A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization *

Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min
School of Computer Science and Engineering
Seoul National University
Seoul, Korea
http://aces.snu.ac.kr
{bernhard,chihun,choonki,yoonsung,jlee}@aces.snu.ac.kr, symin@dandelion.snu.ac.kr

ABSTRACT

In this paper, we propose a fully automatic dynamic scratchpad memory (SPM) management technique for instructions. Our technique loads required code segments into the SPM on demand at runtime. Our approach is based on postpass analysis and optimization techniques, and it handles the whole program, including libraries. The code mapping is determined by solving mixed integer linear programming formulation that approximates our demand paging technique. We increase the effectiveness of demand paging by extracting from functions natural loops that are smaller in size and have a higher instruction fetch count. The postpass optimizer analyzes the object files of an application and transforms them into an application binary image that enables demand paging to the SPM. We evaluate our technique on eleven embedded applications and compare it to a processor core with an instruction cache in terms of its performance and energy consumption. The cache size is about 20% of the executed code size, and the SPM size is chosen such that its die area is equal to that of the cache. The experimental results show that, on average, the processor core and memory subsystem's energy consumption can be reduced by 21.6% and the performance improved by 20.2%. Moreover, in comparison with the optimal static placement strategy, our technique reduces energy consumption by 23.7% and improves performance by 22.9%, on average.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'06, October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-543-6/06/0010 ...$5.00.

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, Demand paging, Heterogeneous memory, Scratchpad memory, Embedded systems.

1. INTRODUCTION

Reducing energy consumption in mobile embedded systems is of great importance these days. The memory subsystem, especially on-chip caches using SRAM, consumes a large portion of the total chip power. Recently, scratchpad memory (SPM) has been proposed as an alternative to caches in order to reduce power and improve performance [6, 23]. While cache behavior is unpredictable to programmers and transparent to applications, programmers can explicitly map the addresses of external memory to the addresses of the SPM because the SPM is addressed using an independent address space.

Many studies [2, 3, 5, 6, 9, 10, 15, 16, 27, 28, 29, 30, 31] have been performed to map code or data to the SPM in order to reduce energy or improve performance. Although dynamic mapping approaches utilize the given SPM space better than static approaches, only a few studies [9, 14, 27, 31] address *dynamic code* mapping. While the dynamic approaches in [27] and [31] are fully automatic, they require analysis of the application's source code and, thus, cannot handle library code unless the library source code is available. In [14], the authors use the program trace to compute the most beneficial set of basic blocks, and could therefore handle libraries. However, their framework does not produce executable images; instead, it analytically computes the expected performance and energy metrics.

This paper focuses on a fully automatic code placement technique for SPM that loads required code sections on demand into the SPM at runtime. The contributions of this paper are as follows.

First our approach is based on compiler postpass optimizations that operate on the object code or binary executable image. This makes our technique applicable to the whole program, including libraries.

Our technique is implemented in a post-pass optimizer. The post-pass optimizer takes object files of an application as inputs and generates a binary executable image that contains a small page manager. The original application code, including libraries, is divided into multiple segments. Each code segment in the application is placed in the SPM or external memory, or it is loaded on demand by the page manager into an execution buffer in the SPM from the external memory. The page manager checks if the target of a function call/return already resides in the execution buffer. If it does, the page manager simply passes control to the target; otherwise, the page manager loads the corresponding code segment into the buffer and then passes control to the target. The optimizer transforms function call/return instructions into a call to the page manager only if the callee (or the function returned to) has been identified as having been paged from external memory.

The second contribution of this paper is that the optimizer extracts natural loops from the application binary and abstracts them as functions (this technique is called function abstraction or function outlining [21]). While placing loops in the SPM is similar to the loop cache approach [11, 18], our placement is controlled by software at runtime, which makes more effective use of the SPM.

Finally, we approximate the dynamic code mapping for demand paging with a mixed integer linear programming (ILP) problem. The classical 0-1 Knapsack formulation is extended to model our demand paging mechanism. The solution of our ILP formulation determines a suboptimal placement of the outlined functions and original functions in the SPM.

We evaluate the performance and energy consumption of our compiler postpass dynamic SPM management technique using eleven realistic embedded applications. Compared to a system with an on-chip instruction cache and similar die size, our technique improves performance by 20.2% and reduces energy consumption by 21.6% on average.

The rest of this paper is organized as follows. Section 2 lists related work. Section 3 describes the overall optimization framework of our approach. Section 4 presents our ILP formulation in detail. Section 5 describes the evaluation environment. Section 6 discusses the experimental results obtained. Finally, Section 7 concludes the paper.

2. RELATED WORK

Steinke *et al.* [27] proposed a technique that dynamically copies code sections into the SPM. At selected program points, they insert copying function calls that copy a corresponding program part (i.e., a loop or a function body) into the SPM. Their technique automatically selects the copy points and program parts that can be active at the same time in the SPM. After determining the candidates for copy points and program parts, they compute energy costs and select an optimal set of program parts for dynamic copying by solving an ILP problem. Verma *et al.* [31] proposed a technique to share the SPM between various memory objects (e.g., global variables, non-scalar local variables, and code segments) using the overlay technique. Their approach is capable of handling both instructions and data in a dynamic way at the same time. They perform lifetime analysis on memory objects by analyzing memory traces to assign the SPM space. The memory objects are copied into the SPM when they are required.

While the approaches in [27] and [31] require the application's source code, our approach requires only object code or binary code. The SPM space is shared between functions and loops in the whole program, including libraries, resulting in better utilization of the SPM space. Automatic code overlay [8, 26] could be used with our approach instead of demand paging. However, automatic code overlay incurs a code copy cost, and it requires the SPM size to be greater than the sum of the function sizes in the application's maximum call chain.

Recently, Janapsatya *et al.* [14] proposed a dynamic SPM allocation strategy based on the so-called *concomitance metric*. Concomitance is a measure to indicate how temporally related two code blocks are. The concomitance is computed based on profiling information, and the resulting dynamic SPM allocation is evaluated using several energy models without producing a running version of the SPM-enabled application.

In [9], Egger *et al.* proposed a dynamic SPM management technique for systems with virtual memory. A postpass optimizer groups code that is to be run from the SPM into pages and places them in a special memory region. At runtime, the memory management unit's page fault exception mechanism is used to track the course of the program and copy the code to the SPM on demand.

