We have implemented two different policies, the *maximum-workingset* (static), and the *on-demand* (adaptive) policy.

For each process $p$, the SPMM maintains a pointer $next_p$ that points to the next page to be replaced (Figure 1 (b)). Whenever a process requests a page, the SPMM loads the code into the designated page. If there is no free page left, the SPMM replaces the oldest page within $p$'s region in a round-robin fashion.

The *dedicated* SPM sharing strategy is fair because each process can only occupy up to $s(p)$ pages. Like the *shared* strategy, the *dedicated* strategy does not cause any task switching overhead for managing the SPM.

Easy adaptation and maximum preservation can be accomplished if we allow the disjoint regions to be discontiguous. To do so, blocks of the SPM are managed with a ring implemented with a doubly-linked list. The pseudo-code in Figure 2 illustrates the process of computing a new SPM allocation. Assume we have $n$ running processes, and $s_{cur}[i]$ stands for the number of SPM blocks currently allocated to process $i$. When a process joins or leaves, the SPMM first computes the new number of SPM blocks $s_{new}[i]$ allocated to each process. It then removes extra blocks from all processes $i$ where $s_{new}[i] < s_{cur}[i]$. By following the round-robin pointer $next[i]$, the SPMM moves the $s_{new}[i] - s_{cur}[i]$ oldest blocks to a temporary *pool* and leaves the more recently allocated blocks allocated to process $i$. Finally, the SPMM assigns the blocks from *pool* to all processes $i$ where $s_{cur}[i] < s_{new}[i]$. The newly assigned blocks are inserted into process $i$'s region at $next[i]$, i.e., on a page fault, process $i$ will first allocate the new blocks before evicting its own code blocks. Since $\sum_{i\ in\ P} s_{cur}[i] = \sum_{i\ in\ P} s_{new}[i]$, the number of reclaimed blocks is equal to the number of reassigned blocks, i.e., *pool* is empty at the end of *AllocateSPM*.

The complexity of the *dedicated* SPM sharing strategy depends on the sharing policy. *Static* policies, i.e., policies that do not change the SPM distribution unless the number of running processes changes, have a very little overhead. *Adaptive* policies monitor the behavior of the running processes and recompute a new SPM distribution when required and thus incur a higher overhead.

**The Maximum-Workingset Policy.** In this policy, the size of a process' private region is proportional to the maximum working set of that process (computed by the postpass optimizer at compile-time, see Section 4). The maximum-workingset policy is *static*, i.e., the distribution of the SPM does not change unless a new process joins or a running process exits.

**The On-Demand Policy.** During its execution, a process may go through various phases with different working sets. The on-demand policy distributes the SPM according to the current working sets of the running processes, $ws(p)$, by keeping track of the average number of pagefaults, $\phi PF(p)$, over the last $k$ epochs of process $p$. $\phi PF(p)$ depends on the relationship of $s(p)$, the number of pages allocated to $p$, and $ws(p)$: if $s(p) \geq ws(p)$, process $p$ will generate no or only compulsory misses, but for $s(p) < ws(p)$, (possibly many) capacity misses occur. The on-demand policy constantly measures the number of pagefaults and updates $\phi PF(p)$. Immediately before a process is scheduled, $\phi PF(p)$ is compared to $\phi PF_{last}(p)$, the average number of pagefaults used when the current SPM distribution was calculated. If $\phi PF(p)$ differs significantly from $\phi PF_{last}$, then the SPM distribution is re-computed. The on-demand policy is *adaptive* as it matches the SPM distribution to the current working set of the running processes.

