Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU [*]

Hyungmin Cho Bernhard Egger Jaejin Lee Heonshik Shin

School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
http://aces.snu.ac.kr
endroit@gmail.com, bernhard@aces.snu.ac.kr, jlee@cse.snu.ac.kr, shinhs@snu.ac.kr

Abstract

In this paper, we propose a dynamic scratchpad memory (SPM) management technique for a horizontally-partitioned memory subsystem with an MMU. The memory subsystem consists of a relatively cheap direct-mapped data cache and SPM. Our technique loads required global data and stack pages into the SPM on demand when a function is called. A scratchpad memory manager loads/unloads the data pages and maintains a page table for the MMU. Our approach is based on post-pass analysis and optimization techniques, and it handles the whole program including libraries. The data page mapping is determined by solving an integer linear programming (ILP) formulation that approximates our demand paging technique. The ILP model uses a dynamic call graph annotated with the number of memory accesses and/or cache misses obtained by profiling. We evaluate our technique on thirteen embedded applications. We compare the results to a reference system with a 4-way set associative data cache and the ideal case with the same 4-way cache and SPM, where all global and stack data is placed in the SPM. On average, our approach reduces the total system energy consumption by 8.1% with no performance degradation. This is equivalent to exploiting 60% of the room available in energy reduction between the reference case and the ideal case.

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

[*] This work was supported in part by the Ministry of Education under the Brain Korea 21 Project, by ETRI Embedded Software Research Division, by ETRI SoC Industry Promotion Center under the Human Resource Development Project for IT SoC Architect, and by MIC & IITA through the IT Leading R&D Support Project. ITC at Seoul National University provided research facilities for this study. Correspondence to: jlee@cse.snu.ac.kr

Keywords Compilers, Post-pass optimization, Demand paging, Scratchpad memory, Horizontally-partitioned Memory

1. Introduction

As the computation power of modern embedded systems continues to grow, reducing the energy consumption remains a major concern. Many studies have shown that the memory system is responsible for a large portion of the total energy consumption of embedded systems. Reducing the off-chip memory accesses, that is, accesses to the external DRAM or Flash memory plays a key role in reducing the energy consumed by the memory system. Not only do off-chip memory accesses constitute for a substantial fraction of the consumed memory system energy itself, but their access latency also increases the execution time and therefore the amount of CPU core and DRAM refresh energy.

Caches and scratchpad memories (SPM) are two memory structures that are very successful in reducing the external memory accesses and thereby also the energy consumed by them. In contrast to caches, the contents of the SPM have to be managed by software. Several studies have discussed various SPM allocation techniques [3, 6, 7, 8, 25, 26, 27]. The contents of the SPM are either static and do not change at run time, or dynamic, that is, the contents change during the course of the application to reflect the most frequently accessed data at any given time. It is not surprising that dynamic allocation techniques are superior over static ones [24]. Except for [8, 20], the size of the SPM must be known at compile-time to determine the optimal SPM allocation, however, this is not a big obstacle for embedded systems since software is often configured and installed on the device and rarely changed thereafter.

In this paper, we propose a dynamic SPM management technique focusing on frequently accessed data. We use profiling information of the original application to determine the program's data access patterns. By correlating the dynamic call graph (DCG) with the profiled memory accesses, we compute the optimal set of data blocks for each node in the DCG. Conceptually, SPM management code is inserted on the DCG's call edges. The SPM allocation depends on the call path, i.e., a data block b can be located in the SPM or in the external memory for the same function depending on the dynamic call path.

Predicting heap access patterns using profiling information is more difficult than predicting the access patterns of global and stack data because the access patterns of heap data are more dependent on the inputs. Therefore, we only target global and stack data in this paper.

We use a horizontally-partitioned memory subsystem [8] for the data side. The original set associative data cache is replaced by an SPM and a direct-mapped cache. The size of the SPM and the cache are chosen in such a way that the die area requirements of the SPM and the cache are less or equal to the size of the original cache. Due to the simpler design of the SPM, the total available data capacity exceeds that of the original cache. All data that is *not* currently residing in the SPM is accessed via the direct-mapped cache.

The storage location of data blocks is dependent on the dynamic call graph. We transparently map data blocks either to the SPM or the external memory using the memory management unit (MMU), i.e., from the application's point of view, data addresses remain constant independent of the current storage location. Because data is mapped using the MMU, the size of a data block corresponds to that of an MMU page. At run time, depending on the dynamic call graph upon entering a function, the SPM management code loads frequently accessed data pages into the SPM, and copies pages back to the external memory if required. By dividing data into pages, we can load only a portion of frequently accessed data into the SPM even if the total size of the data is bigger than the SPM's capacity.

We have implemented our technique in a post-pass optimizer. The profiling data is fed into an integer linear programming (ILP) model. The solution of the ILP determines the storage location of each data block for each node depending on the dynamic call graph. The post-pass optimizer inserts the necessary SPM management code to generate an SPM-optimized binary image.

We have evaluated the proposed data SPM allocation technique on a cycle-accurate ARM9E-S simulator using thirteen applications from the MediaBench [14] and MiBench [10] benchmark suites. We compare our approach with four different sizes of MMU pages to a system with 4-way set associative data cache and the ideal case with the same 4-way cache, where all global and stack data is placed in the SPM. On average, our approach reduces the total system energy consumption by 8.1% with no performance degradation. This is equivalent to exploiting, on average, 60% of the room available in energy reduction between the reference case and the ideal case.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 explains the horizontally-partitioned memory system. Section 4 presents the run time SPM management. Section 5 describes the post-pass optimizer and the ILP formulation. Section 6 explains the evaluation environment, and Section 7 presents our experimental results. Finally, Section 8 concludes the paper.

2. Related Work

Many studies have been done to exploit scratch pad memory in order to reduce energy consumption and/or improve execution time of applications. Some of them target program code placement into the SPM [3, 8], others focus on the data side [2, 4, 6, 13, 25, 26, 27].

SPM allocation schemes for data can be roughly divided into *static* and *dynamic* techniques. Static schemes, such as the ones presented in [3, 4, 13, 25, 27], allocate program data to the SPM when the executable is loaded into memory. The contents of the SPM do not change while the application is executed, i.e., the location of the data remains static.

Wehmeyer *et al.* [13] propose a general static allocation strategy for heterogeneous memory architectures. The location of each data block in the multi-level memory hierarchy is determined by solving an ILP model. Angiolini *et al.* [3] present a static SPM allocation technique that selects code blocks which promise the highest energy savings using an algorithm based on Dynamic Programming.

Dynamic allocation schemes proposed by [6, 7, 26, 28] move program code and/or data between the external DRAM and SPM during the program execution at run time. Egger *et al.* [7] solve an ILP that classifies code into three categories, SPM-resident, paged, and external. The blocks in SPM-resident and the external are placed at fixed locations in the SPM or the external memory, respectively. Paged code is copied to the SPM before execution. In [6, 28], liveness analysis is used to find the optimal program points where to move data blocks to and from the SPM at run time. Verma *et al.* [28] utilizes an ILP model in which the control flow graph and the liveness data are combined with profiling information.

Recently, Egger *et al.* [8] has proposed to use the Memory Management Unit to manage the SPM. They classify program code into paged and unpaged regions based on profiling information. While the program executes, an SPM manager intercepts the MMU prefetch aborts and loads the requested code page into the SPM.

Most data allocation strategies deal only with global data and stack. Heap data is more difficult to exploit because its access patterns are highly depend on the input data sets. Even with profiling data, it is difficult to predict the access patterns at compile time. Nevertheless, Dominguez *et al.* [6] introduce an allocation scheme for heap data to the SPM. They replace the heap management functions and split the SPM into bins of various sizes. Dynamically allocated memory blocks are then allocated to the SPM whenever there is a suitable free bin available.