Beyond these dynamic approaches, some studies have been done on static code mapping to SPM. Steinke *et al.* [28] proposed a static and optimal selection algorithm. The algorithm selects beneficial program parts (e.g., basic blocks) and variables that can be placed in the SPM to save energy. Their mapping problem is formulated as a Knapsack problem. Angiolini *et al.* [2, 3] proposed a postpass and static approach, based on dynamic programming, to find an optimal set of frequently accessed instruction blocks to map to the SPM. Their formulation is a variant of the Knapsack problem, which considers extra instructions to relocate the code blocks. Verma *et al.* [30] proposed a static algorithm and an ILP formulation that can be applied to embedded systems with caches in addition to SPM in order to reduce the energy consumed by instruction fetches. Our approach is different from the static solutions in that we focus on a dynamic code placement technique that is based on demand paging and an approximated mixed ILP formulation.

Many studies on SPM management techniques focus on assigning data objects to the SPM statically or dynamically. Udayakumaran and Barua [29] proposed a compiler algorithm for dynamically managing the SPM to place global and stack variables. Avissar *et al.* [5] proposed a static data partitioning algorithm between SPM and external memory to improve performance. Their technique is based on a binary ILP formulation. Francesco *et al.* [10] proposed a runtime mechanism that uses direct memory access (DMA) engines to reduce the cost of dynamically copying data to the SPM and provides an application programmer's interface (API) to utilize them. Kandemir *et al.* [16, 15] proposed

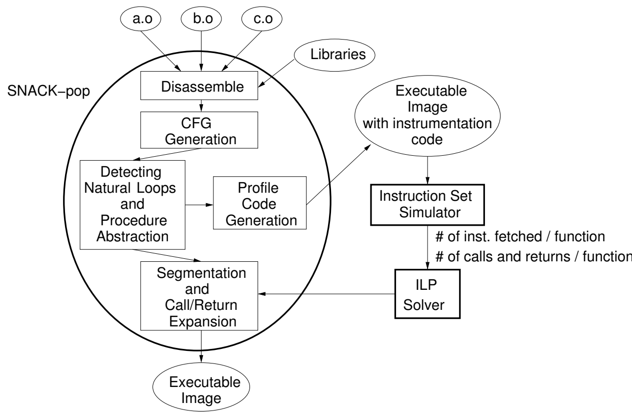


Figure 1: The post-pass optimizer.

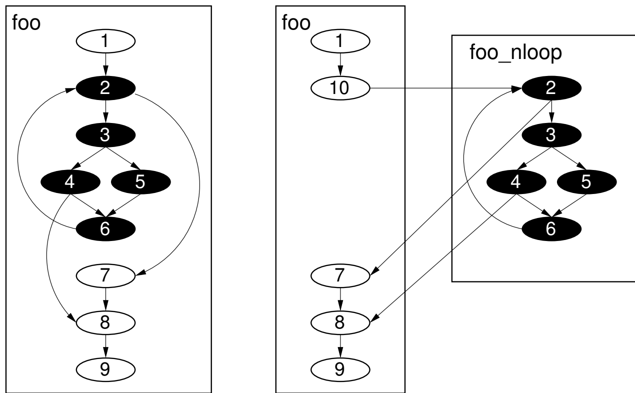


Figure 2: Extracting a natural loop from the control flow graph of a function.

a compiler technique based on loop transformations that dynamically manages SPM for arrays. Panda *et al.* [22] introduced techniques to partition on-chip memory into SPM and cache areas. To improve performance, they statically assign critical data in the program to the SPM.

3. SPM MANAGEMENT

In this section, we describe the basic idea of our post-pass approach and on-demand code mapping techniques.

3.1 The Post-Pass Optimizer

Figure 1 shows the skeleton of our post-pass optimizer, SNACK-pop, which is part of the (**S**eoul **N**ational University **A**dvanced **C**ompiler tool **K**it)[24]. Application and library object files are fed into the post-pass optimizer, disassembled, and then converted into a list of functions. Each function contains a list of instructions, and each instruction has its own relocation information. SNACK-pop detects addresses of constant data in code sections (i.e., .text sections) that are passed as arguments to other functions or that are accessed by other functions directly. Such constant data is copied and clustered together with the corresponding function that receives the address of data as an argument or that accesses the data directly. Then, a basic block control flow graph (CFG) is constructed for each function.

SNACK-pop analyzes each CFG and extracts the natural loops of each function [1, 21]. As shown in Figure 2, each natural loop is replaced with a new node that contains a branch to the header of the natural loop. The natural loop is transformed into a function and added to the function list. Note that the functions outlined in this way do not need to follow the calling convention (i.e., there is no function call overhead, but merely a jump to the loop header). We describe the loop selection criteria later in Section 4.2.

Then, our post-pass optimizer inserts instrumentation code into each function and reassembles the functions. A binary executable image containing instrumentation code is generated and run on a cycle-accurate instruction set simulator, SNACK-armsim [24]. The simulator generates profiling information, such as the number of instructions fetched from each function and the number of calls and returns to each function. This profiling information is then fed into an integer linear programming (ILP) solver that categorizes the functions (including the functions from natural loops) into three classes: SPM, Ext, and Paged. The functions in SPM and Ext are statically placed in the SPM (i.e., SRAM) and external memory (i.e., SDRAM), respectively. The functions in Paged are statically placed in the external memory and are copied to the SPM on demand at runtime.

All functions in Paged are transformed into *segments*. Each segment consists of single or multiple pages. Each page has a fixed size, 64 bytes in our case. Each segment contains exactly one function and at least one page.

SNACK-pop then expands each call instruction to a function in Paged into instructions that jump to a corresponding *paged function table* entry. The paged function table is similar to a jump table and is contained in the page manager.

For paged functions that currently reside in the execution buffer area of the SPM, the table entry contains a direct jump to the function. If the target is not in the execution buffer, the table entry contains a jump to the page manager. The page manager loads the target segment (callee) into the execution buffer, modifies the corresponding table entry, and then jumps to the target.

