A Hardware/Software Framework for Instruction and Data Scratchpad Memory Allocation

ZHONG-HO CHEN and ALVIN W. Y. SU
National Cheng-Kung University, Taiwan

Previous researches show that a scratchpad memory device consumed less energy than a cache device with the same capacity. In this article, we locate the scratchpad memory (SPM) in the top level of the memory hierarchy to reduce the energy consumption. To take the advantage of a SPM, we address two issues of utilizing a SPM. First, the program's locality should be improved. The second issue is SPM management. To tackle these two issues, we present a hardware/software framework for dynamically allocating both instructions and data in SPM. The software flow could be divided into three phases: locality improving, locality extraction, and runtime SPM management. Without modifying the original compiler and the source code, we improve the locality of a program. An optimization algorithm is proposed to extract the SPM allocations. At runtime, an SPM management program is employed. In hardware, an address translation logic (ATL) is proposed to reduce the overhead of SPM management.

The results show that the proposed framework can reduce energy delay product (EDP) by 63%, on average, when compared with the traditional cache architecture. The reduction in EDP is contributed by properly allocating both instructions and data in SPM. By allocating only instructions in SPM, the EDPs are reduced by 45%, on average. By allocating only data in SPM, the EDPs are reduced by 14%, on average.

Categories and Subject Descriptors: B.3.2 [Memory Structure]: Design Style Primary Memory; C.3 [Special-Purpose and Application-Based Systems]: Real-Time and Embedded Systems

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Memory allocation, scratchpad memory, allocation algorithm

ACM Reference Format:

Chen, Z.-H. and Su, A. W. Y. 2010. A hardware/software framework for instruction and data scratchpad memory allocation. *ACM Trans. Architect. Code Optim.* 7, 1, Article 2 (April 2010), 27 pages. DOI = 10.1145/1736065.1736067 <http://doi.acm.org/10.1145/1736065.1736067>

This work is partially supported by NSC, Taiwan under grant 98-2622-E-006-036-CC2.

Authors' addresses: Zhong-Ho Chen, Department of Computer Science and Information Engineering, National Cheng-Kung University, Taiwan; email: zhonghocheng@gmail.com; Alvin W. Y. Su, Department of Computer Science and Information Engineering, National Cheng-Kung University, Taiwan; email: alvinsu@mail.ncku.edu.tw.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1544-3566/2010/04-ART2 \$10.00
DOI 10.1145/1736065.1736067 <http://doi.acm.org/10.1145/1736065.1736067>

ACM Transactions on Architecture and Code Optimization, Vol. 7, No. 1, Article 2, Publication date: April 2010.

1. INTRODUCTION

In a computer memory subsystem, there are many devices of different speeds and capacities. In general, a fast device is smaller and more expensive, and a slow device is larger and cheaper. To construct a system with reasonable cost and performance, a layered memory hierarchy is usually used. A fast memory device is usually located in the higher layer, and it stores subsets of frequently referenced memory. The layered memory hierarchy benefits from a program's locality, which includes spatial locality and temporal locality. Spatial locality means a program usually references the memory location near the one that was just referenced. Temporal locality means a referenced memory location will be referenced again soon. In the desktop system, a cache is a generic memory component, that facilitates the layered memory hierarchy. A cache consists of a comparator, tag SRAM and another data SRAM. The comparator automatically compares the referenced address with tag SRAM to determine whether the content is located inside the data SRAM.

Because of the additional hardware, a cache consumes more energy per access than a SRAM. Therefore, devices which are more power-efficient than a cache may be required. Previous research [Banakar et al. 2002] shows a scratchpad memory (SPM) is more power efficient and smaller. It may have higher performance than a cache with the same capacity. An SPM is a SRAM storing data/instructions and managed by software. It requires neither comparator nor tag SRAM. There are two major reasons why a system with cache is more power consuming than with an SPM. First, a comparator of cache always checks the address of each memory reference. This consumes certain energy. Unlike cache, an SPM performs no runtime check, but it needs software to manage its contents. Second, a cache always relies on the locality of a program. However, a program may reference a memory location and never reference it again. In this case, a cache takes no benefit of loading a memory block from slow and power-hungry external memory.

We address two issues to efficiently utilize an SPM. First, the locality of a program should be improved. Second, SPM should be well managed. Conventionally, the first issue also exists in cache architecture, and it could be tackled by compiler techniques. The locality of a program can be classified into instruction locality and data locality. The technique to improve instruction locality includes code reposition [Pettis and Hansen 1990], and the techniques to improve data locality include data transform [Wolf and Lam 1991] and loop transform (e.g., loop permutation, tiling).

The second issue is how to manage an SPM. The most convenient approach to manage an SPM is using a static SPM allocation [Panda et al. 1997]. In such approaches, the most frequently used data are placed in the SPM and the allocation is fixed during the program execution. However, the frequently referenced memory space varies during program execution. Dynamic SPM allocation [Steinke et al. 2002b] is more efficient than static SPM allocation, since static SPM allocation is a special case of dynamic SPM allocation. In dynamic SPM allocation, the memory regions to be referenced soon will be moved into SPM during program execution. A profile-based optimization is usually used

to determine whether a memory space is going to be frequently referenced. Although many dynamic SPM allocation algorithms were proposed, there are still some disadvantages.

In this article, we propose a dynamic SPM allocation framework to reduce energy consumption of memory subsystem without modifying compiler and source codes. We improve the locality of compiled program images by code repositioning. We also propose a hardware architecture to improve the utilization of SPM. In summary, the proposed framework can reduce the EDP by 63%, on average, when compared with the traditional cache architecture.

The rest of the article is organized as follows: Section 2 reviews the related researches on code reposition and SPM management. Section 3 describes the proposed software optimization methodology. Section 4 presents the hardware architecture to reduce the SPM management overhead. Section 5 describes our evaluation environment and the results. Conclusions are given in Section 6.

2. RELATED WORK

This section describes features of early researches on locality improvement and SPM management.

2.1 Locality Improvement

The locality of a program is classified into instruction locality and data locality. The techniques to improve data locality includes data transformations [Wolf and Lam 1991] and loop transformations (e.g., loop permutation, tiling). These methodologies are compiler assisted and less related to our work.

A program's instruction locality can be improved by gathering memory blocks with high locality correlation. The major problem is how to estimate the locality correlation of two basic blocks and how close they should be placed. Hatfield and Gerald [1971] define a nearness matrix to evaluate the nearness of two procedures. Then two near procedures are placed in the same page in order to reduce the page exception. This method is also applicable to locality improvement. However, the granularity of the nearness matrix is a procedure, and it would be much better to use finer granularity. The PH algorithm [Pettis and Hansen 1990] aims at the basic block level and an ordering algorithm is proposed. The motivation of the PH algorithm is to move infrequently referenced basic blocks out of a procedure.

