Compiler-Directed Scratch Pad Memory Hierarchy Design and Management*

M. Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

A. Choudhary
ECE Department
Northwestern University
Evanston, IL 60208, USA
choudhar@ece.nwu.edu

ABSTRACT

One of the primary challenges in embedded system design is designing the memory hierarchy and restructuring the application to take advantage of it. This task is particularly important for embedded image and video processing applications that make heavy use of large multi-dimensional arrays of signals and nested loops. In this paper, we show that a simple reuse vector/matrix abstraction can provide compiler with useful information in a concise form. Using this information, compiler can either adapt application to an existing memory hierarchy or can come up with a memory hierarchy. Our initial results indicate that the compiler is very successful in both optimizing code for a given memory hierarchy and designing a hierarchy with reasonable performance/size ratio.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; D.3.4 [**Programming Languages**]: Processors—*Compilers;Optimization*

General Terms

Design, Experimentation, Performance

Keywords

Data Reuse, Scratch Pad Memory, Memory Hierarchy

1. INTRODUCTION

Embedded system design has undergone a major renaissance in the last five years. An important characteristic of this change is that software is playing an ever increasing role in system design. Consequently, automatic compiler support for optimizing embedded software is of critical importance. Classical compiler methods alone, however, may not be sufficient for attaining the highest levels of performance from an embedded platform. Instead, embedded system-specific issues should also be accounted for. Software-managed memories (called scratch-pad memories [9]) are an important part of design process, particularly in image and video processing applications that make heavy use of large multi-dimensional arrays of signals and nested loops. Compiler can restructure a given code to adapt it to a given memory hierarchy; but, it can also help designer to come up with a memory hierarchy for a given application. This hierarchy can then be used as a starting point for a more sophisticated memory hierarchy optimization.

While it might be possible (for some applications) to adopt a plain (single level) scratch-pad memory architecture and obtain large savings in performance and energy [9, 7], many access patterns encountered in embedded image and video applications exhibit data reuse in multiple loop levels, thereby necessitating a multi-level memory hierarchy if we are to obtain the best results. Therefore, we believe that as embedded codes get more and more complex, memory architectures that contain multi-level scratch-pad memories will be more widely employed. This will also make it extremely important to provide compiler support for optimizing applications in such hierarchies.

In this paper, we present a compiler-directed memory hierarchy design and code optimization strategy. More specifically, we make the following major contributions:

- We show how an optimizing compiler can manage the flow of data to/from a scratch-pad memory (SPM) hierarchy using a mathematical abstraction based on reuse vectors and reuse matrices.
- We present an algorithm that, given an input code, tries to come up with an SPM hierarchy. This hierarchy might be used as an input for a more detailed systematic analysis and optimization framework.
- We present an algorithm that enforces a memory hierarchy for a given code. For example, using this algorithm, it might be possible to transform a given code in such a way that a majority of array references in the code would work best with a specific depth of memory hierarchy.
- We report experimental data showing the effectiveness of our approach. Our results indicate that the compiler is very successful in both optimizing code for a given hierarchy and designing a hierarchy with reasonable performance/size ratio.

Our approach is different from many previous studies on software-directed memory management. First, in contrast to studies in [9, 10, 7, 2] where the main focus is a single-level memory management, we focus on a memory hierarchy. We focus on both designing a memory hierarchy and exploiting a given hierarchy for current data access pattern. As compared to the design methodology in [3], our approach is simpler as it uses a reuse vector/matrix abstraction (which can be extracted by an optimizing compiler during data access pattern/data dependence analysis). By using this abstraction, we can easily decide the flow of data through memory hierarchy. In addition, this abstraction also allows us to come up with an initial hierarchy design very quickly.

The rest of this paper is organized as follows. Section 2 presents background material and gives details of our optimization strategy. Section 3 discusses our implementation, introduces our experimental framework, and presents performance and energy consumption data. Section 4 summarizes our major results.

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

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA

2. OUR APPROACH

2.1 Background

The iteration space $\mathcal{I}$ of a loop nest contains one point for each iteration. An iteration vector can be used to label each such point in $\mathcal{I}$. We use $\vec{i} = (i_1, ..., i_n)$ to represent the iteration vector for a loop nest of depth n, where each i_k corresponds to a loop index (or its value at a specific point in time), starting with i_1 for the outermost loop index. In fact, an iteration space $\mathcal{I}$ can be viewed as a polyhedron bounded by the loop limits.