Return instructions in each function are expanded to a call to the page manager. The page manager checks whether the target is in the right place. If the target is located in SPM and Ext, then it just jumps to the target. Otherwise, if the target is located in Paged, the page manager further checks whether the target currently resides in the execution buffer. If not, it loads the target segment into the execution buffer, modifies the corresponding table entry, and jumps to the

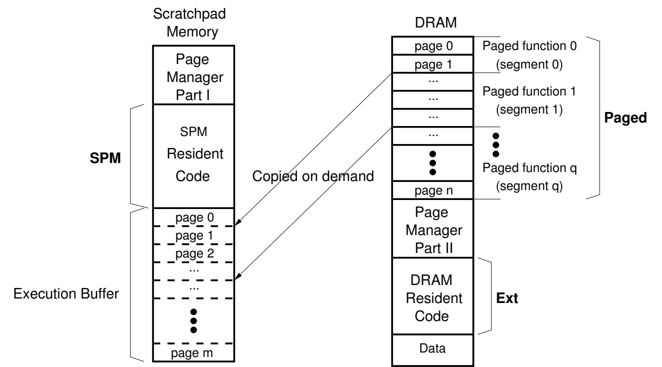


Figure 3: Memory map.

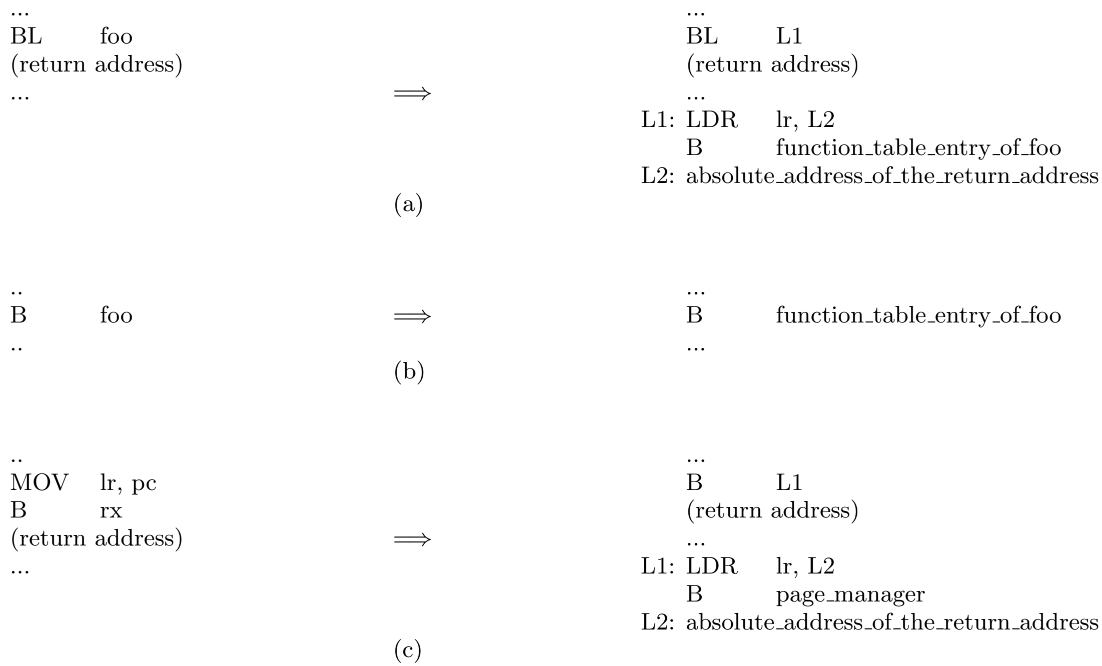


Figure 4: Function call expansion. (a) A function call using the linkage register. (b) A function call without the linkage register. (c) A function call with a function pointer stored in a register `rx` and the linkage register.

target of the return.

Figure 3 shows the memory map used in our scheme. The page manager takes care of loading paged functions into the SPM at runtime. Parts of the page manager that are frequently executed reside in the SPM; the remaining parts reside in the external memory.

Finally, segments in Paged are linked together and become a code section in the final binary image. This section is placed in the external memory at the beginning of execution. The functions in the SPM category are linked together and become another code section in the final image. This section is placed in the SPM at the beginning of execution. The same is true for the functions in Ext, but this section is placed in the external memory. The page manager code is also linked in and becomes a part of the final executable image.

Note that the executable image size is slightly bigger than the original because of the internal fragmentation caused by segmentation. SNACK-pop uses the space that results from internal fragmentation to place the extra instruction generated by the call/return expansion.

3.2 The Paged Function Table and the Page Manager

The page manager contains three tables to manage the execution buffer in the SPM: the paged function table, the page table, and the execution buffer page table.

The *paged function table* contains one entry for each function in Paged. Each entry is two instructions in length (i.e., 8 bytes in our case). Every call to a function in Paged is translated into a call to the corresponding entry in the paged function table. The contents of the first instruction in the table entry change at runtime and depend on the current location of the paged function as follows.

For functions that currently reside in the execution buffer in the SPM, the first instruction in the paged function table entry contains an unconditional branch to the current address of the callee in the execution buffer. The second instruction is not executed in this case. We call this scenario a *paged function table hit*.

When the function does not currently reside in the execution buffer, the first instruction stores the current program counter (PC) register to a specific location p inside the page manager. The second instruction in the paged function table entry contains a branch to the page manager loader function. To identify the paged function to be called, the page manager computes the *page index* of the function by subtracting the paged function table base address from the PC stored at location p. This scenario is called *paged function table miss*.

Like the paged function table, the *page table* also contains one entry per function in Paged. Each entry contains the following information about the corresponding page and the segment (i.e., function) to which it belongs:

- the address of the page in the execution buffer when it is loaded into the buffer
- the information of the segment to which the page belongs (i.e., the length of the segment and its starting address in the external memory)

After computing the page index as described above, the page manager loader function loads the target segment from the external memory into the execution buffer and modifies the first instruction in the corresponding paged function

table entry such that it contains an unconditional branch instruction to the address of the function entry in the execution buffer.

The *execution buffer page table* contains one entry per page in the execution buffer. The entry contains the page table index of the page loaded into the execution buffer page. Using the execution buffer page table and a pointer to the base of the execution buffer page table, the page manager implements a simple round-robin page replacement policy. Because page management is completely done by software and the smallest replacement unit for paging is a segment (i.e., not a page but a function), more sophisticated replacement policies, such as LRU, will increase the complexity of the page manager code resulting in more page manager overhead at run time.

3.3 Call/Return Expansion

When the segment to which the target address of a function call/return belongs has been paged out, the page manager needs to load the target segment into the execution buffer before the actual code can be executed. As mentioned earlier, each call to a function in Paged is translated into a branch instruction to the corresponding function table entry. There are two cases: 1) typical calls where execution resumes immediately after the branch instruction (following the callee's return), and 2) branches that do not return (Figure 4(a) and (b), respectively).

