Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns

Doosan Cho
School of EECS
Seoul National University, Korea
dscho@optimizer.snu.ac.kr

Sudeep Pasricha
Department of Information and Computer Science
University of California, Irvine
sudeep@ics.uci.edu

Ilya Issenin
Department of Information and Computer Science
University of California, Irvine
isse@ics.uci.edu

Nikil Dutt
Department of Information and Computer Science
University of California, Irvine
dutt@ics.uci.edu

Yunheung Paek
School of EECS
Seoul National University, Korea
ypaek@snu.ac.kr

SunJun Ko
Senior Engineer
Samsung Electro-Mechanics Co.,LTD
sunjun.ko@samsung.com

Abstract

Embedded multimedia applications consist of regular and irregular memory access patterns. Particularly, irregular pattern are not amenable to static analysis for extraction of access patterns, and thus prevent efficient use of a Scratch Pad Memory (SPM) hierarchy for performance and energy improvements. To resolve this, we present a compiler strategy to optimize data layout in regular/irregular multimedia applications running on embedded multiprocessor environments. The goal is to maximize the amount of accesses to the SPM over the entire system which leads to a reduction in the energy consumption of the system. This is achieved by optimizing data placement of application-wide reused data so that it resides in the SPMs of processing elements. Specifically, our scheme is based on a profiling that generates a memory access footprint. The memory access footprint is used to identify data elements with fine granularity that can profitably be placed in the SPMs to maximize performance and energy gains. We present a heuristic approach that efficiently exploits the SPMs using memory access footprint. Our experimental results show that our approach is able to reduce energy consumption by 30% and improve performance by 18% over cache based memory subsystems for various multimedia applications.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compiler and optimization

General Terms Algorithms, Design, Experimentation

Keywords Compiler, Memory Hierarchy, Energy Consumption, Data Placement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES'08, June 12–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-60558-104-0/08/06. . . $5.00.

1. Introduction

As embedded systems grow more complex and large to satisfy diverse demands from the market, the processor-memory speed gap is becoming a critical design issue. Since the increase in memory access speed has not kept up with increases in processor speed, memory access contention has increased, resulting in a longer memory access latency in systems today. This makes the memory access cost much greater than the computation cost. In Multi-Processor Systems-on-Chip (MPSoCs), the effect of the widening processor-memory speed gap becomes more significant due to the heavier access contention on the network and the use of shared memory. Thus, 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, a hardware-only cache implementation has several drawbacks. The hardware controlled approach incurs high power and area cost [1]. Moreover, lack of knowledge of future accesses may lead to higher miss rates due to non-optimal data placement in the caches. Besides, it is not possible to achieve effective data prefetching (which helps to hide the access latency) since not all of the programs expose sufficient spatial locality in the data accesses. As a result, it is often unacceptable to use caches because of their unpredictable latency for real embedded applications [2].

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 a 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 [3]. In addition, SPM often allows static timing analysis and thus provides better time predictability required in real-time systems. 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.

Previous work on SPM utilization has focused on development of approaches for efficiently assigning frequently accessed data and instructions to SPM in order to maximize improvement of performance and energy consumption. Many of the approaches for data array placement in SPMs have focused on applications with regular data access patterns. While such regular applications are abundant in the embedded space, there also exist a number of embedded applications whose array access patterns are not analyzed well with accurate compiler static analysis and optimization (we name these as ”irregular applications”). Consequently, it is difficult to achieve efficient SPM hierarchy utilization with such irregular applications. To overcome this difficulty, we present a general purpose scheme for both memory access patterns. Our scheme is a dynamic data reorganization method guided by a profiler. It generates improvements in performance and energy when the gain obtained from reduction in the number of off-chip memory accesses outweighs the cost of data transfers between the SPM and processor/main memory. To that end, our approach solves the following two problems:

1. Identifying parts of the data array that can be copied into the SPM for improving performance and energy consumption, and

2. Maximizing utilization of the SPM space using the selected data elements

We propose a technique for optimizing data placement of application-wide reused data so that it resides in the SPMs of processing elements. Our technique identifies data elements with fine granularity that can profitably be placed in the SPMs to maximize performance and energy gains. We present a heuristic approach that efficiently exploits the SPMs using memory access footprint. Our experimental results indicate that our approach is able to reduce energy consumption by 30% and improve performance by 18% over cache based memory subsystems for various multimedia applications.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes our architecture model. Section 4 presents our approach for data layout reorganization. Section 5 describes how to reorganize data layout using a DMA engine at runtime. Section 6 describes the methodology and experimental setup used to assess the efficiency of our approach and presents detailed simulation results and analysis. Finally, the conclusion is presented in Section 7.

2. Related work

Many papers have focussed on the problem of improving data reuse in caches, primarily by means of loop transformations (e.g. [4, 5]). 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.

There are several prior studies on using SPM for data accesses. The studies can be divided into two parts, static and dynamic. Static methods [6, 1, 7, 8, 9, 10] 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 non-optimal 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 [11, 12, 2, 13, 14, 3]. Cooper et al. [11] proposed using SPM for storing spilled values. Udayakumaran et al. [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 at al. [2] addresses 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 integer-linear programming formulations or similar methods to register allocation to find an optimal dynamic allocation.