The subscript function for a reference to an array U is a mapping from the iteration space $\mathcal{I}$ to the data space $\mathcal{D}$. The data space can also be viewed as a polyhedron bounded by array bounds. A subscript function defined this way maps an iteration vector to an array element. In this paper, we assume that subscript mappings are affine functions of the enclosing loop indices and symbolic constants. Many array references found in embedded image and video processing codes fall into this category. Under this assumption, a reference to an array U can be represented as $L^U\vec{i} + \vec{b}^U$ where L^U is a linear transformation matrix called the access (reference) matrix, $\vec{b}^U$ is the offset vector, and $\vec{i}$ is the iteration vector [12]. If the loop is n-deep and the array U is m-dimensional, L^U is $m \times n$ and $\vec{b}^U$ has m elements. For example, the reference $V[k][j]$ in Figure 1(a) can be written as:

$$L^V\vec{i} + \vec{b}^V = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

According to Li [8], if $\vec{i}$ and $\vec{j}$ are two iteration vectors that access the same memory location, $\vec{r} = \vec{j} - \vec{i}$ is called a reuse vector (assuming that $\vec{j}$ is lexicographically greater than $\vec{i}$). For a loop of depth n, the reuse vector is n dimensional. The loop corresponding to the first non-zero element from top is the one that carries the corresponding reuse. If there are more than one reuse vector in a given direction, they are represented by the lexicographically smallest one. The reuse vector gives us useful information about how array elements are used by different loops in a given nest. In this paper, we mainly focus on self reuses (that is, the reuses originating from individual references) as in very rare circumstances group reuse (that is, the reuse between different references to the same array) brings additional benefits that cannot be exploited by self reuse [8]. There is a temporal reuse due to reference $L^U\vec{i} + \vec{b}^U$ if and only if there exist two different iterations $\vec{i}$ and $\vec{j}$ such that $L^U\vec{i} + \vec{b}^U = L^U\vec{j} + \vec{b}^U$; that is, $\vec{j} - \vec{i} \in Ker\{L^U\}$. This last expression indicates that the temporal reuse vector $\vec{r} = \vec{j} - \vec{i}$ belongs to the kernel set (null set) of L^U [12].

As an example, let us focus on the matrix multiply nest shown in Figure 1(a). The reuse vector due to $U[i][k]$ is $(0, 1, 0)$. Informally, what this means is that, for fixed values of loops i and k, the successive iterations of the j loop access the same element from this array. That is, the j loop carries (exhibits) temporal reuse for this reference.

The reuse vectors coming from individual references make up a reuse matrix, R. In considering the matrix multiply code again, we find that:

$$R = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

The first, second, and third columns in this matrix correspond to the reuse vectors for $W[i][j]$, $U[i][k]$ and $V[k][j]$, respectively. As will be discussed shortly, each column indicates which array sections should be brought into (and discarded from) the SPM at what time.

A linear one-to-one mapping between two iteration spaces can be represented by a square, non-singular matrix T. Using such a transformation, each element $\vec{i}$ of the original iteration space $\mathcal{I}$ is mapped to $\vec{i'} = T\vec{i}$ on the transformed iteration space $\mathcal{I}'$. Converting the original loop nest to the transformed nest is a two-step process [12]. First, each array reference $L^U\vec{i} + \vec{b}^U$ is transformed to $L^UT^{-1}\vec{i'} + \vec{b}^U$. Second, the loop bounds are transformed accordingly. Finding the new loop bounds can necessitate the use of Fourier-Motzkin elimination, details of which can be found in [12]. It can be shown that if T is a linear transformation matrix, $\vec{r}$ is a reuse vector, and R is a reuse matrix, $T\vec{r}$ and TR are the transformed reuse vector and the transformed reuse matrix, respectively [8].

In Sections 2.2 through 2.5, we assume the existence of an SPM hierarchy and focus on how to manage data flow through this hierarchy. In Section 2.6, we show how to design a memory hierarchy based on data reuse. Till Section 2.4, we exclusively focus on a two-level memory hierarchy which contains a large (slow and energy-consuming) memory and a small (fast and less energy-consuming) memory. The problem is then to decide what data to bring to the fast memory at what time and how to decide when data in the fast memory are not useful anymore. We use the terms (software-controlled) memory hierarchy and SPM hierarchy interchangeably.

2.2 Reuse Based Data Transfers

Recall that $U[i][k]$ in Figure 1(a) leads to a reuse vector of $(0, 1, 0)$. Suppose that we have a section of array U in Figure 1(a), $U[i][1 : N]$ (the ith row), residing in the fast memory. Since all loop iterations

$$(i, 1, 1), (i, 1, 2), ..., (i, 1, N), (i, 2, 1), (i, 2, 2), ..., (i, 2, N),$$

$$.....(i, N, 1), (i, N, 2), ..., (i, N, N)$$

access this same array section, this section can be kept in the fast memory (in the fast SPM) during the entire execution of these iterations; and, following the execution of the iteration (i, N, N), it can be discarded from the fast memory. It should also be noted that this section can be brought into the fast memory just before the iteration $(i, 1, 1)$ starts execution (i.e., just before the j loop is entered). For clarity, we represent these iterations collectively using $(i, *, *)$, where $*$ denotes all iterations in the corresponding loop level; we also represent this array section using $U[i][*]$, where $*$ indicates all elements in the corresponding dimension. To sum up, by just considering the value of 1 in this reuse vector (the second entry), we can decide the point in the code at which an array section (in our case, this is the ith row of U) should be brought into the fast memory and the point at which it should be removed from the fast memory. This brings us to the following conclusion:

