# Dynamic Scratch-Pad Memory Management for Irregular Array Access Patterns*

G. Chen, O. Ozturk, M. Kandemir
Computer Science and Engineering Department
Pennsylvania State university
University Park, PA 16802, USA
{gchen,ozturk,kandemir}@cse.psu.edu

M. Karakoy
Department of Computing
Imperial College
London, SW7 2AZ, UK
m.karakoy@ic.ac.uk

## Abstract

*There exist many embedded applications such as those executing on set-top boxes, wireless base stations, HDTV, and mobile handsets that are structured as nested loops and benefit significantly from a software managed memory. Prior work on scratchpad memories (SPMs) focused primarily on applications with regular data access patterns. Unfortunately, some embedded applications do not fit in this category and consequently conventional SPM management schemes will fail to produce the best results for them. In this work, we propose a novel compilation strategy for data SPMs for embedded applications that exhibit irregular data access patterns. Our scheme divides the task of optimization between compiler and runtime. The compiler processes each loop nest and insert code to collect information at runtime. Then, the code is modified in such a fashion that, depending on the collected information, it dynamically chooses to use or not to use the data SPM for a given set of accesses to irregular arrays. Our results indicate that this approach is very successful with the applications that have irregular patterns and improves their execution cycles by about 54% over a state-of-the-art SPM management technique and 23% over the conventional cache memories. Also, the additional code size overhead incurred by our approach is less than 5% for all the applications tested.*

## 1. INTRODUCTION AND MOTIVATION

Recent research shows that scratch-pad memories (SPMs) can be very useful for data-intensive embedded applications that exhibit regular, compiler-analyzable data access patterns. This is because, for such an application, a compiler can strategize data movements in and out of an SPM and this eliminates the potential overheads (e.g., those due to dynamic tag matching) associated with conventional cache memories. Several recent commercial products (e.g., ARM11 [1] and StrongARM [2]) give support for SPMs.

Most of the prior efforts that study compiler support for data SPMs focused explicitly on applications with regular data access patterns. While such applications certainly constitute a large fraction of embedded applications, there also exist a number of embedded applications whose data access patterns do not lend themselves well to accurate compiler analysis and optimization. In this paper, we use the terms "irregular arrays" and "irregular access patterns" to describe these types of cases. One option for such applications is to use conventional cache memories; but, this incurs large power consumption and, in some cases, one may still need to execute such an application (with irregular data access pattern) in an SPM-based embedded architecture. Therefore, it would be very useful in practice to have a technique that enables optimization of such applications in the SPM-based embedded systems.

Our goal is to present and quantify the benefits of a novel scheme that addresses this problem. Our scheme divides the task of optimization between compiler and runtime. The compiler processes each loop nest and insert code to collect information at runtime. Then, the code is modified in such a fashion that, depending on the collected information, it dynamically chooses to use or not to use the data SPM for a given set of accesses to irregular arrays.

We implemented this approach within an optimizing compiler and performed extensive experiments with several embedded applications with irregular data access patterns. Our results indicate that the proposed approach is extremely successful with embedded applications that have irregular data access patterns and improves their execution cycles by about 54% over a state-of-the-art SPM management technique and 23% over conventional cache memories. The results also show that the extra code size overheads incurred by our approach are very small (less than 5% for all the applications tested).

We organized the rest of this paper as follows. Section 2 presents the architectural modeling of an embedded system with a data SPM. Section 3 describes the irregularity in array references contained in the applications targeted by our

* This work is supported in part by NSF Career Award #0093082 and a fund from GSRC.

3-9810801-0-6/DATE06 © 2006 EDAA

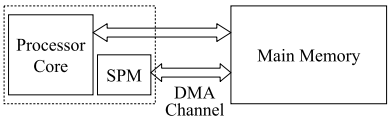


**Figure 1. An embedded architecture with an on-chip scratch-pad memory (SPM).**

approach. Section 4 presents the details of our approach to SPM management for irregular applications. Section 5 discusses the results of our experimental evaluation. Section 6 briefly reviews the related work on SPM management. Section 7 summarizes the major observations we made.