A few approaches have looked at memory system design for MPSoCs [15, 16, 17]. Meftali et al. [15] and Kandemir et al. [16] proposed an optimal memory allocation technique based on ILP for application specific SoCs. Issenin et al. [17] 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 above focused explicitly on regular access patterns, Verma et al [10] and Li et al. [18] proposed approaches that work with irregular array access pattern. Verma et al. [10] 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 [10], 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 run-time in our approach as compared to static placement in [10]. Li et al. [18] 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 pseudo registers, live-range splitting to insert copy statements in an application code, and memory coloring to assign split array slices into the pseudo registers in SPM. However, their approach is prone to internal memory fragmentation when the size of assigned array slices are less than pseudo register size (where the partitioned SPM space). They try to solve this problem by making several sizes of pseudo registers. But, 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 three-dimensional (time, memory hierarchy as a two dimensional space) knapsack problem, that can assign array slices to SPM without any internal fragmentation.

Absar et al. [19] and Chen et al. [20] 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 non-affine reference functions. Additionally, our approach also considers all other types of irregular accesses found in various media applications.

3. Architecture Model

In a general on-chip system, multiple IPs share one on-chip bus to access system memory. Each programmable IP has a DMA (direct memory access) 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 Scratch Pad Memory (SPM), as shown in shown in Figure 1. We restrict the network architecture to a bus-based architecture since it is still the most popular network [21, 22]. 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, such as in the Cell processor [23]. In addition, each processing element executes its own task and the

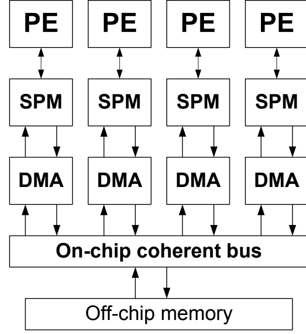

Figure 1. A simplified multi-processor SoC

processing elements share local memory address space. Therefore, we optimize data layout on the on-chip memory address space (local + remote). In order to optimize data layout in the architecture, intra/inter processor data reuse should be considered. They are very common in many array-intensive multimedia applications. Exploiting both data reuses focus on the problem of optimizing data layout considering access patterns of all processors in the system.

4. Strategy for Data Layout Decision

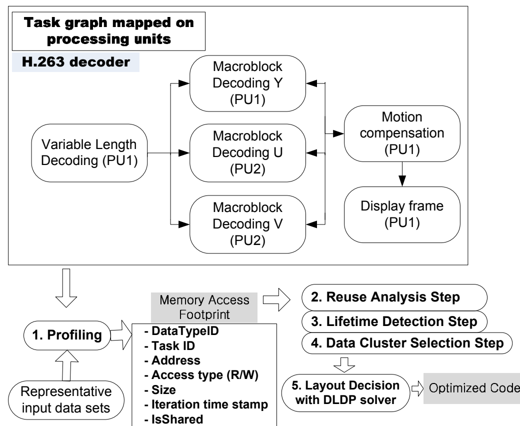

Figure 2. The overall flow of the proposed approach

In order to efficiently solve the data layout decision problem, we break it into two smaller problems. The first problem is to select beneficial array elements that will be copied onto the SPM, which have high data reusability with similar lifetimes. The second problem is to find an optimal layout of data elements in the SPM address space to minimize address fragmentation. Although the second problem is known to be NP-complete [3], we employ heuristics and are able to find near-optimal solutions for both the first and the second problems in polynomial time.

The overall flow of our approach is shown in Figure 2. The input C code is first parallelized at the task level and then mapped to the processing units in the system. For instance, Figure 2 shows the task graph [24] of the H.263 decoder which has been mapped onto processing units. The optimal mapping problem is beyond the scope of this paper so we assume that the mapping has already been performed by a compiler.

The compiler analyzes an application through five steps. First, profiling is used for gathering the array access footprint. During profiling, each program task is run multiple times with representative input data to collect the number of accesses to each array element, and an average is obtained. In the second and third steps, reuse analysis and lifetime detection with the access footprints are performed. 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. By doing so, optimized code with the array slices (clusters) are generated. The following subsections describe this flow in detail.

4.1 Data Selection using Data Reusability and Lifetime Analysis

The usefulness of a memory hierarchy depends on the amount of reuse in data accesses. In MPSoCs, there are two types of data reuse, intra-processor and inter-processor. The intra-reuse refers to the data reuse that occurs across the loop iterations of a task assigned to the same processing element, whereas the inter-reuse represents the data reuse that take place across the loop iterations of tasks assigned to different processing elements. To measure the amount of both reuses, 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 the data reusability factor for the 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 as:

DEFINITION 1. **Data reusability factor (DRF):** *Given the access frequency F corresponding to each array element and the estimated element size of N words, its DRF value T is defined as F/N. The DRF of shared data is referred to as inter-DRF, while the DRF of of unshared (local) data is referred to as intra-DRF.*

Our technique selects candidates of data to be located in SPM when data elements have DRF 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 the SPM as a data cluster. Before we define a data cluster, we first introduce the terms iteration vector, lifetime, and lifetime distance.

DEFINITION 2. **Iteration Vector***: Given a nest of n loops, the iteration vector i of a certain iteration of the innermost loop represents a integer vector that includes the iteration numbers for each loop in order of nesting level. In other words, the iteration vector is given by*

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

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.