Hatfield and Gerald [1971] and the PH algorithm [Pettis and Hansen 1990] optimize the locality of a program using only spatial information. The temporal information is also useful for repositioning a program. Conflict miss graph [Kalamationos and Kaeli 1998] based temporal locality could eliminate the conflict miss between two procedures. Kirovski et al. [1999] also measure the nearness in time for two basic blocks based on temporal correlation. Both methods use temporal locality information. The difference is that one works at the procedure level and the other works at the basic block level.

Due to the large amount of works in this direction, it is impossible to describe all the related work. A comprehensive list will not be presented here, as our focus in this article is on SPM management.

	Instrucion	Global	Stack	Heap
Panda et al. [1997; 2000]		Static		
Sjödin et al. [1998]		Static		
Hallnor and Reinhardt [2000]	Runtime	Runtime	Runtime	Runtime
Kandemir et al. [2001]		Dynamic*		
Sjödin et al. [2001]		Static		
Avissar et al [2001; 2002]		Static	Static	
Banakar et al. [2002]	Static	Static	Static	
Steinke et al. [2002a]	Dynamic			
Steinke et al. [2002b]	Static	Static		
Brockmeyer et al. [2003]		Dynamic*		
Udayakumaran et al. [2003; 2006a; 2006b]		Dynamic	Dynamic	
Verma et al. [2004a]		Dynamic*		
Verma et al. [2004b]	Dynamic	Dynamic		
Janapsatya et al. [2004; 2006]	Dynamic			
Wehmeyer and Marwedel [2004]	Static	Static		
Angiolini et al. [2005]	Static	Static		
Dominguez et al. [2005]				Dynamic
Nguyen et al. [2005]	Static	Static	Static	
Egger et al. [2006]	Dynamic			
Pyka et al. [2007]	Runtime	Runtime		
Park et al. [2007]			Runtime	
This Work	Dynamic	Dynamic	Dynamic	

* Only array variables are handled.

Fig. 1. Research on SPM management.

2.2 SPM Management Scheme

Figure 1 shows previous researches on SPM management. We roughly classify the SPM management schemes into three categories: static allocation, dynamic allocation, and runtime allocation. The SPM allocation of static schemes is fixed during executing a program. On the other hand, the allocation of dynamic schemes varies during program execution according to the locality extracted from profile. The SPM allocation of runtime schemes also varies, but it includes runtime information. The other dimension of SPM management is the managed space. In a program, the memory space is typically divided into four areas, including instruction, global data, stack, and heap. Methods that focus on instruction area place the hottest codes in SPM. Global variables, which are frequently accessed, could be allocated on the SPM. Some applications are loop dominated, so some researches only focus on array variables. It requires additional mechanisms to handle stack variables because they may have multiple instances. Finally, it is difficult to handle heap variables because the sizes of them are only obtained in runtime.

The static allocation approaches place the most frequently referenced data in SPM and this allocation never changes at runtime. Typical static allocation approaches include Panda et al. [1997, 2000], Sjödin et al. [1998, 2001], Avissar et al. [2001, 2002], Banakar et al. [2002], Steinke et al. [2002b], and Wehmeyer and Marwedel [2004]. Panda et al. [1997, 2000] partition scalar and array variables of a program into SPM and DRAM according to the access frequencies. Besides, lifetimes of variables are analyzed in order to overlap variables in the same SPM space. Sjödin et al. [2001] propose a model describing architecture with irregular memory organization and solved the problem by the integer linear programming. Avissar et al. [2001, 2002] propose a distributed stack

applied to stack variables. Angiolini et al. [2005] also facilitate the static allocation, but they arrange the SPM memory space to most frequently accessed variables rather than gather the variables into SPM space. Nguyen et al. [2005] propose an adaptation of static allocation. They compute the allocation during the program loading rather offline. However, static allocation could not always exploit the benefit of program locality during program execution.

Software caching [Hallnor and Reinhardt 2000] is the first method to facilitate runtime SPM allocation. A hardware cache checks every referenced address to determinate whether the data is loaded and a software cache emulates this behavior. The recently used data will be moved into the cache. Therefore, the upcoming access will benefit. Unlike hardware caches, the software caches limit the ability of parallelism, and they involve a lot of overheads due to checking the address of each reference. Although this overhead could be reduced by techniques such as the dynamic compilation, [Cmelik and Keppel 1994], profile-based optimization is more efficient for battery-powered embedded systems. Park et al. [2007] also describe a runtime management scheme on stack memory. The allocation of SPM varies during program execution. It works based on the assumption that a procedure usually references stack frames close to itself. However, it is not applicable to global variables and instructions. Verma et al. [2005] and Pyka et al. [2007] also propose runtime manage strategies for multiprocess applications. An objective function is defined to find the optimal allocation under the management strategies. However, these works eliminate energy spent in context switch between multiprocess rather than energy spent in a single process.

An alternative method to utilize the SPM is based on offline optimization techniques. It is reasonable to use offline optimization technique because a battery-powered embedded system usually runs specific programs and the behavior could be analyzed in offline. The techniques of dynamic SPM allocation usually insert pieces of SPM management codes at specific points of a program. These codes would configure SPM to have a better allocation according to the profile.

One of the methods using dynamic SPM allocation is array-based. Kandemir et al. [2001] adopt loop transformations and data transformations to place a data tile in SPM. They also use Presburger formulas to model the behavior of array variables in a loop. The allocation of SPM changes with the subscript of an array. Brockmeyer et al. [2003] perform the layer assignments by analyzing the lifetime of an array. Verma et al. [2004a] proposed an SPM architecture, which is similar to a loop cache. Array-based analysis method is useful for the applications with huge computation in loops, but it is not general. The other researches focus on dynamic allocation of instructions. The reference frequency of instructions is usually higher than that of data, and they are more deterministic. Steinke et al. [2002a] dynamically copy basic blocks with high energy consumption into SPM. Egger et al. [2006] classify the code segment into *SPM*, *Ext*, and *Paged*. A SPM page is statically placed in SPM because it is referenced most frequently. A Paged page is dynamically copied from external memory at runtime. Ext page is always placed in external memory, because it is unworthy to move it into SPM. Janapsatya et al. [2004, 2006] dynamically

Fig. 2. Profiled-driven optimization methodology.

manage instruction SPMs according to a concomitance table. Concomitance is a metric to estimate the temporal relation of two basic blocks.

The researches on dynamic allocation of data variables include Udayakumaran et al. [2003, 2006a, 2006b], Francesco et al [2004], Verma et al. [2004b], and Dominguez et al. [2005]. Udayakumaran et al. [2003, 2006a, 2006b] propose a compiler-decided dynamic SPM management for global and stack variables. Francesco et al. [2004] propose a modified memory management unit (MMU) to support page-level dynamic SPM management. Verma et al. [2004b] solve the problem of dynamic SPM allocation using ILP formulation techniques. Dominguez et al. [2005] propose a dynamic SPM management for heap variables.

