Software Controlled Memory Layout Reorganization for Irregular Array Access Patterns

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

Ilya Issenin
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
isse@ics.uci.edu

Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
dutt@ics.uci.edu

Jonghee W. Yoon
School of EECS
Seoul National University
jhyoon@optimizer.snu.ac.kr

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

ABSTRACT

Many embedded array-intensive applications have irregular access patterns that 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 power improvement. We present a profiling based strategy that generates a memory access trace which can be used to identify data elements with fine granularity that can profitably be placed in the SPMs to maximize performance and energy gains. We developed an entire toolchain that allows incorporation of the code required to profitably move data to SPMs; visualization of the extracted access pattern after profiling; and evaluation/exploration of the generated application code to steer mapping of data to the SPM to yield performance and energy benefits.

We present a heuristic approach that efficiently exploits the SPM using the profiler-driven access pattern behaviors. Experimental results on EEMBC and other industrial codes obtained with our framework show that we are able to achieve 36% energy reduction and reduce execution time by up to 22% compared to a cache based system.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*optimization*

General Terms

Languages, Management

Keywords

scratch pad memory, data layout, energy consumption

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

Many important embedded applications (e.g., multimedia streaming) are characterized by memory-intensive accesses nested within critical loops. It is essential to utilize temporal locality in such loop intensive applications for improving performance and achieving energy efficiency in the memory subsystem design. Traditional approaches use hardware controlled caches to exploit temporal locality and are typically effective for general purpose architectures. However, a hardware-only implementation has several drawbacks. The hardware controlled approach incurs additional power and area cost [3]. 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 [16].

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 [27]. 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 and TI TMS370CX7X, SPM is used as an alternative of cache or a part of cache. Consequently, an approach for effective utilization is essential for efficacy of SPM based memory subsystems. Studies on SPM utilization have focused on development of approaches for efficiently assigning frequently accessed data and instructions to SPM so as to maximize improvement of performance and energy consumption.

Many prior approaches for data array placement in SPMs have focused on applications with regular data access patterns, typically those with index expressions represented by affine functions of outer loop iterators (we name these as "regular" applications). 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 analy-

sis 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 dynamic data reorganization method guided by profiler. Our approach 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. Identify parts of the data array that can be copied into the SPM for improving run-time and energy consumption, and

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

The solution to these two problems result in code for copying the data from main memory to SPM (using the processor or a DMA controller).

We have developed a complete toolchain that allows for evaluation and exploration of SPM usage for irregular applications. First, the code for copying the data from main memory to SPM is added to the original program and the modified program is compiled using conventional compilers. Next, a profiler we developed generates the array access footprint visually, allowing the designer to analyze the access pattern based on the footprint, and source code generator. Finally, the tool chain produces application code that exploits the SPM to yield performance and energy benefits. We performed experiments with the tool chain for several irregular applications. Our results indicate that the proposed approach is successful with the applications that have irregular data access patterns and improves their energy consumption by about 36% over conventional caches.

The rest of the paper is organized as follows. Section 2 describes a motivational example. Section 3 presents related work. Section 4 presents our approach for data reorganization and describes a code generation method to use selected data and placement with low overhead. Section 5 evaluates the benefits and overheads of the proposed approach. Section 6 concludes the paper.

