Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications

Doosan Cho, *Member, IEEE*, Sudeep Pasricha, *Member, IEEE*, Ilya Issenin, *Member, IEEE*, Nikil D. Dutt, *Fellow, IEEE*, Minwook Ahn, and Yunheung Paek, *Member, IEEE*

***Abstract*—Exploiting runtime memory access traces can be a complementary approach to compiler optimizations for the energy reduction in memory hierarchy. This is particularly important for emerging multimedia applications since they usually have input-sensitive runtime behavior which results in dynamic and/or irregular memory access patterns. These types of applications are normally hard to optimize by static compiler optimizations. The reason is that their behavior stays unknown until runtime and may even change during computation. To tackle this problem, we propose an integrated approach of software [compiler and operating system (OS)] and hardware (data access record table) techniques to exploit data reusability of multimedia applications in Multiprocessor Systems on Chip. Guided by compiler analysis for generating scratch pad data layouts and hardware components for tracking dynamic memory accesses, the scratch pad data layout adapts to an input data pattern with the help of a runtime scratch pad memory manager incorporated in the OS. The runtime data placement strategy presented in this paper provides efficient scratch pad utilization for the dynamic applications. The goal is to minimize the amount of accesses to the main memory over the entire runtime of the system, which leads to a reduction in the energy consumption of the system. Our experimental results show that our approach is able to significantly improve the energy consumption of multimedia applications with dynamic memory access behavior over an existing compiler technique and an alternative hardware technique.**

***Index Terms*—Compiler optimizations, dynamic memory access pattern, multiprocessor system on chip (MPSoC), scratch pad memory (SPM).**

Manuscript received February 10, 2008; revised July 15, 2008 and September 30, 2008. Current version published March 18, 2009. This work was supported in part by the Korean government (MEST) through the Korea Science and Engineering Foundation (KOSEF) NRL Program under Grant R0A-2008-000-20110-0 and in part by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation (KOSEF) under Grant R11-2008-007-01001-0. This paper was recommended by Associate Editor V. Narayanan.

D. Cho, M. Ahn, and Y. Paek are with the Department of Electrical Engineering and Computer Science, Seoul National University, Seoul 151 742, Korea (e-mail: mew26@snu.ac.kr; ypaek@snu.ac.kr).

S. Pasricha is with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523 USA (e-mail: sudeep@engr.colostate.edu).

I. Issenin is with Teradek, Irvine, CA, and also with the Center for Embedded Computer Systems, Department of Computer Science, Donald Bren School of Information and Computer Sciences, University of California at Irvine, Irvine, CA 92697 USA (e-mail: isse@ics.uci.edu).

N. D. Dutt is with the Center for Embedded Computer Systems, Department of Computer Science, Donald Bren School of Information and Computer Sciences, University of California at Irvine, Irvine, CA 92697 USA (e-mail: dutt@ics.uci.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2014002

I. Introduction

In Multiprocessor Systems on Chip (MPSoCs), the effect of the increasing processor–memory speed gap is being more significant due to the heavier access contention on the network and the use of shared memory. Therefore, improvement in memory performance is critical to the successful use of MPSoC systems. In order to narrow the processor–memory speed gap, hardware caches have been widely used to build a memory hierarchy in all kinds of system chips. However, hardware-only cache implementation has several drawbacks. The hardware-controlled approach incurs high power and area cost [1].

An alternative to hardware-controlled cache is a "software-controlled cache," which is essentially a random access memory called scratch pad memory (SPM). The main difference between SPM and hardware-controlled cache is that SPM does not need hardware logic to dynamically map data or instructions from off-chip memory to the cache since it is done by software. This difference makes SPM more energy and cost efficient for embedded applications [2]. Due to these advantages, SPMs are widely used in various types of embedded systems. In some embedded processors such as ARM10E, Analog Devices ADSP TS201S, Motorola M-core MMC221, Renesas SH-X3 and TI TMS370CX7X, SPM is used as an alternative to hardware cache. Consequently, an approach for effective SPM utilization is essential for the efficacy of SPM-based memory subsystems. A previous work on SPM utilization has focused on the development of approaches for efficiently assigning frequently accessed data and instructions to SPM to maximize improvement of performance and energy consumption.

The objective of this paper is to propose an efficient technique to exploit the data reusability of multimedia applications with dynamic behavior on SPM-based MPSoCs. The memory access behavior is not fixed for an embedded system since the accessed values vary dynamically according to the input data. For example, depending on whether the specific input activates one execution path or the other, some of the data accesses may be executed or not. Thus, when multimedia applications are executed on different sets of input data, different memory access patterns emerge. Our methodology allows a designer to identify such situations and generate different data layouts that make the most efficient use of the SPM for each possible behavior. Our approach proactively controls the movement and placement of data in the hierarchy based on runtime data access history. Thus, the data placement is adaptively optimized to the input.

Specifically, our management scheme is based on hardware/software cooperation. The memory manager in the operating

system (OS) determines SPM data placement based on dynamic data access history. The history is recorded through a simple hardware table. To decide good data layouts for various input streams, the compiler first analyzes application behavior with representative input sets. Then, data layouts for each of the input sets are generated. At runtime, the memory manager compares runtime memory access traces with the profiled data access history to determine the data layout for the current input. We show that this extension to the memory subsystem significantly reduces the overall energy consumption of multimedia applications. The energy savings are due to increased SPM hit rates with minimized data movement overhead.

The rest of this paper is organized as follows. Section II discusses related work. Section III presents a motivational example that highlights the need for an adaptive SPM management approach. Section IV describes the main hardware component of our adaptive SPM management approach and associated design considerations. Sections VI and VII describe the software support required from the compiler and OS for our approach. Section VIII describes the methodology and experimental setup used to assess the efficacy of our approach and presents detailed simulation results and analysis. Finally, the conclusion is presented in Section IX.

II. Related Work

Many papers have focused on the problem of improving data reuse in caches, primarily by means of loop transformations (see, e.g. [3] and [4]). However, we do not address this problem in this paper. We assume that all possible loop transformations for improving locality of accesses are already performed before applying the technique presented in this paper.

A. Regular/Irregular Memory Access Pattern Analysis for SPM

There are several prior studies on using SPMs for data accesses. The studies are mostly based on compile-time analysis. They can be categorized into two parts, static and dynamic. Static methods [1], [5]–[9] determine which memory objects (data or instructions) may be located in SPM at compile time, and the decision is fixed during the execution of the program. This may lead to the nonoptimal use of the SPM if behavior of the application during execution is different from compile-time allocation assumptions. Static approaches use greedy strategies to determine which variables to place in SPM, or formulate the problem as an integer-linear programming problem (ILP) or a knapsack problem to find an optimal allocation.

Dynamic SPM allocation approaches include those in [2], [10]–[16]. Cooper and Harvey [10] proposed using SPM for storing spilled values. Ozturk *et al.* [15] proposed to manage the available SPM space in a latency-conscious manner with the help of compilers. Specifically, the proposed scheme places data into the SPM, taking into account the latency variations across the different SPM lines. Francesco *et al.* [16] proposed an integrated hardware/software approach for runtime SPM management. Their hardware consists of SPMs coupled to direct-memory-access (DMA) engines to reduce the copy cost between SPM and main memory, and they provide a high-level programming interface which makes it very easy to manage the SPMs at runtime. In this paper, we also provide high-level programming interface to manage DMA engines described in Section VII. The difference compared to [16] is that they determine copy candidates to map in the SPM at compile time, but we determine them at runtime. In multimedia applications, this difference leads to significant energy saving; we will show the impact of the difference in Section VIII-B. Udayakumaran and Barua [14] proposed an approach that treats each array as one memory object. Placement of parts of array to the SPM is not possible, but the approach does consider all global and stack variables. Kandemir *et al.* [12] address the problem of dynamic placement of array elements in SPM. The solution relies on performing loop transformations first to simplify the reuse pattern or to improve data locality. Dynamic approaches also use ILP formulations or similar methods to register allocation to find an optimal dynamic allocation.

A few approaches have looked at memory system design for MPSoCs [17]–[19]. Meftali *et al.* [17] and Kandemir and Dutt [18] proposed an optimal memory allocation technique based on ILP for application-specific SoCs. Issenin *et al.* [19] introduced a multiprocessor data reuse analysis technique that allows the system designer to explore a wide range of customized memory hierarchy organizations with different sizes and energy profiles.

While the research described earlier focused explicitly on regular access patterns, Verma *et al.* [9] and Li *et al.* [20] proposed approaches that work with irregular array access pattern. Verma *et al.* [9] proposed a static approach to put half of the array to SPM. They also profile an application, find out which half of the array is more often used, and place it in the SPM. However, they do not care if the accesses are regular or not. Unlike in [9], we perform the task of finding a set of array elements to be placed to SPM with finer granularity. In addition to that, the replacement of data in SPM happens at runtime in our approach as compared to static placement in [9]. Li *et al.* [20] introduced a general-purpose compiler approach, called memory coloring, which adapts the array allocation problem to graph coloring for register allocation. The approach operates in three steps: SPM partitioning to pseudoregisters, live-range splitting to insert copy statements in an application code, and memory coloring to assign split array slices into the pseudoregisters in SPM. However, their approach is prone to internal memory fragmentation when the sizes of assigned array slices are less than pseudoregister size (where the partitioned SPM space). They try to solve this problem by making several sizes of pseudoregisters. However, this approach cannot completely solve the problem because the partitioning method uses a constant variable to divide the SPM space, which leads to unavoidable fragmentation. We solve this problem by formulating it as a 2-D (time and space) knapsack problem that can assign array slices to SPM without any internal fragmentation.