Function calls via function pointers need to be treated specially because the target of such calls is usually not known at compile time (Figure 4(c)). The original function call is translated into a direct page manager function call. At runtime, the page manager determines whether the function call target is a paged function and, if so, whether the target function is currently residing in the execution buffer. In the former case, control is immediately transferred to the target address. In the latter case, the page manager needs to load the function into the SPM before branching to the target address.

In all cases except branches that do not return (Figure 4(b)), the return address must be translated into the absolute address and saved in the linkage register before the translated function call occurs. This absolute return address will be used by the page manager later to identify the correct location of the return's target segment (note that the caller may have been paged out when returned). The extra code generated by the translation is placed in the unused space caused by internal fragmentation. If there is not enough space, we allocate one more page to the segment and place the extra code there.

When the callee returns control to a caller in Paged, there is no guarantee that the caller still resides in the execution buffer. This means that the post-pass optimizer always needs to translate function returns to a page manager call. Every return instruction in all functions whose target is a function in Paged is translated into a call to the page manager. To identify these return instructions, SNACK-pop uses dynamic call graph information based on profiling. Whenever control is passed to the page manager by return instructions, the page manager identifies the location of the target segment and computes the real target address using the absolute return address saved in the linkage register. When the target segment does not reside in the execution buffer, the page manager loads the target segment into the execution buffer. Then, control is passed to the target address in the buffer. In addition, the page manager updates the corresponding paged function table, page table, and execution buffer page table entries appropriately. If the target still resides in the execution buffer, the page manager simply branches to the target address.

Note that not all return instructions from functions need to be checked by the page manager at runtime: if all callers of a function f are either located in SPM or Ext, the caller cannot be paged and it is safe to return directly without any additional overhead. This significantly reduces the overhead caused by return expansion. For functions called via function pointers, we generally cannot determine the caller at compile time. In this case, we use profiling information to check whether the callee is ever called by functions located in Paged.

4. INTEGER LINEAR PROGRAMMING FORMULATION

A simple and intuitive solution to SPM code placement without paging is mapping it to the 0-1 Knapsack problem [3, 5, 7, 28]. The objective is to maximize performance and/or minimize the energy consumed by memory accesses. This 0-1 Knapsack formulation gives a statically optimal mapping for SPM. We extend the formulation later to approximate our demand paging mechanism.

4.1 0-1 Knapsack Problem

The problem is mapped to the 0-1 knapsack problem as follows. There are N functions in the program. For every function f_i, we denote the size in bytes of f_i by S_i and the dynamic number of instruction fetches plus the number of accesses to the read-only data located within f_i by A_i, where both S_i and A_i are integers. If SPM size is M bytes, and we want to fetch as many instructions and read-only data as possible from the SPM, what functions should we place in the SPM?

Since SPM (i.e., on-chip SRAM) and external memory (i.e., SDRAM) differ by an order of magnitude in the access time and energy consumption per access, finding the solution to the 0-1 knapsack problem involves *minimizing energy consumption and maximizing performance* simultaneously. Here, we formulate the 0-1 Knapsack problem as a binary integer linear programming problem. This formulation is basically used to map code into the SPM statically in [3, 5, 28] and gives the optimal solution for static mapping. In our formulation, the following symbols are used:

A_i	The number of instruction fetches and read-only data word accesses located in a function f_i
S_i	The size of a function f_i in bytes
N	The number of functions in the application
S_{spm}	The SPM size in bytes
E_{spm}	The energy consumed to fetch an instruction (or a word) from the SPM
E_{ext}	The energy consumed to fetch an instruction (or a word) from the external memory

The binary integer variables I_{spm} and I_{ext} for each function f_i are defined by

$$I_{spm}(i) = \begin{cases} 1 & \text{if } f_i \text{ is placed in the SPM} \\ 0 & \text{otherwise} \end{cases}$$

$$I_{ext}(i) = \begin{cases} 1 & \text{if } f_i \text{ is placed in the external memory} \\ 0 & \text{otherwise} \end{cases}$$

The objective function to be minimized is the total energy consumption by all memory accesses caused by instruction fetches:

$$\sum_{i=1,N} (I_{spm}(i) \cdot A_i \cdot E_{spm} + I_{ext}(i) \cdot A_i \cdot E_{ext})$$

Since each function is located in only one place (i.e., either in SPM or external memory, but not both),

$$I_{spm}(i) + I_{ext}(i) = 1 \text{ for all } 0 < i \leq N$$

In addition, the sum of the sizes of the functions placed in the SPM cannot exceed the SPM size.

$$\sum_{i=1}^{N} I_{spm}(i) \cdot S_i \leq S_{spm}$$

To maximize performance in the same ILP formulation, we can define E_{spm} and E_{ext} as memory access times to fetch an instruction from the SPM and external memory, respectively.

The objective function with the above constraints is solved with an integer linear programming solver. The values of the binary integer variables denoting the function locations (i.e., $I_{spm}(i)$ and $I_{ext}(i)$) are then fed into SNACK-pop.

4.2 The Effect of Natural Loops

In the previous ILP formulation, a small function with high instruction fetch and read-only data access counts is likely to be placed in the SPM rather than in the external memory. Based on this observation, we extract natural loops from functions and abstract the loops into functions. The functions from which the natural loops are extracted are most likely to be placed in the external memory due to their low instruction fetch count.

To determine the natural loops to be abstracted into functions, we start from the outermost loops and move into the inner loops. A loop is selected and abstracted into a function when its static instruction count is less than its dynamic instruction count, and we then stop moving into the loop.

However, if the size of the selected loop is greater than r% of the size of the function to which it belongs, we do not extract the loop from the function. This effectively increases the number of small functions with high instruction fetch counts which results in a better solution of the binary ILP formulation for the utilization of the SPM. In our case, $r = 50$.