2. MOTIVATION

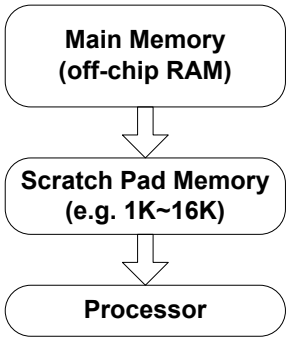

Figure 1: Memory subsystem architecture

Figure 1 shows the data memory architecture with a scratch pad memory. Scratch pad memory is placed close to the processor core like a conventional cache. From viewpoint of applications, accessing array in a scratch pad memory is the same as accessing array in the main memory, but the scratch pad memory is much faster. For the matter of fact, good data layout decisions obtained by using precise locality information is important because that means fewer transfers between SPM and main memory in SPM-based memory sub-systems. A data placement reorganization involves changing the location of the elements of the data array, but not the order in which these elements are referenced. While regular applications can benefit from static analysis to exploit the SPM, irregular applications are not guarantee to static data reuse analysis. However, even irregular applications can benefit from SPMs that need a different approach to exploit layout reorganization of data for the SPM. To demonstrate the potential for layout reorganization of data in SPM consider the following irregular application:

```
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 2: A code fragment with irregular array references

The loop code in Figure 2 is from Reed-Solomon Error Control Code [18], 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 [1].

Therefore, our interest focuses on the copying of the array a to the SPM. 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 possible since the array size can be larger than the SPM size.

The access pattern of array a is precisely known only at runtime. 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 3. The X-axis shows inner-most loop iteration numbers. An iteration is usually iterated by outer loops. Therefore, an iteration has several accesses. Based on the access footprint, we can get information about data reusability and lifetimes of array elements. In this example, several array elements are highly reused. The whole access footprint shows that some parts of array elements are frequently reused, while others are not. Those reused array elements are sparsely spread in the time and space domain. That is why it is difficult to determine frequently reused data block or tile on which to apply existing methods [1, 6].

3. RELATED WORK

Many papers have addressed the problem of improving data reuse in caches, primarily by means of loop transformations (e.g. [5, 19]). However, we do not address this problem since we assume that all possible loop transformations for improving locality of accesses are already performed before applying the technique presented this paper.

There are several prior studies on using SPMs for regular data accesses. The studies can be divided two parts, static and dynamic. Static methods [22, 25, 23, 2, 3, 28] 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 leads to the non-optimal use of the scratch pad memory since during the execution of the program different parts of memory objects may be used. Static approaches use greedy strategies to determine which variables to place in scratch pad memory, or formulate the problem as an integer-linear programming problem (ILP) or a knapsack problem to find an optimal allocation.

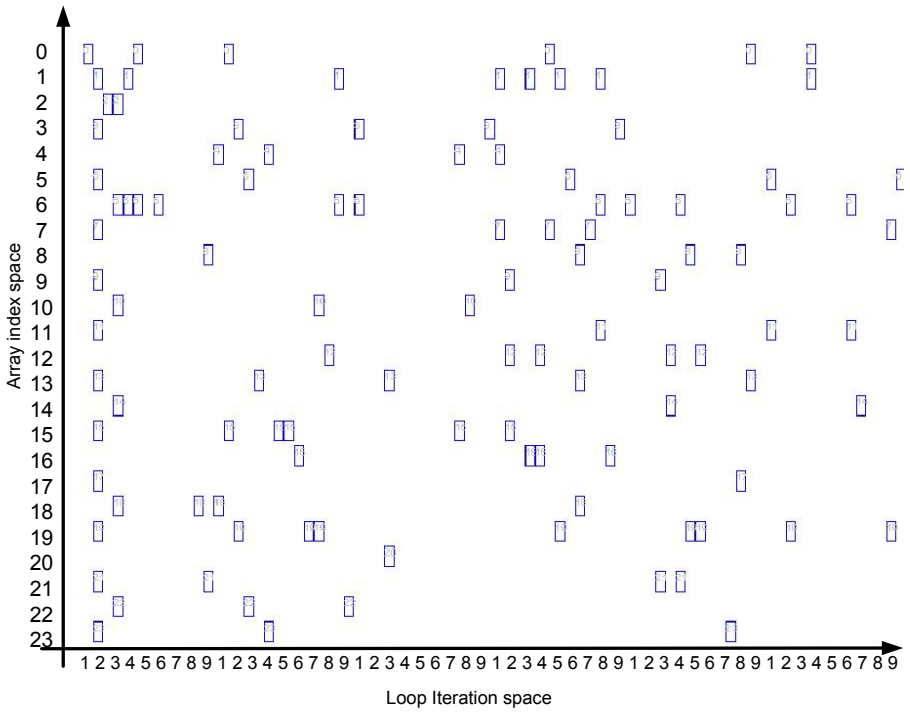

Figure 3: Array access footprint from the example loop kernel

Dynamic SPM allocation approaches include [8, 16, 26, 27, 15, 9]. Cooper et al. [8] proposed to use scratch pad memory for storing spilled values. Udayakumaran et al. [26] proposed an approach that treats each array as one memory object and placement of parts of array to the scratch pad memory is not possible, but it can treat all global and stack variables. Kandemir at al. [16] addresses the problem of dynamic placement of array elements in scratch pad memory. 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.

While research described above focused explicitly on regular access patterns, Verma et al. [28] and Li et al. [17] proposed approaches that work with irregular array access pattern. Verma et al. 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 [28], 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 [28]. Li et al. [17] 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. 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, it cannot completely solve the problem because the partitioning method uses a constant variable to divide the SPM space, where leads to unavoidable fragmentation. We solve this problem by formulating it as a two-dimensional (time and space) knapsack problem, that can assign array slices to SPM without any internal fragmentation.