Absar and Catthoor [21] and Chen *et al.* [22] also present approaches for irregular array accesses. The meaning of irregularity in their work is limited to the case of an indirect indexed array. In addition to that, the indexing array must be

referenced by an affine function. In these works, the authors identify the reused block or tile which is accessed through indirectly indexed arrays in video/image processing applications. Our approach differs from theirs in that we can solve indirect indexed arrays with nonaffine reference functions. In addition, our approach also considers all other types of irregular accesses found in various media applications.

B. Dynamic Behavior Analysis for Cache Memory Subsystems

To the best of our knowledge, there are no existing successful approaches for runtime SPM management. A software caching technique is the only runtime approach proposed so far. It has largely not been successful. Software caching [23]–[25] emulates a cache in SPM using software. By compiler-inserted codes, the tag, data, and valid bits are all managed at each memory access. Even though the compiler already optimizes away some [23], [24], significant software overhead is incurred to manage these fields.

There are some other studies related to the runtime approaches for dynamic behavior optimization on hardware-controlled cache-based systems by Kandemir and Kadayif [26] and Ding and Kennedy [27]. In these studies, the selection of data placement for the cache-based system can be changed by loop-transformation-based approaches at runtime. In [26], the loop transformation is performed over the course of a program's execution according to the data cache miss estimating function. The proposed technique can be used to automatically determine which placements result in minimum cache misses over specific regions of a program while taking into account the added overhead of dynamic layout changes. The approach is based on program's control flow, called nest flow graph. It adds some optimized code at certain program execution points on the graph. The work is good for applications with well-behaved loops. However, the dynamic code selection might result in I-cache misses.

The study in [27] is the first work on improving cache performance for dynamic applications by using a combination of runtime computation and data transformation. This paper is based on an inspector–executor framework [28] that parallelizes unstructured codes for distributed memory machines. Unfortunately, the framework cannot localize nonaffine references. Therefore, this paper would perform as badly as the original computation with such references. Our approach differs from theirs in that we can optimize all types of dynamically behaved loops found in various multimedia applications without increasing code size or causing I-cache misses.

III. Motivational Example

Embedded systems today are evolving into complex multiprocessor systems that incorporate an OS layer in their software. In a general on-chip system, multiple IPs share one on-chip bus to access system memory. Each programmable IP has a DMA to access memory directly instead of relying on the CPU. In our approach, we consider MPSoC systems consisting of several processing elements, and each processing element has its own local SRAM (scratch pad). We restrict the network architecture to a bus-based architecture since it is still the most popular network [29]–[32]. We assume that processing elements communicate with each other through a shared memory. Each processing element has a single port for both local and shared memory accesses, as usually is the case in real systems. In addition, each processing element executes its own task, and the processing elements do not share local memory address space. Therefore, our runtime SPM manager optimizes the data layout with the assigned task on the corresponding local storage.

```
for(k=0; k< frames; k++)
 for (j=0; j<16; j++) {
  for (i=0; i<16; i++) {
  ...
    if (hx && !hy)  ---(1)
      v = ((p1[i]+p1[i+1]+1)>>1) - p2[i];

    else if (!hx && hy) { ---(2)
      v = ((p1[i]+p1a[i]+1)>>1) - p2[i];

    else if (hx && hy) { ---(3)
      v = ((p1[i]+p1[i+1]+p1a[i]+p1a[i+1]+2)>>2) - p2[i];
  ...
  }
 ...
 }
```

Fig. 1. Simplified loop code in the motion estimation.

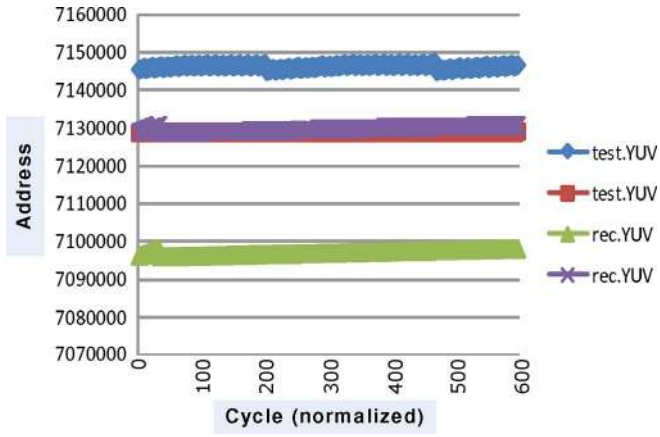

Fig. 2. Memory access distribution with two input sets.

To understand some of the inefficiency of traditional SPM management, it is helpful to first examine the access behavior of a particular application in detail. Fig. 1 shows a loop code from MPEG2 encoder. This loop body constitutes a major part of motion estimation, which takes 37%–84% of total execution time in our studies with various input sets. Most of the memory accesses in MPEG2 encoder occur in this routine.

In order to obtain a clear picture of how data in memory are accessed throughout the program execution, we profiled the accesses and plotted the address distribution for a given execution phase. The profiling results for a 100 000-cycle sample of the MPEG2 encoder are shown in Fig. 2. The memory access distribution for two different input sets is shown in Fig. 2, namely, test.YUV and rec.YUV. The distributions show that two bands of memory are heavily accessed for each input set. These bands are located roughly from addresses 70 950 000 to 71 000 000, 71 250 000 to 71 350 000 with an input file of

rec.YUV and 71 250 000 to 71 350 000, 7 145 000 to 7 150 000 with an input file of test.YUV. In general, the dynamic loop controls lead to differently accessed memory region [27].

Often, traditional schemes analyze and optimize the dynamic behavior through statistical information generated by profiling: A benchmark code is executed with various types of input sets to gather memory access behavior for each input, and statistically, highly accessed data regions are obtained. At compile time, these schemes optimize data layout based on the profiling result.

In Fig. 2, these schemes place either part of the data or the entire data in the accessed bands onto the SPM. If only part of the data is placed in SPM, it loses some portion of reusability of the program, since the access pattern depends on the input. If the entire data are placed in SPM, it causes excessive data transfers, because the program sometimes does not access some parts of the accessed bands. In addition, it is not always possible since there is only a limited SPM space. Therefore, both cases may result in a nonoptimal solution. To solve this problem, we propose an HW/SW integrated approach that analyzes the input characteristic at runtime and optimizes a data layout fitting based on the input. Using the proposed technique takes significantly better energy gain than the traditional schemes illustrated in Section VIII. However, our technique does not predict 100% of dynamic memory access patterns; thus, it may not be applied to real-time systems as itself. By using worst-case execution time (WCET) analysis such as in [33], the proposed technique may apply to soft real-time systems, since WCET analysis generates information whether it is applicable to a certain real-time system.

IV. DART

In this section, we introduce a new hardware component called the data access record table (DART) which records runtime memory access history to support decisions of data placement at runtime. Each DART entry tracks the accesses to particular memory regions, facilitating detection of dynamic memory access behavior across data blocks while they are accessed. The information recorded in DART is used in the OS for making a decision about which region of data block to place onto the SPM. To make a good decision, it should be able to characterize dynamic behavior of a program and identify which input is being processed.

A. WML

Our scheme seeks to optimize a data layout in a manner that is sensitive to the dynamic memory locations accessed. Since there are an excessively large number of memory locations, we introduce the notion of a working memory location (WML). Ideally, track of the usage frequencies of all data in memory should be kept because it would give us the most accurate information. However, it would lead to an unmanageably large amount of information. Instead, we define a region of heavily accessed data block, called WML, that represents heavily accessed memory regions as shown in Fig. 2.

Definition 1: The WML is a region of reused block of data that is used in a contiguous loop-iteration sequence.

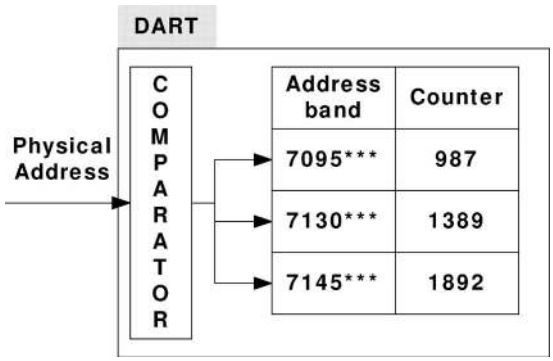

Fig. 3. Memory accesses with DART in Fig. 1.

By observing the access behavior of the WML, our approach can determine what region of data should be placed onto the SPM. A hardware DART is used to maintain and utilize the access behaviors of the WMLs to guide data placement on the SPM. The numbers of WMLs and address bands are measured by profiling. For example, Fig. 2 shows three WMLs (7145×, 7130×, and 7095×).

The DART contains record entries for each WML. Each entry in the table contains an access frequency counter, where the counter value represents the frequency of accesses to the corresponding WML. The additional hardware cost incurred by the DART is relatively small, because it consists of a small number of highly accessed address bands (WMLs) and counter entries. We determine the size of DART based on the maximum number of WMLs in a given application.

An example of a data access operation with DART is shown in Fig. 3, where data in a WML are accessed. The WML address band entry is initialized by the OS when the first loop is started, and the OS updates the address band entry for each loop.