4.3 Extension to Demand Paging

When it comes to demand paging, we have another class of functions: Paged. The functions in this class reside in the external memory, but they are copied to the execution buffer in the SPM on demand when they are needed by the application. We assume that there is another memory location called Paged and modify the object function of the binary integer linear programming formulation. The binary ILP formulation now becomes a mixed integer linear programming formulation. First, we define a binary integer variable I_{paged} for Paged memory as follows:

$$I_{paged}(i) = \begin{cases} 1 & \text{if } f_i \text{ is in Paged and } S_i \leq I_{buffer} \times P \\ 0 & \text{otherwise} \end{cases}$$

Application	Source	Code Size (Byte)
(*quicksort*)	MIBench	
(*dijkstra*)	MIBench	
(*sha*)	MIBench	
(*adpcm-enc*)	MediaBench	
(*adpcm-dec*)	MediaBench	
(*bitcount*)	MIBench	
synthetic	Synthetic	12,328
fft	MIBench	14,688
epic	MediaBench	21,124
unepic	MediaBench	20,128
mpeg4enc	www.xvid.org	49,912
mpeg4dec	www.xvid.org	44,248

Table 1: Applications used. *Synthetic* is a combination of *quicksort*, *dijkstra*, *sha*, *ADPCM-enc*, *ADPCM-dec*, and *bitcount*

Component	Parameters
Core (0.13μm)	Pipeline: ARM926E-S with 4-word instruction prefetch buffer Instruction set: ARMv5TE Clock frequency: 200MHz Power consumption without caches: 0.30mW/MHz, 60mW
Off-Chip Bus	Clock frequency: 66.6MHz Energy consumption per burst mode transfer: 10.0nJ Energy consumption per non-burst mode transfer: 8.0nJ
SPM (0.13μm)	on-chip SRAM, 1 cycle (core clock) access energy/access — Size (KB): Energy (nJ) 1: 0.128 2: 0.134 4: 0.145 6: 0.159 *(interpolated)* 8: 0.173 12: 0.189 *(interpolated)* 16: 0.206
On-Chip Instruction Cache	4-way set associative, 32-byte line size, 1 cycle (core clock) access energy/access — Size (KB): Energy (nJ) 0.5: 0.534 1: 0.538 2: 0.542 4: 0.551 8: 0.564
External Memory	SDRAM, 128MB 24-cycle sequential access, 30-cycle non-sequential access round-trip time (core clock) Standby power consumption: 57.3mW active energy/access (excluding standby power and including off-chip bus energy) — type: Energy (nJ) Burst read (8 words): 19.28 Burst write (8 words): 13.28 Non-burst read: 11.7nJ Non-burst write: 10.4nJ

Table 2: Architecture parameters used in our simulation with SNACK-armsim

where P is the page size and I_{buffer} is a new general integer variable. I_{buffer} represents the size of the execution buffer in number of pages. The constraint for the execution buffer size is as follows:

$$0 \leq I_{buffer} \cdot P \leq S_{spm}$$

The objective function becomes

$$\sum_{i=1,N} (I_{spm}(i) \cdot A_i \cdot E_{spm} + I_{ext}(i) \cdot A_i \cdot E_{ext} + I_{paged}(i) \cdot [A_i \cdot E_{spm} + Penalty_i])$$

where $Penalty_i$ is the energy consumed by copying function f_i from the external memory to the SPM:

$$Penalty_i = (C_i + R_i)(E_{spm} + E_{ext})(\lceil S_i/P \rceil (P/4)) + C_i E_c + R_i E_r$$

with

C_i	Number of calls to f_i
R_i	Number of returns to f_i
E_c	Energy consumed by extra instructions generated by the call expansion
E_r	Energy consumed by extra instructions generated by the return expansion

where $P/4$ is the number of words in a page, and $\lceil S_i/P \rceil (P/4)$ is the total number of words to be copied.

Similar to the formulation in Section 4.1, each function can be located in exactly one place.

$$I_{spm}(i) + I_{ext}(i) + I_{paged}(i) = 1 \text{ for all } 0 < i \leq N$$

In addition, the sum of the sizes of the functions placed in the SPM cannot exceed the size of the SPM minus the execution buffer size.

$$\sum_{i=1,N} I_{spm}(i) \cdot S_i \leq S_{spm} - I_{buffer} \cdot P$$

We conservatively assume that a segment miss always occurs whenever a paged function is called or returned to. In reality, the function called or returned to might still reside in the execution buffer. For this reason, our mixed integer linear programming model produces a suboptimal solution to the dynamic scratchpad memory allocation problem.

To make our demand paging more effective, we break a function into inner natural loops of a size smaller than a predefined threshold value S_t if the outlined function constructed from a natural loop is too big. In our case, $S_t = 256$ bytes. This makes the execution buffer size smaller and results in better SPM space utilization.

5. EVALUATION ENVIRONMENT

5.1 Applications

We evaluated our code placement strategy with eleven embedded applications from MI bench [12], Media bench [17], and www.xvid.org [13], which are summarized in Table 1. The last column in the table shows the size of executed code for each application. To better reflect realistic embedded mobile applications, we converted the file I/O routines in the applications to routines that access memory.

We combine *quicksort*, *dijkstra*, *sha*, *adpcm-enc*, *adpcm-dec*, and *bitcount* to represent a multi-phase (function) application called *synthetic*, that executes one phase at a time. Each of these six applications is called (executed) once.

5.2 Simulation Environment

Evaluation is done using simulation. We use our cycle-accurate ARM architecture simulator, SNACK-armsim, which is one part of the **S**eoul **N**ational University **A**dvanced **C**ompiler tool **K**it. The processor core modeled in SNACK-armsim uses the same 5-stage pipeline and instruction prefetch buffer used in the ARM926E-S core and simulates the ARMv5 instruction set [4]. The processor core also features a memory subsystem similar to ARM926EJ-S, including scratchpad memory, instruction cache, data cache, a write buffer, and a memory management unit (MMU). Note that we do not use the MMU and data cache in our experiments. SNACK-armsim's cycle accuracy (including the memory subsystem) has been validated with the ARM926EJ-S application baseboard from ARM [4]. The simulation parameters are given in Table 2.

5.3 Energy Model

We focus on the energy consumed by the processor core and the memory components: on-chip SRAM (SPM), instruction cache, and external memory (off-chip SDRAM). We can estimate the energy E consumed by the system without caches using the following formula:

$$E = E_{core} + E_{spm} + E_{ext}$$

E_{core} is computed by

$$E_{core} = t \cdot P_{core}$$

where t is the execution time of the application and P_{core} is the average power consumed by the core. We use the values of an ARM946EJ-S embedded core without caches (P_{core} is 0.3mW/MHz with 0.13μm technology[4]).