DEFINITION 3. **Life Time (LT)***: Lifetime of a data element d is defined as the difference between the iteration vector of its first access (FA) and its last access (LA). It represents loop execution time duration, which is a non-empty interval in a loop iteration space.*

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

DEFINITION 4. **LT-Distance (LT-D)***: The lifetime distance is the difference in lifetimes of two elements in a data array.*

LT-D = $|LT(n_a) - LT(n_b)|$, *where the* n_a *and* n_b *represent* a_{th} *and* b_{th} *elements in a data array* 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 5. **Data cluster**: *A data cluster is a union of data elements that have the most beneficial LT-D in a data block.*

The following subsections describe the cluster generation procedures for both regular and irregular memory access patterns. They are orthogonally applied to each type of array element, which may be shared or non-shared. Thus, the generated clusters can be classified into two types: shared and non-shared.

4.1.1 Data clustering for a regular access pattern

The data clustering algorithm is shown in Figure 3.

```
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 DRF
   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
```

Figure 3. Data clustering algorithm for regular access pattern

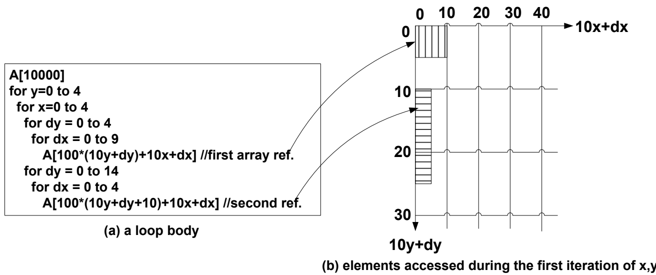

Figure 4. An example access trace with an input

We illustrate our procedure for cluster generation with the program example presented in Figure 4(a), which is extracted from the SUSAN algorithm for image noise filtering. For this program, Figure 4(b) shows the footprint of the addresses of accessed data elements with varying values of iterators y and x. 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, Figure 5 shows a set of 25 elements which is read three times during ten consecutive iterations of two outer loops. Moreover, other 25 element sets which are formed by shifting that set by integer numbers of steps behave exactly 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 these reused data elements can be copied onto 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 first and last accesses of the elements that are reused by the memory access trace. As shown in Figure 5, the candidate elements can be combined into a cluster. The clustering algorithm is performed as follows. The first step is to select one element out of the 25 elements since they have the highest reusability. We select the first element. In the second step, LTs of each element are calculated:

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

In the third step, LT-Ds of all elements with the selected (first) element are calculated:

$Second - First = (0,0,0,0)$
$Third - First = (0,0,0,0)$
...
$25th - First = (0,0,0,0)$

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 combined into 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 applications. In our studies, the beneficial LT-D for regular access patterns ranges from 0 to 5.

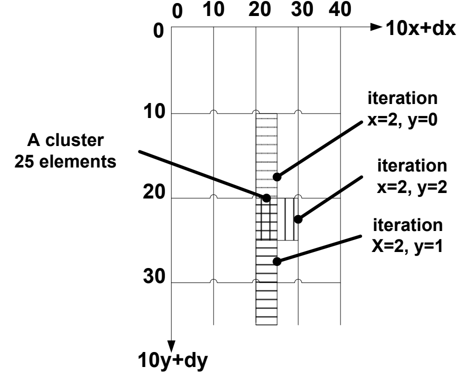

Figure 5. Cluster generation

4.1.2 Data clustering for an irregular access pattern

```
for(j=1; j<n-k+1; j++)
   ...
   for(i=0; i<n; i++)
      ...= S[j] ⊕ (a[((j − 1) ∗ (n − i − 1) + inva[tx[i]])%b]);
...
```

Figure 6. A code fragment with irregular array references

Figure 6 shows a loop code from the Reed-Solomon Error Control Code, which is used in a wide variety of commercial applications in storage devices, wireless communication and digital television. In the figure, four arrays (S, a, $inva$ and tx) are referenced. These four arrays are potential candidates for copying to the SPM. Arrays S and tx are regularly accessed (direct indexed array with affine reference functions), therefore, existing approaches can be used for them. Array $inva$ is indirectly accessed by regularly indexed array tx. It can be also optimized by an existing approach [19]. Since array a is irregularly accessed with a non-affine reference function, current techniques would try to copy the whole array to the scratch pad memory. But that may not be always pos-

sible since the array size can be larger than the SPM size. We are therefore interested in efficiently copying array a onto the SPM.

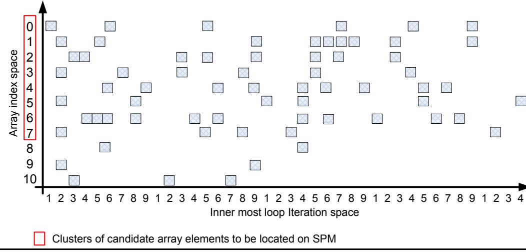

Figure 7. Array access footprint from the example loop kernel

The access pattern of array a is precisely known only at run-time. Therefore, we collect runtime information such as the arrays' access footprint and loop bounds. A part of array access footprint generated by profiling of the application is shown in Figure 7. The X-axis shows inner-most loop iteration numbers. An iteration is iterated by outer loops, thus, an iteration has several accesses. Based on the access footprint, we can get information such as data reusability, LT and LT-D of array elements. In this example, the beneficial LT-D is determined as 15 heuristically. Therefore, the array elements 0-7 are combined into a data cluster. In our studies, the beneficial LT-D of irregular access patterns ranges from 15 to 25. Clustering of an irregular access pattern however has an additional constraint, a regular stride constraint, which is defined below.