Absar et al. [1] and Chen et al. [6] also present approaches for irregular array accesses. The meaning of irregularity of their works is limited to a case of indirect indexed array. In addition to that, the indexing array must be referenced by affine function. In these work, they identify the reused block or tile which 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, and ours also includes all other types of irregular accesses found in various applications such as encryption and communication.

4. STRATEGY FOR DATA LAYOUT REORGANIZATION

In order to efficiently solve the data reorganization problem, we break it into two smaller problems. The first problem is to select array elements copied onto SPM, which have high data reusability with similar lifetimes that are beneficial to copy to SPM. 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 [27], we employ heuristics and are able to find near-optimal solutions for both the first and the second problems in polynomial time.

The workflow of our approach is shown in Figure 4. The first task is to gather array access footprint through profiling. In profiling task, we have several times run our benchmark code with various types of input set to gather access frequency of each array element for each input, and an average is obtained. It represents how frequently an element is reused in the application. After profiling done, data selection task is applied. In the first step of the second task, arrays from the application code are identified, and then locality analysis is performed using array access footprint generated by

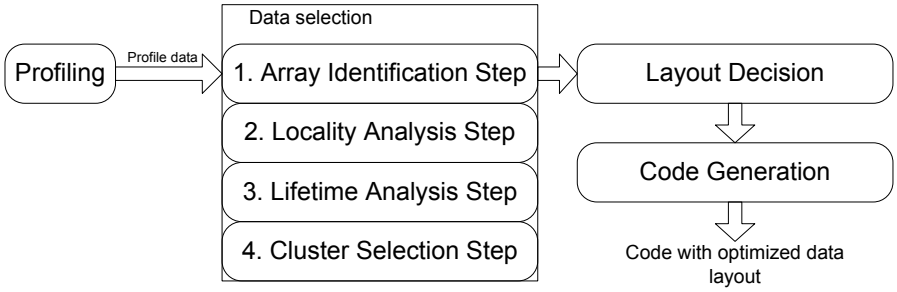

Figure 4: Workflow of the proposed approach

profiling. After that, in the third step, lifetime analysis is performed to determine the live ranges of each array element. Using a formal metric that considers data reusability and lifetime information, candidates of array elements to be copied to SPM are determined for a loop. A cluster of the candidates, (which is determined by lifetime similarity), is a basic unit for assigning to SPM. By doing so, a set of clusters are determined in the final step of second task. The third task is to decide the location of clusters in order to maximally use small SPM space. Finally, optimized code with the array slices (clusters) are generated. The following subsections describe this workflow in detail.

4.1 Data Selection with Data Reusability and Lifetime

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

We use 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} elements of array 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. **Reusability factor :** $T_{n_i} = F/N$.

Our technique selects candidates of data to be located in SPM when array elements have data reusability factor more than two, because those array elements can reduce at least one main memory access. As shown in Figure 5, each candidate of element can be divided by their lifetime in a loop. There are seven candidate clusters of array elements. We developed this data classification method based on lifetime of selected data extracted from the profile-generated array access footprint. This procedure is the following:

- Collect information of the last use for each selected element
- Calculate the Euclidean distance between the last uses of selected elements
- Divide selected array elements into groups with similar lifetimes based on their distance

Based on the Euclidean distance, the lifetimes can be divided into groups with similar lifetimes. We chose the distance as 2 in Figure 5, which is selected to minimize external fragmentation of memory. Alternatively, the selection can be driven by a more sophisticated technique by minimizing the maximum diameter of a cluster (k-clustering) [11]. Finally, we can get data clusters to be transferred into SPM according to this classification. The lifetime groups form several clusters of array elements are shown in Figure 5. These clusters are a basic unit for assigning onto SPM. However, all clusters cannot be assigned to SPM since the SPM size is usually small (e.g., 1K-16K bytes). It is a capacity constraint in the layout reorganization problem. In addition, some of selected clusters may be not assigned to the scratch pad memory because the memory address fragments can be scattered in SPM address space (i.e., memory fragmentation). To solve this 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 subsection.

4.2 Data Layout Decision Problem (DLDP)

A good data layout can place most of the data clusters in the scratch pad memory, as a consequence, which yields the least possible main memory accesses. In general, data layout reorganization problem is to obtain such a good data layout. To make better probability of finding a good layout, minimized fragmentation , which is generated when array clusters are transferred in and out, should be found by a layout decision algorithm in order to increase the chances of finding a large enough free space.