Although the motivations of various SPM allocation techniques are similar, they could have different granularities. The granularity of the SPM allocation could be a variable, an object or a page. Variable-based and object-based allocations benefit in most programs, but it may struggle for recursive programs and may generate memory fragmentation problems due to nonuniform sizes of variables/objects. Page-based memory allocation has the advantage of hardware MMU and there are no memory fragmentation problems. However, it benefits less from a program with small spatial locality. Furthermore, variable-based or object-based methods always modify the addressing mode. It makes system integration difficult. For example, traditional C compilers use stack pointer or frame pointer to address a stack variable, but it is allocated to an absolute SPM address in variable-based methods.

In this article, we propose an integrated framework for battery-powered embedded processor with one instruction SPM and one data SPM. A profile-based optimization algorithm derives the dynamic allocation of SPM. This framework includes software development flow and hardware architecture, and there is no necessity to modify the original compiler and source code.

3. SOFTWARE OPTIMIZATION METHOD

In this work, we adopt the profile-driven optimization methodology, shown in Figure 2. A full-system simulator executes the original program and records all the memory references. Consequently, our optimization algorithm is

Fig. 3. Basic block repositioning.

responsible for repositioning the program image and for extracting the SPM allocation from the trace file. In runtime, an SPM management software is responsible for moving the hottest code/data into SPM according to the allocation.

There are two phases of our optimization algorithm. The first phase is the basic block repositioning, which gathers frequently referenced basic blocks. In other words, it improves the locality of a program. The second phase is the locality extraction. The extracted locality is also the SPM allocation. The basic block repositioning is described in Section 3.1, and locality extraction is described in Section 3.2.

3.1 Basic Block Repositioning

Figure 3 shows the entire flow of the basic block repositioning. Before starting our algorithm, the control flow graph (CFG) of program image should be constructed. There are two stages of CFG construction. One is the basic block identification and the other is the edge detection. A basic block is a piece of code, which has one entry point and one exit point. In addition, a basic block is a continuous code sequence with the maximal extend. The basic block detection procedure scans entire code to detect the boundaries of basic blocks. Exit points of a basic block include: branch, return, call, long jump, indirect jump, and software interrupt. Entry points of a basic block include branch targets, successive instructions of a basic block, entry of a function, and jump targets. The second stage is detecting edges. An edge is the transition from one basic block to another basic block. A basic block is terminated with a return or indirect jump may have more than one successor edge. In addition, there are some nondeterministic edges. The input of our optimization algorithm is the sequence of all memory references generated by a full system simulator, including code, data, and ISR. The CFG may not include the ISR, because ISR is infrequently executed. Therefore, the transition edge to/from an interrupt is filtered in this work.

After constructing the CFG, we would apply the basic block ordering algorithm. It improves the locality of a program. The proposed basic block ordering algorithm is similar to the well-known PH ordering algorithm [Pettis and Hansen 1990]. It is easy to implement for compiled program images and still gets fine performance.

There are some differences between the PH algorithm and ours. The major difference is the ordering scope of basic blocks. In PH algorithm, there are two

Fig. 4. The original control flow graph, the order, and the repositioned image.

position steps, procedure positioning and basic block positioning. However, it is unlikely to place two basic blocks of different procedures in the neighbor. We apply the ordering algorithm to the basic blocks of an entire program image rather than of a procedure. By placing two basic blocks as neighbors, our algorithm tightens the hottest code. Figure 4 shows an example of CFG, derived ordering and reordered basic blocks. The ordering is derived by the PH algorithm.

After the order of basic blocks is derived by the modified PH algorithm, we insert some stubs. Stubs are jump instructions and ensure the semantics of the original program and of the reordered program to be equal. There are two overheads of executing stubs: (i) branch delays and (ii) instruction fetching. Because the branches are always taken, it is easy to eliminate the first overhead by a general hardware branch prediction unit. The instruction fetching delays could be reduced by special hardware prediction unit, such as LA-PC branch prediction [Chen and Baer 1995], but it is possible to mispredict. Therefore, we should carefully insert stubs. We identify some points to insert the stubs: (i) between two discontinuous basic blocks and (ii) after a conditional branch.

The first point is straightforward. After executing the first basic block, it should jump to the successive basic block. The second stubs are also necessary because the new order may make the branch address exceed the relative jump range. For a conditional branch basic block, Figure 5 shows five cases of how to insert stubs.

Cases (a) and (b) are the best cases that short jump is sufficient and no stub is inserted. In these two cases, one of the successor and branch target is right behind the basic block and the other is within the range of branch offset. In Case (c) and (d), one of the successor and the branch target are the next basic block and the other is out of branch offset. We should insert one stub. Case (e) is the worst case; both the successor and branch target are not close to the basic block, and we should insert two stubs. If the next basic block of the branch is the branch target rather than the successor, we should invert the branch condition.

Fig. 5. Stubs insertion for branch. Doted line blocks are within the range of branch offset.

After inserting stubs, the address of each basic block is recalculated. The last step of code repositioning is to relocate the program image. For the code segment, we recalculate the branch offset and long jump addresses. However, it is not trivial for relocating data objects because the content of a data object may be the address of an indirect jump. It is difficult to identify whether a data object requires relocation and the decompilation technique is used. To simplify our work, we use the debug information to identify data objects, which require relocation. The compiler generates a symbol table and embeds it in a program image. It contains the type of a data object and type information. A jump table, usually used in switch/case statements of C language, is also a data object which requires relocation, but there is no debugging information about it. We also identified the base address and range of a jump table.

3.2 Locality Extraction

After repositioning the basic blocks, the next problem is how to extract the locality of a program and how to represent the locality. The extracted locality will be used in SPM allocation. There are two considerations of locality of a program: When the locality is changed and where is the most frequently referenced memory space.

To answer the first question, we use the term “program point” to represent when the locality of a program is likely to be changed. Previous researches suggest some candidate program points: beginning of a function, before a loop, and end of a loop. We used more general criteria, the beginning of all basic blocks. The more candidate program points, the better locality is represented. The second question is much easier. We represent the most frequently referenced memory space as a set of fixed-size memory blocks. It is also possible to implement as variable size memory blocks but memory fragmentation is the price to pay. The choice of the block size is critical, and it is usually application dependent. Although it is more precise to represent the locality in smaller block size, the total size of locality information may be too large. In summary, we represent the locality of a program as an n by m matrix L , where n is the number of basic blocks and m is the number of memory blocks.

Fig. 6. Control flow graph and cost model.