## 2. ARCHITECTURE MODELING

Figure 1 shows the architecture of an embedded system with a scratch-pad memory (SPM). Like a conventional cache, SPM is tightly coupled with the processor core. However, while a cache is normally transparent to the software, an SPM is visible to the software. From the software viewpoint, accessing data in an SPM is the same as accessing data in the main memory except that the SPM is much faster than the off-chip main memory. To accelerate data transfers between the main memory (off-chip) and the SPM, a direct memory access (DMA) channel is also provided. Using this DMA channel, we can transfer data between the main memory and the SPM in bulk without transferring the data through the datapath of the processor. The cost for such a DMA transfer can be expressed as:

$$C_{\mathrm{DMA}} = C_0 + C_1 S,$$

where $C_0$ is the initialization cost for a DMA transaction, $C_1$ is the per byte transfer cost, and $S$ is the size of the data to be transfered. Further, we assume that the per access costs for the main memory and the SPM are $C_2$ and $C_3$, respectively. If a data structure in the main memory is accessed $n$ times, the total cost for accessing this data structure without using the SPM is $C_{\mathrm{memory}} = nC_2$. If, on the other hand, we first load this data structure into the SPM and then access it from there, the total access cost becomes $C_{\mathrm{SPM}} = C_0 + C_1 S + nC_3$, where $S$ is the size of this data structure. If the number of accesses, $n$, is greater than $(C_0 + C_1 S)/(C_2 - C_3)$, we have $C_{\mathrm{SPM}} < C_{\mathrm{memory}}$; that is, loading this data structure into the SPM and accessing it from there is a better option than accessing it directly from the main memory.

## 3. IRREGULARITY IN ARRAY REFERENCES

The body of a loop nest in array-based embedded applications may contain instructions that access the elements of different arrays. A reference to an array can be either *regular* or *irregular*. An array reference is called regular if the subscript of this reference is computed without using the values of any other arrays. On the other hand, a reference to an array is said to be irregular if the subscript of this reference is computed using the values of other arrays. Let us explain these concepts using the following example loop nest:

```
for i1 = u1 to v1
  for i2 = u2 to v2
    for i3 = u3 to v3
      ...A[i1 + X[i1 + i2 + 2][i1 + 3i3]][2i1 + i3 + 2Y[i1][i2] + 1]...
```

The body of this loop nest contains three array references: $X[i_1 + i_2 + 2][i_1 + 3 * i_3]$, $Y[i_1][i_2]$, and $A[i_1 + X[i_1 + i_2 + 2][i_1 + 3i_3]][2i_1 + i_3 + 2Y[i][j] + 1]$. The references to arrays $X$ and $Y$ are regular since the subscript expressions of these references do not depend on the values of any other arrays. The reference to array $A$, however, is irregular since the values of arrays $X$ and $Y$ (which will typically be known only at runtime) are used in computing the subscript of array $A$. We refer to arrays $X$ and $Y$ as the index arrays[1].

For ease of discussion, we say that vector $\vec{I}$ is the *iteration vector* of loop nest $\mathcal{L}$ where each element of $\vec{I}$ corresponds to the index variable of a loop in the loop nest. For example, in the loop nest shown above, we have the iteration vector $\vec{I} = (i_1, i_2, i_3)^{\mathrm{T}}$. In this paper, we assume that the subscript of a regular array reference in a loop nest can be expressed using an affine function such as $M\vec{I} + \vec{o}$, where $M$ is a constant matrix (referred as the access matrix), $\vec{o}$ is a constant vector (referred as the offset vector), and $\vec{I}$ is the iteration vector of the loop nest [16]. For exam-

```
for i1 = 1 to N1
  for i2 = 1 to N2
    S = S + X[i1, i2];
```

(a) Original loop nest.

```
for ii1 = 1 to N1 step T1
  for ii2 = 1 to N2 step T2 {
    X'[1..T1, 1..T2] // in SPM
      <= X[ii1..ii1 + T1, ii2..ii2 + T2];
    for i1 = ii1 to max{N1, i1 + T1 - 1}
      for i2 = ii2 to max{N2, i2 + T2 - 1}
        S = S + X'[i1 - ii1, i2 - ii2];
  }
```

(b) Tiled version.