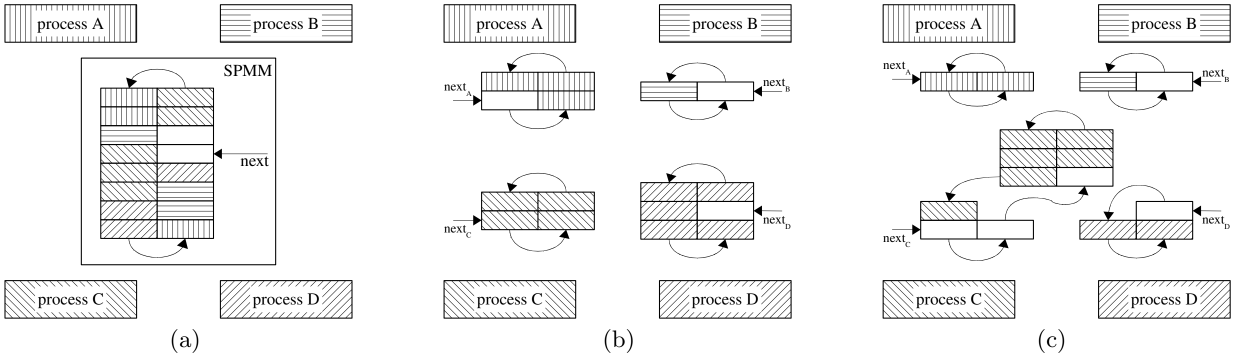


**Figure 1: SPM sharing strategies: (a) shared, (b) dedicated, and (c) dedicated with pool.**

### 3.2.3 The Dedicated with Pool Strategy

The *dedicated with pool* SPM sharing strategy is a mixture between the *shared* and the *dedicated* SPM sharing strategy. A part of the SPM, the *shared pool*, is shared between all processes. The remaining blocks are distributed according to the *dedicated* SPM sharing strategy (Figure 1 (c)). The shared pool is always allocated to the currently running process. The *dedicated with pool* strategy can be considered the common case because a shared pool size of zero blocks yields the identical SPM allocation as the *dedicated* strategy. Likewise, with a shared pool size equal to the number of SPM blocks, the allocation is the same as in the *shared* SPM sharing strategy.

The *dedicated with pool* strategy incurs a small overhead at each task switch when the blocks of the shared pool are moved from the old to the newly scheduled process (Figure 3). Assume that the size of the shared pool is two blocks, i.e., $s_{shared} = 2$, and that these blocks are allocated to the currently running process, $p$. For efficiency reasons, the *dedicated with pool* SPM sharing strategy maintains a per-process $pool\_end[p]$ pointer that is always exactly $s_{shared}$ blocks ahead of $next[p]$. Whenever the running process allocates a new block, both pointers, $next[p]$ and $pool\_end[p]$ are advanced by one block, i.e., at any time the shared pool consists of the $s_{shared}$ oldest blocks between $next[p]$ and $pool\_end[p]$ of the currently running process.

Moving the shared pool from $p$ to $q$ requires the following steps: first, the $s_{shared}$ oldest blocks of $p$ are removed. Since $pool\_end[p]$ is always $s_{shared}$ blocks ahead of $next[p]$, the SPMM links the block preceding $next[p]$ to $pool\_end[p]$ (Figure 3 upper row). The two blocks are then inserted into $q$'s list of SPM blocks as shown in the lower row of Figure 3.

In terms of adaptation, preservation, and computational complexity, the *dedicated with pool* SPM sharing strategy is equal to the *dedicated* strategy. It is also fair because a process can only allocate up to $s_{cur}[p] + s_{shared}$ blocks. At each task switch, the *dedicated with pool* strategy incurs a small overhead caused by moving the shared pool from the old to the newly scheduled process.

Table 1 summarizes the properties of the SPM sharing strategies discussed in this section. The task-switching overhead of the *dedicated with pool* strategy is caused by moving the shared pool. The on-demand policy introduces an additional overhead of keeping track of the number of pagefaults and, if necessary, recomputing a new SPM distribution.

**Table 1: Properties of the Proposed SPM Sharing Strategies. (MWS: Maximum-Workingset Policy, OD: On-Demand Policy, $n$: Number of SPM-Optimized Processes)**

| Property | SPM sharing strategy | | | | |
|---|---|---|---|---|---|
| | shared | dedicated | | dedicated with pool | |
| | | MWS | OD | MWS | OD |
| Easy adaptation | yes | yes | | yes | |
| Maximum Preservation | yes | yes | | yes | |
| Fairness | no | yes | | somewhat | |
| Computational complexity | $O(1)$ | $O(n)$ | | $O(n)$ | |
| Task-switching overhead | no | no | yes | yes | yes |

## 3.3 Predictability and Real-Time Tasks

To give an estimation of the worst-case execution time (WCET), we have to assume that a page fault occurs every time the control flow reaches code in the paged region. A guaranteed lower WCET is difficult to asses because the number of active processes changes dynamically at runtime (Section 5.3 quantifies the penalty incurred for a single page fault). This WCET estimation might not be good enough for real-time tasks with tight timing constraints. The SPMM therefore provides an API for real-time tasks which allows them to exclusively allocate $n$ SPM pages. Internally, the SPMM removes these pages from the list of globally available pages and recomputes a new SPM allocation. This approach works with all proposed SPM sharing strategies.