DEFINITION 6. **Regular stride constraint**: *The index of each array element in a data cluster should have a regular stride. Satisfying this constraint allows a DMA to transfer a cluster in an efficient way, such as in a burst mode.*

If Definition 6 is not satisfied, a cluster will be transferred using several DMA transfers and might incur a large latency waiting for each DMA completion. To avoid this, we add the constraint to our clustering procedure for irregular accesses. Regular stride detection is done using techniques proposed by Y. Wu et al. [25]. The data clustering algorithm for irregular access pattern is shown in Figure 8.

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 DRF
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. Select elements of the LT-D
1-5-1. Combine elements having regular stride among the candidate elements to data clusters
1-6. Remove the elements of the data clusters in the given set of candidates
1-7. Repeat the same procedure with remaining data elements until this procedure generates no data cluster
End

Figure 8. Data clustering algorithm for irregular access pattern

Using this algorithm, a set of data clusters are created for all loops of all the tasks in an application. However, all data clusters cannot be assigned to the SPM since the SPM size is usually small (e.g., 1K-16K bytes). This is essentially 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 SPM address space (i.e., memory fragmentation). The next subsection presents a formal definition of the data cluster allocation problem with the goal of minimizing the fragmentation and with a capacity constraint.

4.2 Data Layout Decision Problem (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 work, 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 three dimensional (time, space, access energy (local, remote, off-chip)) knapsack problem.

Our objective is to minimize the energy consumption within the memory subsystem, and this can be achieved in such a way that each processor can access the most frequently used data from the closest memory location. We assume that the entire data clusters manipulated by an application can be stored in the on-chip and off-chip memory hierarchy. The variables used in our formulation are described below. These are either architecture specific or application dependent.

- N_P: number of processing elements
- N_M: number of memory components including all on-chip and off-chip memories
- $N_{P,T}$: number of tasks assigned to the processing element $p \in P$
- $N_{T,L}$: number of loops belongs to the task $t \in T$
- IS_l: iteration space of loop $l \in L$ given by Definition 2
- D: a set of data clusters generated from an application
- $DRF_{p,t,is,d}$: intra/inter DRF of data cluster $d \in D$ at $is \in IS$ on a task t assigned to the processing element p
- AE_m: the energy required for an access to the memory component m
- ND_{is}: number of data clusters whose lifetime includes an iteration point $is \in IS$
- $D_{is,d}$: a set of data clusters whose lifetime includes an iteration point $is \in IS$

  Each data cluster d exists in its lifetime LT on a certain SPM or the off-chip memory. We use M to identify the location of a shared/non-shared data cluster during the course of loop execution. More specifically,
- $M_{m,is,d}$: indicates whether a data cluster d at a point of loop iteration space is is located on a certain memory component $m \in M$. It is a 0-1 variable whose values we want to determine.

DEFINITION 7. **Energy Equation** *E is used to capture the energy consumption of all processing elements with all tasks. That is,*

$$E = \sum_{p=1}^{N_P} \sum_{t=1}^{N_{P,T}} \sum_{l=1}^{N_{T,L}} \sum_{is=1}^{IS(l)} \sum_{d=1}^{ND_{is}} D_{is,d} \times DRF_{p,t,is,d} \times EP(is,d)$$

$$EP(is,d) = \sum_{m=1}^{N_M} M_{m,is,d} \times AE_m$$

In Definition 7, the equation shows the amount of energy consumption that is calculated by summing up the energy consumption due to data cluster accesses performed by all processing elements for an application. Shared/non-shared data clusters can be located

on local/remote SPM and off-chip memory. Obviously, non-shared clusters are accessed by a certain processor, thus, they belong to the local SPM. But, the scenario is not obvious for shared clusters. For shared clusters, a local SPM is the processor's one which contributes to the largest portion of inter-DRF of a shared cluster.

The placement decision depends on intra/inter DRF. The energy equation guides the data placement decision for improving the energy consumption based on the DRF and energy parameters represented by the EP function. There are two parameters that affect the energy consumption: location and memory size. Accessing non-local memory (remote and off-chip) requires a higher interconnect energy consumption. Similarly the size of the SPM being accessed is another key parameter for energy consumption. We use the function $EP(is, d)$ to capture the energy consumption of a specific access to a cluster d at an iteration point is. EP is expressed as a mathematical formulation by using specific arguments. In the above equation, the location is captured by $\sum_{m=0}^{N_M} M_{m,is,d}$, which returns 1 if a data cluster d is located on a memory component m at an iteration point is. $DRF_{p,t,is,d}$ returns intra/inter DRF for a shared/non-shared data cluster d. Finally, we can decide which data clusters should be kept in local/remote SPMs (in anticipation of future reuse) and which ones can be sent to the off-chip memory based on Definition 7.

DEFINITION 8. **Capacity Constraint***: Let the size of memory component m have a limited capacity C_m. For a set of assigned data clusters $d \in D_m$ to the memory component m, $\sum_{d \in D_m} Size(d)$ must not exceed the capacity of the C_m at each point of loop iteration $is \in IS$, where IS consists of a normalized loop iteration space as given in Definition 2.*

$\sum_{d=1}^{ND_{is}} size(d) \leq C_m \; \forall is$

DEFINITION 9. **Overhead***: Data transfer overhead is represented by $\sum_{m=1}^{N_M} \sum_{d \in D_m} O_d$, which gives the energy needed for data movement.*

DEFINITION 10. **Definition of the DLDP**

- Objective function: find an optimal layout of assigned set of data clusters which minimizes the energy consumption: $E + O$

- Subject to: the capacity constraint in Definition 8

The efficacy of energy consumption of an application in this architecture depends strongly on the proportion of data requests satisfied from the memory hierarchy (local, remote and off-chip memories). Developers typically want all data requests to be satisfied from the local memory (highest priority) or remote memory (normal priority), that is, as far as possible, they do not want to go to the off-chip memory. To achieve this, the objective function determines a mapping of data clusters to memories such that the energy consumption of the application is minimized subject to the capacity constraints on the available memories.