E_{spm} is given by

$$E_{spm} = N_{spm} \cdot E_{sram}$$

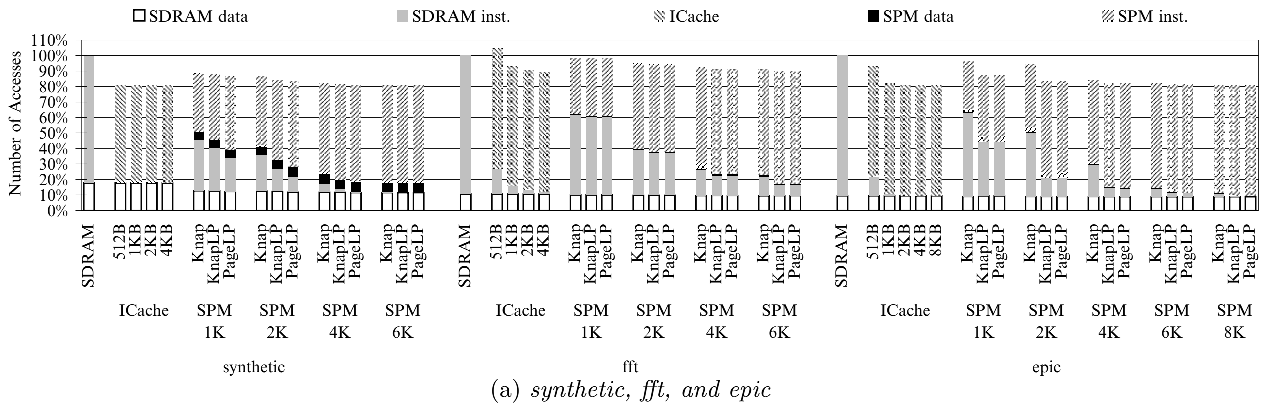
where E_{sram} is the average energy consumed by one SPM access, and N_{spm} is the number of read and write operations to the SPM. We computed the average energy consumption per access (i.e., energy consumed by the data array in the cache) with CACTI [32].

SDRAM energy is composed of static and dynamic energy [20]. We modeled the low power 128MB Micron MT48H8M16LF SDRAM with a memory bus frequency of 100MHz and a supply voltage V_{dd} of 1.8V [19]. The static energy consumption includes the standby power and the power to periodically refresh the SDRAM cells.

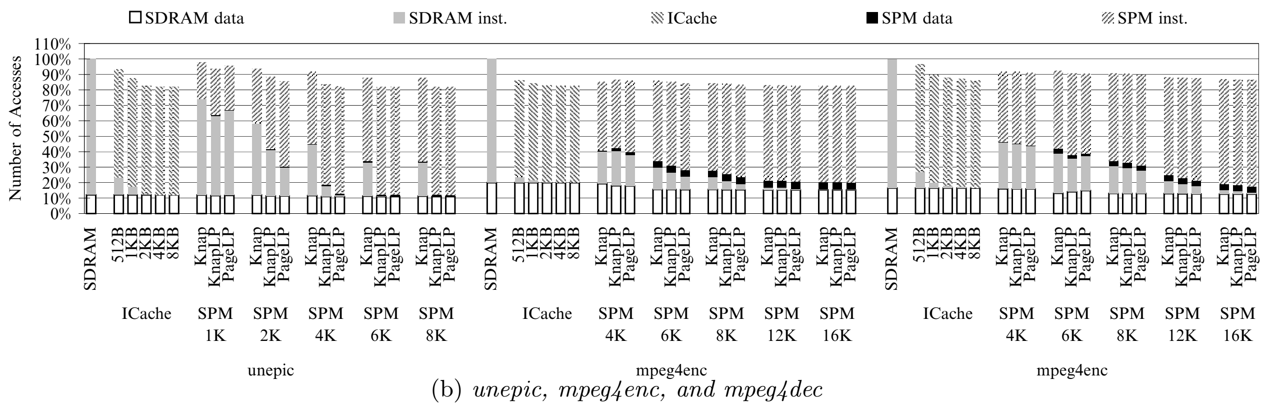
E_{ext} is given by the following formula:

$$\begin{aligned} E_{ext} = \; & t \cdot P_{static} \\ & + N_{ext_read_non-burst} \cdot E_{sdram_read_non-burst} \\ & + N_{ext_write_non-burst} \cdot E_{sdram_write_non-burst} \\ & + N_{ext_read_burst} \cdot E_{sdram_read_burst} \\ & + N_{ext_write_burst} \cdot E_{sdram_write_burst} \end{aligned}$$

where t is the execution time of the application, P_{static} is the average power consumed by the external memory when there are no memory accesses, E_{sdram_type} is the dynamic SDRAM energy consumed by one access whose type is `type` where E_{sdram_type} includes off-chip bus energy per access, and N_{sdram_type} is the number of `type` operations to the external memory. The energy parameters used are given