**Figure 2. An example loop nest (a) and its tiled version (b). In the tiled version, the loops $ii_1$ and $ii_2$ are the inter-tile loops, and the loops $i_1$ and $i_i$ are the intra-tile loops.**

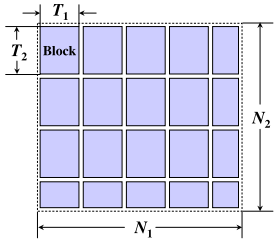


**Figure 3. Dividing a two-dimensional array into data blocks (tiles). All the data blocks have the same size except possibly at the boundaries of the array.**

1 While in this work we focus on irregularity arising from index arrays, our approach can be extend with appropriate modifications to handle the cases where irregularity is due to general functions appearing in array subscripts.

ple, in the loop nest shown above, the subscript of array $X$ can be expressed as:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} + \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

Further, we assume that the subscript of an irregular array reference in a loop nest can be expressed using an affine function as $M\vec{I} + N\vec{J} + \vec{o}$, where both $M$ and $N$ are constant matrices, $\vec{o}$ is a constant vector, $\vec{I}$ is the iteration vector, and $\vec{J}$ is a vector of index array references, i.e., each element of $\vec{J}$ is a regular reference to an index array. This is similar to the notation used in [3]. For example, the subscript of array $A$ in the loop nest presented above can be expressed as:

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} X[i_1 + i_2 + 2][i_1 + 3i_2] \\ Y[i_1][i_2] \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Given two vectors $\vec{U} = (u_1, u_2, ..., u_n)$ and $\vec{V} = (v_1, v_2, ..., v_n)$, we write $\vec{U} \leq \vec{V}$ if and only if $u_i \leq v_i$ for all $i = 1, 2, ..., n$.[2] An important property of the affine function $M\vec{I} + N\vec{J} + \vec{o}$ is that, given $\vec{I}_m \leq \vec{I} \leq \vec{I}_M$ and $\vec{J}_m \leq \vec{J} \leq \vec{J}_M$, we can compute two vectors, $\vec{J}_m$ and $\vec{J}_M$, such that $\vec{J}_m \leq M\vec{I} + N\vec{J} + \vec{o} \leq \vec{J}_M$.

# 4. COMPILER ANALYSIS AND TRANSFORMATION

## 4.1. Loop Tiling

Our focus is on array-based embedded applications from the video/image processing domain. An important characteristic of these applications is that they are constructed from series of nested loops that access multi-dimensional arrays of signals. In an SPM-based memory system, the unit of transfer between the on-chip memory (SPM) and off-chip memory is a *data block* (also called *tile*). Whenever a data item (element) is accessed, the data block that contains that item is brought from the off-chip memory to the on-chip SPM (if it is not already there). Note that, such a transfer can take multiple bus transfers, depending on the data block size and shape used.

Figure 2(a) shows an example loop nest that accesses array $X$ through a reference with affine subscript expression ($X[i_1, i_2]$). This array can be divided into data blocks, as shown in Figure 3. The code in Figure 2(b) gives the blocked (tiled) version of the original loop nest shown in Figure 2(a). Note that the iteration space is divided into tiles (blocks) based on data tiles. In this blocked code, loops $ii_1$ and $ii_2$ iterate over the data blocks, and are called the *inter-tile iterators* (or *block iterators*). In comparison, $i_1$ and $i_2$ are referred to as the *intra-tile iterators,* and iterate over the elements of a given data block (indexed by $ii_1$ and $ii_2$). The explicit data block transfers between the SPM and the off-chip memory occur only across the iterations of $ii_1$ and $ii_2$. In this paper, when we say "inter-tile iteration", we mean a vector $(ii_i, ii_2)^{\mathrm{T}}$. Self temporal reuse is said to exist when an array reference in a loop nest accesses the same data in different loop iterations. Similarly, if a reference accesses nearby data in different iterations, we say that there exists a self spatial reuse [16]. It should be emphasized that, in an SPM-based architecture, the most useful forms of data reuse (temporal or spatial) are those that can be exploited by reusing data from the SPM. Therefore, a good SPM management strategy is the one that converts the inherent temporal and spatial data reuses in the application code into data locality in the SPM. That is, the available SPM space should be managed in such a way that a vast majority of the data requests from the CPU should be satisfied from the SPM without going to the off-chip memory.

```
for t = ... {
  // initialization phase
  X[s1, s2] = ...;
  Y[s3, s4] = ...;
  ...
  // execution phase
  L: for i = ...
    for j = ...
      SUM+ = A[2 * X[i, j]][Y[i, j]]
           +B[X[i, j] + 1][Y[i, j]] + B[X[i, j]][Y[i, j] + 1];
}
```