## 4. SPM-OPTIMIZED APPLICATIONS

In order to achieve the best performance while saving as much energy as possible, frequently executed code should be run from the SPM whereas rarely accessed instructions should be accessed via a cache or directly from the external memory. The postpass optimizer is based on the one presented in [7]. It disassembles ARM ELF binaries, analyzes the traces of training runs, and sorts the code of an application based on the access frequency. Temporally local code is clustered into pages the size of an MMU memory page. The postpass optimizer also constructs the dynamic control flow graph of the entire application and detects loops. Once all code has been placed in pages, the loop hierarchy is used to compute the maximal working set of an application.

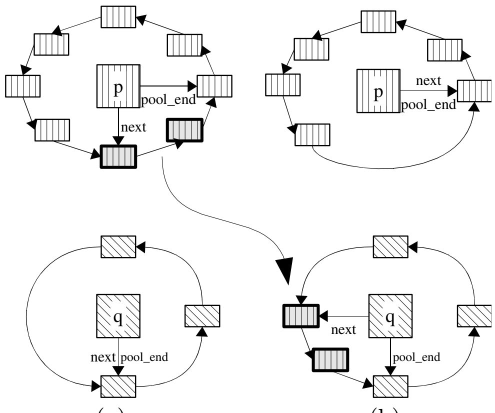
To improve code density within the pages, the postpass optimizer performs several optimizations. On the function

```
procedure AllocateSPM(Thread list $P$)
begin
  compute $s_{new}$ according to the active policy

  $pool := \phi$
  for $i$ in $P$ do begin
    if ($s_{new}[i] < s_{cur}[i]$) then begin
      move ($s_{cur}[i] - s_{new}[i]$) blocks from process $i$ to $pool$
    end
  end

  for $i$ in $P$ do begin
    if ($s_{new}[i] > s_{cur}[i]$) then begin
      move ($s_{new}[i] - s_{cur}[i]$) blocks from $pool$ to process $i$
    end
    $s_{cur}[i] := s_{new}[i]$
  end
end

policy MaxWorkingSet(Thread list $P$, distribution $s_{new}$)
begin
  for $i$ in $P$ do begin
    $s_{new}[i] := \frac{max\ working\ set[i]}{\sum_{j\ in\ P} max\ working\ set[j]} \cdot \#\ of\ SPM\ pages$
  end
end

policy OnDemand(Thread list $P$, distribution $s_{new}$)
begin
  for $i$ in $P$ do begin
    $s_{new}[i] := \frac{\phi PF[i]}{\sum_{j\ in\ P} \phi PF[j]} \cdot \#\ of\ SPM\ pages$
  end

  for $i$ in $P$ do $\phi PF_{last}[i] := \phi PF[i]$
end
```

**Figure 2: Computing a new SPM allocation for the *dedicated* SPM strategy depending on the policy. $s_{cur}[i]$ contains the number of blocks currently assigned to process $i$.**

level, infrequently executed basic blocks are separated from frequently executed ones and allocated to different pages. On the loop level, functions are sorted according to their access frequency and placed in descending order. The functions of the innermost loops are allocated to pages first, and code of the outer loop is only placed in those pages to reduce internal fragmentation. ARM code includes small data sections, so called *data pools*, that contain constants that are to big to be encoded as an immediate operand or global data addresses. The postpass optimizer extracts the constant pools from the code sections because if they are placed in the instruction SPM, data reads incur an additional two cycle penalty. Like code, the extracted constant pools are clustered into pages. To satisfy the constraints imposed by the limited range of immediate operands, those data pages have to be placed in the near vicinity of the code page(s) referencing them. The final layout of an SPM-optimized application thus does not contain separated text and data sections, but rather single pages containing code or data.

After the final code and data layout has been computed, the postpass optimizer inserts a code block map listing each block's access frequency and loop affiliation. Finally, the postpass optimizer generates a new ELF binary. It is noteworthy that the SPM-optimized binaries run unmodified on systems with no SPM.