The comparator tests if the current address matches with the address entry. If the result is yes, then an increment of the corresponding DART counter will be performed. Each entry covers an accessed region corresponding to a WML. An increment operation on the counter is performed in parallel with the data access.

V. Overall Workflow With Compiler and OS

The overall workflow of our approach is shown in Fig. 4. The proposed approach consists of three parts: DART, compiler, and OS. The input to the system is C code that has already been parallelized at the task level and mapped to processing units.[1] For instance, the task graph [34] of H.263 decoder is shown in Fig. 4. A brief overview of the workflow involving the compiler and OS is presented next.

The compiler analyzes each input task through five steps. First, profiling is used for gathering data access traces. During profiling, each program task is run multiple times with representative input data to collect the number of accesses to each array element; memory access traces are obtained for each input data. In the second and third steps, reuse analysis and lifetime

[1]The optimal mapping problem is beyond the scope of this paper, so we assume that mapping is already done by a compiler.

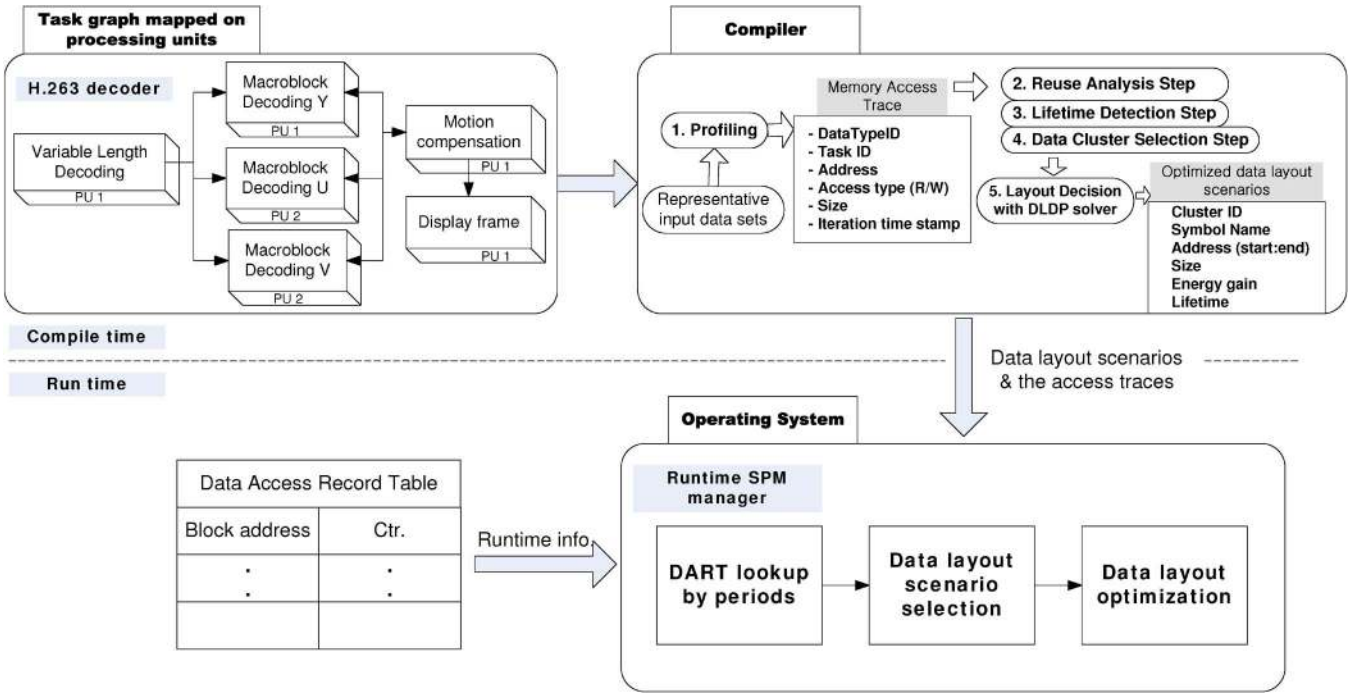

Fig. 4. Overall workflow for proposed approach.

(LT) detection with the traces are performed. By using formal metrics, candidates of data clusters to be copied to SPM are determined in the fourth step. The data cluster is defined in more detail in the next section. The final step is to decide the location of data clusters in order to maximally use the limited SPM space. In this step, data layouts are made by the compiler for the representative input sets. These layouts are used as inputs to the OS for optimizing data layout.

The runtime SPM manager in the OS periodically refers to the DART and the access trace, and selects a data layout for the current input data. Finally, the DMA places data clusters onto the SPM for the selected layout. In the next section, we describe how data layouts are generated in our approach.

VI. Data Layout Candidate Generation by Compiler

The problem of efficient data layout generation is to find a good layout of data elements in the SPM address space to minimize address fragmentation. Although the problem is known to be NP-complete [2], we employ heuristics and are able to find a good (suboptimal) solution for the problem in polynomial time.

Briefly, a designer identifies a set of representative inputs that can trigger the different behaviors that will arise at runtime. The applications are then fed with the different input patterns, and for each execution, a different data layout scenario is obtained. The following sections describe these tasks in more detail.

A. Data Selection Using Data Reusability and LT Analysis

The usefulness of memory hierarchy depends on the amount of reuse in data accesses. To measure the amount of reuse, we now present a data reusability model that we used to determine candidates of data elements to be copied to SPM.

We use a data reusability factor as a metric which measures how many references access the same location during different loop iterations. Let T_{n_i} be data reusability factor for i_{th} element of data block n, which depends on the estimated element size of N words, as well as on the access frequency F corresponding to each array element, which is obtained by profiling. The reusability factor is defined next.

Definition 2—(Reusability Factor) $T_{n_i} = F/N$*:* Our technique selects candidates of data to be located in SPM when data elements have data reusability factor of more than one, because those data elements can reduce at least one main memory access (supported by DMA engine). Since the number of the elements is excessively large, we transfer the data elements onto SPM as a data cluster. Before we define a data cluster, we first introduce the terms iteration vector, LT, and LT distance (LT-D).

Definition 3—Iteration Vector: Given a nest of n loops, the iteration vector i of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level. In other words, the iteration vector is given by

$$I = \{(i_1, i_2, \ldots, i_n) | Lbound_{i_k} \leq i_k \leq Ubound_{i_k}\}$$

where i_k, $1 \leq k \leq n$, represents the iteration number of the loop at nesting level k, and each L/U bound denotes lower/upper bound of corresponding loop nest. The set of all possible iteration vectors for a loop statement is an iteration space.

Each data access can be represented by the iteration vector. If a data element is accessed again, then both accesses have their own iteration vectors.

The Algorithm for data clustering
Input: candidate arrays
Output: data clusters
Begin
1. For each data array do
1-1. Select an element which has the highest reusability value
1-2. Calculate LT of each element in the given data array
1-3. Compute LT-Ds between the selected element and the others
1-4. Determine the most beneficial LT-D heuristically
1-5. Combine elements of the LT-D to a data cluster
1-6. Remove the elements of the data cluster in the given set of candidates
1-7. Repeat the same procedure with remaining data elements until this procedure generates no data cluster
End

Fig. 5. Data clustering algorithm.

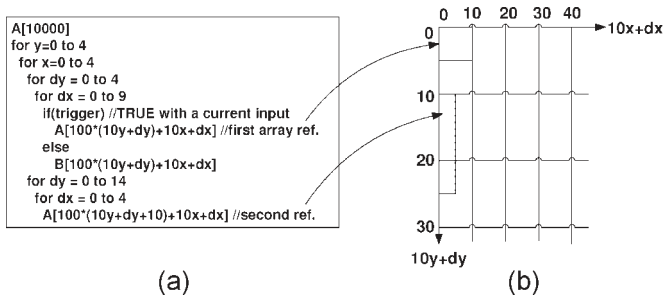

Fig. 6. Example access trace with an input. (a) a loop body. (b) elements accessed during the first iteration of x, y.

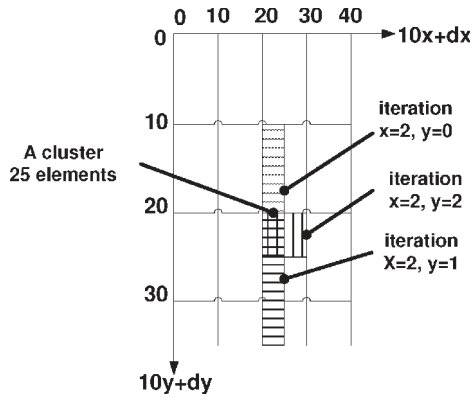

Fig. 7. Cluster generation.

Definition 4—LT: LT of a data element d is defined as a difference between the iteration vector of the first access (FA) and the last access (LA). It represents the loop execution time duration, which is a nonempty interval in a loop-iteration space

$$LT(d) = LA(i_1, i_2, \ldots, i_n) - FA(i_1, i_2, \ldots, i_n).$$

Definition 5—LT-D: The LT-D is a difference in LTs of two elements in a data array.

$LT\text{-}D = |LT(n_a) - LT(n_b)|$, where n_a and n_b represent a_{th} and b_{th} elements in a data block n.

Making LT-D small minimizes fragmentation that might appear when data objects are transferred in and out of the SPM. This increases the chance of finding a large-enough block of free space in the SPM, which results in the least possible main memory accesses. For this purpose, we introduce the notion of a data cluster.