**Figure 4. A code fragment with irregular array references.**

```
for each loop nest L in program P {
  apply loop tiling to L based on the access patterns of
      regular array references;
    for each assignment to index array X
      update the block minimum and maximum values of X;
    compute the set of array elements that are irregularly
        referenced in the current inter-tile iteration;
    compare the memory access costs for using and not using SPM;
    if(using SPM is beneficial)
      execute the intra-tile loop iterations by using the SPM
    else
      execute the intra-tile loop iterations by not using the SPM
}
```

**Figure 5. Sketch of our approach.**

## 4.2. Our Approach

Figure 4 shows the structure of a code fragment with irregular array references. Most array-based embedded codes with irregular accesses have this structure to begin with, and those without this structure can be transformed to have it. Loop $t$ is the timing loop, which contains an initialization phase and an execution phase. In the initialization phase, we initialize/update the values of only the index arrays. The execution phase typically contains a loop nest that accesses irregularly referenced arrays.[3] Figure 5 shows the sketch of our approach. Our approach optimizes each loop nest $\mathcal{L}$ in a given program $\mathcal{P}$ using the following three steps.

2 This ordering relationship is different from lexicographic ordering on vectors.

3 Sometimes we have a single initialization phase and multiple execution phases within the same timing loop.

**Step 1.** We tile the target loop nest such that the blocks of the index arrays accessed by each tile of the loop nest are small enough to be put in the SPM. Since using loop tiling to improve data locality of regular array accesses has been well studied in the compiler literature [16, 17], we do not elaborate on it any further. The problem we try to solve is, given a program whose regular array references have already been optimized for the SPM, how to further optimize it such that the irregular array references can also benefit from the SPM.

**Step 2.** At runtime, when computing the values of an index array $X$, we also compute the maximum and minimum values of the elements in each block (tile) of $X$ and store these values in two auxiliary arrays $X_M$ and $X_m$. We refer to the maximum and the minimum values of a block of an array as the *block maximum* and *block minimum* values, respectively.

**Step 3.** At runtime, at each tile of the loop nest, for each indirect reference to array $A$, we conservatively compute the set of elements of array $A$ that might be accessed during the execution of this tile. When computing this set, we use the maximum and minimum values of the index array blocks that are used in this tile. Based on the number of array elements that need to be loaded into the SPM and the number of references to the loaded elements (which can be computed at compilation time), we can decide whether to load the data used in this tile into the SPM or access the data directly from the main memory without loading it into the SPM.

Figure 6 shows an example application of our approach to the example code shown in Figure 4. Our approach first splits the loop nest $\mathcal{L}$ in the execution phase into tiles of size $T_1 \times T_2$. At each inter-tile iteration, $(ii, jj)^{\mathrm{T}}$, we access the data blocks $X[ii..ii+T_1][jj..jj+T_2]$ and $Y[ii..ii+T_1][jj..jj+T_2]$ of index arrays $X$ and $Y$. When updating the value of an element of an index array in the initialization phase, we also update the maximum and the minimum values of the data block that contains this element. For example, when we update the value of $X[s_1, s_2]$, we also update the values $X_M[s_1/T_1, s_2/T_2]$ and $X_m[s_1/T_1, s_2/T_2]$. Note that $X_M[s_1/T_1, s_2/T_2]$ and $X_m[s_1/T_1, s_2/T_2]$ contain, respectively, the maximum and the minimum values of the block of array $X$ that contains $X[s_1, s_2]$.

At each inter-tile iteration in the execution phase, for each array that is irregularly referenced, we compute the set of elements that might be referenced by the intra-tile loop iterations. As we mentioned above, the subscript of an irregular array reference is an affine function of the index variables of the loops and the values of the elements of the index arrays. Since the range of the index variable for each loop can be statically determined at compilation time, and the range of the values of the elements for each index array block that can be accessed in this inter-tile iteration has been computed and stored in the auxiliary arrays, we can conservatively compute the range of the subscripts of the irregular array references using the techniques presented in Chapter 8 of [16]. Note that we must be conservative in computing the set of the elements for an irregularly referenced array; that is, all the elements of an irregularly referenced array that might be accessed in the intra-tile loop iterations of the current inter-tile loop iteration must be included in the resulting set.