Data allocation schemes that do not use an MMU have some limitations. If the size of the frequently accessed data, often an array, is larger than the SPM size, the data has to be allocated to the external memory because the addresses of the elements in an array must be contiguous. Verma *et al.* [27] overcomes this limitation by checking the offsets of every access to the data to determine its current location. However, these offset checks cause a rather big overhead.

In this paper, we introduce a dynamic SPM allocation technique for global and stack data. Similar to the approach in [8], we use the MMU's page fault abort exceptions to intercept accesses to data that needs to be copied into the SPM.

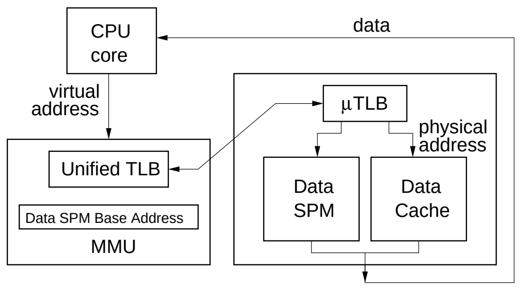


Figure 1. Horizontally Partitioned Memory Subsystem

3. The Memory Subsystem

Egger *et al.* [8] have proposed a horizontally-partitioned, on-chip memory subsystem for the instruction side of a Harvard architecture. The original instruction cache is replaced by an SPM and a minicache. Both the SPM and the minicache are physically addressed. The address translation is serialized with the SPM/minicache access. The resulting physical address is compared to the SPM base register to decide which of the memories, the SPM, the minicache, or the external memory should be accessed.

In this paper, we aim at a horizontally-partitioned memory architecture that is similar to the memory architecture proposed in [8]. We focus on the data side of the memory subsystem. The architecture of our memory subsystem is shown in Figure 1. Basically, it consists of a relatively cheap direct-mapped data cache, a data

8-entry μTLB		0.22ns	
16KB SPM	16KB direct-mapped cache	0.35ns	0.33ns
Total latency (μTLB $\rightarrow$ SPM or cache)		0.57ns	

Table 1. Cycle Access Time

SPM, a data μTLB, a unified TLB, the data SPM base register, and an MMU.

When accessing a datum, the MMU 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. The result of the translation is cached in the μTLB. When the PA is located in the SPM address range, only the SPM is accessed to retrieve the datum. If not, then the datum must be fetched from the data cache, in which case the SPM is not accessed.

Compared to a traditional cache, serializing the address translation with the SPM or cache access increases the latency of data access by approximately the access time of the μTLB. With the current 0.13μm manufacturing process, however, core clocks of up to 1GHz can easily be supported with a 1-cycle latency [8]. Table 1 shows cycle access times (i.e., the minimal time between two subsequent requests) for μTLB, cache, and scratchpad memory. The numbers were obtained with CACTI [29].

Due to the missing cache control logic and the simpler design, scratchpad memories are more efficient than caches in terms of both energy and die area. We can replace the cache with scratchpad memory and a smaller direct-mapped cache whose total capacity is bigger than the original cache, yet still achieve a reduction in the required die area.

Note that only the data cache is replaced by an SPM and a smaller direct-mapped cache. We do not modify the instruction side of the L1 memory hierarchy.

4. Runtime SPM Management

On cached cores, binaries run without further software support. On cores equipped with scratchpad memory, however, the SPM has to be explicitly managed either by the running application or by a dedicated SPM manager that is part of the run-time environment.

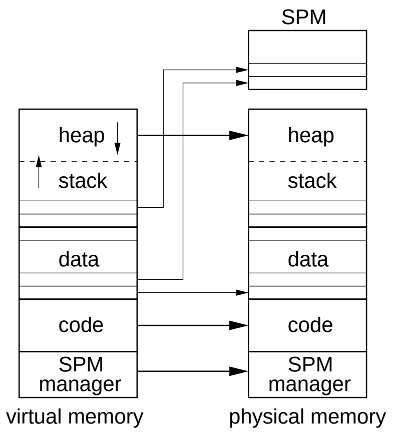


Figure 2. Virtual to physical page mapping on global data and stack.

Our scratchpad memory manager (SPMM) manages the SPM at run time. The management information is inserted into the application binaries, and it is given to the SPMM at run time. The SPMM uses demand-paging techniques similar to those in virtual memory systems [9] to track the course of the application and load pages into the SPM on demand.

Loading global data pages, which are required to execute a function, to the SPM is done by the SPMM just before the function is called. The data pages that are loaded by the function are stored to the external memory just after the function returns. To do so, SPM manager calls are inserted before and after the function call site. Whenever the SPMM is called, it loads or stores appropriate pages directed by the argument of the call, and modifies the virtual to physical address mapping in its page table. The MMU uses this page table and performs virtual to physical address translation. Figure 2 shows the virtual to physical address mapping in our approach. The mapping of code and heap areas is fixed.

If the pages are loaded and stored every time a function is called, the page movement overhead will be huge. Consequently, only frequently accessed data pages should be placed in the SPM.

A simple and intuitive solution to the problem of global data placement in SPM without an MMU (i.e., without demand paging) is using the solution of 0-1 Knapsack problem [5]. This can be easily done with the number of global data accesses in each page. The objective is to maximize performance and/or minimize the energy consumed by memory accesses. Placing pages for global data in data SPM according to the solution of the 0-1 Knapsack formulation will give us a statically optimal mapping for global data into SPM. However, the static allocation is only good for embedded systems without data caches. It cannot compete with cached systems because the data SPM size is much smaller than the global data in an application in general.

To effectively utilize the SPM with a limited size, we can exploit profiling information that includes function call sequences and the global data pages required to execute the functions called in the sequence. Consider the function call trace $f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_n$ that is obtained by profiling an application. Each caller and callee pair $f_i \rightarrow f_j$ in the trace is annotated with the global data pages accessed in function f_j and the number of accesses in the pages. Whenever a function f_j is called in the trace, we need to load the data pages required to run the function into the SPM just before the function is executed in order to save energy. Of course we need to consider the cost of copying pages from the external memory to the SPM. However, some required pages may not be loaded into the SPM when a function is executed because the data SPM size may be smaller than the global data size accessed in the function. Therefore, whenever a function is called, it is important to load the optimal set of pages for the function call into the SPM. We can try all possible combinations of the page assignment to the SPM for each caller and callee pair in the trace to find an assignment that consumes the minimum energy. With the optimal page assignment information for each function call, the SPMM can load the optimal set of pages into the SPM at run time.

For this mechanism to work, the SPMM needs to keep track of how many times each function is called and which pages need to be loaded into the SPM for each call of the function. The amount of information recorded is almost the same as the amount of trace information we obtained before, and this introduces a huge management overhead. Consequently, this approach is impractical. Instead, we try to approximate the trace information with a dynamic call graph. We treat each path from the root (the node without any predecessor in the graph) to a node without any successors in the dynamic call graph as a trace mentioned above. This can be modeled with an integer linear programming (ILP) problem and our ILP formulation will be explained in detail in Section 5.2 and Section 5.3.

5. The Post-Pass Optimizer

In this section, we describe our compiler post-pass approach and on-demand data mapping techniques in detail.