The one dimensional 0-1 (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 near-optimal solution by using a best-first search with a heuristic. In the next subsection we describe our approach to solve the DLDP.

4.3 The DLDP Solver

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

C.1) Divide Step: The cluster generation methods described in Section 4.1 are first used to generate a set of data clusters for an application. This creates an initial problem instance. Figure 9 shows an example of an initial problem instance, with cluster lifetimes and sizes. We subsequently simplify the initial problem instance using the following methods.

The simplification 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 Figure 9, which shows an instance of DLDP that has seven clusters, $d1, ..., d7$, that need to be allocated to SPMs (each with capacity=10). The X-axis represents the capacity. Results are shown for 10 iteration time units IS ={1,2,3,...,10} (y-axis).

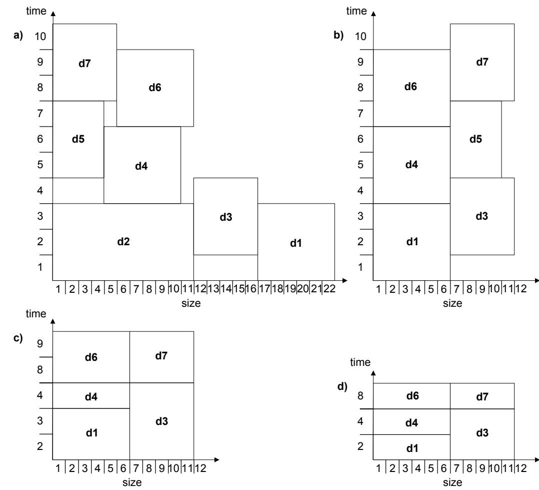

Figure 9. An instance of DLDP to which the reduce and split operators are applied

The reduce operator performs two kinds of simplifications. 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 Figure 9(a), block d2 has size 11 at time $IS = (1, 2, 3)$, yet the SPM capacity is only 10. So, the reduce operator removes d2 from the instance, which results in the instance shown in Figure 9(b).

The second kind of simplification removes unnecessary times from the instance. In the instance of Figure 9(b), at times 1, 5, 6, 7 and 10 the total size does not exceed the capacity. Since the constraint at these five times are satisfied in all assignments of the clusters, these times can be removed from the instance. Thus the reduce operator can also remove d5, resulting in the instance displayed in Figure 9(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 same capacity, then either time can be removed. In the instance of Figure 9(c), time 2 and 3 impose the same size constraint as do time 8 and 9. Thus, time 3 and 9 can be removed from the instance, resulting in the instance of Figure 9(d).

The split operator decomposes a problem instance into subproblems that can be solved independently. A split can be performed between any two adjacent iteration times, is and is', such that $D_{is} \cap D_{is'} == \phi$. In the instance of Figure 9(d), a split can be made in between time 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.

Figure 10 shows the procedure (Simplify()) that uses the Reduce and Split operations for a DLDP instance P. In the procedure, min and max(IS) returns the minimum and maximum iteration time $is \in IS$. D_{is} represents a set of data clusters whose LT includes the iteration time is, and $next(is)$ represents the next iteration time $is+1$.

```
Simplify(dldp* P){ //P: an instance of DLDP
  if(D == empty) then return;
  else {
    if IS >= 2 then {
      is_a = min(IS);
      while is_a != max(IS) do {
        is_b = next(is_a);
        // time reduction operation
        if Dis_a == Dis_b then {
          remove is from IS;
          for d ∈ Dis do
            remove is from LT(d);
        } else
            is_a = is_b;
      } //end of while
    }

    init(is_a); //initialize is_a to perform split operation
    while is_a != max(IS) do {
      is_b = next(is_a);
      // problem split operation
      if (next(is_a) == is_b) && (intersect(Dis_a, Dis_b) == empty){
        p1 = times{is | is <= is_a} and D{d | LT_end(d) <= is_a}
        p2 = times{is | is >= is_b} and D{d | LT_start(d) >= is_b}
      }
    } //end of while
  }
}
```

Figure 10. A procedure of the divide step with simplification

C.2) The Conquer Step with the k-way Best First Search: In the previous subsection the Split procedure divides a problem instance (P) of DLDP into smaller problem instances (p_1,p_2,...,p_l). Each $p \in P$ is an objective of the k-way best first search [26] in the conquer step. The k-way best-first search is used to search for the near-optimal cluster ordering in on-chip memory address space of the SPMs. 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. The 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 consumption and the minimum energy consumption on the objective function for the DLDP instance.

DEFINITION 11. **Metric for searching:**
$E_{MIN} = min_{d' \in children(d)}(current_{MIN} + child(d, d'))$
$E_{MAX} = max_{d' \in children(d)}(current_{MAX} + child(d, d'))$
where child(d, d') is the energy consumption 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 (1) selects an unprocessed cluster, (2) processes the cluster and then creates its k best number of children, and (3) propagates new max and min values by Definition 11 through the tree and uses these values 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 clusters which have the k-best values. Let us now consider the three major steps in more detail.

The first step is to find the node to process next. 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-best values at unobserved clusters. Our implementation orders the children from left to right so that their values are non-decreasing with a priority queue.