After computing the set of elements of an irregular array, say $A$, we can compute the benefit (profit) of using the SPM for these elements as follows:

$$C_A = N_A C_2 - (C_0 + C_1|S_A| + N_A C_3),$$

where $N_A$ is the number of accesses to array $A$, which is calculated at compilation time, and $C_0$ through $C_3$ are the costs, as defined in Section 2. In the transformed code given in Figure 6, since the body of the loop nest contains a single instruction that accesses array $A$ and two instructions that access array $B$, we have $N_A$ and $N_B$ equal to $T_1T_2$ and $2T_1T_2$, respectively, where $T_1T_2$ gives the number of intra-tile loop iterations that are executed in the current inter-tile iteration. For each array, if the number of data elements that need to be loaded into the SPM is less than the total available space in the SPM for the irregular arrays and the memory access cost for using the SPM is lower than the cost obtained when not using the SPM (which is indicated by $C_A > 0$ and $C_B > 0$ for arrays $A$ and $B$, respectively), we load the elements of this array into the SPM and all the references to this array in the current inter-tile iteration are redirected to the SPM; otherwise, we access the elements of this array directly from the main memory. Further, for a loop nest that accesses multiple arrays and the available space in the SPM cannot contain all the arrays, the array with the larger number of accesses has a higher priority to be loaded into the SPM, though, in principle, our approach can accommodate any other dynamic SPM management strategy. As an example, in the code given in Figure 6, when the available space in the SPM cannot hold both arrays $A$ and $B$, we try to load the tile of array $B$ first since $N_B > N_A$.

During the execution phase, we need to check whether an irregularly referenced array is in the SPM or in the main memory. Such checks usually involve conditional branches. Fortunately, the conditions indicated by these branches do not change across the intra-tile loop iterations within the same inter-tile loop iteration. Therefore, we can avoid the overheads due to these conditional branches by dynamic code modification. For example, in the code given in Figure 6, the values of $\theta_A$ and $\theta_B$ are determined before entering loop nest $\mathcal{L}'$ and are never changed during the execution of $\mathcal{L}'$. Therefore, we can dynamically replace the conditional branches that are based on $\theta_A$ and $\theta_B$ with appropriate unconditional jumps before entering loop nest $\mathcal{L}'$.

## 5. EXPERIMENTAL ANALYSIS

To measure the effectiveness of our approach in improving performance, we implemented it within an experimental compilation framework (built upon the SUIF infrastructure from Stanford University [4]) and tested it using a set of seven embedded application codes that have irregular data access patterns. The code size overheads introduced by our

```
// compute the value of index array X
for t = ... {
  // initialization phase
  X[s1, s2] = ...;
  if(X[s1, s2] > X_M[s1/T1, s2/T2]) X_M[s1/T1, s2/T2] = X[s1, s2];
  if(X[s1, s2] < X_m[s1/T1, s2/T2]) X_m[s1/T1, s2/T2] = X[s1, s2];
  Y[s3, s4] = ...;
  if(Y[s3, s4] > Y_M[s3/T1, s4/T2]) Y_M[s3/T1, s3/T2] = Y[s3, s4];
  if(Y[s3, s4] < Y_m[s3/T1, s4/T2]) Y_m[s3/T1, s3/T2] = Y[s3, s4];
  ...
  // execution phase
  L': for ii = ...
    for jj = ...
      S_A = {A[2x, y] | X_m[ii, jj] ≤ x ≤ X_M[ii, jj]
                        ∧ Y_m[ii, jj] ≤ y ≤ Y_M[ii, jj]};
        // S_A: the set of the elements of A that need to be loaded into the SPM
      C_A = N_A C_2 − (C_0 + C_1|S_A| + N_A C_3)
        //C_A: the benefit of loading A into the SPM
        //N_A: the number of references to A in the execution phase
      S_B = {B[x + 1, y], B[x, y + 1] | X_m[ii, jj] ≤ x ≤ X_M[ii, jj]
                                        ∧ Y_m[ii, jj] ≤ y ≤ Y_M[ii, jj]};
        // S_B: the set of the elements of B that need to be loaded into the SPM
      C_B = N_B C_2 − (C_0 + C_1|S_B| + N_B C_3)
        //C_B: the benefit of loading B to the SPM
        //N_B: the number of references to B in the execution phase
      θ_A = false; θ_B = false
      S_SPM = the total size of the SPM available for irregularly referenced arrays;
      if(C_B > 0 ∧ |S_B| ≤ S_SPM) {
        B'[...] ⇐ S_B; // load data elements into the SPM
        θ_B = true; S_SPM = S_SPM − |S_B|;
      }
      if(C_A > 0 ∧ |S_A| ≤ S_SPM) {
        A'[...] ⇐ S_A; // load data elements into the SPM
        θ_A = true; S_SPM = S_SPM − |S_A|;
      }
      for i = ii to ii + T_1
        for j = jj to jj + T_2 {
          if(θ_A)
            r_1 = A'[2 * X[i, j]][Y[i, j]] //load from the SPM
          else
            r_1 = A[2 * X[i, j]][Y[i, j]] // load from the main memory
          if(θ_B) {
            r_2 = B'[X[i, j] + 1][Y[i, j]] //load from the SPM
            r_3 = B'[X[i, j]][Y[i, j] + 1] //load from the SPM
          } else {
            r_2 = B[X[i, j] + 1][Y[i, j]] //load from the off-chip main memory
            r_3 = B[X[i, j]][Y[i, j] + 1] //load from the off-chip main memory
          }
          SUM = r_1 + r_2 + r_3 //access data elements in the SPM
        }
}
```