DEFINITION 2. • **Definition of the DLDP**

- *Given:*

  - a data cluster set C for each arrays in an application code

  - Loop execution times T represented as loop iteration number

  - Capacity, SPM capacity

  - Benefit(c), a product of Size(c) by a sum of data reusability factor

  For each cluster c ∈ C

  - Size(c), a total size of elements belonging to cluster c

  - Lifetime(c) = [start(c),..., end(c)], a non-empty interval in time T

  - Tr(c), data transfer cost represented as the number of data transferring of c

- *Find: an assigned set of clusters* $C_a \subseteq C$
- *Maximize:* $\sum_{c \in C_a}(Benefit(c) - Tr(c))$
- *Subject to:* $Size(clusters(t)) \leq Capacity$*, for each* $t \in T$

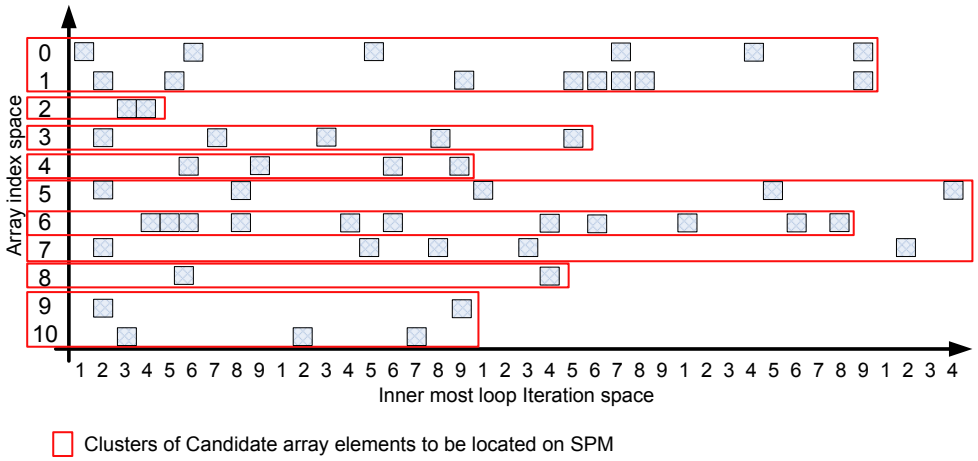

Figure 5: Cluster selection

- *where clusters(t) represents a cluster set $C_t \subseteq C_a$ at a time t.*

In this work, data layout decision problem is to find a particular ordering for each selected cluster in SPM address space on execution time stamp of 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 two dimensional (time and space) knapsack problem. The problem forms the knapsack with a fixed SPM size k with loop iteration time t. A formal statement of the data layout decision problem mapping to the two dimensional knapsack is given in Definition 2. The objective function of this problem is to maximize the benefit with minimizing data transfer cost. Data transfer cost can be two types in this work. The first is memory access latency executed by DMA since the transferring can be performed by DMA circuits. The second is memory access latency executed by processor using load/store instructions. These costs depend on a memory configuration of the target architecture.

The one dimensional (space) knapsack problem for memory object movement into scratch pad memory is formulated in [3]. It is a special case of DLDP in which there is only a static time. Since the problem is NP-Complete and is a special case of DLDP, DLDP is also NP-Complete. We can search a near-optimal solution by a best-first search with a heuristic. In the next section 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 maximize the objective function in Definition 2 because the search space of clusters consists exponentially many sets. Our algorithm for solving the DLDP has two steps. Section 4.3.1 gives the algorithm of the divide step. Section 4.3.2 presents a best first search method for each problem instance, as a conquer step.

4.3.1 Divide Step with Simplification

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 as following with Figure 6.

Figure 6 shows an instance of DLDP that has seven array clusters, c1,...,c7, and time T ={1,2,3,...,10} (loop iteration number), which are displayed on each rectangles and the Y-axis. This example has a capacity constraint 10 in T.

The reduce operator performs two kinds of simplification. The first kind removes from the problem instance any cluster c whose size exceeds the capacity available in its lifetime(c). For example, in the instance of Figure 6.(a), cluster c2 has size 11 at time $T = 1, 2, 3$, yet the capacity is only 10. So, the reduce operator removes c2 from the instance, which results in the instance shown in Figure6.(b).

The second kind of simplification removes unnecessary times from the instance. In the instance of Figure 6.(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, thereby, the reduce operator can also remove c5, resulting in the instance displayed in Figure 6.(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 cluster. If the two times have the same capacity, then either time can be removed. In the instance of Figure 6.(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 6.(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 clusters(t) $\cap$ clusters(t') == ϕ. In the instance of Figure 6.(d), a split can be made in between time 4 and 8 resulting in two subproblems: one comprising times 2 and 4 and clusters c1, c3 and c4; and a second comprising time 8 and clusters c6 and c7.

In Figure 7, let Size(clusters(t)) be the total required size at time t - that is $\sum_{c \in clusters(t)} size(c)$. Also let 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 Figure 7.

4.3.2 The Conquer Step with the k-way Best First Search

In the previous subsection the Split procedure divides an problem instance (P) of DLDP to smaller problem instance (p_1,p_2,...,p_l). Each $p \in P$ is objective of the k-way best first search [10] in conquer step. The k-way best-first search is used to search for the near-optimal cluster ordering in scratch pad memory space. In-

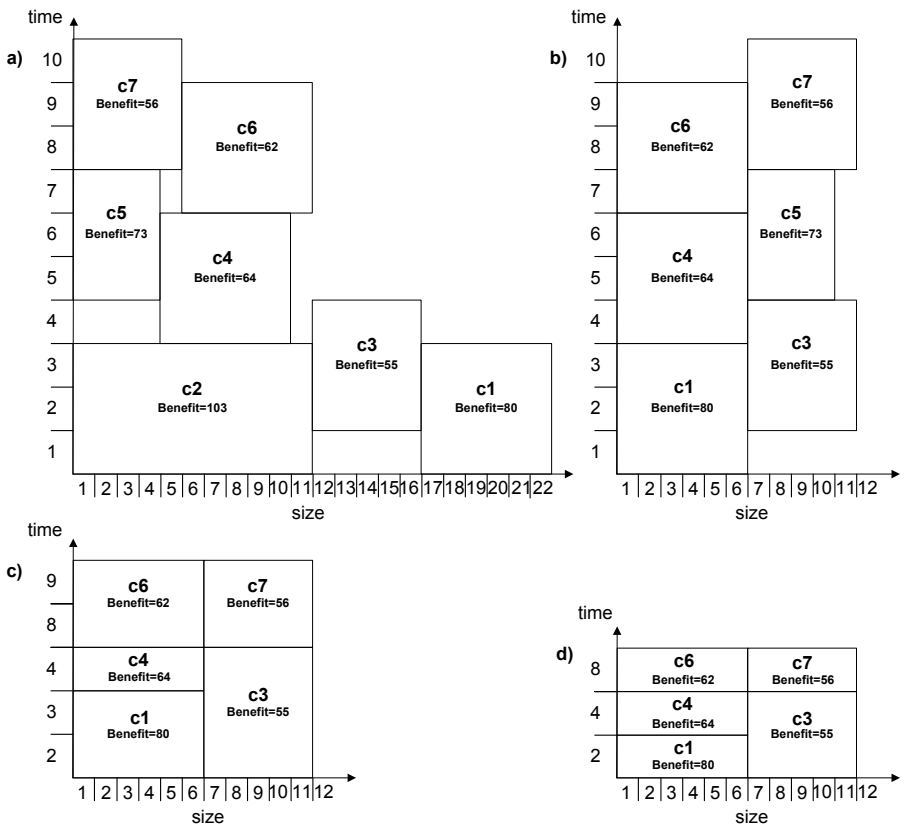

Figure 6: An instance of DLDP to which the reduce and split operators are applied