> *An array section should remain in the fast memory from the start of the loop that carries reuse for the corresponding array reference to the termination of the same loop.*

Let us consider the following generic n-deep loop nest that accesses an m-dimensional array:

$$\begin{array}{l} for(i_1 = {i_1}^l; i_1 \leq {i_1}^u; i_1++) \\ \quad for(i_2 = {i_2}^l; i_2 \leq {i_2}^u; i_2++) \\ \quad\quad for(i_3 = {i_3}^l; i_3 \leq {i_3}^u; i_3++) \\ \quad\quad ... \\ \quad\quad\quad for(i_n = {i_n}^l; i_n \leq {i_n}^u; i_n++) \\ \quad\quad\quad\quad U[f_1(i_1, i_2, .., i_n)][f_2(i_1, i_2, .., i_n)]...[f_m(i_1, i_2, ..., i_n)] \end{array}$$

In this nest, each $f_j(.)$ $(1 \leq j \leq m)$ is an affine function corresponding to a subscript expression. ${i_k}^l$ and ${i_k}^u$ are the lower and upper bounds for loop i_k. Assume that $\vec{r} = (r_1, r_2, ..., r_n)$ is the corresponding reuse vector. Assume further (for now) that there is only a single non-zero element (say r_k) in $\vec{r}$; all other elements are assumed to be zero. In other words, the loop i_k is the one that carries the reuse.

In this case, the compiler can make use of the fast memory as follows. Let us use $f_l(i_1, i_2, .., i_{k-1}, *, *, .., *)$ to denote all indices enumerated in a dimension l by fixing the first $k-1$ loops at specific values $(i_1, i_2, .., i_{k-1})$ and considering all values (within the loop bound ranges) for all other loop index positions. Just before executing the loop i_k, the compiler brings the array section containing all elements represented by $U[f_1(i_1, i_2, .., i_{k-1}, *, *, .., *)]\,[f_2(i_1, i_2, .., i_{k-1}, *, *,$

$.., *)]$ ... $[f_m(i_1, i_2, .., i_{k-1}, *, *, .., *)]$ into the fast memory, and keeps them there while executing all iterations in $(i_1, i_2, .., i_{k-1}, *, *, .., *)$. When the last iteration in this set has been executed, the section can be removed from the fast memory. If this is done, the said set of elements would reside in the fast memory as long as they are reused, and as soon as the last iteration that reuses the elements is completed, the set can be removed from the fast memory. It should be emphasized that if any element in this set is updated while it is in the fast memory, it should be written back to slow memory.

If there are multiple references to the same array, the reuse vectors are considered together and the corresponding data transfers are combined as much as possible. Consider the nested loop in Figure 1(b). We have two reuse vectors: $(0, 1, 0)$ (due to reference $U[i][k]$) and $(0, 0, 1)$ (due to reference $U[i][j]$). We note that the data transfer indicated by $(0, 0, 1)$ is subsumed by the transfer indicated by $(0, 1, 0)$; that is, once the transfer of the ith row to the fast memory is complete (just before entering the j loop), there is no need for performing an extra transfer for the element $U[i][j]$. Let us now focus on the code given in Figure 1(c). In this case, the reuse vectors are $(0, 1, 0)$ and $(0, 1, 0)$. Consequently, both data transfers should be performed before the j loop is entered. However, one data transfer contains the ith row of the array whereas the other contains the ith column. Except for one element, these two transfers do not overlap; so, they should be performed separately. Our current implementation uses the following rule to decide whether the transformations required by different references to the same array can be combined. Let ${L_1}^U \vec{i} + \vec{b_1}^U$ and ${L_2}^U \vec{i} + \vec{b_2}^U$ be two references to array U. If ${L_1}^U = {L_2}^U$, there is a good chance that there exists significant amount of data reuse between these two references (even if $\vec{b_1}^U \neq \vec{b_2}^U$). Consequently, in this case, the data transfers due to these references can be combined and performed together. On the other hand, if ${L_1}^U \neq {L_2}^U$, these two references are treated as if they belong to different arrays. If a nest contains references to multiple arrays, each array is treated independently and a separate data transfer is performed for each array.