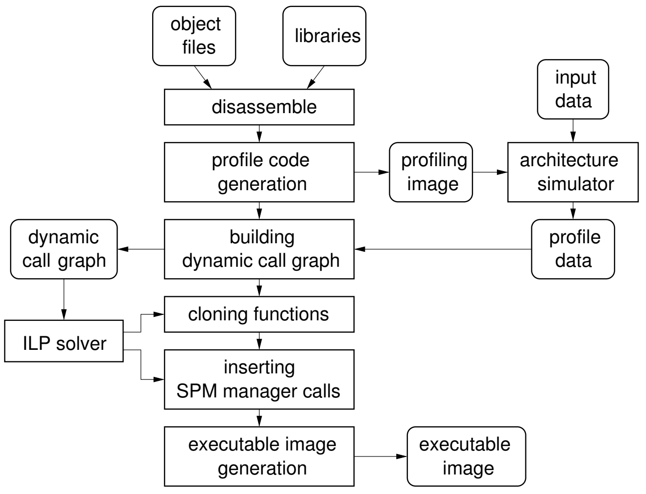


Figure 3. The post-pass optimizer.

5.1 Overview

Figure 3 shows the skeleton of our post-pass optimizer, SNACK-pop, which is a part of the **S**eoul **N**ational University **A**dvanced **C**ompiler tool **K**it[1]. Application and library object (ELF binaries) files are fed into the post-pass optimizer and disassembled. With the disassembled image, a profiling image is generated. The profiling image runs on an architecture simulator. The simulator generates profile data for the application, which contains the number of accesses in each global data and stack page, the pages accessed by each function, caller and callee relationship, and the number of calls to each function. The profile data is then fed into the dynamic call graph building phase. The generated dynamic call graph annotated with profiling information is used by the integer linear programming (ILP) solver. The solution of the ILP solver is used to insert SPM manager calls for each function. Finally, the optimizer generates an executable image that can direct the SPMM for data SPM management.

In the following sections, we elaborate on important stages of our post-pass optimization.

5.2 Profiling

The first step of generating an executable image is profiling the original application.

We run the original application on our architecture simulator and obtain a dynamic call graph. We want to allocate data pages into the SPM along each path from the root node to a leaf node (i.e., a node with no successors) in the graph. To consider each path in the dynamic call graph, we perform node splitting and transform the graph into a graph in which no node has multiple predecessors except for leaf nodes.

For example, consider a dynamic call graph in Figure 4(a). First, this graph is converted into an acyclic graph by merging all the nodes in a cycle into a single representative node. The cycles in the graph are generated by recursive function calls. In Figure 4(b), f_1 and f_2 are merged into a single representative node. Then, we find a node n with multiple predecessors in the acyclic graph (say k predecessors, p_1, p_2, ..., and p_n, where $k > 1$). The node n is replaced by k copies of itself, n_1, n_2, ..., and n_k. A copy n_i

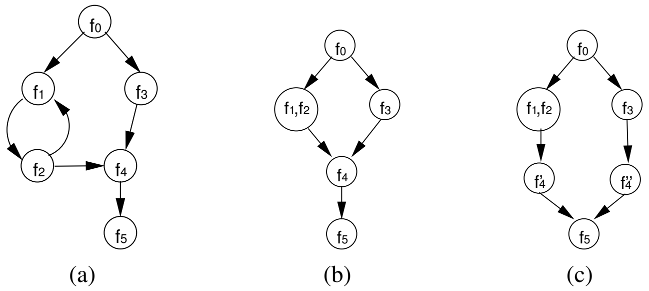


Figure 4. Converting the dynamic call graph into a call graph in which no node has multiple predecessors except for leaf nodes. (a) The original dynamic call graph. (b) Merging nodes in a cycle. (c) Splitting nodes.

corresponds to a predecessor p_i. Each edge $p_i \rightarrow n$ becomes $p_i \rightarrow n_i$. Similarly, an outgoing edge $n \rightarrow s$ to n's successor s is replaced by k edges of $n_i \rightarrow s$. If n is a leaf node, this step will not be applied. The final dynamic call tree is shown in Figure 4(c). In fact, this process is identical to assigning a different node to the same function for each different caller of the function.

Each edge $e = f \rightarrow g$ in the converted dynamic call graph is annotated with the following information for all pages p:

- $R(p, e)$: the sum of all global data reads in page p caused by function g for all calls from function f to function g.
- $W(p, e)$: the sum of all global data writes in page p caused by function g for all calls from function f to function g.
- $Calls(e)$: the number of calls from function f to function g

Without loss of generality, we assume the size of each global data access is a word (4 bytes).

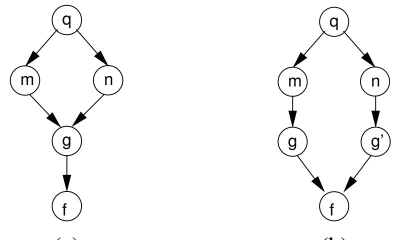


Figure 5. (a) Dynamic call graph (b) The dynamic call graph after node splitting.

The data pages for a function that must be copied in to the data SPM or copied out from the SPM to minimize energy consumption are dependent upon the call paths to which the function belongs. We say that those data pages required to execute a function are copied in to the data SPM or copied out from the SPM at the call edge to the function in the dynamic call graph. Note that each node (i.e., function) in the call graph has only one predecessor except for leaf nodes.

Our dynamic call graph distinguishes multiple call paths that reach a function. Consider the call graph in Figure 5(a). The data pages to be loaded into the SPM for the function f at the edge $g \rightarrow f$ will dependent upon the real call path taken at run time, either $q \rightarrow m \rightarrow g \rightarrow f$ or $q \rightarrow n \rightarrow g \rightarrow f$. In the dynamic call graph obtained after node splitting in Figure 5(b), these two paths are explicitly distinguished.

5.3 ILP Formulation

We approximate our problem with integer linear programming. Our goal is to find an assignment of pages to each edge in each call path from the root to a leaf node that consumes the minimum energy. If a page is assigned to an edge, the page is copied in to the data SPM or copied out from the SPM when the corresponding call occurs. We assume that the page loaded into the SPM due to a function call is copied out to the external memory when the function returns.

In our ILP formulation, the following symbols are used:

N_p	The number of pages in the global data area
N_e	The number of edges in the dynamic call graph
S_{spm}	The SPM size in the number of pages
E_{spm_R}	The energy consumed to read a word from the SPM
E_{ext_R}	The energy consumed to read a word from the external memory
E_{spm_W}	The energy consumed to write a word to the SPM
E_{ext_W}	The energy consumed to write a word to the external memory
E_{page_load}	The energy consumed to read a page from the external memory and write it to SPM
E_{page_store}	The energy consumed to read a page from SPM and write it to the external memory

The binary integer variables $LOAD(p,e)$, $STAY(p,e)$, and $STORE(p,e)$ represent the actions taken for a page p in the global data area. If none of the three variables is 1 for page p at edge e, then page p is found in the external memory at edge e.

$$LOAD(p,e) = \begin{cases} 1 & \text{if page } p \text{ is not in the SPM and is loaded into SPM at edge } e \\ 0 & \text{otherwise} \end{cases}$$

$$STAY(p,e) = \begin{cases} 1 & \text{if page } p \text{ has already been present in the SPM at edge } e \\ 0 & \text{otherwise} \end{cases}$$

$$STORE(p,e) = \begin{cases} 1 & \text{if page } p \text{ resides in the SPM and is copied out to the external memory at edge } e \\ 0 & \text{otherwise} \end{cases}$$

During execution along a call path from the root, if a page p is loaded to the SPM in an edge $j = m \rightarrow n$, on the path (i.e., $LOAD(p,j)$ = 1), it remains in the SPM until it is copied out to the external memory at an edge $k = v \rightarrow w$, (i.e., $STORE(p,k)$ = 1) that appears after the edge j on the path. On the subpath from n to v, the binary variable $STAY(p,e)$ must be 1 for all edges e on the subpath. We assume that when the function that is the target of the edge k where $LOAD(p,k) = 1$ returns, the page p is always copied out from the SPM. Thus, either only one of the binary variables can be 1 or all of them are zero at an edge. This is represented by the following constraint for all pages p and edges e:

$$LOAD(p,e) + STAY(p,e) + STORE(p,e) \leq 1 \tag{1}$$

Note that this also implies for all pages p and edges e,

$$\begin{aligned} LOAD(p,e) + STAY(p,e) &\leq 1 & (2)\\ LOAD(p,e) + STORE(p,e) &\leq 1 & (3)\\ STORE(p,e) + STAY(p,e) &\leq 1 & (4) \end{aligned}$$

When a page p is loaded into the SPM at an edge e, p must remain in the SPM or copied out of the SPM at the edge next to e in the call path. Similarly, when a page p has already been present in the SPM at an edge e, p must remain in the SPM or copied out of the SPM at the edge next to e in the call path. Thus, for all pages p and edges e, the following constraint holds for all outgoing edges k from e's target node:

$$LOAD(p,e) + STAY(p,e) \leq STAY(p,k) + STORE(p,k) \tag{5}$$

This constraint also implies that a page p cannot be loaded again into the SPM at the edge next to e in the call path.