Definition 6—Data Cluster: A data cluster is a union of data elements that have the most beneficial LT-D in an array.

The data clustering algorithm is shown in Fig. 5. In the algorithm, the most beneficial LT-D should be carefully determined because it determines the efficiency of our technique. We determine LT-D with the help of a simple and practical heuristic. The decision procedure is like this: First, the smallest LT-D is taken. Second, clusters with the LT-D are generated. Next, the energy saving from allocation of the clusters is estimated. In the fourth step, LT-D is increased to the next larger one. Steps 2–4 are repeatedly executed until an LT-D is found whose gain does not exceed the others.

We illustrate our procedure for cluster generation with the program example shown in Fig. 6(a), which is extracted from the SUSAN algorithm for image noise filtering. For this program, the footprint of the addresses of accessed data elements with varying values of iterators y and x is shown in Fig. 6(b). With every increment of the iterator x, the footprint shifts right by ten elements. If we increment y by one, the footprint shifts down by ten elements. If we continue iterating over x and y, we can notice that some of the elements are read more than once. For example, Fig. 7 shows a set of 25 elements, which is read three times during ten consecutive iterations of two outer loops. Moreover, the other 25-element set, which is formed by shifting that set by integer numbers of steps, behaves exactly the same way: After they are read for the first time, in five iterations, they are read again, and in another five iterations, they are read for the last time.

All those reused data elements can be copied into the SPM when they are accessed for the first time and can be read from there later. We can compute the iteration vectors of the FA and LA to the elements that are reused by the memory access trace.

As shown in Fig. 7, the candidate elements can be combined into a cluster. The clustering algorithm is performed as follows. The first step is to select one element in the 25 elements since they have the highest reusability. The first element is selected. Second, LTs of each element are calculated

$$\text{First} : LA(2,2,0,0) - FA(0,2,10,0) = (2,0,-10,0)$$
$$\text{Second} : LA(2,2,0,1) - FA(0,2,10,1) = (2,0,-10,0)$$
$$\cdots$$
$$25\text{th} : LA(2,2,4,4) - FA(0,2,14,4) = (2,0,-10,0).$$

Third, LT-Ds are calculated

$$\text{Second}-\text{First} = (0,0,0,0)$$
$$\text{Third}-\text{First} = (0,0,0,0)$$
$$\cdots$$
$$25\text{th} - \text{First} = (0,0,0,0).$$

In this case, all the distances of the candidate elements are zero. Thus, the most beneficial LT-D is determined as zero. Finally, the unit of 25 elements is obtained as a data cluster. The clustering procedure is iteratively applied to the remaining part of the array A. The most beneficial LT-D depends on the memory access pattern of an application. It is determined by the amount of energy saving estimated by Definition 10.

In general, all data clusters cannot be assigned to SPM since the SPM size is usually small (e.g., 1–16 kB). It is a capacity constraint in a data placement problem. In addition, some of selected clusters may not be assigned to the SPM because the memory address fragments can be scattered in the SPM address space (i.e., memory fragmentation). To solve this data cluster allocation problem with the goal of minimizing the fragmentation and with a capacity constraint, the formal definition of the problem is presented in the next section.

B. DLDP

A good data layout can place most of the data clusters in the SPM, which yields the least possible main memory accesses. In general, the data layout reorganization problem is to obtain such a good data layout. To obtain a better probability of finding a good layout, with minimal fragmentation, when clusters are transferred in and out, we propose to use a layout decision algorithm that increases the chances of finding a large-enough free space.

In this paper, the data layout decision problem (DLDP) is to find a particular ordering for each selected data cluster in the SPM address space for each loop in an application; the clusters should fit temporally and spatially into the SPM and deliver the highest overall energy saving by the method. To solve this problem, we formulate the problem as a 2-D (time and space) knapsack problem. A formal statement of the DLDP mapping to the 2-D knapsack is given in Definition 10, following the definition of several terms used.

Definition 7—Capacity Constraint: Let the SPM have a limited capacity C. For a set of assigned data clusters $d \in D_a$, $\sum_{d \in D_a} Size(d)$ must not exceed the total capacity of the SPM C in iteration time T, where T consists a normalized loop-iteration space as given in Definition 3.

Definition 8—Profit: This is the amount of energy saving calculated by how many main memory accesses can be reduced by maximizing SPM utilization, $E_{\text{profit}}(d)$.

Definition 9—Overhead: Data transfer overhead is represented by $E_{\text{overhead}}(d)$, which gives the energy needed for that data movement at runtime.

Definition 10—Definition of the DLDP:

1) Objective function: Find a good (suboptimal) layout of assigned set of data clusters D_a which maximizes the energy saving: $E_{\text{saving}} = \sum_{d \in D_a}(E_{\text{profit}}(d) - E_{\text{overhead}}(d))$.
2) Subject to: the capacity constraint in Definition 7.

The objective function of this problem is to minimize total energy consumption while maximizing the profit. The energy formulation used in the objective function is described in more detail in [35].

The 1-D (space) knapsack problem for memory object movement into SPM is formulated in [1]. It is a special case of DLDP in which there is only a static time; the formulation has no consideration of the time dimension. Since the problem is NP-complete and is a special case of DLDP, DLDP is also NP-complete. We can search for a good-enough solution by using a best-first search with a heuristic. In the next section, we describe our approach to solve the DLDP.

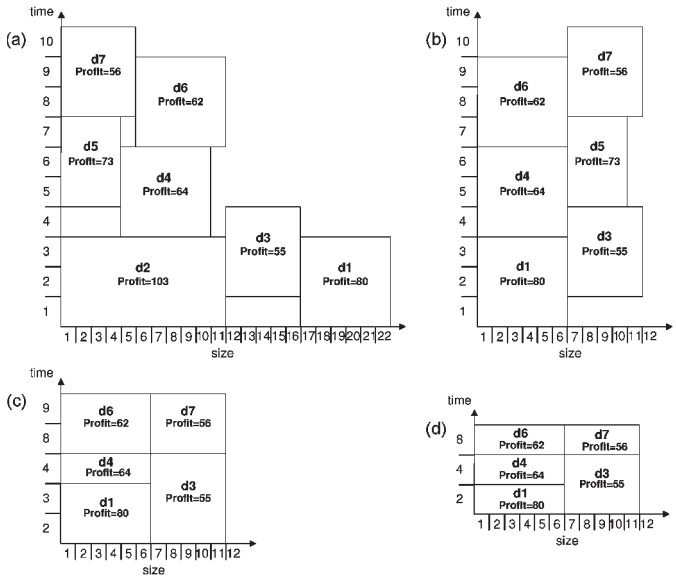

Fig. 8. Instance of DLDP to which the reduce and split operators are applied.

C. DLDP Solver

Our approach exploits a divide and conquer principle to effectively seek a solution to maximize the objective function in Definition 10. Our algorithm for solving the DLDP has two steps. Section VI-C-1 gives the algorithm of the divide step. Section VI-C-2 presents a best first search method for each problem instance, as a conquer step.

1) Divide Step: This procedure employs two basic operations: reduce, which simplifies a problem instance, and split, which decomposes a problem instance into smaller independent problem instances. An example of the two operations is shown in Fig. 8.

Fig. 8 shows an instance of DLDP that has seven clusters $d1, \ldots, d7$ that need to be allocated to an SPM of size 10 (x-axis). Results are shown for ten time units $T = \{1, 2, 3, \ldots, 10\}$ (y-axis).

The reduce operator performs two kinds of simplification. The first kind removes from the problem instance any clusters d whose size exceeds the capacity available in its $LT(d)$. For example, in the instance of Fig. 8(a), block $d2$ has size 11 at time $T = 1, 2, 3$, yet the SPM capacity is only 10. Therefore, the reduce operator removes $d2$ from the instance, which results in the instance shown in Fig. 8(b).

The second kind of simplification removes unnecessary times from the instance. In the instance of Fig. 8(b), at times 1, 5, 6, 7, and 10, the total size does not exceed the capacity. Since the constraints at these five times are satisfied in all assignments of the clusters, these times can be removed from the instance; thereby, the reduce operator can also remove $d5$, resulting in the instance shown in Fig. 8(c).

There is a second method by which the reduce operator removes times from an instance. It is often the case that two adjacent times have the same block. If the two times have the

```
Simplify(dldp* P){ //P: an instance of DLDP
  if(D == empty) then return;
  else {
    if T >= 2 then {
      ta = min(T);
      while ta != max(T) do {
        tb = next(ta);
        // time reduction operation
        if clusters(ta) == clusters(tb) then {
          remove t from T;
          for d∈ clusters(t) do
            remove t from LT(d);
        } else
            ta = tb;
      } //end of while
    }

    init(ta); //initialize ta to perform split operation
    while ta != max(T) do {
      tb = next(ta);
      // problem split operation
      if (next(ta) == tb) && (intersect(clusters(ta), clusters(tb)) == empty){
        p1 = times{t | t <= ta} and clusters{d | end(d) <= ta}
        p2 = times{t | t >= tb} and clusters{d | start(d) >= tb}
      }
    } //end of while
  }
}
```

Fig. 9. Procedure of the divide step with simplification.

same capacity, then either time can be removed. In the instance of Fig. 8(c), times 2 and 3 impose the same size constraint as do times 8 and 9. Thus, times 3 and 9 can be removed from the instance, resulting in the instance of Fig. 8(d).