(a) (b)

**Figure 3: Moving a shared pool of two blocks from $p$ to $q$. (a) before (b) after moving the shared pool.**

**Table 2: Access Latencies in CPU Cycles**

| Memory | Hit | Miss |
|---|---|---|
| Cache | 1 | 2 + writeback + line fetch |
| unified TLB | 1 | 3 + MMU page table walk |
| $\mu$TLB | 1 | 2 + unified TLB access |
| Memory | Read | Write |
| SPM | 1 | 1 |
| SDRAM | | |
| sequential | 24 | 24 |
| non-seq. | 27 | 27 |

# 5. EVALUATION ENVIRONMENT

## 5.1 Simulation Environment

The multiprocess benchmarks are executed on a cycle-accurate ARM architecture simulator [14]. The simulator models the ARM9E-S processor core and supports the ARMv5TE instruction set. It includes timing models for the pipelined ARM9E-S core, the MMU with the unified TLB, caches with $\mu$TLBs, scratchpad memory, the AMBA AHB bus, and external memory. For the simulations, the processor core clock is set to 200 MHz. The data cache and the instruction cache of the reference case are virtually-indexed, virtually-tagged, i.e., accesses to the caches do not require prior address translation by the MMU. This is in contrast to the horizontally partitioned memory architecture where both the SPM and the minicache are physically addressed. The page size of an MMU page is set to 256 bytes. The latencies of the caches, the SPM, the unified and the $\mu$TLBs, and the external memory (SDRAM) are shown in Table 2.

## 5.2 Memory System

The SPMM and the SPM sharing strategies are designed for high performance embedded processors with both a cache and an SPM with the same access latency, such as the ARM11 processors. The ARM11 chip, however, accesses both the cache and the SPM for every access, independent of the actual location of the requested instruction. To obtain optimal energy savings, we have therefore proposed a horizontally-partitioned memory system where the address translation is performed first and then only the memory containing the address is accessed [7]. We have shown that the increased latency introduced by the prior address translation limits the maximum clock frequency of the level-one memory system to about 1.5GHz for 0.13-$\mu$m manufacturing technology. This is, however, not a problem for contemporary embedded processors where clock frequencies typically still are below 1 GHz.

## 5.3 The Runtime Environment

We have implemented a small RTE consisting of a loader, a scheduler, and the SPMM. The scheduler is a preempting round-robin scheduler. All processes have equal priority. Processes in the scheduler queue are always ready-to-run because there are no data dependencies between processes and all I/O data is placed in external memory. The loader loads processes from a RAM file system and assigns stack and heap areas to newly created processes.

Whenever a process accesses an unmapped instruction, on average, the penalty incurred by loading the corresponding page into the SPM is 69 instructions, or 240 core clocks (1.2ms). The interrupt handler is responsible for 7 instructions, and the SPMM requires 48 instructions for managing the SPM, advancing the round robin pointers and disabling/enabling the memory mappings. Copying a page of 256 bytes requires 14 load/store multiple instructions.

## 5.4 Performance Metrics

We use the total energy consumed by the core and the memory subsystem as the energy metric and the total execution time as the performance metric. Additionally, we define the *throughput* of the RTE as the amount of work per time. The amount of work, i.e., running a benchmark from start to the end, is constant, thus

$$throughput = \frac{c}{T_{total}}$$

The simulator computes the total number of core clocks from the start till the end of a run. The end of a run is reached as soon as the last single process applications of a multiprocess benchmark ends. The execution time is computed by dividing the measured number of core clocks by the core clock frequency:

$$T_{total} = \frac{\#\, core\ clocks}{core\ frequency}$$

The energy consumption is calculated by summing up the the core energy, the on-chip memory system with both $\mu$TLBs (if present), the unified TLB, the instruction and the data cache, the SPM (if present), the off-chip bus, and the external memory (SDRAM).

For the horizontally partitioned memory system, the energies consumed by the TLBs, the caches and the SPM are computed by

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

where $e_{TLB}$, $e_{cache}$, and $e_{SPM}$ denote the access energy for the respective memory type. *Hit* and *miss* denote the number of hits and the number of misses for the corresponding memory structures, respectively. The $\mu$TLBs and the unified TLB are modeled as caches with a 4-byte linesize, hence writing a datum costs one word write. The cache energy is computed accordingly, only this time with the corresponding linesize. The SPM energy is simply the access energy multiplied by the sum of reads and writes.