**Figure 6. The optimized code for the example in Figure 4. During the initialization phase, the code inserted by our compiler tracks the block maximum and minimum values for each index array. Before going into the execution phase, we dynamically determine which arrays need to be loaded into the SPM.**

approach was less than 5% for all the applications tested. For each of these applications, we collected results with three different versions, referred to as *dynamic*, *static*, and *hybrid*. The dynamic scheme makes use of a conventional cache memory (16KB; direct-mapped; 16 byte block size) as its on-chip memory. The contents of this on-chip cache memory are managed by the hardware at runtime. The other two schemes work with an SPM. The scheme called static represents the state-of-art in SPM optimization for data accesses. Specifically, it analyzes the code and identifies, for each loop nest, the data blocks that would benefit from SPM placement. Note that, this is, in a sense, also a dynamic scheme since the contents of the SPM are changed dynamically as we move from one loop nest to another. However, unlike the case with a conventional cache memory, these dynamic data movements are decided at compile time (and this is why we call it static). This implementation is based on the approach presented in [10]. The third scheme evaluated, hybrid, is the one proposed in this paper.

| Application | Description | Data Size (KB) | Execution Cycles (M) |
|---|---|---|---|
| dither | Image dithering and resolution tuner | 327.31 | 821.12 |
| power-x | Adaptive image recognition | 266.88 | 734.33 |
| ur-direct | Motion estimation | 505.47 | 907.66 |
| med 3.0 | Medical imaging | 492.70 | 882.63 |
| mesh | Parallel mesh generator | 617.54 | 1,117.12 |
| trio-encr | Public key encryption | 498.09 | 963.15 |
| terpa 1.1 | Medical imaging | 673.35 | 1,306.72 |

**Table 1. Application codes.**

We evaluated these three SPM management schemes using a simulation environment built upon SimpleScalar [5]. We assume that the embedded system uses a 2-issue processor core with 400MHz clock frequency. The SPM size is 16KB with 1 cycle hit latency and 90ns miss latency.

Table 1 lists the applications used in this study and their important properties. An important characteristic of these applications is that the data array accesses in all of them are dominated by irregular accesses (through index arrays). The second column of Table 1 gives a description of each application and the third column gives the amount of data it manipulates. The next column gives the number of execution cycles taken by the dynamic scheme (i.e., the one with the conventional cache memory). In the rest of this paper, the execution cycles results of the schemes static and hybrid are given as normalized values with respect to those of the dynamic scheme.

Figure 7 shows the normalized execution cycles for our applications under the three different schemes explained above. Our first observation from these results is that the static scheme (which represents one of state-of-the-art methods in SPM management) performs very poorly as compared to the dynamic scheme and slows down the applications in our suite by 67.8% on the average.[4] This shows that the current SPM management approaches may not be a good choice for this type of *irregular applications,* where a compiler cannot fully extract the data access pattern (due to irregular data accesses). This general trend is reversed in only one application (trio-encr) since, in that application, a single loop nest, which can partially be analyzable by the compiler, dominates the execution time. When we compare the dynamic and hybrid schemes on the other hand, we see that the hybrid scheme generates much better results than the dynamic scheme. Specifically, it reduces execution cycles by 23.2% on average. In fact, it generates better results than the dynamic scheme for all the application codes we have.