L is a (0-1)-matrix, that is, a matrix each of whose elements, denoted by l_{ij} , is either one or zero. One means the memory block is frequently referenced in corresponding basic block. In this work, a frequently referenced memory block, that is, $l_{ij} = 1$, is placed in SPM and an infrequently reference memory block, that is, $l_{ij} = 0$, is placed in external memory. A row of the L matrix, denoted by l_i , is the locality of the basic block. A cost model could be derived from the CFG of a program and a given locality matrix:

$$Cost = C_1 \sum_{i=0}^n \sum_{j=0}^m (1 - l_{ij}) r_{ij} + C_2 \sum_{i,j} e_{ij} D(l_i, l_j), r_{ij} \in R, e_{ij} \in E, l_{ij} \in L.$$

The cost here is the clock cycles spent on memory subsystem excluding instruction execution cycles. R is the reference matrix and each element, r_{ij} , indicates the reference count of a memory block in the basic block, and it could be obtained from the trace file. The first summation, $\sum_{i=0}^n \sum_{j=0}^m (1 - l_{ij}) r_{ij}$, is the reference count of infrequently referenced memory blocks and C_1 is the difference in clock cycles per reference between SPM and external memory. The second summation, $\sum_{i,j} e_{ij} D(l_i, l_j)$, is the count of swaps. C_2 is the number of cycles to swap a memory block between SPM and external memory. E is the adjacency matrix of the CFG of the program and each element indicates the count of control transfers between two basic blocks. $D(l_i, l_j)$ is the locality difference matrix between two basic blocks, and it is represented as:

$$D(l_i, l_j) = \sum_{k=0}^m |l_{ik} - l_{jk}|.$$

Figure 6 illustrates an example of the cost model. In this example, we use a 1,024KB page as the memory blocks, and there are three memory pages and eight basic blocks. Therefore, R and L are 8×3 matrices, and E is an

```

0: Scratchpad Allocation Algorithm
1: Construct CFG,  $E$  and  $R$ 
2: Set  $L$  all zeros
3: for nodes  $u, v$  in descending order of the connected edges weight
4:      $do\_local\_opt(u, v)$ 
5:     if  $l_u = l_v$ 
6:          $merge(u, v)$ 
7:         goto line 3
8:     endif
9: endfor

```

Fig. 7. SPM allocation algorithm.

8×8 matrix. Basic block 0 is a virtual basic block, which neither exists in the original program image nor references any memory. R and E are obtained from the CFG and trace file. L will be derived from the algorithm described in following section. We assume C_1 and C_2 are 14 and 1,396, respectively, and the cost of the control flow graph is 20,944.

Since the cost model is defined, the problem of locality extraction is to find L , which generates the minimum cost. One of the solutions of L is the matrix with all 1s, but it is a trivial solution. Since the frequently referenced memory blocks should be placed in SPM, there is a constraint for the solution of L : The size of memory blocks within SPM must be less or equal to the size of SPM. In other words, the number of elements in a row with value one should be less or equal to SPM size divided by the size of memory block, denoted by S . Under such a constraint, L with the minimum cost is also the optimized SPM allocation. Note that the problem cannot be solved by ILP because of the nonlinear definition of $D(l_i, l_j)$. Since the size of solution space is $(\sum_{k=0}^S \binom{m}{k})^n$, it is difficult to find an optimal solution. A heuristic algorithm is proposed to find the allocation. In short, our SPM allocation algorithm is a procedure of merging two CFG nodes with the similar locality. It considers the CFG as its input and generates the optimized locality matrix. Because references to codes and data are in different memory spaces, we construct two CFGs, one is the code graph and the other is the data graph. These two graphs have the same edge matrix but the different reference matrix. The algorithm deals with these two graphs and gets two locality matrices.

Figure 7 shows the proposed SPM allocation algorithm. In the beginning, the elements of the locality matrix are initialized as zeros. The algorithm traverses each edge and finds the local optimal SPM allocation of nodes connected by the edge. The traversal is in descending order of edge weight, because the nodes with heavy transitions are very critical and are likely to be merged. The condition to merge two nodes is that they have similar locality. Nodes with the similar locality should get the same SPM allocation and the program points between these nodes are redundant. The subfunction *merge* accumulates the count of referenced memory blocks of these two nodes and redirects the edges associated with these two nodes. A merge operation implies the elimination of the cost of the swapping memory blocks. Because merge operation changes the

Fig. 8. Concept of local optimization.

weight of the edges, our algorithm restarts to the edge with the largest count after the merge. The worst case of the algorithm is merging two nodes with least count every time, so the complexity of the traversal is $O(e^2)$.

Figure 8 illustrates the concept of local optimization of the nodes u and v . The local optimization algorithm decides no SPM allocation except for these two nodes. There are two candidates of SPM allocations for a node. The first is the referenced memory blocks in the node. The other is the memory blocks propagated from the allocation of neighborhood nodes. At the end of the local optimization, the algorithm would derive the allocation of these two nodes according to the referenced memory blocks and the allocation of the neighborhood nodes. To vote the best SPM allocation, we associate a gain with each memory block. The gain is the cost difference between the cases with and without the memory block in the SPM allocation. The gain of an individual node could be derived from the defined cost model:

$$G_u(m) = \text{Cost}(l_{um} = 0) - \text{Cost}(l_{um} = 1) = C_1 r_{um} + C_2 \sum_{x \neq v} (e_{ux} + e_{xu}) (-1)^{(1-l_{xm})}.$$

We skip the details of derivation, but we briefly explain the earlier equation. The first term of the gain is easy to understand. In case that a memory block exists in the allocation of the node u , the reference of this memory yields no cost, and thus the gain is $C_1 r_{um}$. The second term is a little complicated, and it depends on the allocation of neighborhood nodes. In case that the memory block m exists in the allocation of neighborhood node x but doesn't exist in current node u , then there is cost of swapping the memory blocks. Therefore, the gain of the memory block m in node u is $C_2(e_{ux} + e_{xu})$. In the opposite case, the gain is $-C_2(e_{ux} + e_{xu})$. The edge between u and v is ignored because we don't know the exact allocation of node v . Next, we derive the coupled-gain $G_{uv}(x, y)$ as:

$$G_{uv}(x, y) = \begin{cases} G_u(x) + G_v(y) & x = y \\ G_u(x) + G_v(y) - 2C_2(e_{uv} + e_{vu}) & x \neq y \end{cases}.$$

The coupled gain is the performance cost reduced with the memory block x in the allocation of the node u and the memory block y in the allocation of the node v . In case that these two memory blocks are identical, the gain is simply

```

0: do-local-opt(u,v)
1:   calculate  $G_u(m)$  and  $G_v(m)$ 
2:   calculate  $G_{uv}(x,y)$ 
3:   for  $i = 0 \rightarrow S-1$ 
4:     pickup maximum  $G_{uv}(x,y)$ ,  $l_{ux} \neq 1$  AND  $l_{vy} \neq 1$ 
5:     if  $G_{uv}(x,y) > 0$ 
6:       Set  $l_{ux}$  and  $l_{vy}$ 
7:       if  $x \neq y$ 
8:          $G_{uv}(y,x) = G_{uv}(y,x) + 2C_2(e_{uv} + e_{vu})$ 
9:       endif
10:    endfor

```

Fig. 9. Local optimization algorithm.