The split operator decomposes a problem instance into subproblems that can be solved independently. A split can be performed between any two adjacent times, t and t', such that $\text{clusters}(t) \cap \text{clusters}(t') == \phi$. In the instance of Fig. 8(d), a split can be made in between times 4 and 8, resulting in two subproblems: one comprising times 2 and 4 and clusters $d1$, $d3$, and $d4$; and a second comprising time 8 and clusters $d6$ and $d7$.

In Fig. 9, let $Size(\text{clusters}(t))$ be the total required size at time t—that is $\sum_{d \in \text{clusters(t)}} size(d)$, where $\text{clusters}(t)$ represents assigned data clusters at the time t. In addition, let $\text{next}(t)$ be the next time $t+1$. The main procedure (Simplify()) uses the reduce and split operations for DLDP instance P as shown in Fig. 9.

2) Conquer Step With the k-Way Best First Search: In the previous section, the split procedure divides a problem instance (P) of DLDP to smaller problem instances $(p_1, p_2, \ldots, p_l)$. Each $p \in P$ is an objective of the k-way best first search [36] in the conquer step. The k-way best-first search is used to search for good-enough clusters ordering in SPM space. Instead of searching the whole set of orderings of clusters (which contains $D!$ orderings), the algorithm selects the k-best clusters in a sorted manner. Selection of the value of k should be done based on time-complexity and solution-optimality requirements.

The search algorithm builds a search tree and stores at each node the maximum energy gain and the minimum energy gain on the objective function for the DLDP instance.

Definition 11—Metric for Searching:

$$E_{\text{MAX}} = \max_{d' \in children(d)} (current_{\text{MAX}} + child(d, d'))$$

$$E_{\text{MIN}} = \min_{d' \in children(d)} (current_{\text{MIN}} + child(d, d'))$$

where $child(d, d')$ is the profit of the cluster assigned in moving from node d to node d', and $children(d)$ is the set of nodes that are children of d.

The search scheme repeatedly performs the following: 1) Select an unprocessed cluster; 2) process the cluster and then create its k-best number of children; and 3) propagate new max and min profits by Definition 11 through the tree, and use these profits to select the k clusters. It performs this sequence of three stages until the search tree contains no more unprocessed clusters. Notice that whenever a cluster is observed, its k children are immediately created, producing a search tree. Thus, the search tree's new leaves are always the children blocks which have the k-highest profit. Let us now consider the three major steps in more detail.

The first step finds the next node to process. The k-way best-first search selects leaf clusters by descending the search tree, starting at the root and taking the children with the k-highest profits at unobserved clusters. Our implementation orders the children from left to right so that their profits are nondecreasing with a priority queue.

The second step processes and expands the node. For each of these unobserved nodes, the maximum and minimum energy gain values on its objective function are obtained, and k-best children nodes are chosen to branch on. The k nodes are created and then processed and expanded in the same way. At each step, the set of nodes contributing to the $current_{\text{MAX}}$ is stored to a solution set.

The third step propagates the new energy gain and prunes some nodes. Starting at nodes just created and working up the tree to the root, the values of the maximum energy gain and the minimum energy gain are updated for each node. As this stage assigns and reassigns energy gain, it checks to see if any node has one child whose maximum energy gain does not exceed the minimum energy gain of the other child. In such a case, the reused block of maximum can be no better than that of the block of minimum, so the cluster of maximum and all its descendants are removed from the tree.

This search procedure produces a placement of clusters for an input. By iterating these compiler procedures, data layout scenarios for the representative input sets are obtained.

VII. Cluster Management Scheme in OS

This section introduces some new components at the OS level, which are capable of automatically managing the content of the SPM. The goal is to make the approach absolutely transparent, from the developer's point of view.

A. Data Layout Adaptation

To invoke the runtime adaptation procedure, we provide a high-level function Remap() which executes the procedure to determine an adapted layout corresponding to the current input at runtime. To increase efficiency of our system, the runtime SPM manager starts in a segment of the SPM.

Our management scheme periodically checks the information in DART and the memory access traces to optimize the current data layout based on the current input data. This adaptation

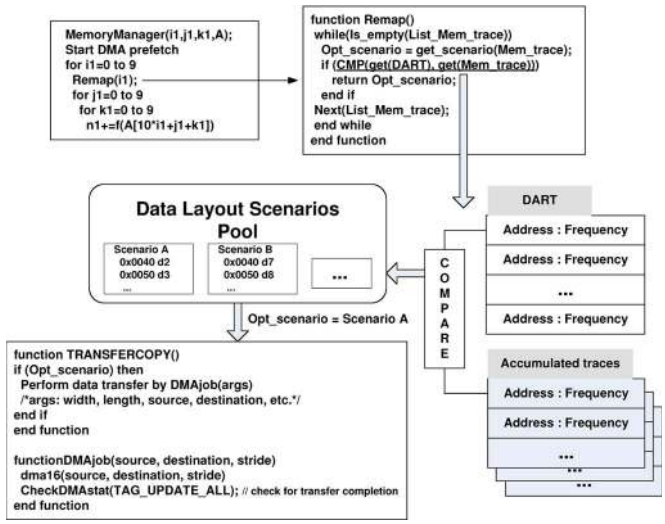

Fig. 10. Runtime data layout adapting procedure.

procedure does not come for free. The cost of our system includes data transfer latency during layout reorganizing and trace comparison latency for a data layout scenario selection. Both costs can be balanced by adjusting the frequency of Remap() calls. Thus, the costs are amortized over multiple computation iterations. The compiler should make sure that this cost does not outweigh any energy or performance gain by either adapting infrequently or making it adjustable at runtime. The adaptation period is determined by the compiler as follows.

Step 1) In a given loop hierarchy, the iteration step of the outer most loop is the initial period of the adaptation procedure.

Step 2) With the given period, energy gain/overhead is measured.

Step 3) To find the best period, binary search is used. One way is to increase the period by two times the current value. The other way is to divide that by a factor of two. Thus, each step gets closer to the best solution.

Step 4) Iterate Step 2–3 for finding the best period until no closer solution exists.

As will be shown in Section VIII-D, the procedure of data layout adaptation incurs negligible overhead in practice.

The runtime SPM manager integrated as part of the OS requires some preparatory steps. To make trace handling more manageable, the access frequencies of each trace are accumulated in the region of WML. This accumulated trace is used in the comparison procedure as shown in Fig. 10.

By comparing the traces with records of DART, the runtime SPM manager can predict what part of data would be frequently used in the current input data. Fig. 10 shows the pseudocode of how to select the data layout.

After the decision of the layout scenario to move a set of clusters, the TransferCopy() function is internally invoked. This function contains a set of rules that, according to the data that are requested to be moved, selects the appropriate action to be taken during DMA transfer. In the function, the call of TransferCopy() leads to the transfers as described hereinafter.

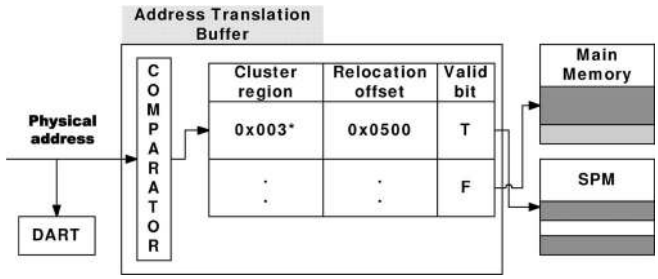

Fig. 11. Address translation.

B. DMA for Preloading Clusters

The layouts (including dynamically allocated clusters) have to be loaded by the SPM manager. Those clusters should be placed into the SPM at runtime. For that, we use a DMA engine to minimize any latency due to the movement.

The use of a DMA engine requires a coordinated hardware/software effort: The software must decide whether to use the DMA module for any given transfer or send commands to do the transfer. To that end, the hardware generates signals (interrupt) when data transfer is completed. With the signals, the software can synchronize while working in other tasks.

The DMA is controlled by the OS. To implement this, we propose the utilization of a tailored data transfer function. Depending on the selected layout, the function decides, at runtime, how to execute the data transfer: using a DMA resource, giving a high priority to the transfer, or not.

The DMA routines are used to specify the memory transfers between the SPM and the main memory (TransferCopy() shown in Fig. 10). A DMA job is initialized using DMAjob(). Its first two arguments (width and length) specify the shape of the cluster transfer. Subsequent arguments contain information on the address and stride. This information is needed by the transfer engine to generate the addresses for the burst/copy operations. The source/destination arguments indicate the direction of the data transfer. If the CheckDMAstat() function is called, the processor will be stalled until a DMA transfer completion. Otherwise, the processor continues its execution in parallel with the DMA.

C. Address Translation at Runtime

The decision whether to put an object into the SPM and where it should be located is moved from the application to the runtime system in OS. Therefore, the actual address of a memory object may change at runtime. An additional address translation buffer is used (as shown in Fig. 11) to dynamically translate a memory access to the desired address decided by our runtime system. This address translator is implemented by a set of address relocation registers. These registers are built with high-speed logic to make the address translation efficient. Each access to memory must go through the address translator. Only the OS can change a memory map using privileged instructions that load or modify the relocation registers.