```
Simplify(dldp* P){ //P: an instance of DLDP
  if(C == 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 c clusters(t) do
            remove t from Lifetime(c);
        } 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{c | end(c) <= ta}
        p2 = times{t | t >= tb} and clusters{c | start(c) >= tb}
      }
    }  //end of while
  }
}
```

Figure 7: A procedure of the divide step with simplification

stead of searching the whole set of cluster orderings (which contains $C!$ orderings), the algorithm selects the k best clusters in a sorted manner. This k-way best first search may incur the overhead of expanding k-1 unnecessary search space, and selection of the k should be determined with considering time complexity and solution optimality. Each cluster selection by the search builds a search tree as shown in Figure 8.

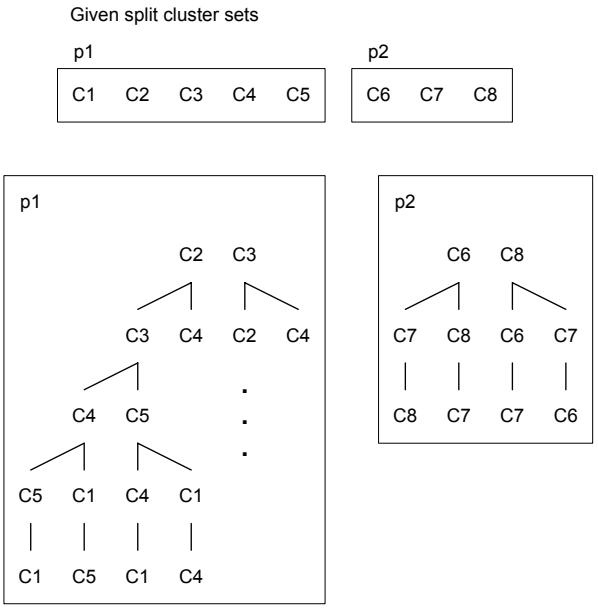

Figure 8: A search tree example with k=2

The search algorithm stores at each node the maximum benefit and the minimum benefit on the objective function for the DLDP instance.

DEFINITION 3. **Metric for searching**
$Benefit_{MAX}$ = $max_{c' \in children(c)}(current_{MAX}(c')$ $+child(c, c'))$
$Benefit_{MIN}$ = $min_{c' \in children(c)}(current_{MIN}(c')$ $+child(c, c'))$
where child(c, c') is the sum of the benefits of the cluster assigned in moving from node c to node c',and children(c) is the set of nodes that are children of c.

The search scheme repeatedly (1) selects an unobserved cluster, (2) observes the cluster and then creates its k number of children, and (3) propagates new max and min benefits by Definition 3 through the tree and uses these benefits to select the k clusters. It performs this sequence of three stages until the tree contains no nodes to observe. 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-highest benefit. Let us now consider the three major steps in more detail.

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

The second step is that processes and expands the node. For each of these unobserved nodes, max and min benefits on its objective function is obtained. In the case of the node to observe a child node, an unobserved cluster is chosen to branch on, and its k children nodes are created, the highest one in which the cluster is recorded as a solution. Created k nodes are then processed and expanded in the same way.

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

Finally, this search procedure produces a placement of cluster ordering as a near-optimal solution. The ordering is sequentially mapped to address of the SPM.

4.4 Code Generation

After our method determines the layout of the array elements in SPM, our tool transforms a given code into a code to implement the desired memory allocation and transfers. Code generation in our approach involves changing the original code in three steps. First, for each original array in the application (e.g., $array$) which is moved to the scratch-pad during lifetime(c), where c $\in$ clusters C, the compiler declares a new array buffer (e.g., spm_array) in the application corresponding to the copy of $array$ in the scratch-pad. The original array $array$ is allocated to main memory. By the same way, the compiler can allocate $array$ and spm_array to different address regions in memory. These regions later should be mapped to different physical memories (main memory and SPM). Second, the compiler replaces array references by the statements that address SPM buffer if the buffer holds a copy of the data requested. Third, data-transfers between main memory and SPM are inserted at certain program points, to evict some data elements and copy others, as decided by the placement selection procedure.

Since the SPM buffers of data elements (declared above) have limited lifetimes, our approach is developed to a dynamic method. As a consequence, different clusters with non-overlapping lifetimes may have overlapping offsets in the scratch-pad address space. Furthermore, a single cluster may be allocated to the scratch-pad at different addresses at different times. Clearly, there is a need for additional code that maps the iteration number into SPM address space offset. This can be easily accomplished by using a table of address map, as shown in Figure 9. The map table is used for mapping from an iteration number to corresponding an element in SPM buffers. To build this table, two intermediate tables are necessary. We refer to these intermediate tables as address lookup tables. These lookup tables are temporarily created by our code generator based on array access footprint, and linker uses these tables in order to build the address map table. The lookup tables in Figure 9 are implemented by a sorted associative container that associates an iteration number of type key with an array index of type $data$ and an array index of type key with SPM buffer address of type $data$. Using these tables, linker builds an address map table onto SPM in order to translate each iteration number to scratch pad memory address for correct array accessing. Next section explains a table construction procedure in detail.