The second step is to process and expand the node. For each of these unobserved nodes, maximum and minimum energy consumption on its objective function is obtained, and k best unobserved clusters 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 $current_{MIN}$ is stored to a solution set.

The third step is to propagate the new energy consumption and prune the tree. Starting at the nodes just created and working up the tree to the root, the value of the maximum energy consumption and the minimum energy consumption are updated for each node. As this stage assigns and reassigns energy consumption, it checks to see if any node has one child whose minimum energy consumption exceeds the maximum energy consumption of the other child. In such a case the cluster of minimum can be no better than that of the cluster of maximum, so the cluster of minimum and all its descendants are removed from the tree. Finally, this search procedure produces a placement of clusters as a near-optimal solution.

5. DMA for preloading clusters

The layouts including dynamically allocated clusters have to be loaded onto the SPMs at runtime. For that, we make use of a DMA engine to minimize any latency due to the movement. The utilization of DMA component requires a co-operation of hardware/software. The software determines whether to utilize the DMA for any given transfer and send commands to do the transfer. Then, the hardware generates transfer completion signals (interrupt) of DMA so that the software can synchronize while still working in other tasks during the process. To achieve this, we use a data transfer function that decides how to execute the data transfer (using a DMA resource, giving a high priority to the transfer), at runtime, depending on the generated clusters layout.

```
function TRANSFERCOPY()
  Perform data transfer by DMAjob(args)
  /*args: width, length, source, destination, etc.*/
end function

functionDMAjob(source, destination, stride)
  dma16(source, destination, stride)
  CheckDMAstat(TAG_UPDATE_ALL); //wait for DMA completion
end function
```

Figure 11. DMA routines

The DMA routines are used to specify the memory transfers between the SPM and the main memory (TransferCopy() shown in Figure 11). 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.

5.1 Address Translation at runtime

The generation of small sized clusters can improve the utilization of the SPM. However, it may also make it extremely difficult to generate an address translation code for transformed data references and hinder optimization of the address generation code. To avoid this overhead, an additional address translation buffer is used (as shown in Figure 12) to dynamically translate a memory access to the desired address decided by our DLDP solver. This address translator

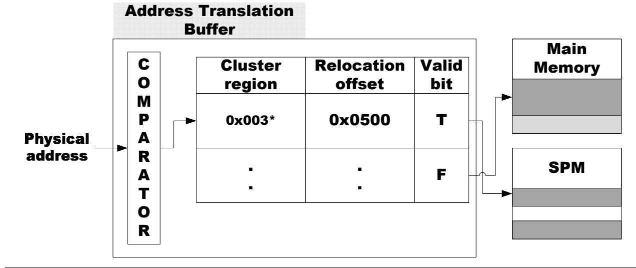

Figure 12. The address translation

is implemented by a set of registers. These registers are built with high speed logic to make the address translation efficient. Every access to memory must go through the address translator. The CPU dispatcher reloads these registers like that it reloads the other registers. Only the operating system can change the memory map by privileged instructions to load or modify the relocation registers.

For correct SPM addressing, each entry of the translator records the address region of each cluster. The OS initializes and updates the table entries when the application is loaded. The difference between the main memory and SPM address of a cluster is the relocation offset. On an access, the comparator tests whether the memory access belongs to a cluster region in the buffer. If it does, then the address translation is applied. 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 has been transferred to the SPM by the DMA. The buffer has a valid bit which indicates whether data transfer to the corresponding SPM has completed. The bit is triggered by a signal on DMA completion.

6. Experiments

We conducted several experiments to assess the effectiveness of our proposed approach. We explored how much our approach influences the energy consumption while minimizing the overhead. We chose to base our analysis upon benchmark traces taken from Mediabench [27]. The goal of our experiments was to compare our approach for the SPM based subsystem against a traditional cache based memory subsystem and to estimate the ability of these approaches to exploit data reusability of the data accesses for a number of multimedia applications.

6.1 Experimental setup

We created a tool set that implemented our proposed technique. The tool provided us with the code versions that implemented necessary data transfers between the SPM and the main memory for the selected clusters. We used a Pentium 4 workstation for profiling purposes for each task. The SimpleScalar simulator [28] based CMP simulator (with four processing elements) was used for obtaining the number of misses for the caches. An external DRAM with 18-cycle latency and a local SRAM (scratch-pad) with 1-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 [29]. We used CACTI [29] for energy estimation of both the cache and SPM at 90nm technology. To generate SDRAM energy parameters, a detailed model of a mobile SDRAM from Micron Technologies [30], (MT48V2M32LFC-8 2.5V) was selected for our experiments.

The input to the experiments were a set of codes obtained from MediaBench [27] with various size (7.2KB-504KB) of input data. A brief description of the applications is given in Table 1. The benchmarks were compiled with gcc with optimizations turned on (-O2). The experiments were performed with 2,4 and 8K SPM and with the same size of the cache (1-8 way set-associative).

Table 1. Program codes and inputs

Program	Abbreviation	Description
MPEG2 Encocder	MPEG-E	The generic coding of moving picture and associated audio
MPEG2 Decocder	MPEG-D	The generic decoding of moving picture and associated audio
Mp3 encoder (Lame)	MP3-E	MPEG-1 audio layer 3 audio encoding format
Jpeg decoder	JPEG-D	Decoder of the image compression
Susan	Susan	Image noise filter
H263 decoder	H263-D	The standard video codec

6.2 Results for energy and runtime reduction

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 goal was to evaluate the energy efficiency of our compiler driven data layout optimization compared to one implemented using a hardware cache controller with the LRU replacement policy. The sizes of the SPM and the cache were selected to be the closest values that are powers of two. The cache line size was selected to be the minimal allowed by the simulator [28] (8 bytes, which is 2 data elements in all benchmarks). In this way, we limited our comparison to the data reusability of the data exploited without considering spatial locality issues.