2.3 Transformations

In some cases, it may not be clear which loop carries the reuse. To make the loop carrying the reuse explicit, it may be necessary to apply loop transformations. As an example, consider the loop nest shown in Figure 1(d). When considered alone, none of the references in this nest has temporal reuse. However, there is a group reuse in this nest with a reuse vector of $\vec{r} = (1, -1)$. Consequently, it is not trivial to find the point in the code to perform memory transfers so that we can exploit the fast memory. However, if we transform this nest using the loop transformation matrix

$$T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

we obtain the nest in Figure 1(e). We now see that the reuse vector for this transformed nest is $\vec{r} = (0, 1)$. This means that only the innermost loop carries reuse, and we can make use of the fast memory. To see how the reuse is exploited in this case, let us list the array elements accessed by a few initial iterations:

$$\begin{aligned}
(i' = 3, j' = 2) &\Rightarrow U[2][1] \text{ and } U[1][2] \\
(i' = 4, j =' 2) &\Rightarrow U[2][2] \text{ and } U[1][3] \\
(i' = 4, j' = 3) &\Rightarrow U[3][1] \text{ and } U[2][2] \\
(i' = 5, j' = 2) &\Rightarrow U[2][3] \text{ and } U[1][4] \\
(i' = 5, j' = 3) &\Rightarrow U[3][2] \text{ and } U[2][3] \\
(i' = 5, j' = 4) &\Rightarrow U[4][1] \text{ and } U[3][2]
\end{aligned}$$

We observe that (for a specific i') the iterations in $(i', *)$ cause the reuse of elements $U[j'][i' - j']$, where $max(2, i' - N + 1) \leq j' \leq min(N, i' - 1) - 1$. Therefore, just before entering the j' loop (in the transformed nest), these elements can be brought into the fast memory. As compared to the matrix multiply case, this example illustrates two interesting points. First, in some cases, loop transformations might be necessary for identifying where (in the code) to perform data transfers. Second, the data transfers for different values of the outer loop are of different sizes. So, if we are to design a memory hierarchy, in determining the size of the fast memory, we should consider the maximum number of reused elements over all i' values.

Another question in this example is how to determine the transformation matrix to use. We can determine the transformation matrix using the default and desired reuse vectors. For our current example, the default reuse vector is $\vec{r_d} = (1, -1)$. Since we have only two loops in the nest, if want to make sure that the reuse will be carried by one of them, the desired reuse vector should $\vec{r_s} = (0, 1)$. Then, from $\vec{r_s} = T\vec{r_d}$, we can determine the entries of T. Note that we used loop transformations in this section to modify the group temporal reuse vector so as to exploit a two-level memory hierarchy. As will be discussed shortly, loop transformations can also be used for modifying the temporal reuse vectors.

2.4 Multi-Level Hierarchy

When there are multiple non-zero elements in a given reuse vector, we can use this fact for exploiting a multi-level memory hierarchy. In the most general case, we can exploit a memory level for each non-zero entry in the reuse vector. To illustrate this, let us consider the four-deep nest in Figure 1(f). For $U[i][k]$, the reuse vector is $(0, 1, 0, 1)$ which indicates that there are reuses in j and l loops (although the reuse itself is carried by the j loop). We can exploit these reuses as follows. Before entering the j loop, the i row of this array can be transferred from the slow memory (called Level 1) to the fast memory (called Level 2). In addition to that, if we have an even faster memory (called Level 3), the element $U[i][k]$ can be transferred to this memory (just before the l loop) and can be used from there throughout the execution of the l loop. In other words, at a given time, one row of array U can be in Level 2 SPM and one element from this row can be in Level 1 SPM (the fastest level). This is depicted in Figure 2. A similar data transfer scheme is valid for reference $V[j][l]$ as well. Before entering the i loop, the entire array can be transferred to Level 2 (if we have that much space in Level 2), and a row the array can be transferred to Level 1 before the k loop.

It is important to note that just the fact that there are two non-zero entries in the temporal reuse vector does not mean that we can use the nest in question only with a three-level memory hierarchy. In fact, we can still use it with a two-level memory hierarchy and the fact that there are two non-zero entries in the reuse vector gives the compiler flexibility in scheduling data transfers. For example, focusing on reference $U[i][k]$ in Figure 1(f), we have two options assuming a two-level memory: (i) we can perform data transfer before the j loop, or (ii) we can perform data transfer before the l loop. Note that transfer before the j loop requires a larger fast memory space, but it is also expected to give better results. It is also important to emphasize that loop transformations can make a nested loop written for a two-level of memory hierarchy suitable for a three-level of memory hierarchy, and vice versa. For example, the loop transformation matrix

$$T = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

can transform a reuse vector $(0, 1, 0, 0)$ to $(1, 0, 1, 0)$. Note that while $(0, 1, 0, 0)$ is suitable for a two-level of memory hierarchy, $(1, 0, 1, 0)$ is suitable for a three-level of memory hierarchy.