**Table 3: Multiprocess Benchmarks**

| Benchmark | Applications | | Code | Benchmark | Applications | | Code |
|---|---|---|---|---|---|---|---|
| | | Start | Size | | | Start | Size |
| | | [ticks] | [KB] | | | [ticks] | [KB] |
| dsp | combine | 0 | 21 | multimedia 1 | epic | 0 | 69 |
| | fft | 0 | | | fft | ? | |
| internet 1 | pgpd | 0 | 108 | | pgpe | ? | |
| | pgpe | 0 | | multimedia 2 | mp3 | 0 | 166 |
| | unepic | 0 | | | mp4d | 103 | |
| high load | combine | 0 | 186 | | pgpd | 50 | |
| | fft | 0 | | | pgpe | 179 | |
| | mp3 | 0 | | | unepic | 5 | |
| | mp4d | 0 | | multimedia 3 | epic | 36 | 230 |
| | pgpd | 0 | | | h264 | 0 | |
| | pgpe | 0 | | | mp4e | 93 | |
| | unepic | 0 | | | pgpe | 147 | |
| internet 2 | combine | 63 | 133 | | unepic | 43 | |
| | epic | 0 | | multimedia 4 | fft | 0 | 141 |
| | fft | 112 | | | mp3 | 69 | |
| | mp4d | 7 | | | pgpd | 53 | |
| | pgpe | 139 | | | pgpe | 141 | |
| | unepic | 193 | | | unepic | 189 | |

The SDRAM energy is composed of static and dynamic energy [17]. We have modeled the low power 64-MB Samsung K4X51163PC SDRAM [22] with a memory bus frequency $f_{mem}$ = 66 MHz and a supply voltage $V_{dd}$ = 1.8 V. The static energy consumption includes the standby power and the power to periodically refresh the SDRAM cells and is computed by multiplying the execution time by the static power consumption. The dynamic energy comprises of the read random, write random, read burst, and write burst energy. It includes both the SDRAM dynamic energy and the memory bus energy.

## 5.5 Benchmarks

Lacking a suitable benchmark suite consisting of multiprocess applications for portable devices, we use fifteen embedded applications to construct representative multiprocess benchmarks. The applications include nine benchmarks from MiBench [8] and MediaBench [13], a H.264 video decoder [9], the official ISO MP3 decoder [18], MPEG-4 XviD encoding/decoding [28], and a public key encryption tool, Pretty Good Privacy (PGP) [21]. We chained the applications *quicksort, dijkstra, SHA, ADPCM-enc, ADPCM-dec,* and *bitcount* together into one application called *combine.* Each of the smaller applications is executed once in *combine* to represent an embedded program with multiple phases.

Each of the multi-process benchmarks comprises of several applications. The benchmarks *dsp*, *internet 1*, and *high load* represent increasing workloads, and all applications are started simultaneously. The benchmarks *internet 2* and *multimedia 1-4* represent dynamic workloads and were generated by randomly selecting both the applications contained within and their starting time.

Table 3 summarizes the characteristics of each multiprocess benchmark. The columns *Benchmark* show the eight multiprocess benchmarks. The columns *Applications* list the single process applications of each benchmark and their starting time. A starting time of zero implies that the application is started as soon as the scheduler starts running. A starting time of $t$ ticks denotes that the application is started after $t$ scheduler ticks.

Table 4: **Energy Consumption and Execution Time of the Different SPM Sharing Strategies in Comparison with a Fully-Cached System.** $E$ **stands for** *Energy*, **and** $T$ **denotes the** *Throughput*

| | | | SPM sharing strategy | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | ideal | | divided | | hybrid with on-demand policy | | | | | | global | |
| | | | | | 1/4 | | 2/4 | | 3/4 | | | |
| | $E$ | $T$ | $E$ | $T$ | $E$ | $T$ | $E$ | $T$ | $E$ | $T$ | $E$ | $T$ |
| dsp | 63% | 148% | 75% | 126% | 66% | 141% | 63% | 147% | 63% | 147% | **63%** | **147%** |
| internet 1 | 75% | 118% | 76% | 117% | 75% | 117% | 75% | 117% | 75% | 117% | **75%** | **117%** |
| high load | 65% | 154% | 113% | 96% | 88% | 119% | 79% | 130% | 73% | 140% | **66%** | **152%** |
| internet 2 | 69% | 135% | 102% | 98% | 81% | 119% | 76% | 126% | 72% | 131% | **70%** | **134%** |
| multimedia 1 | 73% | 123% | 87% | 107% | 78% | 116% | 75% | 120% | 74% | 122% | **74%** | **122%** |
| multimedia 2 | 51% | 227% | 91% | 140% | 80% | 157% | 70% | 176% | 65% | 188% | **57%** | **210%** |
| multimedia 3 | 71% | 130% | 110% | 96% | 102% | 102% | 98% | 105% | 96% | 108% | **95%** | **108%** |
| multimedia 4 | 48% | 237% | 59% | 202% | 55% | 214% | 53% | 221% | 53% | 223% | **53%** | **224%** |
| Geo.mean | 64% | 154% | 87% | 119% | 77% | 132% | 73% | 139% | 70% | 143% | **68%** | **147%** |