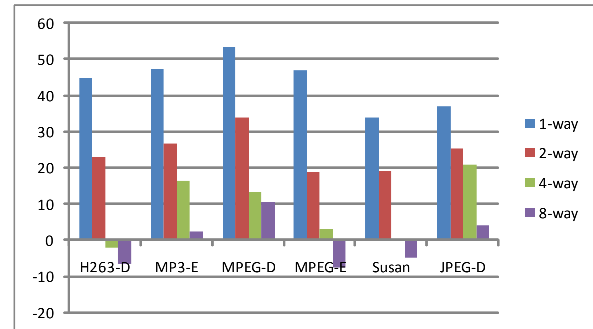

Figure 13. Runtime reduction

Figure 14 and Figure 13 shows the impact of our SPM management approach on energy and performance reduction, for varying cache associativity and size across the six benchmarks. In Figure 14, energy consumption of each cache configuration is considered as the base line (100%), and each bar represents energy consumption of the SPM of the same size as the cache. 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 [29]). The energy savings when using SPM are due to the more optimal data placement decisions that are possible for the SPM compared to that made according to the LRU policy of the cache controller, which result 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, the 8-way set associative is the best. But, the energy consumption per access is much higher than SPM. As a result, the SPM based memory subsystem consumes 9% to 40% less energy than the system with caches of the same size.

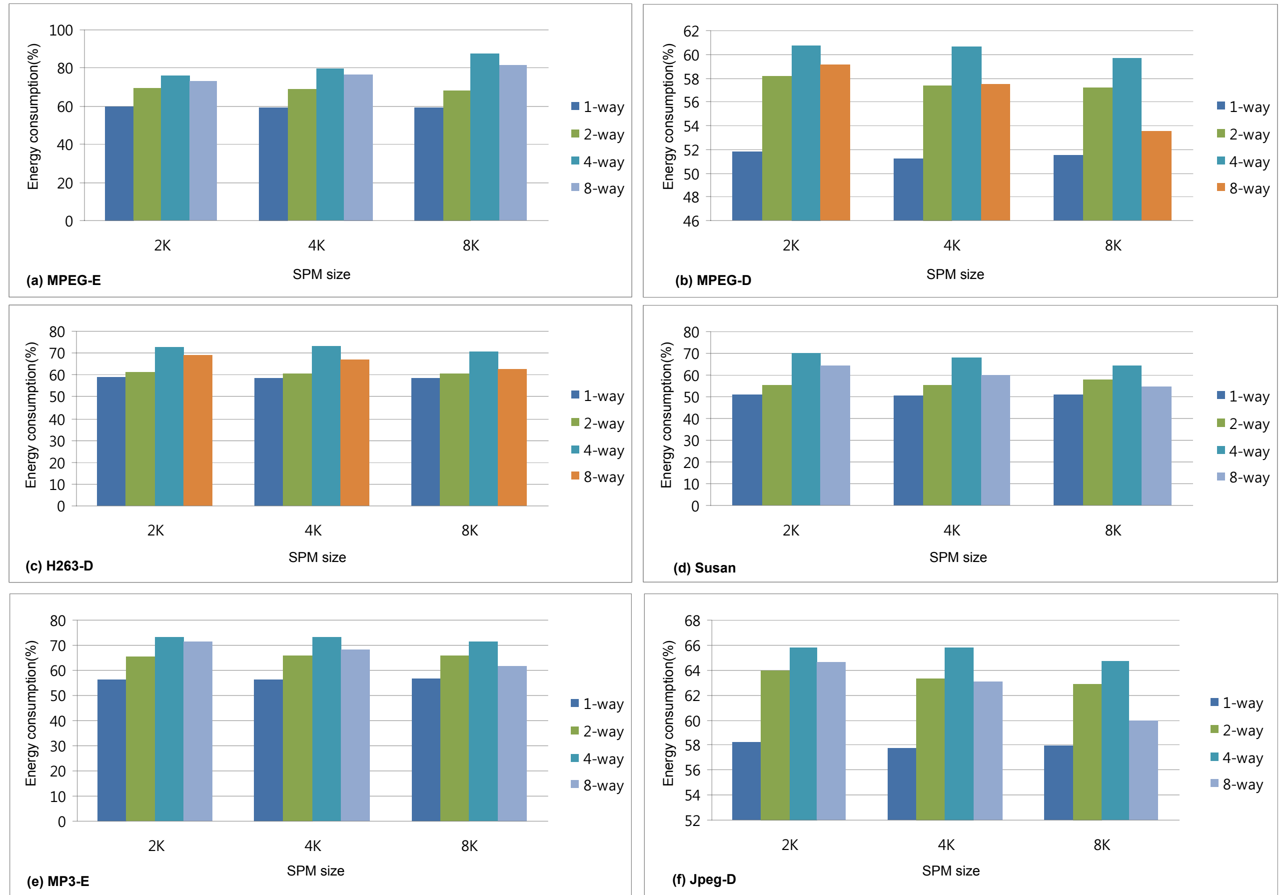

Figure 14. Energy savings for various cache sizes and associativities

The results show that using SPM instead of a conventional cache decreases the total data access energy by 30% on the average (i.e., across all benchmarks and cache configurations experimented with). Note that similar results have also been reported by Benini et al. [31]. In addition to that, we made observations of that increasing the associativity from 1 (direct-mapped case) to 4 improves cache performance. However, increasing the associativity from 4 to 8 decreases the performance of the conventional cache due to the overhead factor we used and the deficiency of a significant drop in the number of conflict misses as a result of increased associativity. These results clearly show that the SPM outperforms the cache even under higher associativities. From the experiments, we found that the additional hardware component (address translator) does not result in a significant overhead for the proposed approach. The address translator consumes 13.6% energy of the 2K directed mapped cache of the simplest configuration. Overall, the energy consumption of the proposed approach is always lower than all cache configurations in our studies.