2.5 Imperfectly Nested Loops

While all examples given so far have focused on perfect nests only, our reuse vector/matrix based strategy can work with imperfectly-nested loops as well. An imperfect nest is the one that contains assignment statements between loops, whereas in a perfect nest all assignment statements are nested within the innermost loop. In our framework, imperfectly-nested loop nests in general present opportunities for min-

```
for(i = 1; i ≤ N; i++)
 for(j = 1; j ≤ N; j++)
  for(k = 1; k ≤ N; k++)
   W[i][j]+ = U[i][k] * V[k][j];
```
(a)

```
for(i = 1; i ≤ N; i++)
 for(j = 1; j ≤ N; j++)
  for(k = 1; k ≤ N; k++)
   c+ = U[i][k] + U[i][j];
```
(b)

```
for(i = 1; i ≤ N; i++)
 for(j = 1; j ≤ N; j++)
  for(k = 1; k ≤ N; k++)
   c+ = U[i][k] + U[k][i];
```
(c)

```
for(i = 2; i ≤ N; i++)
 for(j = 1; j ≤ N − 1; j++)
  k+ = U[i][j] + U[i − 1][j + 1];
```
(d)

```
for(i' = 3; i' ≤ 2N − 1; i'++)
 for(j' = max(2, i' − N + 1); j' ≤ min(N, i' − 1); j'++)
  k+ = U[j'][i' − j'] + U[j' − 1][i' − j' + 1];
```
(e)

```
for(i = 1; i ≤ N; i++)
 for(j = 1; j ≤ N; j++)
  for(k = 1; k ≤ N; k++)
   for(l = 1; l ≤ N; l++)
    d+ = (U[i][k] * U[i][k]) − V[j][l];
```
(f)

```
for(i = 1; i ≤ N; i++)
 for(j = 1; j ≤ N; j++)
  for(k = 1; k ≤ N; k++)
   c+ = U[i][k] * V[j][k];
 for(j = 1; j ≤ N; j++)
  for(k = 1; k ≤ N; k++)
   d = c + U[k][i] + U[i][k];
```
(g)

```
for(jt = 1; jt ≤ N; jt = jt + S)
 for(i = 1; i ≤ N; i++)
  for(j = jt; j ≤ jt + S − 1; j++)
   for(k = 1; k ≤ N; k++)
    W[i][j]+ = U[i][k] * V[k][j];
```
(h)

Figure 1: Different code fragments.

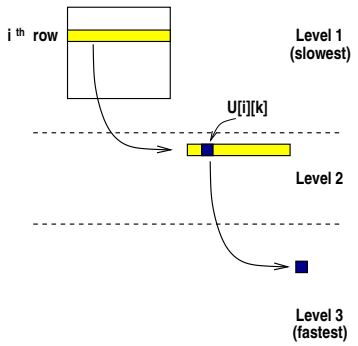

Figure 2: Exploiting three-level memory.

```
compute reuse matrix R = [r⃗1, r⃗2, ..., r⃗s]
level=1;
for i = 1, s
  k = number of non-zeros in r⃗i;
  if (k > level) then level = k; endif
endfor;
for l = 2, level
  capacity[level] = 0;
endfor;
for i = 1, s
  current_level = 2;
  let r⃗i be [r1i, r2i, ..., rni];
  for j = 1, n
    if (rji ≠ 0) then
      amount = F(reference(r⃗i));
      capacity[current_level] = capacity[current_level] + amount;
      current_level = current_level + 1;
    endif;
  endfor;
endfor;
return(level,capacity[]);
```

Figure 3: Memory hierarchy design algorithm.

imizing data transfers by overlapping or combining them with each other. Consider, for example, the code fragment shown in Figure 1(g). Let us focus on a two-level memory hierarchy and on data transfers due to array U. Since both the sub-nests in this code has the same reference $U[i][k]$, the data transfer due to this reference can be performed just before the first j loop; there is no need to perform data transfer (due to this reference) before the second j loop. The second sub-nest has another reference to array U: $U[k][i]$. The data transfer due to this reference can be performed along with that due to $U[i][k]$ before the first j loop. After the execution of the second j loop, the elements of array U which reside in the fast memory can be discarded (as their reuse ends). Our current implementation handles perfectly-nested and imperfectly-nested loops of any depth. But, it does not attempt to optimize the hierarchical SPM usage across multiple, independent imperfectly-nested loops. That is, it does not try to take advantage of inter-nest data reuse. However, it should also be noted that handling imperfectly-nested loops means that our approach can be extended operate on an entire program as the entire code can be thought of as an imperfectly nested loop enclosed by an imaginary outermost loop that iterates only once.