For all pages p and edges e, the following constraint holds for all outgoing edges k from e's target node:

$$STAY(p,k) \leq LOAD(p,e) + STAY(p,e) \tag{6}$$

This means that if a page p resides in the SPM at an edge k, p has stayed or been loaded in the SPM at the edge just before k in the call path.

The last constraint describes that the size of all the pages in the SPM at each edge must be smaller or equal to the size of the SPM. For all edges e, the following size constraint holds:

$$\sum_{i}^{N_p} LOAD(i,e) + STAY(i,e) \leq S_{spm} \tag{7}$$

The objective function to be maximized is the total energy reduction:

$$\sum_{i=1}^{N_p}\sum_{j=1}^{N_e} \begin{array}{l} (ExtR_{ij} + ExtW_{ij}) \\ -(SPMR_{ij} + SPMW_{ij}) \\ -CopyIn_{ij} - CopyOut_{ij} \end{array} \tag{8}$$

$$\begin{aligned} ExtR_{ij} &= (LOAD(i,j) + STAY(i,j)) \cdot R(i,j) \cdot E_{ext\_R} \\ ExtW_{ij} &= (LOAD(i,j) + STAY(i,j)) \cdot W(i,j) \cdot E_{ext\_W} \\ SPMR_{ij} &= (LOAD(i,j) + STAY(i,j)) \cdot R(i,j) \cdot E_{spm\_R} \\ SPMW_{ij} &= (LOAD(i,j) + STAY(i,j)) \cdot W(i,j) \cdot E_{spm\_W} \\ CopyIn_{ij} &= Calls(j) \cdot LOAD(i,j) \cdot E_{page\_load} \\ CopyOut_{ij} &= Calls(j) \cdot STORE(i,j) \cdot E_{page\_store} \end{aligned}$$

The objective function includes the energy overhead for page movements between SPM and the external memory. The formulated problem is solved with an integer linear programming solver. The values of the binary integer variables denoting actions for data page placement (i.e., $LOAD(p,e)$, $STAY(p,e)$, and $STORE(p,e)$) are then fed into the post-pass optimizer.

5.4 Cloning Functions

Consider the call graph in Figure 5(b). There are two call paths from q to f: $P = q \rightarrow m \rightarrow g \rightarrow f$ and $Q = q \rightarrow n \rightarrow g' \rightarrow f$. Suppose that our ILP solution recommends that page i is loaded at $g \rightarrow f$ and page $j \neq i$ is loaded at $g' \rightarrow f$. If the function g in the original call graph in Figure 5(a) is not cloned, then we cannot tell the SPMM which page should be loaded when the function f is called in g. This is because determining the pages to be loaded depends on the call paths taken at run time. Therefore, in addition to cloning nodes in the dynamic call graph, we need to clone functions that have multiple callers. This may exponentially increases the code size.

However, we do not need to clone all the functions with multiple callers. Even though the corresponding node to the function

g has two split nodes g' and g'' in our dynamic call graph, if $LOAD(p, e') = LOAD(p, e'')$ and $STORE(p, e') = STORE(p, e'')$ for all pages p (e' and e'' are the outgoing edges of g' and g'' respectively), the function g does not need to be cloned. Note that leaf nodes are not split in our call graph even though they have multiple predecessors.

5.5 Inserting SPM Manager Calls

Since our scheme selects pages to be loaded to the SPM at each function call edge in the call graph, SPMM calls to load/store the pages are inserted at the call site of the callee in the caller.

```
FUNC  foo                          FUNC  foo
                                         ...
      ...                                B L1 // branch
                                   R1:   (return address)
      // branch and link                 ...
      BL bar                             // call SPMM
      (return address)             L1:   BL spmm_enter_foo_bar
                                         BL bar
      ...                                // call SPMM
                                         BL spmm_leave_foo_bar
                                         B R1
            (a)                                (b)
```

Figure 6. Inserting SPMM calls. (a) The original code. (b) After inserting SPMM calls.

The SPMM calls are inserted only at the call sites that corresponds to the call edge e where $LOAD(p, e) = 1$ or $STORE(p, e) = 1$ for some page p. Each call site to the SPMM has its own ID and the SPMM identifies which call site (i.e., call edge) calls the SPMM with the ID. The SPMM contains page load/store information for each call edge.

Figure 6 shows how to insert the SPMM calls when the call edge from function `foo` to function `bar` has page movements (i.e., page load/stores). If a data page is loaded to the SPM just before a function is called, that page is stored to the external memory when the function returns. Thus we need to insert SPM manager calls just before (`spmm_enter`) and after (`spmm_leave`) the function call.

For functions called via function pointers, we insert pointer comparison code to check whether the callee matches the edge $caller \rightarrow callee$ in the call graph (Figure 7.

After the SPMM copies in/out data pages to/from the SPM, it updates the page table with the new locations of the data pages moved. The TLB entries for the pages moved are invalidated because the TLB may contain stale entries for those pages. The SPMM writes pages to the external memory before loading pages in order to make room for the pages to be loaded.

We do not need a page replacement policy for the SPM because the maximum number of pages that reside in the SPM at every call edge is smaller than the size of SPM in the ILP model.

5.6 An Extension to Stack Pages

Unlike global data, the position of the stack frame for a function in memory varies depending on the call path to the function. However, the position is fixed in memory for each node in our dynamic call graph because our call graph distinguishes different call paths to a function. Consequently, we can treat the stack frame pages just like we treat global data pages.

All we have to do is to keep track of the number of accesses in each stack frame page at profiling stage. Then, we solve the ILP problem without distinguishing stack frame pages and global data pages.

```
FUNC  foo                          FUNC  foo
                                         ...
                                         B L1
                                   R1:   (return address)
      ...                                ...
                                         // load bar's address
                                   L1:   LDR lr, L3
                                         // compare the pointer
      MOV lr, pc                         // and bar's address
      B rx                               CMP rx, lr
      (return address)                   // if not the same,
                                         // call with the pointer
                                         BNE L2
                                         // otherwise, call SPMM
                                         BL spmm_enter_foo_bar
      ...                                BL bar
                                         // call SPMM
                                         BL spmm_leave_foo_bar
                                         B R1
                                         // the original call
                                   L2:   LDR lr, R1
                                         B rx
                                   L3:   DCD address_of_bar
            (a)                                (b)
```

Figure 7. Inserting SPMM calls for functions called with pointers. (a) The original code. Register `rx` contains the pointers to the callee. (b) After inserting SPMM calls.