4.4.1 Construction of an address map table

Memory access footprint generated by profiling is essential information in constructing an address map table, since it affects correctness and efficiency of the table. In the table construction

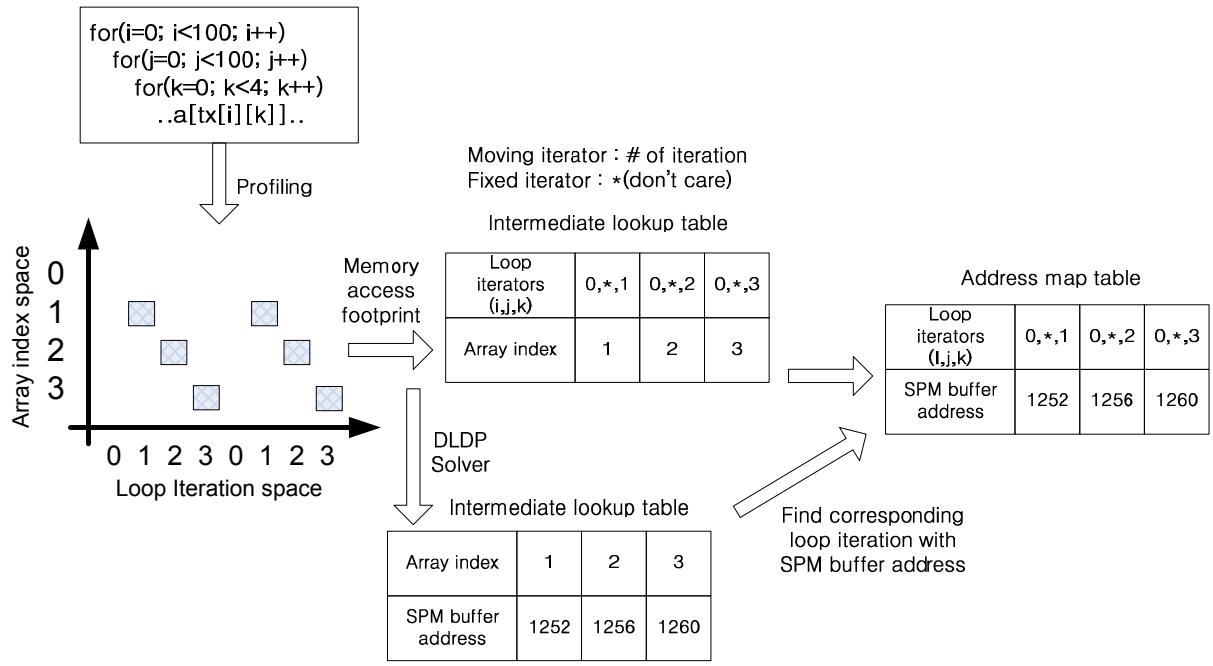

Figure 9: The table construction procedure

process, the most important concern is size efficiency. To minimize table size, it needs to eliminate unnecessary information in the table. Initial lookup table records all iteration numbers on a given loop, but some loop iterations do not participate in array accessing. To classify these iterations, it needs a method to classify iterators to participation group and non-participation group in array accessing represented in a given array reference function. We call the non-participation iterators to fixed iterators, since they iterate array access trace generated by participation iterators. Hence, we can eliminate those iterations generated by fixed iterators in the address map table records. To achieve this, we use similar concept described in [15] that loop iterators are classified into two groups; moving iterator and fixed iterator. We extract array access trace of moving iterators, and build lookup tables based on the memory trace. Then, the map table can record necessary information only. The table building procedure is as following.

- Obtain array access footprint through profiling
- Classify loop iterators into two groups and extract array access footprint of moving iterators from the profiling results
- Build a first lookup table which has two entries : moving iteration numbers and corresponding array indexes
- Build a second lookup table which has two entries : array indexes and corresponding SPM buffer addresses
- Build a map table which has two entries : moving iteration numbers and corresponding SPM buffer address

Figure 9 shows the building procedures. There are five steps. The first step is to identify array reference function. The function includes a part of iterators or the whole of iterators on a given loop. All the iterators used in a loop hierarchy are split into two groups: moving iterators (M) and fixed iterators (F). To determine this classification, iterators should be added one by one to a set M of moving iterators as long as the following property remains true:

- All iterators I in a given loop
- All iterators I' used in an array of reference function R
- Fixed iterators $F = I - I'$
- Moving iterators $M = I \cap I'$
- where a reference function is consisted with iterators M, a given loop nests is consisted with loop iterators $M + F$

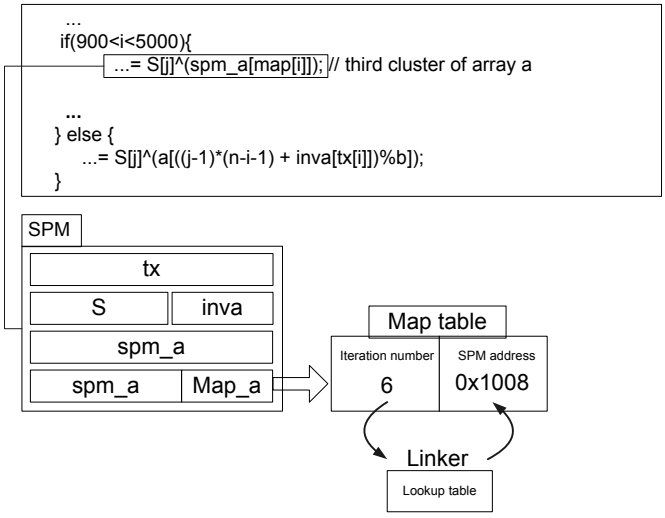

Figure 10: An address mapping operation

This classification for loop iterators is necessary in our approach. It allows easy determination of the repeating addresses in the address footprint of the accessed array elements between iterations of

moving iterators when the fixed iterators do not change their values and the moving iterators are iterating over their complete loop bounds. This allows keeping the reused array element index efficiently during iteration over fixed iterators since it does not affect to addresses of accessed array elements. Therefore, the table entry is consisted with moving iterations only, not fixed iterations.

The third step is construction of first lookup table. The first lookup table has two entries that are loop iterations of moving iterators and the corresponding index of array elements. Each entry is extracted from memory access footprint shown in Figure 9. The fourth step is construction of second lookup table. The second lookup table has two entries that are array index of corresponding array element and its address of SPM. The SPM address is decided by DLDP solver. The output of DLDP solver has two entries: array index and its SPM address. By the solution of DLDP, the second table can be built. Finally, an address map table is consisted with moving iterators and addresses of SPM buffer without the array index entry, since array index information does not necessary in address mapping process in run time.