# 6. EXPERIMENTAL RESULTS

The reference case for our measurements is defined by running all benchmarks in the RTE on an ARM926EJ-S core with virtually-indexed, virtually-tagged caches. The instruction cache size of the reference case is set to the smallest cache that achieves a cache miss ratio of about 1% (4 KB for all benchmarks). The size of the data cache is fixed to 16 KB. For the reference case, the multiprocess benchmarks are composed of the original (SPM-unaware) single process applications.

A theoretical lower bound, the so-called *ideal* case, is obtained by running each benchmark with the original (SPM-unaware) applications on the horizontally partitioned memory architecture. We assume that all instruction fetches are covered by the SPM. The SPM is not large enough to hold all code at once, but we assume that it always contains the required instructions, i.e., no pagefaults occur and no code needs to be copied from the external memory into the SPM. We choose this setup because the access energy of an SPM that is large enough to hold all code is so high that the energy consumption is higher than that of the reference case. This means that, in practice, this lower bound is unachievable, but we will show that the proposed SPM sharing strategies approach the lower bound if the number of pagefaults is reasonably small.

The results for the proposed dynamic SPM allocation with the *shared*, *dedicated*, and *dedicated with pool* SPM sharing schemes are obtained by running the benchmarks comprising of the SPM-optimized images on an ARM926EJ-S core with a horizontally partitioned memory system on the instruction side. The original 4-KB, 4-way set-associative cache is replaced by an 8-KB SPM and a 256-byte direct-mapped cache. Thanks to the simpler design of the SPM, the horizontally partitioned memory system fits into the same die area as the original cache. The data side is left unmodified.

Figures 4 and 5 compare the throughput and the energy consumption of the reference images, denoted *ref*, to the ideal case (*ideal*), the three SPM sharing strategies, *dedicated*, *dedicated with pool* (with a pool size of 1/4, 2/4, and 3/4 of the available SPM size), and the *shared* SPM sharing strategy. Both the *dedicated* and the *dedicated with pool* sharing strategies use the *on-demand* policy which adapts the SPM distribution on-demand based on the number of pagefaults.

The energy consumption in Figure 5 is split up into CPU core, SDRAM (includes static and dynamic energy), TLB (includes the energy of the unified and the $\mu$TLBs), instruction cache, data cache, and SPM energy. The energy consumption of the CPU core is directly proportional to the execution time, i.e., the execution time is represented by the fraction of the CPU Core bar. Note that we assume virtually-indexed, virtually-tagged caches, thus, the reference case consumes no TLB energy except for a couple of unified TLB misses. The other cases, on the other hand, require address translation by the $\mu$TLBs prior to accessing the correct memory and consequently spend about 9% of the total energy in the TLBs.

Figure 6, finally, shows the number of pagefaults normalized to the *dedicated* case (no pagefaults occur in the reference or ideal case). The number above the *dedicated* bar represents the absolute number of pagefaults.

We have run all benchmarks at a scheduling frequency of 100 and 300 Hz, as well as two reference cache sizes (4 KB and 8 KB). The results of the different configurations are comparable, and we only show the results for the 4-KB reference case running at a scheduling frequency of 100 Hz.

Compared to the reference case, the *ideal* case consumes only about 64% of the energy and achieves a throughput of 154%. The SPM uses significantly less energy than the instruction cache, and since we assume that all instructions reside in SPM, the SDRAM energy is also reduced because of the absent cache misses (reduced dynamic energy) and the shorter runtime (reduced static energy). The *dedicated* case performs not very well. Figure 6 reveals that the completely separated, private SPMs assigned to each running application are too small and result in a huge number of pagefaults. The exception, *internet 1*, has only 771 pagefaults because the working sets of the three applications it is comprised of almost fit into the available SPM.

As we increase the size of the shared pool in the *dedicated with pool* cases, the number of pagefaults drops, and both the energy consumption and the throughput significantly outperform the reference case. The *shared* SPM sharing strategy, finally, achieves the best results overall, both in terms of throughput and energy consumption. Figure 6 shows that this is due to the overall smallest number of pagefaults. While the *shared* strategy is susceptible to applications with a big working set that could lead to the eviction of other processes' pages, this disadvantage is not as big as expected.