5.7 Cache Misses as a Profiling Metric

Instead of using the number of access in each page as a metric for determining whether the page is loaded to the SPM or not, we can use the number of cache misses in each page to make the decision. Profiling information for the number of cache misses in each page will be obtained in our profiling stage, just like we did for the number of accesses.

In our ILP formulation, the objective function is changed to:

$$\sum_{i=1}^{N_p}\sum_{j=1}^{N_e} \begin{array}{l} (CacheRH_{ij} + CacheRM_{ij}) \\ +(CacheWH_{ij} + CacheWM_{ij}) \\ -(SPMR_{ij} + SPMW_{ij}) \\ -CopyIn_{ij} - CopyOut_{ij} \end{array} \tag{9}$$

$$\begin{array}{rcl} CacheRH_{ij} & = & RH_{ij} \cdot E_{cache\_read} \\ CacheRM_{ij} & = & RM_{ij} \cdot cache\_line\_size \\ & & \cdot(E_{ext\_read} + E_{cache\_write}) \\ CacheWH_{ij} & = & WH_{ij} \cdot E_{cache\_write} \\ CacheWM_{ij} & = & WM_{ij} \cdot E_{ext\_write} \\ RH_{ij} & = & \text{\# of cache read hits for page } i \text{ at edge } j \\ RM_{ij} & = & \text{\# of cache read misses for page } i \text{ at edge } j \\ WH_{ij} & = & \text{\# of cache write hits for page } i \text{ at edge } j \\ WM_{ij} & = & \text{\# of cache write misses for page } i \text{ at edge } j \end{array}$$

6. Evaluation Environment

We have evaluated the effectiveness of our dynamic data SPM management techniques for the horizontally partitioned memory system with a cycle accurate simulator, SNACK-armsim[1].

6.1 The Architecture Simulator

SNACK-armsim models the ARM9E-S 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 μTLBs, scratchpad memory, the AMBA AHB bus, and external memory. The simulator's on-chip memory system accurately models the horizontally partitioned memory system as presented in section 3.

Memory	Hit	Miss
Cache	1	2 + writeback + line fetch
unified TLB	1	3 + MMU page table walk
μTLB	1	2 + unified TLB access

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

Table 2. Access latencies in CPU cycles

On ARM architectures, a 1MB area of memory mapped with tiny (1024 byte) pages consists of 1024 entries. Each entry is 4 bytes wide, thus the size of the (second level) page table is 4KB. We use four different page sizes in our evaluation: 128B, 256B, 512B, and 1KB. For example, a page size of 256B requires a 16KB second level page table. The MMU in the simulator is modified to accommodate different page sizes.

The processor core clock in the simulator is set to 200MHz. The latencies of the cache, the SPM, the unified and the μTLBs, and the external memory (SDRAM) are shown in Table 2. The page table walk consists of one or two non-sequential memory accesses.

6.2 Performance Metrics

The execution time is computed by dividing the measured number of core clocks by the core clock frequency

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

The consumed energy is computed by summing up the the core energy, the on-chip memory system with both μTLBs, the unified TLB, the instruction and the data cache, the SPM, the off-chip bus, and the external memory (SDRAM)

$$\begin{aligned} E_{total} &= E_{core} + E_{unified\ TLB} \\ &\quad + E_{i-\mu TLB} + E_{d-\mu TLB} \\ &\quad + E_{icache} + E_{dcache} + E_{SPM} \\ &\quad + E_{ext\_static} + E_{ext\_dynamic} \end{aligned}$$

The core energy is computed by

$$E_{core} = T_{total} \cdot P_{core} \cdot f_{core}$$

where f_{core} is the core frequency in MHz and p_{core} the *power per MHz* parameter from Table 3 (a). The energy consumed by the TLBs, the caches and the SPM is 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} are taken from Table 3 (a) and (b). *Hit* and *miss* denote the number of hits and the number of misses for the corresponding memory structures, respectively. The μTLBs 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.

Direct-Mapped Cache			SPM	
size [KB]	linesize [words]	energy [nJ]	size [KB]	energy [nJ]
0.5	4	0.196	1	0.128
1	8	0.197	2	0.134
2	8	0.203	3	0.139
4	8	0.215	4	0.145
8	8	0.237	6	0.160
16	8	0.277	8	0.175

4-way Set Associative Cache			Core	
size [KB]	linesize [words]	energy [nJ]	model	power [mW/MHz]
1	8	0.538	ARM926EJ-S	0.360
2	8	0.542		
4	8	0.550		
8	8	0.564		

SDRAM			
dynamic	energy [nJ]	static	power [mW]
read random	11.747	standby	9.600
read burst	3.373		
write random	10.397		
write burst	1.659		

TLB			
type	assoc	entries	energy [nJ]
unified TLB	2-way	64	0.141
μTLB	full	16	0.125

Table 3. Per-word access energy and power parameters.

The SDRAM energy is composed of static and dynamic energy [18]. We have modeled the low power 64MB Samsung K4X51163PC SDRAM [22] with a memory bus frequency $f_{mem} = 66MHz$ and a supply voltage $V_{dd} = 1.8V$. The static energy consumption, E_{ext_static}, includes the standby power and the power to periodically refresh the SDRAM cells and is computed by

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

where $P_{standby}$ is the static power consumption of the SDRAM (Table 3 (a)). The dynamic energy,

$$\begin{aligned} E_{ext\_dynamic} &= e_{read_{random}} \cdot read_{random} \\ &\quad + e_{read_{burst}} \cdot read_{burst} \\ &\quad + e_{write_{random}} \cdot write_{random} \\ &\quad + e_{write_{burst}} \cdot write_{burst} \end{aligned}$$

includes both SDRAM dynamic energy and the memory bus energy. The energy $e_{read/write_{random/burst}}$ denote the per-word access energy for a random/burst read/write access, respectively.

Table 3 lists the values used for the energy calculations. All energy parameters are the *energy required per word (4 byte) access*, including the values for *SDRAM read/write burst*. The cache, SPM, and TLB access energies were computed for $0.13\mu m$ technology using CACTI [29]. We modified CACTI to eliminate the energy computation procedures for the unnecessary components, such as comparators and tag arrays, in the direct mapped cache to compute the SPM access energy. The core power consumption for a $0.13\mu m$ ARM926EJ-S core without caches was taken from [15]. The static and dynamic energy of the SDRAM were computed using the System Power Calculator from [17], and the bus energy was taken from [23].

Application	Global Data Size [KB]	Maximum Stack Used [KB]	Reference Case DCache [KB]	Reference Case ICache [KB]	Ideal Case DCache [KB]	Ideal Case SPM [KB]	Ideal Case per-access energy of the SPM
fft	67.3	1.0	1	8	1	68	4KB SPM
mp3	102.5	2.0	4	8	4	108	8KB SPM
mpeg4d	487.0	2.3	8	8	8	492	12KB SPM
mpeg4e	488.3	2.3	8	8	8	492	12KB SPM
pgpd	262.5	7.3	1	8	1	272	4KB SPM
pgpe	262.5	7.3	8	8	8	272	12KB SPM
epic	2.3	324.5	4	8	4	328	8KB SPM
unepic	2.5	1.0	8	8	8	4	4KB SPM
synthetic	46.0	8.8	1	8	1	56	4KB SPM

Table 4. Characteristics of each application and horizontally partitioned memory system setup for reference and ideal cases.

6.3 Benchmark Applications

We use thirteen embedded applications to evaluate our evaluation. These include nine benchmarks from MiBench [10] and MediaBench [14], the official ISO MP3 decoder [19], MPEG-4 XviD encoding/decoding [30], and a public key encryption/decryption tool, Pretty Good Privacy (PGP) [21]. We chained the benchmarks *Dijkstra, SHA, ADPCM-enc, ADPCM-dec*, and *Bitcount* together into one benchmark called *synthetic*. Each of the smaller benchmarks is executed once in *synthetic* to represent an embedded application with multiple phases. The characteristics of the applications are summarized in Table 4.