Fig. 10. Local optimization algorithm.

accumulated. Otherwise, the swapping cost should be subtracted from the gain. Figure 9 describes the local optimization algorithm.

The first two lines of the algorithm calculate the gains defined in this section. The iteration between lines 3 and 10 allocates a couple of memory blocks in two nodes, and it is repeated S times. Only those unallocated memory blocks with maximum and positive $G_{uv}(x,y)$ would be allocated. After these memory blocks are allocated, $G_{uv}(y,x)$ is recalculated because the partial allocation of u and v is known now.

Figure 10 shows an example of the optimization process. Since the edges with the maximum weights are between nodes b_1 and b_2 , the local optimization is applied on them. Because l_0 and l_1 are identical, these two nodes are merged. Figure 10(a) shows the result after merging b_1 and b_2 . The merging process includes merging referenced pages and redirecting edges connected with these two nodes. Figure 10(b) shows the results after the second merge, b_4 and b_5 . Subsequently, b_6 and b_7 are merged and finally b_3 are merged with b_4 and b_5 as shown in Figure 10(c).

Fig. 11. (a) Optimized instruction CFG, (b) optimized data cfg, (c) the region table, (d) the allocation table, and (e) the entry table.

3.3 Region Table

After applying the proposed algorithm, we would get two optimized CFGs and regions. Figure 11(a) and 11(b) show an example of optimized CFG. All basic blocks inside a region have the identical instruction and data allocations. Each region consists of a text allocation, a data allocation, and entries. The regions are represented by the region tables, as shown in Figure 11(c). The instruction and data allocation indexes point to the instruction and data allocation table, respectively. The allocation table consists of multiple rows and each row indicates a specific SPM allocation. An SPM allocation is a set of memory pages that will be placed in the SPM. Figure 11(d) is the instruction allocation table with the two pages. Entries of a region are the targets of edges whose sources are the other regions. In Figure 11(b), the entries of region 2 are b_4 and b_5 . The entry table collects all entries of all regions. In region table, an entry index and the number of entries are used to identify the entries of a region.

4. MEMORY ARCHITECTURE AND RUNTIME SYSTEM

4.1 Memory Architecture

Figure 12 shows the architecture of the proposed memory subsystem. There are five memory devices, instruction scratchpad memory (ISPM), data scratchpad memory (DSPM), instruction cache (IC), data cache (DC), and DRAM, and they are all located in the same physical memory space. IC and DC can efficiently reduce the energy consumption of DRAM. Both caches are direct mapped and the capacities are 1KB. The CPU core could simultaneously issue an instruction in ISPM and a data in DSPM in one cycle. The virtual address is translated

Fig. 12. Proposed architecture with two SPMs.

into the physical address by instruction and data memory management unit (IMMU and DMMU). Translation look-aside buffers (TLBs) are implemented in IMMU and DMMU to decrease the translation latency and table look-up logic is designed to reduce the TLB miss penalty. The address translation logic (ATL) acts as the hardware accelerator for the SPM management. The detail functions of the ATL are described in Section 4.3.

4.2 SPM Management Runtime System

Before executing a program, the entire program image is loaded in DRAM, and the instructions of all entry points are modified as software interrupts (SWIs), each associated with a region number as the parameter. When the program hits a software interrupt, an SPM management interrupt service routine (SPMISR) is responsible for forcing the ATL to swap regions. Used registers should be stored and subsequently restored. The overhead of the SPMISR includes the cycles to execute it and to flush instructions located in the cache. If SPMISR is frequently executed, the system performance is reduced. Therefore, it is important to write an efficient SPMISR code. In this work, the SPMISR code size is around 120 bytes.

4.3 Address Translation Logic

After the SPMISR requests a region swapping, the ATL takes over the follow-up tasks. There are five steps in the ATL task flow, as shown in Figure 13. The first task is to access the requested entry of the region table. The register RTAB is used to point to the region table in the SDRAM.

Then, the ATL checks whether the requested and current SPM allocations are identical. If the SPM allocations are different, the ATL is responsible for copying pages into ISPM and modifies the corresponding page table entries. However, the ATL performs these operations indirectly. The ATL sends a command to the DMA, which then performs the copy operations. Consequently, the ATL writes to the corresponding TLBs to modify the page table. Pages not in the allocation are swapped out of the ISPM. A dirty bit is implemented to avoid swapping a clean page. Figure 14 shows an example to swap pages among three SPM allocations. Let $\{p_0, p_1\}$ be in the ISPM, and the allocation is configured

Fig. 13. The ATL task flow.

Fig. 14. SPM allocation table and example of swapping pages.

as $\{p_0, p_2\}$. The ATL replaces p_2 with p_1 . The physical address of a page in SPM may be different even with the same allocation and the ATL calculates the physical addresses. Moreover, inconsistency may happen when swapping a page in the cache into the SPM. The page should be invalidated before swapped into the SPM.

After configuring the allocation, the ATL removes the software interrupts for the requested region and inserts the software interrupts for the current region. To remove a software interrupt requires restoring the original instruction which is stored in the entry table. The ETAB register is used to point to the entry table, which stores all entries.

When these tasks are finished, the SPMISR returns to the instruction, which hits the software interrupt. The program continues its execution until hitting another software interrupt.

5. RESULTS

5.1 Simulation Environment

Our simulation environment [Chen 2009] is a full system simulator written in SystemC. The processor accesses the memory through two memory interfaces,

Table I. (a) Energy and Delay of SDRAM and Caches, (b) Energy Consumption of SPMs, (c) Static Power Consumption of CPU Core and SDRAM, and (d) ATL Dynamic Power.

SDRAM			Cache			
Access Type	Delay (cpu cycles)	Energy (nj)	Size (KB)	Associative	Block Size (Byte)	Energy (nj)
Random Read	16	10.352	1	1	32	0.197
Burst Read (32Byte)	30	17.655	4	4	32	0.551
Random write	20	10.370	8	4	32	0.564
Burst Write (32Byte)	34	16.218	16	4	32	0.605

(a)

SPM		
Size (KB)	Bit-width (bit)	Energy (nj)
4	32	0.145
8	32	0.160
16	32	0.190

(b)

Static Power	
Model	Static Power
AT91SAM9261	0.66mW
MT48H4M32LF	6.5mW

(c)

ATL	
Swapping Energy (nj)	1,620

(d)

one for instruction fetching and the other for data load/store. There is a system bus connecting an external memory (SDRAM) and a Flash. There are four general bus operations that can be performed, including single read/write and burst read/write. The bus contention is modeled and the first-in-first-out arbitration is applied. It is important to model burst operations, because they are more power-efficient than the single random access operations.