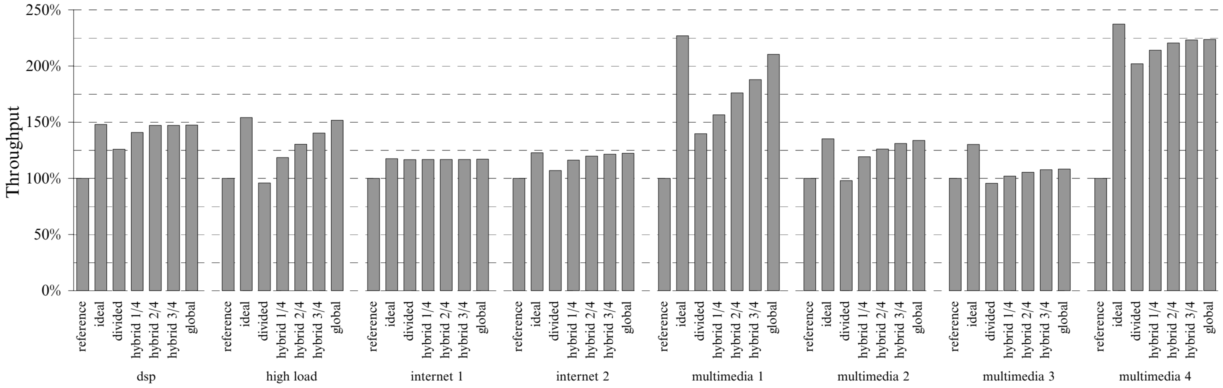


**Figure 4: Throughput.**

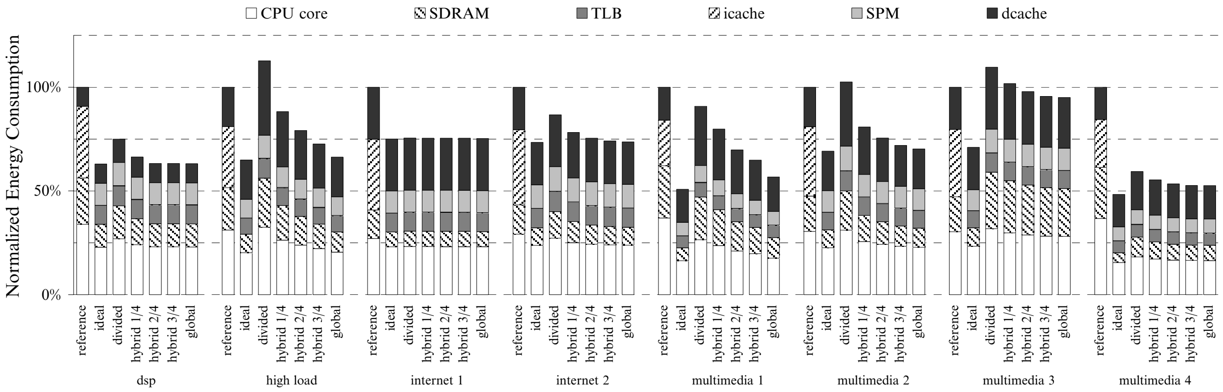


**Figure 5: Energy consumption of core and memory subsystem.**

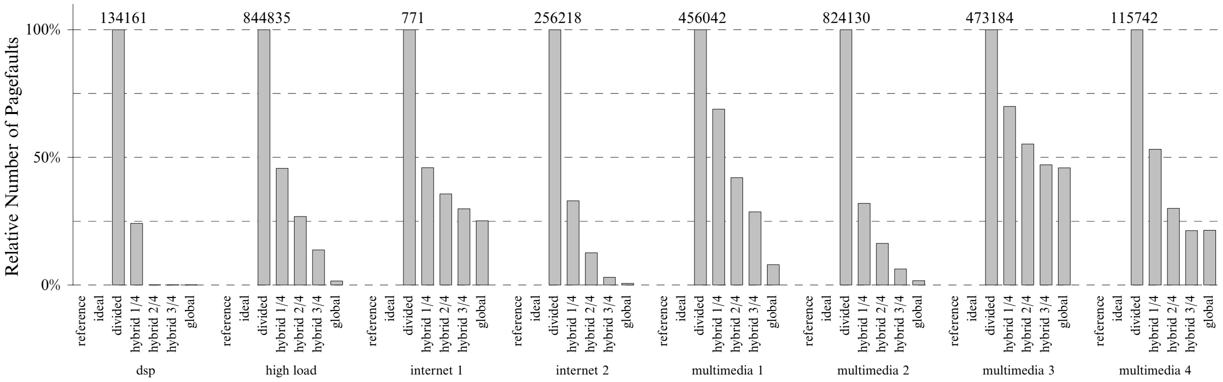


**Figure 6: Relative and absolute number of pagefaults.**

On average, the *dedicated* SPM sharing strategy achieves a 19% increase in throughput and a 13% reduction in energy consumption (Table 4). The *dedicated with pool* SPM sharing strategy achieves a 32%, 39%, and 43% improvement in throughput and a 23%, 27%, and 30% reduction in energy consumption for a shared pool size of 1/4, 2/4, and 3/4 of the SPM size, respectively. Finally, the *shared* SPM sharing strategy achieves a 47% improvement in throughput and a 32% reduction in energy consumption over a fully-cached ARM926EJ-S core.

A comparison of the *shared* SPM sharing strategy with the (in practice unachievable) *ideal* case shows that the *shared* SPM strategy achieves a very good performance and exploits, on average, 87% of the runtime improvements and 89% of the energy savings possible.

## 7. CONCLUSION AND FUTURE WORK

In this paper we have introduced an SPM management technique for systems with virtual memory and preemptive multitasking. We propose three SPM sharing strategies, the *shared*, the *dedicated*, and the *dedicated with pool* SPM sharing strategy, and analyze them in terms of implementation issues and with regard to preferable properties of multiprocess SPM allocation schemes.

The SPM-optimized applications are generated by a postpass optimizer that clusters temporally local code into pages. At runtime, the pages are loaded into the SPM on demand by an SPM manager (SPMM) by intercepting the MMU's page fault exception.

We have evaluated the three strategies on a horizontally partitioned memory system with a physically-addressed SPM and a small minicache. The multiprocess benchmarks developed for this work are run on a simple runtime environment (RTE) with a preempting round-robin scheduler, a loader, and the SPMM. We compare the proposed multiprocess SPM sharing strategies to a fully-cached reference system.

On average, we achieve a 47% improvement in throughput, and a 32% reduction in energy consumption with the *shared* strategy. Compared to the ideal case, the *shared* sharing strategy exploits 87% of the throughput improvements and 89% of the energy savings possible.