## 6. RELATED WORK

There exist several prior studies on using SPMs for instruction accesses. For example, Sias et al. [13] present

4 This is a completely different result than those obtained with *regular codes* by the prior work.

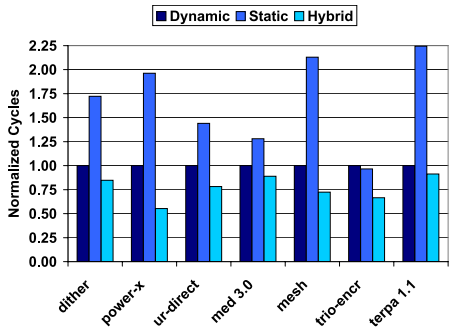


**Figure 7. Normalized execution cycles.**

compiler techniques, which arrange for 70-99% of the fetched operations to come from a statically managed 256-instruction loop buffer. Bellas et al. [6] present an SPM management scheme for instruction accesses. Steinke et al. [14] focus on a strategy for placing program and data objects into an SPM for saving energy. Lee et al. [11] focus on reducing the energy consumption due to instruction accesses using a software-managed SPM (called loop cache). As compared to our approach presented in this paper, all these schemes focus on instruction accesses. In comparison, we target data references. On the data accesses side, several studies focused on the effective use of SPMs. Absar and Catthoor [3] present a compiler-based approach for exploiting SPM in presence of irregular array references. To our knowledge, this is the only other SPM work that targets irregular applications. Our approach differs from theirs in that we use runtime information to decide whether to load data into the SPM and which data need be loaded, while their approach makes all decisions during compilation time. Francesco et al. [8] present an integrated hardware/software solution to support SPMs. Panda et al. [12] present a powerful static data partitioning scheme for efficient utilization of scratch-pad memories when they co-exist with conventional caches. Benini et al. [7] discuss an elegant memory management scheme that is based on keeping the most frequently used data items in a software-managed memory (instead of a conventional cache). Kandemir et al. [10] propose a dynamic SPM management scheme for data accesses. Hallnor and Reinhardt [9] present a software-managed cache architecture and a data replacement algorithm. Wang et al. [15] discuss a framework for analyzing the flow of values and data reuse for on-chip memories. In contrast to these studies, our work exploits both compilation time and runtime information to improve the behavior of a data SPM in the presence irregular array references, while the prior efforts do not take irregular array references into account.

## 7. CONCLUDING REMARKS

Scratch-pad memories (SPMs) are being increasing used in embedded systems and recent research has studied several compiler optimizations, designed specifically for these software-managed on-chip memories. Most of these techniques are able to handle only applications with regular data access patterns. Unfortunately, not all embedded applications exhibit only regular data access patterns (e.g., some codes from embedded multi-media) and we need compiler techniques to optimize their behavior when they need to be executed in the SPM-based architectures. This paper addresses this problem by proposing a scheme that enlists both compiler's and runtime system's help. Our experiments with seven embedded applications dominated by irregular data access patterns show that this hybrid scheme is very effective in practice.

## References

[1] Arm11 family. http://www.arm.com/products/CPUs/families/ARM11Family.html.

[2] Intel application processors. http://developer.intel.com/ design/ pca/ applicationsprocessors/.

[3] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *Proc. Design, Automation and Test in Europe*, volume 2, pages 1162–1167, 2005.

[4] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[5] T. M. Austin and D. Burger. The simplescalar architectural research tool set. http://www.cs.wisc.edu/∼mscalar/simplescalar.html.

[6] N. Bellas, I. N. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *International Conference on Computer Design*, 1999.

[7] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. In *IEEE Design & Test of Computers*, Apr. 2000.

[8] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for runtime scratchpad management. pages 238–243, 2004.

[9] E. G. Hallnor and S. K. Reinhardt. A fully-associative software-managed cache design. In *International Conference on Computer Architecture*, 2000.

[10] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *In Proc. the 38th conference on Design automation*, June 2001.

[11] L. H. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *ISLPED*, Aug. 1999.

[12] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad-memory in embedded processor applications. In *European Design and Test Conference*, Mar. 1997.

[13] J. Sias, H. Hunter, and W. Hwu. Enhancing loop buffering of media and telecommunication applications using low-overhead predication. In *the Annual International Symposium on Microarchitecture*, Dec. 2001.

[14] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[15] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In *9th International Conference on Compiler Construction*, Mar. 2000.

[16] M. Wolf. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, CA, 1996.

[17] M. Wolfe. Parallelizing compilers. *ACM Comput. Surv.*, 28(1):261–262, 1996.