5.2 Energy and Delay Model

In our experiments, the energy consumption of the memory subsystem is estimated. The CPU clock rate is set to 200MHz and the SDRAM runs at 100MHz. The energy model for our memory subsystem is:

$$E_{mem} = (E_{ispm} + E_{ic}) + (E_{dspm} + E_{dc}) + E_{dram} + E_{static}.$$

The energy consumption of memory subsystem could be divided into four classes. The first three classes are dynamic power, and the last one is the static power. The instruction energy is due to ISPM and IC and the data energy is due to DSPM and DC. The energy consumption of SPM and caches is calculated based on the number of reads/writes. There are four types of DRAM accesses, read, write, burst read, and burst write. The width of DRAM is 32-bit and the burst length is 8. The cache used in our experiment is 4-way associative. Table I shows the energy and delay models used in the simulation. The energy consumption of SDRAM is calculated for MT48H4M32LF-75 using system power calculator [Micron Technique, Inc 2009]. The energy consumption of caches is calculated for 0.13 μ m process technique using CACTI [Wilton and Jouppi 1996]. The static power of the CPU core is estimated with ARM9-based processor AT91SAM9261. The energy consumption of the proposed ATL is estimated by synopsys designvision with TSMC 0.13 μ m cell library.

The delay of an application is modeled by the sum of instruction count, fetch delay, load/store delay, and idle time as:

$$D_{ap} = N_{inst} \times CPI_{core} + D_{fetch} + D_{ls}.$$

N_{inst} is the total number of executed instructions of an application. CPI_{core} is the cycles per instruction without the memory access delay, and it depends on the micro-architecture, pipeline, superscalar, branch prediction, and so on. Since this research focuses only on the memory subsystem, simplification on processor timing models is made. In this article, it is assumed that CPI_{core} is always one. D_{fetch} is the number of cycles for instruction fetching and D_{ls} is the number of cycles for data loading/storing.

Ten benchmarks are used to evaluate our work, including *adpcm*, *blowfish*, *cjpeg* (*jpeg encoder*), *crc*, *dijkstra*, *djpeg* (*jpeg decoder*), *fft*, *stringsearch*, *sha*, and *susan* (*susan smoothing*). All source codes come from MiBench [Guthaus et al. 2001]. The compiler was modified from lcc [Fraser 1991], a retargetable compiler for ANSI C. The operation system is uC/OS-II with a FAT32 file system based on the Flash memory. The virtual memory system is implemented as a two-level page table with ATL described in Section 4, and the page size is the 1K bytes. Figure 15 shows the characteristics of each benchmark. For most benchmarks, most energy is consumed in the IC and DC. The proposed SPM allocation would like to reduce it by replacing these two devices with ISPM and DSPM. The code size of *dijkstra* is quite small, but a lot of data are required. Because there is no floating point logic in our CPU, the benchmark, *fft*, will consume a lot of cycles in fetching instructions. The bottleneck of *susan* is also the instruction fetching.

5.3 Dynamic SPM Allocation for Instruction

The first experiment evaluates the proposed dynamic SPM allocation for instructions. Three memory capacities are tested, including 4KB, 8KB, and 16KB. Table II shows six configurations that are evaluated in this experiment. For comparison, a 16K, 4-way associative cache is adopted as the data memory of all configurations. The configuration *Cache* uses the traditional cache architecture as the instruction memory device. The configurations, *Cache+PH* and *Cache+MPH*, also adopt the traditional cache architecture, but the codes are optimized with the PH and the modified PH (MPH) reposition algorithm, respectively. The configuration *SPM* adopts the proposed SPM architecture. Similarly, the configurations, *SPM+PH* and *SPM+MPH*, adopt the proposed SPM architecture and the codes are optimized with the PH and modified PH reposition algorithm, respectively.

In CMOS VLSI systems, the supply voltage could be scaled. Increasing the voltage also increases the energy but decreases the delay of the system. Here, we use the energy delay product (EDP) as the metric. The EDPs are normalized with the configuration *Cache* with the same memory capacity. Figure 16 shows the normalized EDPs reduction of all benchmarks. The result shows the proposed method, *SPM+MPH*, could reduce the EDP by 35% to 79%, averaging

(a) Energy

(b) Delay

Fig. 15. Energy and delay of each benchmark using 4KB 4-way associative IC and DC.

Table II. Configurations in the Experiment of Dynamic SPM Allocation for Instruction.

Configuration	Instruction Memory	Data Memory	Reposition Algorithm
Cache			None
Cache+PH	4/816KB Cache (4-way)		PH
Cache+MPH			MPH
SPM		16KB Cache (4-way)	None
SPM+PH	4/8/16KB SPM,		PH
SPM+MPH	1KB Cache (1-way)		MPH

45%. The reduction is contributed from the code repositioning and the proposed SPM allocation.

Specifically, the major EDP reduction is achieved by the proposed SPM allocation. In the benchmark *fft* with 4KB capacity, the EDP is reduced by 79%. Although the proposed SPM allocation may reduce the EDP, the performance may be unacceptable if program locality is not improved at the same time. By

Fig. 16. EDP reduction of each benchmark with different memory architectures.

taking the benchmark *stringsearch* with 4KB capacity as an example, the EDP of *SPM* is even higher than the traditional cache architecture. After code repositioning using PH or MPH, the EDP is efficiently reduced by 57%. When the memory capacity increases, the reduction of EDPs seems to saturate. In such case, nearly all instructions are fetched from IC or ISPM because the code size is close to the memory capacity.

Because of the dynamic SPM allocation, traffic between SPM and SDRAM is produced. Table III shows the SDRAM traffic contributed from IC and ISPM. Typically, increasing the size of SPM would reduce the traffic between SPM and SDRAM. In *adpcm*, the traffic is 2290KB when we have 4KB ISPM. It is reduced to 257KB and 26KB when we have 8KB and 16KB ISPM, respectively.

Table III. SDRAM Traffic Contributed from IC and ISPM.

Benchmark	4KB		8KB		16KB	
	IC (KB)	ISPM (KB)	IC (KB)	ISPM (KB)	IC (KB)	ISPM (KB)
adpcm	2,062.1	2,290.0	85.2	257.0	13.5	26.0
blowfish	1,221.1	2,291.0	34.3	28.0	10.4	24.0
cjpeg	450.0	2,288.0	203.8	1,237.0	145.1	861.0
crc	23.0	35.0	13.4	23.0	1.4	16.0
dijkstra	38.6	614.0	1.7	8.0	0.4	10.0
djpeg	527.0	3,061.0	152.3	581.0	109.9	154.0
fft	238,627.0	44.0	41,026.0	48.0	2,157.2	56.0
sha	419.6	46.0	17.2	20.0	3.8	16.0
stringsearch	2.8	13.0	1.4	8.0	0.4	9.0
susan	25,009.4	6,717.0	6,972.0	6,748.0	198.6	135.0

Fig. 17. EDP reduction of Egger's and the proposed SPM allocation.