For correct SPM addressing, each entry field of the translator records the address region of each cluster. The runtime SPM

manager updates the entry field when Remap() is called. The difference between main memory and SPM address of a cluster is called the relocation offset. On an access, the comparator checks if the memory access belongs to a cluster region in the buffer, and applies address translation if there is a match. The input address of the address translation buffer is added to the corresponding relocation offset to generate the data address on the SPM. However, the final address is only valid if the corresponding data have been transferred to the SPM by the DMA. The buffer has a valid bit field that indicates if the data transfer to the SPM has completed. The bit is triggered by a signal on DMA completion.

VIII. Experiments

We conducted several tests to assess the functionality and performance of our proposed approach. We explore how much our runtime adaptation procedure influences the energy consumption while minimizing the overhead. Next, we test the effect of the adaptation period on energy-delay efficiency. We chose to base our analysis on benchmark traces taken from real-world data processing algorithms, like transformations, filters, etc., most of which are derived from MediaBench [37]. The goal of our experiments is to compare our runtime adaptation approach with alternative software/hardware approaches and to estimate the ability of these approaches to exploit data reusability of the data accesses for a number of multimedia applications.

A. Experimental Setup

We created a tool set that implements our proposed technique. We used a Pentium 4 workstation for profiling each task. The SimpleScalar simulator-based [38] chip multi-processors (CMP) simulator (with four processing elements) was used for obtaining the number of misses for the cache. The SimpleScalar simulator was augmented and modified to provide several CMP features [39]. An external main memory with 18-cycle latency and the local SRAM (scratch-pad) with one-cycle latency were simulated in the default configuration. In our experiments, we used a relatively small off-chip memory and did not account for the energy dissipation in the off-chip buses due to limitations of the used energy model [40], [41]. We used CACTI [40], [41] for energy estimation of both the cache and SPM at 90-nm technology. The SPM is implemented at the same technology as the cache, but we remove the tag memory array, tag column multiplexers, tag sense amplifiers, and tag output drivers in CACTI that are not needed for the SPM. The other hardware components such as DART are estimated by WATTCH [42]. The simulator [39] is integrated with WATTCH. To generate the main memory energy parameter, a detailed model of a mobile SDRAM, from Micron Technologies [43], was selected in our experiments. We chose Micron's mobile SDRAM as it represents one of industry's best low-power SDRAMs and is widely used in media devices.

The experimental input is a set of codes obtained from MediaBench [37] with various sizes (7.2–504 kB) of input data. A brief description of the applications and their input data sets is given in Table I. For each benchmark, we selected two input data sets exhibiting significantly different characteristics. For the MPEG2, we chose video of CIF and QCIF resolutions, with a fast and relatively slow moving scenes (fast.m2v and slow.m2v), and low and high change in background scenery (low.YUV and high.YUV). For the H263, our input data include slow/short and relatively long moving scenes (small.263 and large.263). For JPEG and Susan, we selected images with different sizes and different content (corners.pgm, color.pgm, snap.jpg, and small.jpg). The images include substantial amount of color and background detail or black and white images with and without sharp boundaries. For MP3, our input data set includes a simple voice of a man and sound of a crowd of people laughing (voice.pcm and crowd.pcm). The benchmarks were compiled using gcc with optimizations turned on (−O2). The experiments are performed with 2k, 4k, and 8k SPM sizes and with the same size of the cache (1–8 way set-associative).

TABLE I
Program Codes and Inputs

Program	*Abbreviation*	*Description*	*Inputs*
MPEG2 Encocder	MPEG-E	The generic coding of moving picture and associated audio	high.YUV(504K) low.YUV(72K)
MPEG2 Decocder	MPEG-D	The generic decoding of moving picture and associated audio	fast.m2v(35K) slow.m2v(14K)
MP3 encoder (Lame)	MP3-E	MPEG-1 audio layer 3 audio encoding format	voice.wav(26K) crowd.wav(177K)
Jpeg decoder	JPEG-D	Decoder of the image compression	small.jpg(19.6K) snap.jpg(121.6K)
Susan	Susan	Image noise filter	color.pgm(7.2K) corners.pgm(14K)
H263 decoder	H263-D	The standard video codec	small.263(33.5K) large.263(114K)

B. Comparison With an Existing Method

This section presents the results obtained by comparing our method for dynamic behavior of multimedia applications against one of the best existing methods. For comparison, since there exist no other automatic compiler methods to handle dynamic behavior of multimedia application for SPM, we use the most general existing compiler-directed SPM allocation method for regular/irregular memory access patterns from [44]. The existing method represents one of the state-of-the-art.

Fig. 12 compares the energy consumption of multimedia applications using our approach versus the existing approach which is the baseline with all SPM sizes. With the comparison, the SPM is managed only by compile-time decision. With our approach, the SPM is managed by the combination of compile time and runtime decision. The figure shows an average reduction of 28.8% in energy consumption using our method. This result demonstrates that our approach has the potential to significantly improve energy consumption beyond today's state-of the-art. The energy reduction with our approach is because memory accesses in a dynamically behaved loop are not easily amenable to optimization by the existing approach.

In general, energy improvements from allocation by runtime decision to SPM are proportional to the percentage of data

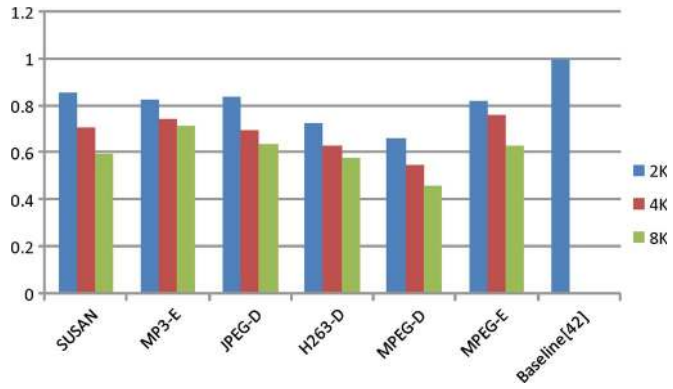

Fig. 12. Energy consumption with the proposed approach against one existing SPM management technique [44].

accesses made to dynamically behaved loop execution. Our scheme will not benefit greatly from applications with a small portion of accesses going to data consumed by the dynamic execution. In our studies, applications are almost dominated by dynamically accessed data, and more than half of data accesses are made to dynamic execution flows. Some applications may have very high percentages of dynamically executed accesses but cannot be easily placed as a whole data cluster due to allocation pressure from limited SPM sizes. This is the case in 2k SPM.

Fig. 12 also shows the effect of increasing SPM size in energy consumption from our approach. The average energy saving from our method varies from 14.4% to 51.3%, when the SPM size is varied from 2k to 8k. In our studies, our approach is always better than the comparison. It is not a surprising result since multimedia applications have lots of dynamic memory accesses. It is also important to note that increasing SPM size sometimes gives a relatively small additional benefit on average. The reason is that a small fraction of data clusters are frequently reused in some cases. Such a case is also seen for caches. A very large cache does not always lead to great performance improvement than a moderate size of cache [45].

This experiment implicitly shows relative memory accesses to dynamically accessed data going to main memory after applying our approach. Sometimes, main memory to SPM copying codes (usually DMA call) increase the number of main memory accesses, but it is reduced much more by the improved locality afforded by SPM. Thus, the average reduction across benchmarks is a significant 36% reduction in main memory accesses from our method. In our observation, the proposed approach was able to make a decision that placed many important data clusters into SPM with minimizing transfers. This was sometimes correlated with a small increase in transfers for less important data, which were evicted to make space for the more frequently accessed data. That explains the high reduction in main memory accesses for many benchmarks.

C. Comparison With Caches

This section compares the energy efficiency of software/hardware steered data reorganization approach versus alternative architectures using a hardware cache controller with the least recently used (LRU) replacement policy. It is important to note that our method is useful regardless of this comparison because there are a great number of embedded architectures which have SPM and main memory but have no data cache. These architectures are popular because SPMs are simple to design and verify, and provide better real-time guarantees for global and stack data, power consumption, and cost [1], [8], [14] compared to caches. Nevertheless, it is interesting to see how our method compares against processors containing caches.

We examined the effect of using the SPM on the reduction of traffic to main memory as well as on the energy spent in the memory subsystem. The sizes of the SPM and the cache have been selected to be the closest values that are powers of two. The cache line size has been selected to be the minimal allowed by the simulator [38] (8 bytes, which is two data elements in all benchmarks). In this way, we compare solely how well is the data reusability of the data exploited without considering spatial locality issues.

Fig. 13 shows the impact of our approach on energy reduction over varying cache associativity and size. Energy consumption of each cache configuration is the base line (100%). Each bar represents energy consumption of SPM of the same size. The energy savings when using SPM in comparison with cache arise from two sources. It can be seen that the SPM consumes less energy than a cache of the same size per access (about two times less for direct mapped cache [40]). It is possible to make better data placement decisions for the SPM compared to that made according to the LRU policy of the cache controller, which results in less accesses to the main memory.

In the case of direct mapped cache, the SPM data placement is always better for all studied cases. In other cases, 8-way set associative is the best. However, the energy consumption per access is much higher than SPM. As a result, the scratch pad-based memory subsystem consumes 11%–49% less energy than the system with a cache of the same size. From the results, we see that the hardware-controlled cache-only approach performs significantly worse than ours on average, and we found that very small caches perform very poorly for applications. The additional hardware components such as DART and address translator do not significantly affect the proposed approach since they are simple buffer memories. Therefore, the energy consumption of the proposed approach is always lower than all cache configurations. These results show that, on the average (i.e., across all benchmarks and cache configurations we experimented with), using SPM instead of a conventional cache decreases the total data access energy by 36.2% (assuming that the cache and SPM have the same capacity).