6.4 Other Evaluation Parameters

We compare the horizontally partitioned memory system with our dynamic data SPM management technique to a fully cached system. For the fully cached system (the reference case), both the instruction and data caches are 4-way set associative caches. The instruction cache size is 8KB for all applications. The data cache size varies from 1KB to 8KB for different applications, and it is set to the smallest cache that achieves a cache miss ratio below 5%. Table 4 lists the reference case for each benchmark.

To know how much room is available for improvement in the reference case for each application, we compare our technique to the *ideal* case where all global data and stack frames are placed in the SPM, i.e., there is no page fault and no copy-in/out overhead. Table 4 also lists the ideal case for each benchmark. The data cache in the ideal case is the same as in the reference case. The heap data is cached in the data cache. The per-access energy of the SPM in the ideal case is the per-access energy of the SPM whose die area is comparable to that of the data cache. The ideal case is a theoretical lower bound and typically unachievable in practice.

We solve our ILP models with a commercial solver [11] running on a window based PC (Pentium 4 2.4Ghz with 1GB main memory).

Moving a single page with a p-byte size from the SPM to the external memory or from the external memory to the SPM requires (28 + p/8) instructions. For example, to copy-in/copy-out a page with sizes of 1KB, 512B, 256B, and 128B, we need to run 156, 92, 60, and 44 instructions, respectively.

7. Experimental Results

Figure 8 and Figure 9 show the normalized energy consumption, the normalized number of memory accesses, the data cache hit rate, and the number of pages in/out for each application with different memory subsystem configurations and different data SPM management techniques. The page size varies from 128B to 1KB. The execution time is proportional to the energy consumption of the processor core itself (CORE).

The bars labeled ref show the results of the reference case and ideal show the results of the ideal case. Among the remaining bars for each application, DDxK+SPMyK simulates the horizontally partitioned memory subsystem configuration with a xKB directed mapped cache and yKB SPM. The group of bars right to ideal uses the ILP formulation with the number of data cache misses. The next group of bars uses the ILP formulation with the number of external memory accesses mentioned in Section 5. Note that ref and ideal contain the TLB and μTLB access energy (TLB) too because typically the MMU needs to be enabled in order to use the data cache [12, 16].

The sum of die areas of the directed mapped data cache and SPM in our memory subsystem configuration is comparable to (actually less than) the area of the 4-way set associative data cache used in ref.

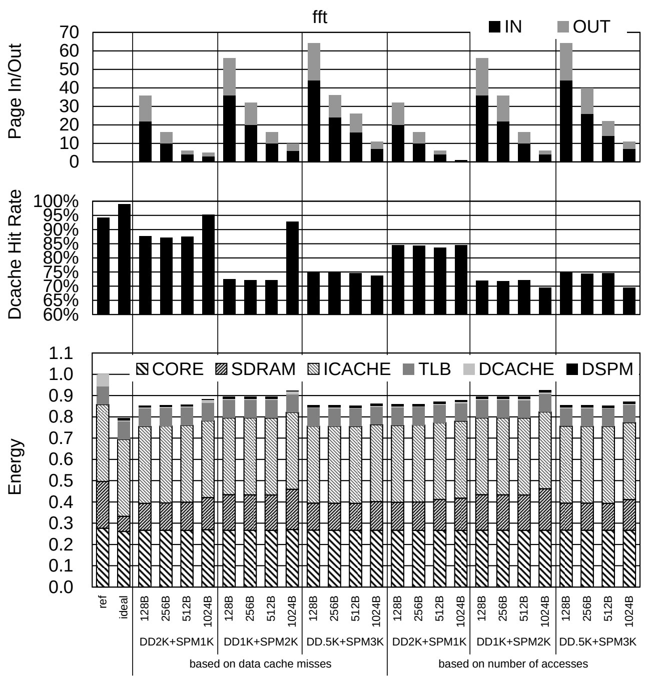
For all applications but *pgpd* and *pgpe*, it takes less than 10 minutes to obtain the solutions of our ILP models. For *pgpd* and *pgpe*, it takes about 30 minutes.

Since the size of the global data and the maximum stack size for *unepic* is smaller than the size of the SPM, our ILP formulation is not needed in this case. Thus, the result of only one horizontally partitioned memory subsystem configuration is reported for *unepic*.

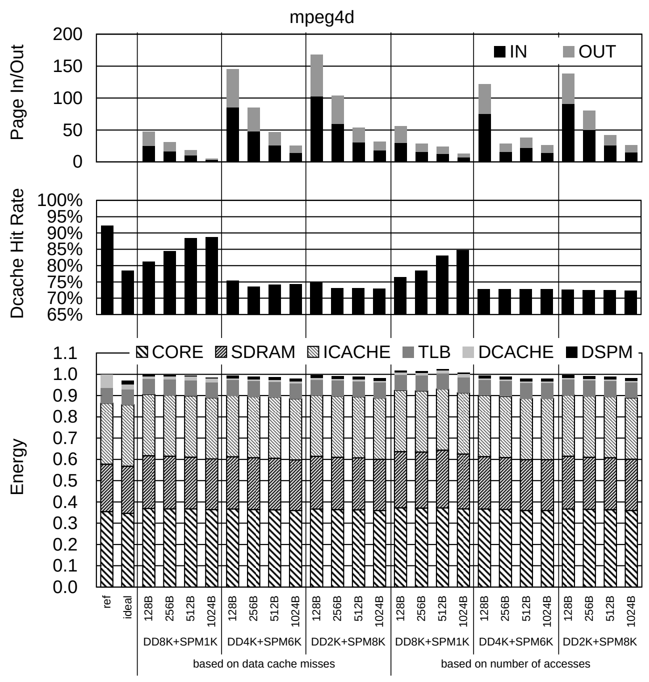
In general we see that the number of pages loaded/stored to/from the SPM increases when we increase the size of the SPM. Bigger SPM has more room to accommodate data pages. Consequently, the ILP model places more pages in the SPM resulting in a high number of pages in/out. However, due to the limited die area, the size of the direct mapped data cache decreases as the size of the SPM increases. This results in low hit ratios for the data cache. Since the pages with a larger number of cache misses are placed in the SPM by our cache-miss based ILP model, its data cache hit ratios are typically larger than those of the other ILP model. In addition, data cache pollution by copying data pages in/out to/from the SPM also accounts for the smaller miss ratio compared to the reference case.

The impact of the page size is different from application to application. Due to the finer granularity of small pages, the ILP model can better pinpoint frequently accessed data regions with small pages. *Fft*, *pgpd*, and *synthetic* belong to this category.