Figure 10 shows address mapping operation with the final address map table. The final table bridges an iteration number to a corresponding SPM element's address. By the table, iteration number can be easily replace into correct SPM address. The size of the initial address map table is the same as the lifetime duration of the clusters selected by DLDP solver. The use of the mapping tables may incur large memory space overhead. Fortunately, the size can be compressed by hashing for redundant elements of SPM addresses. To compress the table by hashing, hash key should completely distinguish different SPM index into corresponding loop iterations. For our benchmarks, the average size of the table was relatively small (only about 5% of the SPM data) as shown in Table 2.

5. EXPERIMENTAL RESULTS

One of the goals of our experiments is to compare our approach for the scratch pad memory based subsystem management against a traditional cache based memory subsystem for their ability to exploit data reusability of the data accesses in a number of multimedia, communication, and encryption applications. We also study the overheads incurred in using our approach.

We have created a tool that implements the described technique. We have used a Pentium 4 workstation for profiling purposes. The SimpleScalar simulator [4] has been used for obtaining the number of misses for cache and measuring runtime. We have used the CACTI model [24] for energy estimation of both the cache and scratch pad memories at 130nm technology.

Table 1: Benchmarks used in experiments

Programs	VITERBI	RS	PANAMA	AIFFTR	IDCTRN
Description	Error-correction for communication	Error-correction for storage and communication	Encryption for stream data	Fast Fourier Transform	Inverse Discrete Cosine Transform

The experimental input is a set of full codes obtained from EEMBC [29], and other industrial applications with (4K-93K) data stream. Table 1 lists benchmarks used for our experiments. The tool has provided us with the code versions that implement necessary data transfers between the scratch pad memory and the main memory for the selected clusters.

We have examined the effect of using a scratch pad memory on the reduction of traffic to main memory as well as on the energy spent in the memory subsystem. The comparison of the scratch pad based memory architecture has been made against a system having a direct mapped data cache of the same size (1-4KB). The goal is to evaluate the energy efficiency of our software steered data reorganization approach compared to that one implemented by a hardware cache controller i.e., with the LRU replacement policy. The sizes of the scratch pad memory and the cache have been selected to be the closest values that are powers of two while being greater or equal than the buffer size required in the scratch pad configuration. The cache line size has been selected to be the minimal allowed by the simulator [4] (8 bytes, which is 2 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.

The energy savings when using scratch pad in comparison with cache are coming from two sources. First, a scratch pad memory consumes less energy than a cache of the same size per one access (about two times less for direct mapped cache [24]). Second, it is possible to make the more optimal data placement decisions for the scratch pad memory compared to that ones made according to the LRU policy of the cache controller, which results in less accesses to the main memory for all studied cases. In our experiments we have used a relatively small off-chip memory and have not accounted for the energy dissipation in the off-chip buses due to limitations of the used energy model [24]. As we can see from Table 2, the scratch pad based memory subsystem consumes 21% to 48% less energy than the system with a cache of the same size. We also estimate the amount of address mapping table size needed to perform the described in Section 4.4. On average the size of address tables is 92 bytes. It is about 5% of the total data in the SPM.

6. CONCLUSION

We developed a profiling based strategy to efficiently use software controlled on-chip memory in memory hierarchies of embedded systems. Our technique is geared towards array-intensive applications that expose irregular accesses patterns. The new 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 memory. Experimental results obtained with our tool show that we are able to achieve 36% energy reduction compared to a cache based system.

7. ACKNOWLEDGMENTS