With larger SPM, we have fewer probabilities of swapping out pages that would be referenced in near future. In *fft*, however, the traffic increases when ISPM capacity becomes larger. It is because more memory pages are placed in larger SPM. For those applications that reference vast memory space that cannot be all placed in SPM, the SDRAM traffic from ISPM may be less but the traffic from IC would be much. Placing more pages in SPM would reduce the references to the IC and also reduce the block conflicts. Though the traffic from SPM might increase, the overall traffic is reduced.

The comparison of the proposed SPM allocation with the work of Egger et al. [2006] is depicted in Figure 17. In this experiment, the size of ISPM is 4KB and the size of IC is 1KB. Table IV details the delay and energy consumed in each benchmark. The energy and delay are normalized with respect to the cache architecture. In most benchmarks, the proposed method outperforms Egger's method. The proposed method achieves uniform performance while some results of Egger's method are unsatisfactory. In *blowfish*, *fft*, and *susan*, the EDPs are even higher than the cache architecture. One possible reason is that Egger et al. [2006] adopt some empirical factors, such as the cache miss ratio and the average number of page misses. These factors are used in

Table IV. Comparison Between Egger's Method and the Proposed SPM Allocation.

Benchmark	Energy (%)		Delay(%)	
	Egger's	Proposed	Egger's	Proposed
adpcm	72	59	105	102
blowfish	102	57	154	95
cjpeg	63	63	100	102
crc	58	55	102	100
dijkstra	57	52	103	101
djpeg	76	68	96	96
fft	273	41	101	51
stringsearch	50	50	87	87
sha	61	55	100	101
susan	164	55	96	77

code classification and reposition. We observe that the practical miss ratio is between 0.108 and 0.001, while Egger et al. use 0.002 as their miss ratio. The other reason is the page size. When page conflicts happen, it is required to replace a page in SPM. The larger page size causes more page conflicts. By reducing page size, the page conflicts are reduced, but it also increases the size of page table and the power consumption of the virtual memory system. On the other hand, the proposed SPM considers the overhead of page conflicts, which only happens when changing regions.

5.4 PEfficiency of Code Repositioning

Figure 18 shows the results corresponding to the code repositioning. Figure 18(a) is the normalized reduction of fetching cycles after code repositioning. The result shows the repositioning can efficiently reduce the instruction fetching cycles. However, the repositioning may increase the instruction count that decreases overall EDP. Figure 18(b) shows the percentages of increased instruction counts. By eliminating jump instructions between two concatenated basic blocks, the instruction count may decrease, such as *fft*. The modified PH algorithm can reduce more instruction fetching delay, but it also increases more instruction counts.

Depending on the bottleneck of a program, repositioning may benefit or be harmful to the overall EDP. When a program, such as *fft*, *string search*, and *susan*, spends much energy and time on fetching instructions, the repositioning can efficiently reduce the EDP. In *fft* with 4KB memory capacity, the EDP of configuration *Cache+PH* is even reduced by 42%. In other cases, the EDPs may increase after repositioning. For example, 4KB instruction memory is enough for benchmarks, *adpcm*, *cjpeg*, *dijkstra*, and *sha*.

Moreover, the efficiencies of different repositioning algorithms are dependent on the memory architecture. In *susan* with 4KB capacity, the EDP of the *Cache+PH* is lower than the EDP of the *Cache+MPH*. However, the EDP of the *SPM+PH* is higher than the EDP of the *SPM+MPH*. In short, the performance of the modified repositioning algorithm matches quite well with the proposed SPM architecture.

Fig. 18. Increase of instruction count and decrease of fetch cycles after repositioning.

5.5 Dynamic SPM Allocation for Data

In this section, we evaluate the proposed SPM allocation for data. For comparison, a 16K, 4-way associative cache is adopted as the instruction memory. All program images are also repositioned by the modified PH algorithm. The EDPs are normalized with respect to the traditional cache architecture with the same memory capacity. Figure 19 shows the normalized reduction of EDPs. The EDPs are reduced by 14%, on average.

In general, the proposed allocation could reduce the EDP by around 25%. In *blowfish*, *cjpeg*, and *djpeg*, unfortunately, the EDPs are even higher than those of the cache architecture. The reason is: The proposed SPM allocation assumes that a basic block only references certain memory spaces. So, the SPM allocation for a basic block is invariant. In these benchmarks, some basic blocks usually reference vast memory spaces. For example, when *cjpeg* and *djpeg* process a large image and it is impossible to allocation entire image data into the SPM. Therefore, the proposed SPM allocation is less suited for loop-dominated applications.

Fig. 19. The normalized EDPs reduction of the proposed SPM allocation for data memory.

Fig. 20. The normalized EDP reduction of the proposed SPM allocation for both instruction and data.

5.6 Dynamic SPM Allocation for Both Instruction and Data

In this section, we evaluate the proposed SPM allocation for both instruction and data. In this experiment, all program images are also repositioned by the modified PH algorithm. The EDPs are normalized with respect to the traditional cache architecture with the same memory capacity. Figure 20 shows the normalized reduction of EDPs. By integrating SPM management for both instructions and data, the proposed method is quite effective. All EDPs of the proposed SPM allocation are lower than those of the traditional cache architecture. On average, the EDP is reduced by 63%. Especially in *fft* with 4KB capacity, the EDP is further reduced by 90%.

6. CONCLUSION AND FUTURE WORK

This article presents a hardware/software framework for dynamically allocating both instructions and data in SPM. The software flow could be divided into three phases, locality improvement, locality extraction, and runtime SPM management. The optimization is done in binary code level. Without modifying the original compiler and source code, the locality of a program is improved. An optimization algorithm is proposed to extract the SPM allocations. In runtime, an SPM manager is responsible for managing the SPM. In the hardware side, we propose the address translation logic (ATL) to reduce the overhead of SPM management. In summary, the proposed framework can reduce the EDP by 63%, on average, when compared to the traditional cache architecture. The reduction is contributed from allocating both instruction and data in SPM.

By allocating only instructions in SPM, the EDPs are reduced by 45%, on average. These reductions are contributed from the code repositioning and the proposed SPM allocation. Depending on the bottleneck of a program, repositioning may benefit or be harmful to the overall EDP. When a program spends a lot of energy and time on fetching instructions, the repositioning can efficiently reduce the EDP. Moreover, a program is repositioned based on only spatial information, and it would be more efficient if the temporal information are included. Compared to the work of Egger et al. [2006], the proposed method gives more stable performance because all EDPs are less than the traditional cache architecture.

In general, the proposed allocation could reduce the EDP by around 25%. Unfortunately, the proposed SPM allocation is less suited for loop-dominated applications. It may require compiler-assist or programmer-assist methods to improve the SPM allocation.