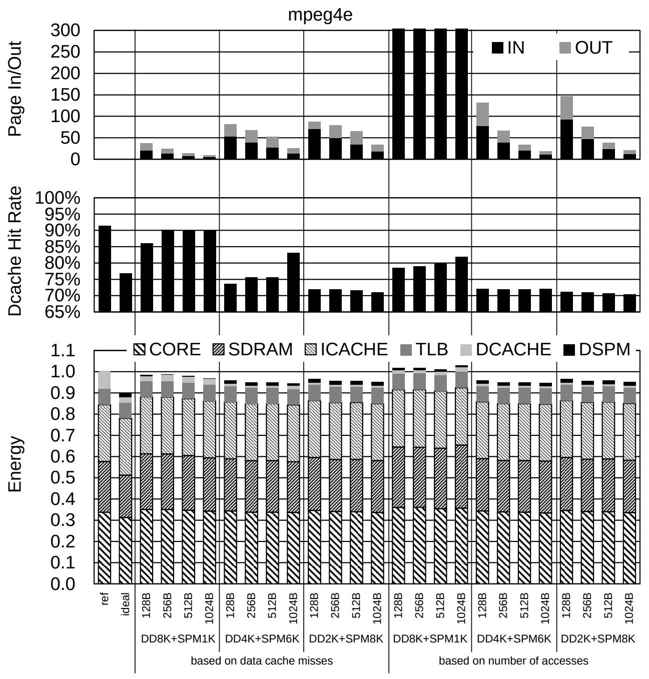
On the other hand, small pages cause more TLB misses than large pages in addition to increasing the number of pages in/out. The large number of TLB misses prolongs the execution time. This is especially true for *mp3* and *pgpe*. For these two applications with small pages, SDRAM is much bigger than *ref* in spite of the large data cache hit rate. This is mostly due to a large number of TLB misses.



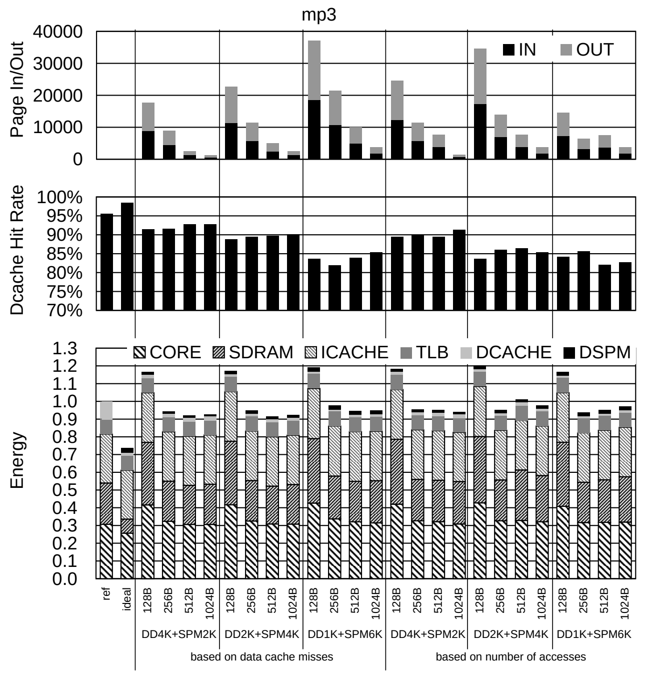
(a)



(b)

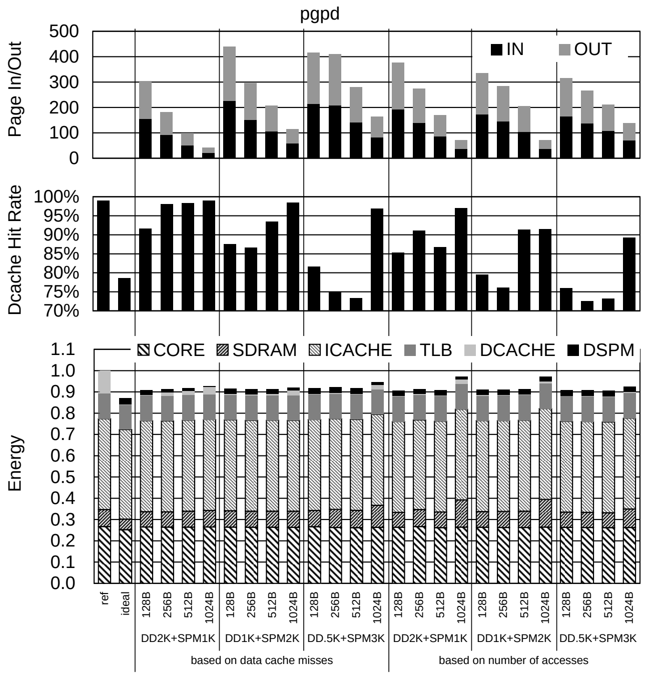


(c)

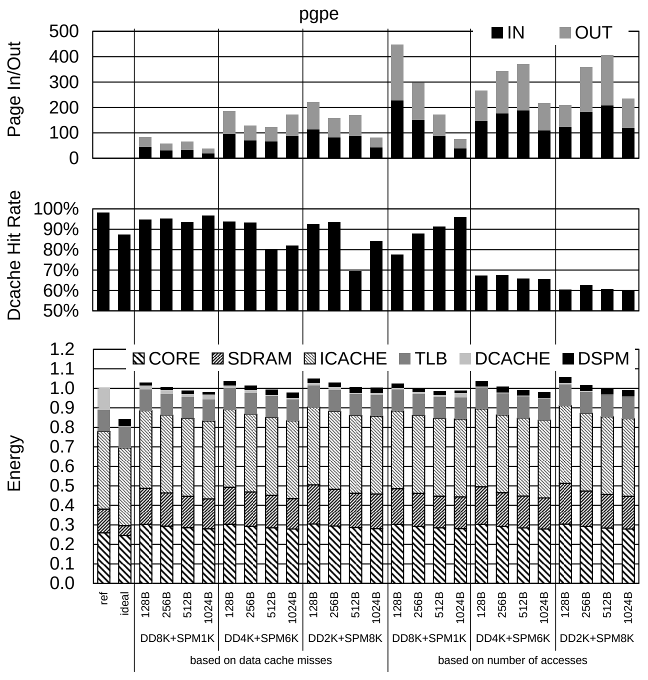


(d)

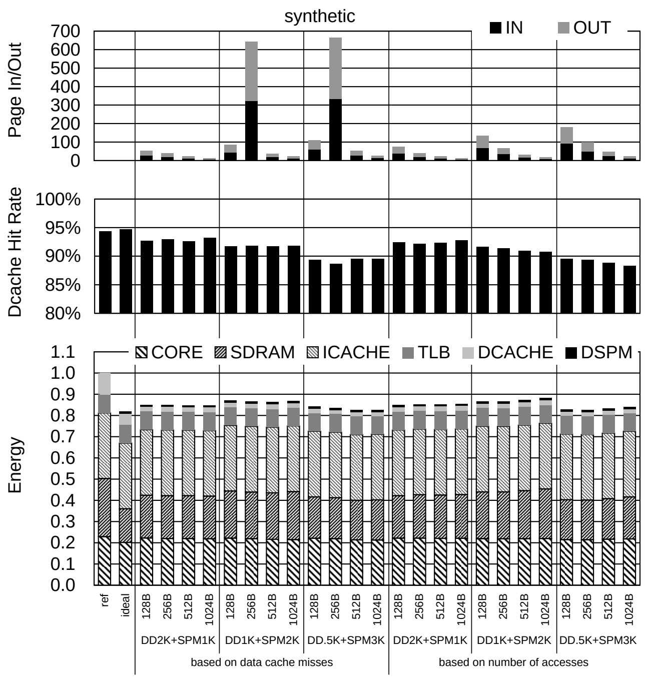
Figure 8. Energy consumption, data cache hit rate, and pages in/out for *fft*, *mpeg4d*, *mpeg4e*, and *mp3*. (a) *fft* (b) *mpeg4d* (c) *mpeg4e* (d)*mp3*