2.6 Memory Design Algorithm

So far, we have assumed that a memory hierarchy already exists and tried to determine how this hierarchy can be exploited by capturing the reuse of data using reuse vectors/matrices. Our reuse vector/matrix based analysis however can also be used for designing a suitable SPM hierarchy for a given code. In this subsection, we address this problem. Note that after obtaining an initial memory hierarchy using the technique presented in this subsection, more sophisticated techniques [3] can be used, if desired, to refine the hierarchy.

2.6.1 Hierarchy Design for a Given Code

We first present a strategy to determine the cache configuration given a loop nest. A sketch of our approach is given in Figure 3. Our approach tries to build a separate hierarchy for each nest and then combines these hierarchies into a single hierarchy by choosing the largest memory required for each level. The algorithm first builds the reuse matrix and then it constructs the memory hierarchy incrementally. It starts with the first reuse vector and determines how much SPM space we need in each level. It then moves to the second reuse vector. Considering this vector, the algorithm can increase either the number of levels or the space needed by existing level(s) or both. These incremental updates stop once the last reuse vector has been processed.

As an example, consider once more the matrix multiply code in Figure 1(a) and its reuse matrix in Section 2.1. Considering the first reuse

vector, $(0, 0, 1)$, we decide that the hierarchy should be at least two levels and that the smaller (and faster) memory should be able to hold $W[i][j]$. Considering the second reuse vector, $(0, 1, 0)$, we can see that the number of levels does not need to be increased (as this reuse vector has the same number of non-zero elements as the previous one). However, it should be large enough to hold an entire row of U. Finally, the last reuse vector $(1, 0, 0)$ implies that the fast memory should be able to hold the entire array V. Based on these, our algorithm decides that we need a two-level SPM hierarchy and that the faster level should have a capacity of at least N^2+N+1 elements. By default, we assume that the slower level has a capacity to hold the entire working set (N^3 elements). In the algorithm in Figure 3, we first compute the number of levels in the hierarchy and then find the capacity of each level. The function 'reference(.)' gives the array reference corresponding to the reuse vector under consideration, and the '$F(.)$' function computes the SPM space demanded by that reference (see Section 2.2).

We next discuss how to modify this strategy if the underlying system has some memory space constraints. For example, in case we may not be able to store the entire V array (in our current example) in the fast memory, we need to modify the code slightly. We note that strip-mining, a loop transformation that divides the iteration space of a given loop into stripes [12], can be useful in this context. In our example, we can strip-mine the j loop as shown in Figure 1(h). In this code, S is the stripe size. Now, the total space requirement for the fast memory (considering only i, j, and k loops) is $NS + N + 1$. By choosing a suitable value for S, the space demand (put on the fast memory) can be kept under control.

2.6.2 Enforcing a Hierarchy

The algorithm presented in the previous subsection does not try to modify the code to enforce it to adopt a pre-specified memory hierarchy; that is, it does not modify the reuse matrix; it just builds a hierarchy considering the current reuse matrix. In many cases, however, we might want to modify the reuse matrix to control the number of levels in the hierarchy. A reuse matrix can be changed using two types of transformations:

- The transformations that change the number of zeroes and non-zeroes in the reuse vectors.
- The transformations that re-order zeroes and non-zeroes in the reuse vectors (without changing their numbers).

Since in general the second type of transformation can be reduced to the first type, we focus only on the first types of transformations here. Our approach can take a reuse matrix and transform it to another reuse matrix such that in the resulting matrix most reuse vectors demand a t-level memory hierarchy, where t being a value between 2 and n (the number of loops in the nest). For the clarity of presentation, however, we present the algorithm for only the case when $t = 3$ (see Figure 4); that is, for each reuse vector in R, we would like to make all the entries in the vector zero, except two entries (these two entries will represent the loops that exhibit reuse). Note that in general the order in which the reuse vectors in a given reuse matrix are processed might also be important. Our current approach orders reuse vectors from left to right considering the number of times each reuse vector occurs in the code. The most frequently used reuse vector occupies the leftmost column whereas the least frequently used one sits in the rightmost column. Then, for the most frequent reuse vector, the algorithm makes sure that only two non-zero elements are obtained (after the transformation). It achieves this by selecting suitable vectors for the first $n-2$ rows of T. In doing so, data dependences are also checked using the approach in [8] (this is not shown in the code for clarity). It then processes the remaining reuse vectors and selects suitable vectors for the remaining two rows. As soon as all rows of T have been determined, the algorithm returns the transformed reuse matrix (TR) and terminates.