Finally, we present a brief discussion on the strengths and weakness of our method over caches. Firstly, like caches our approach gives preference to more frequently accessed data by allocating it into SPMs. Secondly, one downside of our approach is that a cache dynamically retains the used part of data, while our approach retains a part of data decided at compile time. Thirdly, an advantage of our approach is that it avoids copying infrequently used data to the SPM; a cache copies in infrequently used data when accessed, possibly evicting frequently used data. On the whole, the runtime results shown in Figure 13 indicate that we come out slightly ahead over the caches (an 18.7% runtime improvement on average).

7. Conclusion

In this paper, we developed a compiler-driven strategy to reduce the number of off-chip memory references by allowing the parallel processing elements in an embedded MPSoC to co-operate with each other by caching data on behalf of each other. Our technique is geared towards array-intensive applications that expose regular/irregular memory accesses patterns. Our strategy identifies data elements to be mapped to SPM with fine granularity and derives an energy and performance-efficient layout of data in on-chip memories. Experimental results obtained with our tool show that we are able to achieve 30% energy reduction and an 18.7% runtime improvement on average, compared to cache based memory subsystems.

8. Acknowledgments

This work is partially supported by a generous grant from IDEC, MKE (Ministry of Knowledge Economy), Korea, under ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) (IITA-2008-C1090-0801-0020), the IT R&D program of MIC/IITA. [2006-S-006-01, Components/Module technology for Ubiquitous Terminals], Human Resource Development Project for IT SoC Architect, Nano IP/SoC promotion group of Seoul R&BD Program in 2008.

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. of the 10th International Workshop on Hardware/Software Codesign*, May 2002.

[2] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Design Automation Conference*, 2001, pp. 690–695.

[3] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," in *Proc. of the 2nd ACM international conference on Hardware/software codesign and system synthesis*, 2004, pp. 104–109.

[4] 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, 1994.

[5] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. on Programming Languages and Systems*, vol. 18, no. 4, pp. 424–453, July 1996.

[6] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. on Embedded Computing Sys.*, vol. 1, no. 1, pp. 6–26, 2002.

[7] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proc. of the European conference on Design and Test*, 1997, p. 7.

[8] J. Sjödin and C. von Platen, "Storage allocation for embedded processors," in *Proc. of the international conference on Compilers, architecture, and synthesis for embedded systems*, 2001, pp. 15–23.

[9] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. of the conference on Design, automation and test in Europe*, 2002, p. 409.

[10] M. Verma, S. Steinke, and P. Marwedel, "Data partitioning for maximal scratchpad usage," in *Proc. of Asia South Pacific Design Automated Conference*, 2003.

[11] K. D. Cooper and T. J. Harvey, "Compiler-controlled memory," in *Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 2–11.

[12] 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.

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

[14] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. of the international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 276–286.

[15] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip," in *Proc. of the 14th international symposium on Systems synthesis*, 2001, pp. 19–24.

[16] M. Kandemir and N. Dutt, *Memory Systems and Compiler Support for MPSoC Architectures*. Morgan Kaufmann, 2005.

[17] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *Proc. of the 43rd annual conference on Design automation*, 2006, pp. 49–52.

[18] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," in *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005, pp. 329–338.

[19] M. J. Absar and F. Catthoor, "Compiler-based approach for exploiting scratch-pad in presence of irregular array access," in *Proc. of the conference on Design, Automation and Test in Europe*, 2005, pp. 1162–1167.

[20] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy, "Dynamic scratch-pad memory management for irregular array access patterns," in *Proc. of the conference on Design, automation and test in Europe*. European Design and Automation Association, 2006, pp. 931–936.

[21] "Arm advanced micro bus architecture (amba)," ARM. [Online]. Available: http://www.arm.com/products/solutions/AMBAHomePage.html

[22] "Sonics, integration architectures." [Online]. Available: http://www.sonicsinc.com

[23] M. Gschwind, H. P. Hofstee, B. K. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture." *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[24] W. J. S. H. Yongjoo Kim, Seongnam Kwon and Y. Paek, "An openmp translator with retargetable parallel programming model for mpsoc," in *Proc. of Intl. Conf. on Ubiquitous Information Technologies and Applications*, 2007.

[25] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," *SIGPLAN Not.*, vol. 37, no. 5, pp. 210–221, 2002.

[26] A. Felner and S. Kraus, "Kbfs: K-best-first search," in *Annuals of Mathematics and Artificial Intelligence*, 2003, pp. 19–39.

[27] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *International Symposium on Microarchitecture*, 1997, pp. 330–335.

[28] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, 1997.

[29] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model."

[30] "128 mbit micron mobile sdram data sheet." Micron Technology Incorporated. [Online]. Available: http://www.micron.com

[31] 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, 2000.