REFERENCES

- ANGIOLINI, F., BENINI, L., AND CAPRARA, A. 2005. An efficient profile-based algorithm for scratchpad memory partitioning. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 24, 1660–1676.
- AVISSAR, O., BARUA, R., AND STEWART, D. 2001. Heterogeneous memory management for embedded systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, New York, 34–43.
- AVISSAR, O., BARUA, R., AND STEWART, D. 2002. An optimal memory allocation scheme for scratchpad-based embedded systems. *ACM Trans. Embed. Comput. Syst.* 1, 1, 6–26.
- BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*. ACM, New York, 73–78.
- BROCKMEYER, E., MIRANDA, M., CORPORAAL, H., AND CATTHOOR, F. 2003. Layer assignment techniques for low energy in multi-layered memory organizations. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, Los Alamitos, CA.
- CHEN, T.-F. AND BAER, J.-L. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5, 609–623.
- CHEN, Z.-H. 2009. NCKU SPM Simulator. <http://code.google.com/p/nckuspsimulator/> In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, ACM, 128–137.
- DOMINGUEZ, A., UDAYAKUMARAN, S., AND BARUA, R. 2005. Heap data allocation to scratch-pad memory in embedded systems. *J. Embed. Comput.* 1, 4, 521–540.
- EGGER, B., LEE, J., AND SHIN, H. 2006. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th International Conference on Embedded Software*. ACM, New York, 321–330.

- FRANCESCO, P., MARCHAL, P., ATIENZA, D., BENINI, L., CATTHOOR, F., MENDIAS, J. M. 2004. An integrated hardware/software approach for runtime scratchpad management. In *Proceedings of the 41st Annual Conference on Design Automation*. ACM, New York, 238–243.
- FRASER, C. W. 1991. A retargetable compiler for ANSI C. *SIGPLAN Notice*, 26, 29–43.
- GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*. IEEE, Los Alamitos, CA, 3–14.
- HALLNOR, E. G. AND REINHARDT, S. K. 2000. A fully associative software-managed cache design. *Comput. Archit. News* 28, 2, 107–116.
- HATFIELD, D. J. AND GERALD, J. G. 1971. Program restructuring for virtual memory. *IBM Syst. J.* 10, 3, 168–192.
- JANAPSATYA, A., PARAMESWARAN, S., AND IGNJATOVI, A. 2004. Hardware/software managed scratchpad memory for embedded system. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE, Los Almitos, CA, 370–377.
- JANAPSATYA, A., IGNJATOVI, A., AND PARAMESWARAN, S. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the Conference on Asia South Pacific Design Automation*. IEEE, Los Alamitos, CA, 612–627.
- KALAMATIONOS, J. AND KAEI, D. R. 1998. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 244–253.
- KANDEMIR, M., RAMANUJAM, J., IRWIN, M. J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2001. Dynamic management of scratchpad memory space. In *Proceedings of the 38th Annual Design Automation Conference*. ACM, New York, 690–695.
- KANDEMIR, M., KADAYIF, I., AND SEZER, U. 2001. Exploiting scratchpad memory using Presburger formulas. In *Proceedings of the 14th International Symposium on Systems Synthesis*. ACM, New York.
- KIROVSKI, D., LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1999. Application-driven synthesis of memory-intensive systems-on-chip. *IEEE Trans. Comput. Aid. Des. Integr. Circuits Syst.* 18, 9, 1316–1326.
- MICRON TECHNIQUE, INC. 2009. System Power Calculator. http://www.micron.com/support/part_info/powercalc.
- NGUYEN, N., DOMINGUEZ, A., AND BARUA, R. 2005. Memory allocation for embedded systems with a compile-time-unknown scratchpad size. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, New York, 115–125.
- PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1997. Efficient utilization of scratchpad memory in embedded processor applications. In *Proceedings of the European Conference on Design and Test*. IEEE, Los Alamitos, CA, 7–11.
- PANDA, P. R., NIKIL, D. D., AND ALEXANDRU, N. 2000. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Trans. Autom. Electron. Syst.* 5, 3, 682–704.
- PARK, S., PARK, H.-W., AND HA, S. 2007. A novel technique to use scratch-pad memory for stack management. In *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM, New York, 1478–1483.
- PETTIS, K. AND HANSEN, R. C. 1990. Profile guided code positioning. *SIGPLAN Notice*, 25, 6, 16–27.
- PYKA, R., FABACH, C., VERMA, M., FALK, H., AND MARWEDEL, P. 2007. Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications. In *Proceedings of the 10th International Workshop on Software and Compilers for Ecosystems*. ACM, New York, 41–50.
- SJODIN, J., FR DERBERG, B., AND LINDGREN, T. 1998. Allocation of global data objects in on-chip RAM. In *Proceedings of the Conference on Compiler and Architecture Support for Embedded Computing Systems*. ACM, New York.
- SJODIN, J. AND PLATEN, C. V. 2001. Storage allocation for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, New York, 15–23.
- STEINKE, S., GRUNWALD, N., WEHMEYER, L., BANAKAR, R., BALAKRISHNAN, M., AND MARWEDEL, P. 2002a. Reducing energy consumption by dynamic copying of instructions onto on-chip memory. In *Proceedings of the 15th International Symposium on System Synthesis*. ACM, New York, 213–218.

- STEINKE, S., WEHMEYER, L., LEE, B.-S., AND MARWEDEL, P. 2002b. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, Los Alamitos, CA, 409–415.
- UDAYAKUMARAN, S. AND BARUA, R. 2003. Compiler-decided dynamic memory allocation for scratchpad-based embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, New York, 276–286.
- UDAYAKUMARAN, S., DOMINGUEZ, A., AND BARUA, R. 2006a. Dynamic allocation for scratchpad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2, 472–511.
- UDAYAKUMARAN, S. AND BARUA, R. 2006b. An integrated scratchpad allocator for affine and non-affine code. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, Los Alamitos, CA, 925–930.
- VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004a. Cache-aware scratchpad allocation algorithm. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, IEEE, Los Alamitos, CA, 1264–1269.
- VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004b. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd International Conference on Hardware/Software Co-Design and System Synthesis*. ACM, New York, 104–109.
- VERMA, M., PETZOLD, K., WEHMEYER, L., FALK, H., AND MARWEDEL, P. 2005. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia*. IEEE, Los Alamitos, CA, 115–120.
- WEHMEYER, L., HELMIG, U., AND MARWEDEL, P. 2004. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*. ACM, New York, 114–120.
- WILTON, S. AND JOUPPI, N. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid State Circuits* 31, 5, 677–688.
- WOLF, M. E. AND LAM, M. S. 1991. A data locality optimizing algorithm. *SIGPLAN Notice* 26, 6, 30–44.

Received November 2009; revised November 2009; accepted December 2009