compute reuse matrix $R = [\vec{r_1}, \vec{r_2}, \ldots, \vec{r_s}]$
let T be the transformation matrix and $\vec{t_1}, \vec{t_2}, \ldots, \vec{t_n}$ be the rows of T;
finished=false;
select $\vec{t_1}, \vec{t_2}, \ldots, \vec{t_{n-3}}, \vec{t_{n-2}}$ such that
 $\vec{t_1} \in Ker\{\vec{r_1}\}, \vec{t_2} \in Ker\{\vec{r_1}\} \ldots \vec{t_{n-3}} \in Ker\{\vec{r_1}\}, \vec{t_{n-2}} \in Ker\{\vec{r_1}\}$;
j = 2;
while (not finished)
 $bool_1$ = ($\vec{t_1} \in Ker\{\vec{r_j}\}$);
 $bool_2$ = ($\vec{t_2} \in Ker\{\vec{r_j}\}$);
 ....
 $bool_{n-2}$ = ($\vec{t_3} \in Ker\{\vec{r_j}\}$);
 k = number of trues in {$bool_1$,$bool_2$,...,$bool_{n-2}$};
 if (k $\leq$ (n-4)) then
  set $\vec{t_{n-1}}$ to vector in $Ker\{\vec{r_j}\}$;
  set $\vec{t_n}$ to vector in $Ker\{\vec{r_j}\}$;
  finished = true;
 else
  if (k = (n-3)) then
   if ($\vec{t_{n-1}}$ is not set) then
    set $\vec{t_{n-1}}$ to vector in $Ker\{\vec{r_j}\}$;
   else
   if ($\vec{t_n}$ is not set) then
    set $\vec{t_n}$ to vector in $Ker\{\vec{r_j}\}$;
    finished = true;
   endif;
  endif;
 endif;
 j = j + 1;
endwhile;
return(TR);

Figure 4: Memory hierarchy transformation algorithm.

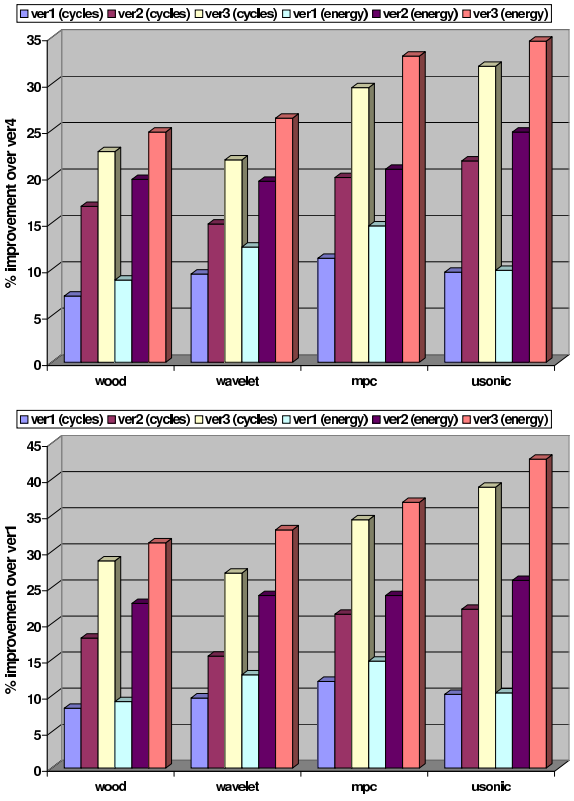

Figure 5: Percentage energy and performance improvements over `ver4`. Top: two-level memory hierarchy; Bottom: three-level memory hierarchy.

3. EXPERIMENTS

We used array-based versions of four embedded image processing applications to evaluate the effectiveness of our strategy. `wood` is a

color-based visual surface inspection method for wood properties. It builds the color histogram percentile features of the image to recognize wood surface defects with relatively low complexity. `wavelet` is a wavelet-based noise reduction and restoration application. `mpc` is a medical pattern classification application. It is used mainly for choosing therapy for peptic ulcers. Finally, `usonic` is a feature-based estimation algorithm for ultrasonic image sequences. The total input sizes used for `wood`, `wavelet`, `mpc`, and `usonic` are 271KB, 306KB, 586KB, and 410KB, respectively. For each benchmark code, we experimented with four different versions. `ver1` is a static SPM optimization strategy. It profiles the code and selects a set of array elements that has the highest reuse. It then puts these elements in the fast memory (at the beginning of the program) and keeps them there throughout the execution. `ver2` is a nest-based dynamic SPM management strategy. It changes the contents of SPM depending on the current access pattern; its details can be found elsewhere [7]. `ver3` is the reuse vector/matrix based SPM management strategy discussed in this paper. `ver4` is a version that uses a cache memory (instead of SPM) in conjunction with the locality-optimized version of each application. All versions tested here have been implemented in a source-to-source translator using the SUIF infrastructure [1]. Our compiler takes a C code as input and optimizes it taking into account the memory hierarchy. The compiler output (which is also a C code) is then fed into a custom simulator. The simulator models a memory hierarchy (which is constructed using SPMs) and assumes a simple, single-issue embedded core (running on 200MHz). For cache experiments, the simulator directly calls Dinero [4].