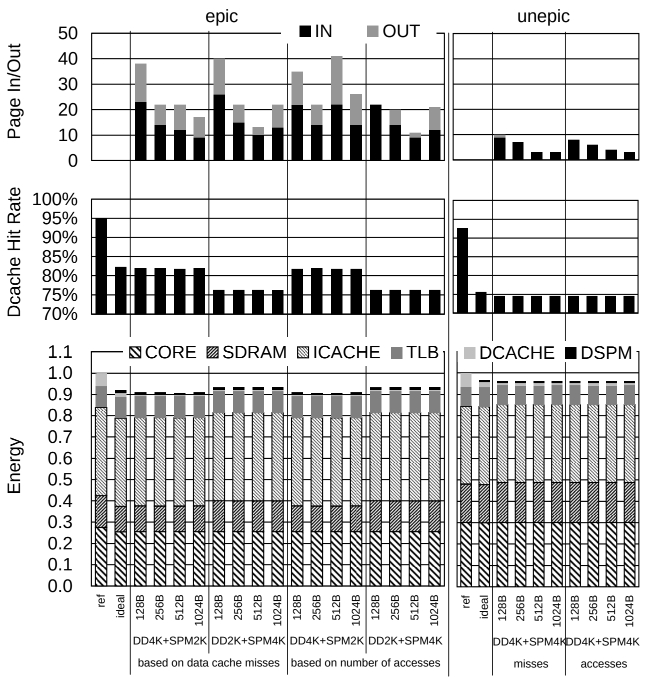
(a)



(b)



(c)



(d)

Figure 9. Energy consumption, data cache hit rate, and pages in/out for *pgpd*, *pgpe*, *synthetic*, *epic*, and *unepic* (a)*pgpd* (b)*pgpe* (c) *synthetic* (d) *epic* and *unepic*

Application	ref	ideal (energy)	the best conf. (time)	the best conf. (energy)	degree (the larger, the better)	configuration
fft	1	0.795	0.959	0.854	71.4%	DD2K+SPM1K, 128B
mp3	1	0.736	1.003	0.924	28.8%	DD2K+SPM4K, 1KB
mpeg4d	1	0.968	1.015	0.978	68.4%	DD4K+SPM6K, 1KB
mpeg4e	1	0.898	0.998	0.945	53.8%	DD4K+SPM6K, 1KB
pgpd	1	0.869	0.993	0.905	72.5%	DD2K+SPM1K, 128B
pgpe	1	0.843	1.075	0.977	14.4%	DD4K+SPM6K, 1KB
epic	1	0.920	0.939	0.908	115.3%	DD4K+SPM2K, 128B
unepic	1	0.967	1.000	0.962	113.3%	DD4K+SPM4K, 128B
synthetic	1	0.819	0.961	0.826	96.1%	DD.5K+SPM3K, 128B
Geometric mean	1	0.865	0.993	0.919	59.9%	

Table 5. The degree of exploiting the room between ref and ideal in energy reduction for each application with the best configuration.

The overhead of copy-in/out a single page increases about 1.5 times when we double the page size (Section 6.4). As long as the ratio of the numbers of pages in/out for the two consecutive page sizes is around 1.5 and the ILP model does not affect the paging behavior that much, there is no difference in the total overhead of copying pages in/out. For the case of *mp3* with a page size of 128B, a much larger SDRAM energy than in the 256B case and a ratio of the pages in/out bigger than 1.5 between 128B and 256B shows that the 128B case has a large copy-in/out overhead. The page-in/out overhead does not affect that much the execution time and energy consumption for all applications but *mp3* because the numbers of pages in/out are not that large. Due to the large number of pages in/out in *mp3*, the page-in/out overhead directly affect its performance and energy consumption.

We see that the best combination of the data cache size, data SPM size, and page size is largely dependent upon the application. In addition we observe that our cache-miss based ILP model works slightly better than our memory-access based ILP model.

We compare our approach with the ideal case (ideal). It has the same cache as the one in the reference case, and all global and stack data is placed in the SPM. It conservatively approximates the energy consumption in a theoretical ideal case (Section 6.4). The difference in energy consumption between ref and ideal is the room available for improvement in energy reduction. The degree of exploitation of this room is represented by,

$$degree = \frac{\mathsf{ref} - the\ best\ configuration}{\mathsf{ref} - \mathsf{ideal}}$$

Table 5 summarizes the effectiveness of our approach for each application with the best memory subsystem configuration and the cache-miss based ILP model. The reason why the *degree* is greater than 100% for *epic* is that the SPM size (2KB) in the best configuration is smaller than the SPM size (4KB) used in ideal for its energy calculation, and for *unepic*, it is due to the direct mapped cache in the best configuration that consumes less energy than the 4-way set associative cache in ideal. Overall, our approach reduces the total energy consumption by 8.1% on average with no performance degradation. This is equivalent to exploiting 60% of the room available in energy reduction between ref and ideal.

8. Conclusions

We introduced a dynamic data SPM allocation technique for contemporary embedded systems with an MMU. The data cache is replaced by a horizontally-partitioned memory subsystem consisting of an SPM and a relatively cheap direct-mapped cache. Global and stack data pages are copied to and from the SPM based on the dynamic call graph. Because we use the MMU to map pages either to the SPM or the external memory, data addresses remain constant for the application. The data page mapping is determined by solving an ILP formulation that approximates our demand paging technique. The ILP model considers each call path from the root to a leaf node in the call graph. One of our ILP models uses a dynamic call graph annotated with the number of memory accesses and the other uses the number of cache misses obtained by profiling.

We see that the best combination of the sizes of the direct-mapped data cache, SPM, and data page is dependent upon the application. In additon, the cache-miss based ILP model works slightly better than the other ILP model.

We compare our approach to a system with a 4-way set associative data cache and the ideal system with the same 4-way cache with all global and stack data placed in the SPM. On average, our approach reduces the total system energy consumption by 8.1% with no performance degradation. This is equivalent to exploiting 60% of the room available in energy reduction between the reference case and the ideal case.

Acknowledgments

We would like to thank Choonki Jang for his comments on the preliminary work of this paper and the anonymous reviewers for their helpful feedback.

References

[1] Advanced Compiler Research Laboratory. Seoul National University Advanced Compiler Tool Kit (SNACK). `http://aces.snu.ac.kr/snack.html`, 2004.

[2] Federico Angiolini, Luca Benini, and Alberto 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.

[3] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro 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.

[4] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

[7] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul 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] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT'06: Proceedings of the sixth ACM & IEEE internationel conference on Embedded Software*, 2006.

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

[10] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, December 1998.

[11] Lindo Systems Inc. What's Best! 8.0. `http://www.lindo.com/products/wb/wbm.html`, 2007.

[12] Intel. *Intel XScale Core Developers Manual*, January 2004.

[13] Peter Marwedel Lars Wehmeyer, Urs Helmig. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004), Munich, Germany*, Jun 2004.

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

[15] ARM Ltd. ARM926EJ-S,. `http://www.arm.com/products/CPUs/ARM926EJ-S.html`, 2002.

[16] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, 2003.

[17] Micron Technology, Inc. MT48H8M16LF Mobile SDRAM. `http://www.micron.com/products/dram/mobilesdram/`, 2003.

[18] Micron Technology, Inc. Mobile SDRAM Power Calc 10. `http://www.micron.com/systemcalc`, 2004.

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

[20] Nghi Nguyen, Angel Dominguez, and Rajeev 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.

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

[22] Samsung Semiconductor. K4X51163PC Mobile DDR SRAM. `http://www.samsung.com/products/semiconductor/MobileSDRAM/`, 2005.

[23] Aviral Shrivastava, Ilya Issenin, and Nikil 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] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter 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] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 409, 2002.

[26] Sumesh Udayakumaran and Rajeev 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.

[27] Manish Verma, Stefan Steinke, and Peter Marwedel. Data partitioning for maximal scratchpad usage. In *Proceedings of the 2003 conference on Asia South Pacific design automation, KitaKyushu, Japan*, Jan 2003.

[28] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis ,Stockholm, Sweden*, 2004.

[29] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid State Circuits*, 31(5):677–688, 1996.

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