In our future work, we will employ more advanced SPM page replacement techniques in the SPMM, and extend our work to multiprocess systems without an MMU.

## 8. REFERENCES

[1] 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.

[2] ARM Architecture Version 6 (ARMv6). http://www.arm.com, 2002.

[3] 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.

[4] H. Cho, B. Egger, J. Lee, and H. Shin. Dynamic data scratchpad memory management for a memory subsystem with an MMU. In *Proc. of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers and tool support for embedded systems (LCTES)*, June 2007.

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

[6] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT'06: Proceedings of the sixth ACM & IEEE international conference on Embedded Software*, 2006.

[7] B. Egger, J. Lee, and H. Shin. Dynamic scratchpad memory management for code in portable systems with an MMU. *Trans. on Embedded Computing Sys.*, 7(2):1–38, 2008.

[8] 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.

[9] H.264 Video Codec. http://www.itu.int/rec/T-REC-H.264, 2003.

[10] Intel XScale Architecture. http://www.intel.com, 2002.

[11] 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.

[12] 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.

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

[14] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. Facsim: A fast and cycle-accurate architecture simulator for embedded systems. In *Proc. of the 2008 ACM SIGPLAN/SIGBED conference on Languages, compilers and tool support for embedded systems (LCTES)*, June 2008.

[15] 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.

[16] P. Machanick, P. Salverda, and L. Pompe. Hardware-software trade-offs in a direct rambus implementation of the rampage memory hierarchy. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 105–114, 1998.

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

[18] MP3 Reference Decoder. http://www.mp3-tech.org/programmer/sources/dist10.tgz, 1996.

[19] 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.

[20] 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.

[21] Pretty Good Privacy (PGPi). http://www.pgpi.org/, 2002.

[22] Samsung Semiconductor. K4X51163PC Mobile DDR SRAM. 2005.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] Xvid MPEG-4 Video Codec. http://www.xvid.org, 2005.