We also observe that increasing the associativity from 1 to 4 improves cache performance, whereas going from 4 to 8 in general degrades the performance of the conventional cache version. The reason is that the increment of hardware complexity (overhead) factor we used is not amortized by a significant drop in the number of conflict misses as a result of increased associativity. These results clearly show that, even under the higher associativities, the SPM version significantly outperforms the cache version. Benini *et al.* [46] also reported the similar results.

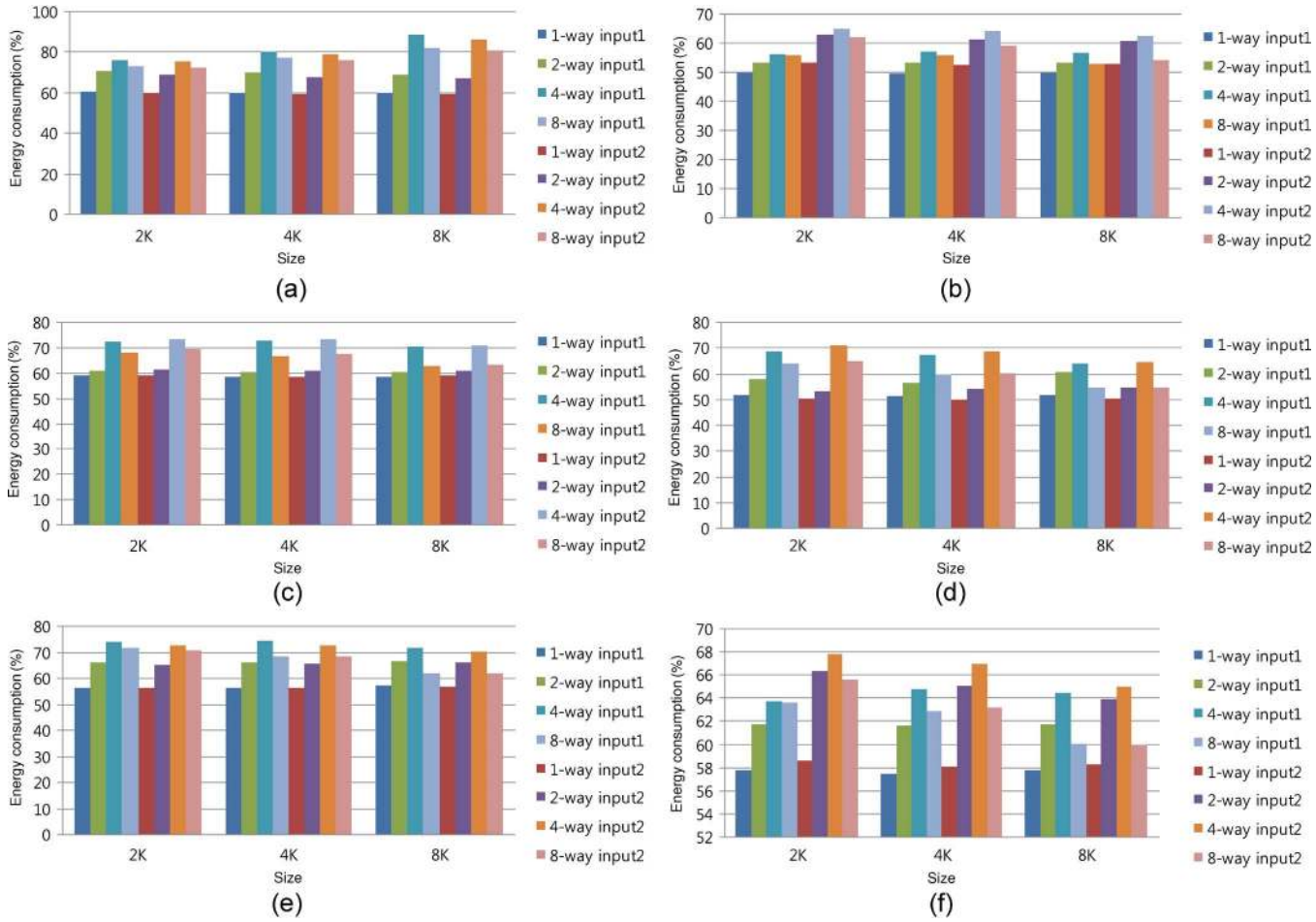

Fig. 13. Energy savings with various sizes of cache.

From this experiment, we see that our SPM management method not only shares similarities with a cache memory design but also has some important differences. Like caches, our method gives preference to more frequently accessed variables by allocating them more space in SPM. One advantage of our technique is that it avoids copying infrequently used data to fast memory; a cache copies in infrequent data when accessed, possibly evicting frequent data. One downside of our technique is that ours is based on profiling results. Thus, the accuracy of the preference to more frequently accessed variables depends on the coverage of the possible memory access histories generated through profiling. To increase the accuracy, we did profiling multiple times with representative input data sets to cover all possible (realistic) memory access histories. By doing this, our approach can take 98% accuracy in our studies.

D. Effect of Energy-Delay Cost With Varying Adaptation Period

As earlier mentioned, in the layout adaptation procedure, the proposed approach yields two kinds of latency, which are data transfer latency (occurred in layout reorganization) and comparison latency (occurred in tracking of the dynamic behavior). These latencies are proportional to the decision of invocation period of Remap() call. Both costs can be balanced by adjusting frequency of Remap() call. Thus, the cost of layout adaptation should be analyzed to prove the system's efficacy. The algorithm described in Section VII-A is used to find a practically good period.

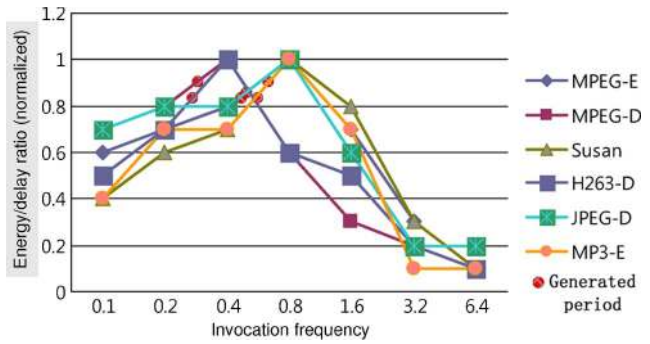

Fig. 14. Effect on the various adaptation periods.

We performed experiments with the two inputs, as shown in Table I, over the direct mapped cache. The average is shown in Fig. 14. The results show the impact of various invocation periods on the energy-delay ratio. We plot the normalized energy-delay ratio on the y-axis (the value one is the best) and the invocation frequency on the x-axis. The frequency represents a rate over the outer most loop-iteration length; the rate is the outer most loop-iteration length over invocation period length.

As shown in Fig. 14, the longest (0.1) period may miss a lot of dynamic access behavior. As a result, it misses reuse advantages in part of the clusters; thus, the energy saving is slightly decreased. The shortest (6.4) period has a large comparison overhead and frequent DMA calls with partial data clusters. Thus, it has a larger delay and energy overhead. We observe from these results that eliminating the extra off-chip memory accesses (due to the frequent DMA call with small array slices) helps to improve energy-delay ratio in all benchmarks. Fig. 14 shows that the periods generated by our approach range from 0.83 to 0.91 on the y-axis (energy-delay ratio). In particular, the largest energy savings are obtained for the MPEG-D and Susan codes, with generated periods of 0.91 and 0.9 (energy-delay ratio), respectively. As expected, our approach generates good-enough savings with small delay. The results clearly emphasize the importance of selecting an appropriate invocation period.

IX. Conclusion

In this paper, we presented a method to improve the efficiency of SPM management in the memory hierarchy by tracking dynamic memory access behavior due to input data patterns to execute irregular memory access and/or different control flow in applications. By tracking memory usage, our proposed approach allows more frequently accessed data to remain in SPM longer and therefore have a larger chance of reuse. The reuse block choices are made with the help of a DART, which records dynamic reference analysis in a location-sensitive manner. We also introduced the concept of a WML, which allows the DART to feasibly characterize the accessed memory locations.

This paper complements conventional SPM management methods to enable tracking of the runtime behavior of media applications. The proposed method can be used with existing optimizations for memory subsystem as well as locality optimizations or transformations. In addition, if more input data are used, our approach can provide even more benefits.

For future work, we will examine more sophisticated memory access pattern scenarios, analyze more sophisticated selection algorithms, and improve our adaptive SPM management scheme. In general, we believe that the schemes presented in this paper can be extended into a more general framework for intelligent runtime management of the SPM memory hierarchy.

References

[1] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proc. 10th Int. Workshop Hardware/Software Codesign, CODES*, Estes Park, CO, May 2002, pp. 73–78.

[2] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratch-pad memory for energy minimization," in *Proc. 2nd IEEE/ACM/IFIP Int. Conf. CODES+ISSS*, 2004, pp. 104–109.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[4] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, Jul. 1996.

[5] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, no. 1, pp. 6–26, Nov. 2002.

[6] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proc. EDTC*, 1997, p. 7.

[7] J. Sjödin and C. von Platen, "Storage allocation for embedded processors," in *Proc. Int. Conf. CASES*, 2001, pp. 15–23.