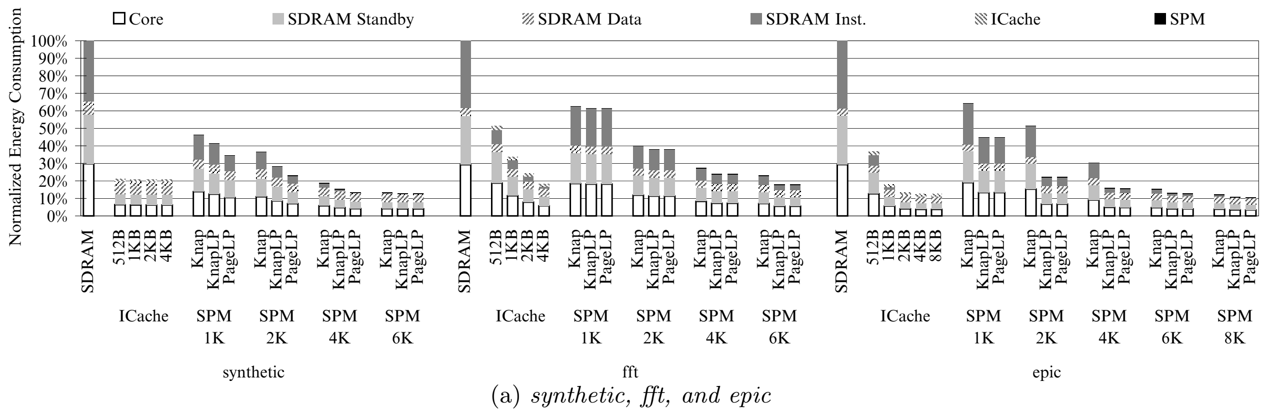


(a) *synthetic, fft, and epic*

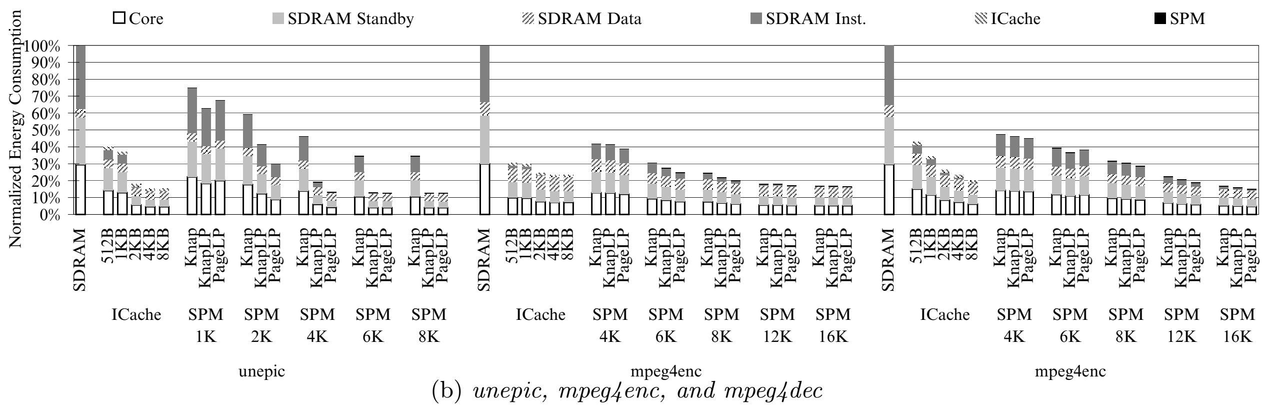


(b) *unepic, mpeg4enc, and mpeg4dec*

Figure 5: Number of memory accesses for different code placement strategies.



(a) *synthetic, fft, and epic*



(b) *unepic, mpeg4enc, and mpeg4dec*

Figure 6: Energy consumption for different code placement strategies.

Application	Executed code size	ICache size	Comparable SPM size	Exec. Time to ICache (%)	Exec. Time to Knap (%)	Total Energy to ICache (%)	Total Energy to Knap (%)	Number of SPM pinned library/user functions	Number of paged library/user functions
synthetic	12KB	2KB	6KB	64.7	97.5	62.6	97.3	11/35	0/3
fft	14KB	2KB	6KB	70.1	78.2	75.3	77.7	14/6	0/0
epic	21KB	4KB	8KB	90.5	88.3	85.1	87.3	14/15	0/5
unepic	20KB	4KB	8KB	87.6	38.2	83.4	36.8	15/41	0/6
mpeg4enc	49KB	8KB	12KB	74.9	95.9	73.7	95.7	8/45	0/15
mpeg4dec	43KB	8KB	12KB	96.1	85.2	94.8	84.7	15/16	0/12
Average (geometric mean)				79.8	77.1	78.4	76.3		

Table 3: The relative performance and energy consumption of PageLP. The instruction cache size is about 20% of the executed code size. The SPM size is chosen such that the die area occupied by the SPM is comparable to the die area of the cache.

in Table 2. The dynamic energy of the SDRAM is also computed from [19]. The parameters for off-chip bus energy per access are taken from [25].

For the system with an instruction cache, the energy formula for E is given by

$$E_{icache} = N_{icache_hit} \cdot E_{icache} + N_{icache_miss} \cdot (E_{icache} + L_{icache} \cdot E_{sram})$$

where N_{icache_hit} and N_{icache_miss} are the numbers of instruction cache hits and misses respectively, E_{icache} is the average energy consumption per access, and L_{icache} is the line size of the instruction cache in words. E_{icache} is obtained from CACTI [32].

6. EXPERIMENTAL RESULTS

Figure 5 compares the breakdown of the number of memory accesses for each application under different code placement strategies with various instruction cache and SPM sizes while providing ample external memory (SDRAM). The bar SDRAM simulates the case when all the code and data are placed in the external SDRAM. The number of total memory accesses for each bar in the figure is normalized to SDRAM. The next group of bars, ICache shows the breakdown of memory accesses when an instruction cache is used without SPM.

Among the remaining bars for each application, Knap simulates the ILP formulation for the function placement strategy without the natural loop extraction described in Section 4.1. This is the optimal static allocation strategy proposed by [3, 28]. KnapLP is for the strategy proposed in this paper where natural loops are extracted from the application, abstracted as functions, and then allocated statically with the ILP formulation (Section 4.2). Finally, PageLP is the case when the ILP formulation for demand paging in Section 4.3 is applied to the image where natural loops are outlined.

The SPM size varies from 1KB to 8KB for all applications except *mpeg4enc* and *mpeg4dec*. For *mpeg4enc* and *mpeg4dec*, the SPM size varies from 4KB to 16KB. For each application, each bar is divided into sections that represent the percent numbers of SDRAM data accesses, SDRAM instruction fetches, instruction cache accesses, SPM data accesses, and SPM instruction fetches to the total number of memory accesses in SDRAM.

The total number of memory accesses of each case in ICache and SPM is smaller than each case in SDRAM except for *fft* ICache 512B. This is due to the 4-word instruction prefetch buffer that can be found in common ARM processors. When the instruction cache is used, the prefetch buffer is not used. Thus, inexact instruction prefetches that are due to branch instructions will not occur with an instruction cache. Because of the high number of instruction cache misses in the case of *fft* ICache 512B, the total number of memory accesses is larger than that of SDRAM (note that an instruction cache miss is counted as one cache access).

With SPM, some part of the code is placed in the SPM. For instructions placed in the SPM, the instruction prefetch buffer will not be used because the SPM guarantees one cycle access. This results in a smaller number of instruction prefetches. As the size of the SPM increases, the total number of memory accesses decreases, as can bee seen in Figure 5.

For the cases with an instruction cache, we see that the number of SDRAM accesses decreases as cache size increases, except for *synthetic*. For *synthetic*, a 512B instruction cache is already enough to avoid any SDRAM instruction accesses, except for cold misses.