We performed two sets of evaluations. In the first set, we fixed the SPM hierarchy and evaluated the performance and energy consumption of each version. In the second set, for each version, we determined the total fast level memory size and energy consumption under a given performance bound. We calculated the access latencies for different sizes of SPMs and data caches using the CACTI tool [6]. To compute the per access energy costs for different sizes of memories, we employed a slightly modified form of the model proposed by Shiue and Chakrabarti [11]. While our simulator also gives the energy spent in interconnect, we found this value very small for all benchmarks used.

Figure 5 gives the performance (execution cycles) and energy results for two-level (top) and three-level (bottom) memory hierarchies. All values presented are normalized with respect to the corresponding value of `ver4` (the cache version). In the two-level memory, the simulated faster memory (SPM) size is 4KB; the cache used in the `ver4` version is also 4KB (two-way associative with a block size of 32 bytes). In the three-level memory hierarchy, the fastest SPM size is 2KB and the middle level memory is 6KB. In all cases, the first level memory (the slowest memory) is 1MB. These results show that our strategy outperforms both `ver1` and `ver2` in both energy and performance. We also note that the gains are larger with the three-level memory (as our reuse vector based strategy utilizes all the fast levels fully). The average performance improvements due to `ver1`, `ver2`, and `ver3` are 9.4%, 18.3%, and 26.5%, respectively, for the two-level memory. The corresponding values for the three-level memory are 10.0%, 19.2%, and 32.3%. Figure 6 shows the percentage increase in memory capacity and energy consumption (with respect to our approach) when all version generate the same performance results. Our experimentation methodology is as follows. First, we ran the algorithm in Figure 3 and determined a memory hierarchy for our version (`ver3`). We then recorded the total memory size (excluding the slowest memory, the size of which is always fixed at 1MB) used by our version and its execution time. Then, we ran other versions using several different levels and capacities, and found (for each version) the configuration that gave the same execution time as `ver3` (within an error margin of 2%). We observe from these results that in order to catch the performance of `ver3`, the other versions demand much larger memory space and (because of this) significantly larger energy consumption. Even `ver2` (which is the best among the remaining ones) increases the memory capacity (omitting the slowest one) by 35.4% and energy consumption by 16.5% on the average.

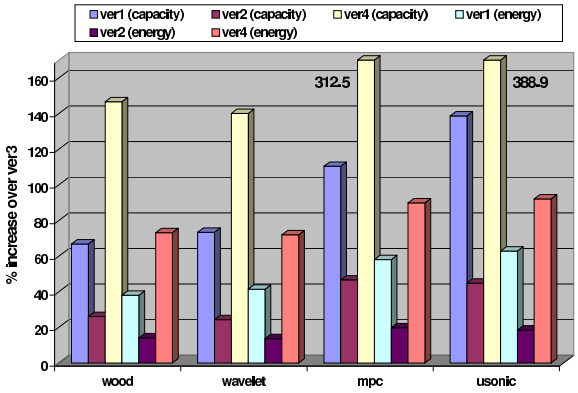

Figure 6: Percentage increase in memory size and energy over `ver3` under constant performance.

4. CONCLUSIONS

We presented and discussed a scratch-pad memory (SPM) hierarchy design and optimization framework. In this framework, the compiler has a central role in the sense that it manages the flow of data across a given hierarchy (by staging computation and data). Given an access pattern, the compiler can also come up with a memory hierarchy which might be used as a starting point for further analysis and synthesis. The effectiveness of this strategy was measured using four complete applications from the embedded image processing domain. Our results reveal that this optimization framework is very successful in reducing both energy and execution cycles and that it outperforms two previous approaches to SPM management.

5. REFERENCES

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

[2] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers*, pages 74–85, April-June, 2000.

[3] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. Custom memory management methodology – exploration of memory organization for embedded multimedia system design. *Kluwer Academic Publishers*, June, 1998.

[4] Dinero IV Trace-Driven Uniprocessor Cache Simulator. URL: http://www.cs.wisc.edu/∼markhill/DineroIV/

[5] J. Eyre and J. Bier. DSP processors hit the mainstream. *IEEE Computer Magazine*, pp. 51–59, August 1998.

[6] N. P. Jouppi and S. J. E. Wilton. An enhanced access and cycle time model for on-chip caches. *Research Report 93/5*, Compaq WRL, Palo Alto, CA, July 1994.

[7] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In Proc. *the 38th Design Automation Conference*, Las Vegas, NV, June 2001.

[8] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1993.

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

[10] P. R. Panda, N. D. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In Proc. *ISSS'97*, Antwerp, September 1997.

[11] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In Proc. *Design Automation Conference*, New Orleans, Louisiana, 1999.

[12] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.