[8] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. Conf. DATE*, 2002, p. 409.

[9] M. Verma, S. Steinke, and P. Marwedel, "Data partitioning for maximal scratchpad usage," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2003, pp. 77–83.

[10] K. D. Cooper and T. J. Harvey, "Compiler-controlled memory," in *Proc. Architectural Support Program. Lang. Oper. Syst.*, 1998, pp. 2–11.

[11] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 2, p. 15, 2007.

[12] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proc. Des. Autom. Conf.*, 2001, pp. 690–695.

[13] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embed. Comput.*, vol. 1, no. 4, pp. 521–540, Dec. 2005.

[14] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. Int. Conf. CASES*, 2003, pp. 276–286.

[15] O. Ozturk, G. Chen, M. Kandemir, and M. Karakoy, "Compiler-directed variable latency aware SPM management to cope with timing problems," in *Proc. Int. Symp. CGO*, 2007, pp. 232–243.

[16] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An integrated hardware/software approach for run-time scratchpad management," in *Proc. 41st Annu. DAC*, 2004, pp. 238–243.

[17] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip," in *Proc. 14th ISSS*, 2001, pp. 19–24.

[18] M. Kandemir and N. Dutt, *Memory Systems and Compiler Support for MPSoC Architectures*. San Mateo, CA: Morgan Kaufmann, 2005.

[19] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *Proc. 43rd Annu. DAC*, 2006, pp. 49–52.

[20] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," in *Proc. 14th Int. Conf. PACT*, 2005, pp. 329–338.

[21] M. J. Absar and F. Catthoor, "Compiler-based approach for exploiting scratch-pad in presence of irregular array access," in *Proc. Conf. DATE*, 2005, pp. 1162–1167.

[22] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy, "Dynamic scratch-pad memory management for irregular array access patterns," in *Proc. Conf. DATE*, 2006, pp. 931–936.

[23] C. A. Moritz, M. Frank, and S. P. Amarasinghe, "Flexcache: A framework for flexible compiler generated data caching," in *Proc. Revised Papers 2nd Int. Workshop IMS*, 2001, pp. 135–146.

[24] C. M. Huneycutt, J. B. Fryman, and K. M. Mackenzie, "Software caching using dynamic binary rewriting for embedded devices," in *Proc. ICPP*, 2002, p. 621.

[25] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proc. 27th Annu. ISCA*, 2000, pp. 107–116.

[26] M. Kandemir and I. Kadayif, "Compiler-directed selection of dynamic memory layouts," in *Proc. 9th Int. Symp. CODES*, 2001, pp. 219–224.

[27] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," *SIGPLAN Not.*, vol. 34, no. 5, pp. 229–241, May 1999.

[28] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel Distrib. Comput.*, vol. 8, no. 4, pp. 303–312, Apr. 1990.

[29] *On-Chip Coreconnect Bus Architecture*, New York: IBM. [Online]. Available: http://www.chips.ibm.com/products/coreconnect/index.html

[30] *Arm Advanced Micro Bus Architecture (amba)*, ARM. [Online]. Available: http://www.arm.com/products/solutions/AMBAHomePage.html

[31] *Sonics, Integration Architectures*. [Online]. Available: http://www.sonicsinc.com

[32] *Wishbone System-on-Chip Interconnection Architecture for Portable ip Cores Silicore and Opencores*. [Online]. Available: http://www.opercores.org

[33] G. Bernat, A. Colin, and S. Petters, "PWCET: A tool for probabilistic worstcase execution time analysis of real-time systems," Univ. York, York, U.K., 2003. Tech. Rep.

[34] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for mpsoc," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 3, pp. 1–18, Jul. 2008.
[35] I. Issenin and N. Dutt, "Data reuse driven energy-aware mpsoc co-synthesis of memory and communication architecture for streaming applications," in *Proc. 4th Int. Conf. CODES+ISSS*, 2006, pp. 294–299.
[36] A. Felner and S. Kraus, "Kbfs: K-best-first search," in *Ann. Math. Artif. Intell.*, 2003, pp. 19–39.
[37] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 330–335.
[38] D. C. Burger and T. M. Austin, "The simplescalar tool set," Tech. Rep. CS-TR-1997-1342, 1997, version 2.0.
[39] N. Manjikian, "Multiprocessor enhancements of the simplescalar tool set," *SIGARCH Comput. Archit. News*, vol. 29, no. 1, pp. 8–15, Mar. 2001.
[40] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," HP Labs, Palo Alto, CA, Tech. Rep. WRL-2001-2.
[41] *Cacti 3.2. p. shivaumar and n.p. jouppi*, COMPAQ. revised 2004. [Online]. Available: http://research.compaq.com/wrl/people/-jouppi/CACTI.html
[42] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. ISCA*, 2000, pp. 83–94.
[43] *128 mbit Micron Mobile Sdram Data Sheet*, Micron Technol. Incorporated, Boise, ID. [Online]. Available: http://www.micron.com
[44] J. Absar and F. Catthoor, "Reuse analysis of indirectly indexed arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 2, pp. 282–305, Apr. 2006.
[45] J. Hennessy and D. Patterson, *Computer Architecture a Quantitative Approach*, 3rd ed. Palo Alto, CA: Morgan Kaufmann, 2002.
[46] L. Benini, A. Macii, E. Macii, and M. Poncino, "Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation," *IEEE Des. Test*, vol. 17, no. 2, pp. 74–85, Apr./Jun. 2000.

Doosan Cho (M'06) received the B.S. degree in digital information engineering from Hankuk University of Foreign Studies, Seoul, Korea, in 2001, and the M.S. degree in electrical engineering from Korea University, Seoul, in 2003. He is currently working toward the Ph.D. degree in electrical engineering and computer science at Seoul National University, Seoul.

He is the author of some papers in his areas of research interest, which include memory subsystem optimization and embedded system optimization techniques. His research interests lie at the intersection of compilers and computer architecture, with a particular interest in embedded systems.

Sudeep Pasricha (M'02) received the B.E. degree in electronics and communication engineering from Delhi Institute of Technology, Delhi, India, in 2000, and the M.S. and Ph.D. degrees in computer science from the University of California, Irvine, in 2005 and 2008, respectively.

He has four years of work experience in semiconductor companies, including STMicroelectronics and Conexant. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins. He has published more than 35 research articles in peer-reviewed conferences and journals and presented several tutorials in the area of on-chip communication architecture design at leading conferences. He recently coauthored a book entitled *On-chip Communication Architectures* (Morgan Kauffman/Elsevier 2008). His research interests are in the areas of multiprocessor embedded system design, networks-on-chip and emerging interconnect technologies, system-level modeling languages and design methodologies, and computer architecture.

Dr. Pasricha is currently in the program committee of the ISQED and VLSID conferences. He has received a Best Paper Award at ASPDAC 2006, a Best Paper Award nomination at DAC 2005, and several fellowships and awards for excellence in research.

Ilya Issenin (M'97) received the degree in electrical engineering from the Moscow Engineering Physics Institute, Moscow, Russia, in 1998, and the M.S. and Ph.D. degrees in information and computer science from the University of California, Irvine, in 2001 and 2007, respectively.

He is currently with Teradek, Irvine, CA, working on low-power video designs. He is also with the Center for Embedded Computer Systems, Department of Computer Science, Donald Bren School of Information and Computer Sciences, University of California, Irvine. He is the author of a number of papers in his areas of research interest, which include memory and communication subsystems optimization and synthesis, and low-power architecture optimization techniques.

Nikil D. Dutt (S'84–M'89–SM'96–F'08) received the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Urbana, in 1989.

He is currently a Chancellor's Professor with the University of California, Irvine, with academic appointments in the Center for Embedded Computer Systems, Department of Computer Science, Donald Bren School of Information and Computer Sciences and in the Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering. His research interests are in embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems.

Dr. Dutt was an ACM Special Interest Group on Design Automation distinguished Lecturer during 2001–2002 and an IEEE Computer Society distinguished Visitor for 2003–2005. He has served and currently serves on the steering, organizing, and program committees of numerous premier conferences and workshops. He also serves as the Associate Editors and Editor-in-Chief for various journals and transactions. He is an ACM distinguished Scientist and an IFIP Silver Core awardee. He received the Best Paper Awards at the Conference on Computer Hardware Description Languages and Their Applications (CHDL)89, CHDL91, VLSIDesign2003, CODES + ISSS 2003, IEEE Consumer Communications and Networking Conference 2006, and ASPDAC-2006.

Minwook Ahn received the Ph.D. degree from Seoul National University, Seoul, Korea, in 2009.

He is currently a Postdoctoral Researcher with the Department of Electrical Engineering and Computer Science, Seoul National University. His research interests are in embedded systems, optimizing compiler, computer architecture, and reconfigurable devices.

Yunheung Paek (M'99) received the B.S. and M.S. degrees in computer engineering from Seoul National University (SNU), Seoul, Korea, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Urbana.

He gained a national scholarship from the Ministry of Education. He is currently a Full Professor with the School of Electrical Engineering, SNU. Before he came to SNU, he was with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon, Korea, as an Associate Professor for one year and an Assistant Professor for two and a half years. He has served on the program committees and organizing committees for numerous conferences and also worked for editors for various journals and transactions. His current research interests include embedded software, embedded system development tools, retargetable compiler, and MPSoC.