We see that the number of instruction fetches from the SPM increases as the size of the SPM increases. In addition, interestingly enough, some read-only data accesses in SDRAM are converted to SPM data accesses because the functions that contain the read-only data are placed in the SPM.

Among our SPM code placement strategies, PageLP has the largest SPM inst., except for *fft* and *epic*. This results PageLP consuming less energy than Knap and KnapLP. For *fft* and *epic*, KnapLP performs almost equally well. This is because our ILP solutions for KnapLP and PageLP are identical at 1KB and 2KB for *epic* and at all sizes for *fft*. Moreover, KnapLP always has a larger SPM inst. than Knap due to natural loop outlining.

The effect of demand paging with larger SPM sizes is marginal compared to smaller SPM sizes. The reason is that most of the important functions are pinned in the SPM for larger SPM sizes.

Page loading from the external memory to the SPM is performed with load and store instructions by the page manager. We see that the paging overhead is very small because the difference between SPM data in KnapLP and PageLP is negligible in the figure. If the paging overhead were large, the SPM data section in PageLP would be larger than KnapLP.

Figure 6 compares energy consumption of the processor core and the memory subsystem for each application under different code placement strategies with various sizes of instruction cache and SPM. Each bar is normalized to the value of SDRAM. In each bar, Core represents the amount of energy consumed by core. SDRAM Standby is the amount of energy caused by the standby power consumption of the external memory. SDRAM data and SDRAM inst. represent the energy consumed by SDRAM accesses for data and instruction fetches, respectively. ICache represents the amount of energy consumed due to cache accesses, and SPM represents the energy consumed by SPM accesses.

Since we multiply a constant power consumption with the execution time of each case to obtain the energy consumption of the processor core (i.e., Core), Core is proportional to the execution time. The same is true for SDRAM Standby. Thus, the height of Core or SDRAM Standby represents the execution time of each case relative to the execution time of SDRAM.

Thanks to the reduced number of memory accesses to SDRAM, PageLP is faster than Knap and KnapLP. However, when the size of the SPM is large enough that almost all instruction fetches go to the SPM (i.e., the SPM is *saturated*), Knap, KnapLP, and PageLP show comparable performance. See, for example, the 16KB case of *MPEG4-enc*.

Due to the reduced execution time with a bigger instruction cache or SPM, Core and SDRAM Standby decrease as the size of the cache or SPM increases. In addition, SDRAM inst. decreases as as the size of the cache or SPM increases.

Table 3 summarizes the performance and energy consumption of PageLP relative to ICache and Knap (the optimal static allocation strategy) when the instruction cache size is about 20% of the executed code size in the unmodified application. The SPM size is chosen such that the die area occupied by the SPM is comparable to the die area of the instruction cache of a given size. The die areas for both instruction cache and SPM are computed with CACTI [32]. Table 3 also shows the number of SPM pinned library/user functions and paged library/user functions for PageLP.

Again from Table 3, we see that our demand paging mechanism for SPM reduces the energy consumption by 21.6% and improves performance by 20.2% compared to an instruction cache occupying a comparable die area. If we compare our demand paging to the optimal static placement strategy used in [3, 28], the total energy consumption is reduced by 23.7% and performance is improved by 22.9% on average. This shows that our code placement strategy is very effective in reducing energy consumption and increasing performance. In addition, many library functions are placed in the SPM, which is impossible unless a postpass approach is used.

7. CONCLUSIONS

This paper introduced a fully automatic and dynamic code placement technique for on-chip scratchpad memory for embedded processors. Our approach is based on demand paging techniques and profiling information. We approximate our code placement technique for demand paging with a mixed ILP problem. Our technique has been implemented in a postpass optimizer that handles the whole application binary, including libraries. No application or library source code is needed. Our experimental results show that our postpass and dynamic approach can reduce energy consumption and improving performance compared to systems with an instruction cache or the optimal static code placement.

8. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-Time Algorithm for On-chip Scratchpad Memory Partitioning. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2003)*, October 2003.

[3] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Oliveri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2004)*, September 2004.

[4] ARM. *http://www.arm.com.*

[5] Oren Avissar and Rajeev Barua. An Optimal Memory Allocation Scheme for Scratchpad-Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[6] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *The tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill, 1990.

[8] Ron Cytron and Paul G. Loewner. An Automatic Overlay Generator. *IBM Journal of Research and Development*, 30(6):603–608, 1986.

[9] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *The ACM Conference on Embedded Software (EMSoft 2006)*, October 2006.

[10] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In *The 41st Design Automation Conference (DAC 2004)*, pages 238–243, June 2004.

[11] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *IEEE Computer Architecture Letters*, January 2002.

[12] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Aommercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, December 1998.

[13] http://www.xvid.org. Xvid MPEG-4 Video Codec. 2004.

[14] Andhi Janapsatya, Aleksandar Ignjatovic;, and Sri 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, New York, NY, USA, 2006. ACM Press.

[15] M. Kandemir and A. Choudhary. Compiler-Directed Scratch Pad Memory Hierarchy Design and Management. In *Annual ACM IEEE Design Automation Conference*, June 2002.

[16] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Annual ACM IEEE Design Automation Conference*, June 2001.

[17] Chunho Lee, Miograg Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[18] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design (ISLPED99)*, pages 267–269, February 1999.

[19] Inc Micron Technology. Mt48h8m16lf mobile sdram. 2003.

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

[21] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. Alto : A Link-Time Optimizer for the Compaq Alpha. *Software Practice and Experience*, 31:67–101, 2001.

[22] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[23] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design Automation and Test Conference*, March 1997.

[24] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min. Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory. In *The ACM Conference on Embedded Software (EMSoft 2004)*, September 2004.

[25] Aviral Shrivastava, Ilya Issenin, and Nikil Dutt. Compilation Techniques for Energy Reduction in Horizontally Partitioned Cache Architectures. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2005)*, September 2005.

[26] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley and Sons, Inc., 2003.

[27] 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 *Proceedings of the 15th international symposium on System Synthesis (ISSS'02)*, October 2002.

[28] Stefen Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Design, Automation and Test in Europe Conference and Exposition (DATE 2002)*, pages 409–417, February 2002.

[29] Sumesh Udayakumaran and Rajeev Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2003)*, October 2003.

[30] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Design, Automation and Test in Europe Conference and Exposition (DATE 2004)*, pages 1264–1269, February 2004.

[31] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *International Conference on Hardware/Software Codesign and System Synthesis*, September 2004.

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