This work was partially supported by NSF grant CNS-0615438, the IT R&D program of Ministry of Information and Communication (MIC)/Institute of Information Technology Assessment (IITA) [2007-S001-01, Components/Module technology for Ubiquitous Terminals], the ITRC (Inofrmation Technology Research Center) support program supervised by the IITA (IITA-2005-C1090-0502-0031), Nano IP/SoC promotion group of Seoul R&BD Program in 2007, MIC(A1100-0501-0004), and Korea Ministry of Science and Technology (M103 BY010004-05B2501-00411).

8. REFERENCES

[1] J. Absar and F. Catthoor. Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access. In *Proceedings of DATE*, 2005.

[2] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch Pad Based Embedded Systems. *ACM Trans. on Embedded Systems*, 1(1), September 2002.

[3] R. Banakar, S. Steinke, B-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In

Table 2: Energy, main memory accesses, and runtime reduction compared to the cache based memory subsystem

Benchmark Codes	Energy reduction (%)	Main memory access reduction (%)	Runtime reduction (%)	Address map table size (bytes)
REED SOLOMON	35.9	46.1	8.8	87
VITERBI	38	32.6	11.5	112
PANAMA	21.8	20	22.7	56
IDCTRN	40.2	21	6.3	55
AIFFTR	48.3	41.1	9.8	153
average	36.8	32.1	11.8	-

International Symposium on Hardware/Software Codesign, May 2002.

[4] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. In*Technical Report 1342*, University of Wisconsin-Madison, CS Department, June 1997.

[5] D.F. Bacon, S.L. Graham et al. Compiler Transformations for High Performance Computing. In *ACM Computing Survey*, 26(4). 1994.

[6] G. Chen, O. Ozturk, M. Kandemir and M. Karakoy. Dynamic Scratch-Pad Memory Management for Irregular Array Access Patterns. In *DATE*, March 2006.

[7] M. Co, Dee A.B. Weikle, and K. Skadron A Break-Even Formulation for Evaluating Branch Predictor Energy Efficiency. In *Proceedings of Complexity-Effective Design*, 2005.

[8] K. Cooper, T. Harvey. Compiler-Controlled Memory. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, 2-11, 1998.

[9] A. Dominguez, S. Udayakumaran, and R. Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. In *Journal of Embedded Computing*, 2005.

[10] A. Felner, S. Kraus, and R.E. Korf. KBFS: K-Best-First-Search. In *Annuals of Mathematics and Artificial Intelligence*, 19-39, 2003.

[11] D.S. Hochbaum and D.B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. In *Journal of the ACM*, 33, 1986.

[12] I. Issenin, and N. Dutt. Data Reuse Driven Memory and Network-on-Chip Co-Synthesis. In *Proceedings of IESS*, 2007.

[13] I. Issenin, and N. Dutt. Data Reuse Driven Energy-Aware MPSoC Co-Synthesis of Memory and Communication Architecture for Streaming Applications. In *Proceedings of CODES-ISSS*, 2006.

[14] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt. Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies. In *Proceedings of DAC*, 2006.

[15] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies. In *Proceedings of DATE*, 2004.

[16] M. Kandemir, J. Ramanujam, M.J.Irwin, N. Vijaykrishnan, I. Kadayif, and, A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proceedings of the Design Automation Conference*, 690-695, June 2001.

[17] Lian Li, Lin Gao and Jingling Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *PACT*, October 2005.

[18] S. Lin, D. Costello. Error Control Coding: Fundamentals and Applications. Book, Practice Hall, October 1982.

[19] K. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. In *ACM Trans. on Programming Languages and Systems*, 18(4), July 1996.

[20] M. Palkovic, M. Miranda, K. Denolf, P. Vos, and F. Catthoor. Systematic Address and Control Code Transformations for Performance Optimisation of a MPEG-4 Video Decoder. In *Proceedings of ASP-DAC*, 2002.

[21] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of PLDI*, 1998.

[22] P. Panda, N. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of DATE*, 1997.

[23] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of DATE*, 2002.

[24] P. Shivakumar, N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Technical Report, August 2001.

[25] J. Sjodin and C. Platen. Storage Allocation for Embedded Processors. In *Proceedings of the Conference on Compilers, and Architecture, and Synthesis for Embedded Systems*, 15-23, 2001.

[26] S. Udayakumaran and R. Barua. Compiler-decided Dynamic Memory Allocation for Scratch-Pad based Embedded Systems. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 276-286, 2003.

[27] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *Proceedings of CODES*, 2004.

[28] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of ASPDAC*, 2003.

[29] EDN Embedded Microprocessor Benchmark Consortium, www.eembc.org.

[30] Intel Corporation, http://developer.intel.com/design/intelxscale/index.htm.