

DRDU: A Data Reuse Analysis Technique for Efficient Scratch-Pad Memory Management

ILYA ISSENIN

University of California, Irvine

ERIK BROCKMEYER and MIGUEL MIRANDA

IMEC, Belgium

and

NIKIL DUTT

University of California, Irvine

In multimedia and other streaming applications, a significant portion of energy is spent on data transfers. Exploiting data reuse opportunities in the application, we can reduce this energy by making copies of frequently used data in a small local memory and replacing speed- and power-inefficient transfers from main off-chip memory by more efficient local data transfers. In this article we present an automated approach for analyzing these opportunities in a program that allows modification of the program to use custom scratch-pad memory configurations comprising a hierarchical set of buffers for local storage of frequently reused data. Using our approach we are able to both reduce energy consumption of the memory subsystem when using a scratch-pad memory by about a factor of two, on average, and improve memory system performance compared to a cache of the same size.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; C.3 [**Computer systems organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Performance, Algorithms

Additional Key Words and Phrases: Scratch-pad memory management, memory hierarchy, data reuse analysis, compiler analysis

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712, and is an expanded version of the *DATE 2004 Conference* paper [Issenin et al. 2004].

Authors' addresses: I. Issenin and N. Dutt, Center for Embedded Computer Systems, School of Information and Computer Science, University of California at Irvine, Irvine, CA 92697; email: {isse, dutt}@ics.uci.edu; E. Brockmeyer and M. Miranda, IMEC, Kapeldreef 75, B-3001 Leuven, Belgium; email: {miranda, brockmey}@imec.be.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1(212)869-0481, or permissions@acm.org

© 2007 ACM 1084-4309/2007/04-ART15 \$5.00 DOI 10.1145/1230800.1230807 <http://doi.acm.org/10.1145/1230800.1230807>

ACM Reference Format:

Issenin, I., Brockmeyer, E., Miranda, M., and Dutt, N. 2007. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.* 12, 2, Article 15 (April 2007), 28 pages. DOI = 10.1145/1230800.1230807 <http://10.1145/1230800.1230807>.

1. INTRODUCTION

Exploiting data reuse opportunities in loop-dominant applications is essential for energy-efficient memory hierarchies. The traditional approach for solving the data reuse problem employs the use of hardware-controlled caches. While this has been successfully used for general-purpose architectures, a hardware-only implementation has several drawbacks. A hardware controller approach adds additional power and area cost [Banakar et al. 2002]. Due to the lack of knowledge of future accesses, the placement of data in the cache is not optimal, which leads to higher miss rates. Besides this, it is not possible to achieve effective data prefetch (which helps to hide access latency), since not all of the programs expose sufficient spatial locality in the data accesses. For real, time applications, it is often unacceptable to use caches because of the difficulty in obtaining tight bound on worst-case execution time [Kandemir et al. 2001].

A proposed alternative to hardware caches is a “software-controlled cache.” For this, the decisions on when to allocate reused data to intermediate buffers (stored in scratch-pad memory) are done after analyzing the algorithm at compile time. The code for copying the data from main memory to buffers (using the processor or a DMA controller) is added to the original program, and the modified program is compiled using conventional compilers. In this scheme, the size of the buffers required to partially or completely eliminate repeated accesses to main memory determines the optimal memory hierarchy.

In this article we present an automated approach for performing data reuse analysis and show how it can be used to build a power-efficient scratch-pad-memory-based memory subsystem. Specifically, our approach creates a customized scratch-pad memory that employs a hierarchical buffer organization, and also inserts the appropriate code in the source to perform the necessary transfers to and from this customized scratch-pad memory organization. Our analysis technique allows exact detection of the amount of reused data, which leads to reduced size of a buffer (necessary for keeping this data) in comparison with other known techniques. We show the efficacy of our approach on several multimedia and streaming benchmark kernels that generate about a factor of two reduction in memory energy consumption.

The rest of the article is organized as follows. Section 2 presents related work. Section 3 describes our data reuse analysis technique and shows its place in our approach for memory subsystem optimization. Section 4 evaluates the benefits and overheads of our approach. Section 5 concludes the article.

2. RELATED WORK

Many papers have addressed the problem of data reuse in caches by improving locality of accesses, primarily by means of loop transformations (e.g., Bacon

et al. [1994] and McKinley et al. [1996]). However, we do not address this problem and assume that all possible loop transformations for improving locality are already performed before applying the technique presented in this article.

Panda et al. [1997] propose an approach that uses a scratch-pad memory to store scalars and some of the arrays of the application. The partitioning of data between the scratch pad and cache is done at compile time and does not change during the execution of the program. This leads to nonoptimal use of the scratch-pad memory, since during the execution of the program different parts of arrays and different scalars may be reused. The same drawback is inherent to the approaches presented by Steinke et al. [2002] and Verma et al. [2003], where the most frequently used data structures and basic blocks are statically placed in the scratch-pad memory. Udayakumaran et al. [2003] improved this scheme by dividing the program into regions. For each region a static allocation is derived by compiler. Between the regions necessary data transfers are performed to change the variable assignments. However, this approach can place only whole arrays in the scratch-pad memory. Cooper et al. [1998] proposed to use scratch-pad memory for storing spilled register values. Sjödin et al. [2001] and Avissar et al. [2002] used an ILP approach to determine which variables to place in scratch-pad memory. Verma et al. [2004] proposed to solve the problem of dynamic placement of memory objects to the scratch-pad memory in a way similar to register allocation. As in Udayakumaran et al. [2003], these approaches treat each array as one memory object, and placement of parts of array to the scratch-pad memory is not possible. Ozturk et al. [2004] presented an approach which uses compression to free space in scratch-pad memory when the space is needed for a new array tile.

At IMEC, a methodology for data transfer and storage exploration (DTSE) has been developed which includes a data reuse optimization step [Diguet et al. 1997]. However, no algorithm to automate this step was proposed. Van Achteren et al. [2003] describe an attempt to explore tradeoffs between scratch-pad memory size and power, assuming an optimal dynamic (runtime) placement of array elements in scratch-pad memory. However, no technique was presented for implementing such optimal placement. In another paper [Van Achteren et al. 2002] they present a technique for optimal placement, but it is limited to applications that contain two nested loops with one array reference inside.

Kandemir et al. [2002; 2001] addressed 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. However, if changing the order of accesses is not possible (e.g., due to dependencies) and the reused areas are not contiguous, the memory requirements for the scratch pad in their approach may significantly exceed the amount of actually reused data, hence leading to suboptimal results. Moreover, the proposed data replacement scheme is not designed to update only a part of the buffer while keeping the rest of the buffer intact. This means that the data which is no longer to be accessed is still taking space in the buffer in the scratch-pad memory, hence leading to increased memory size requirements.

<pre> for i=0 to 10 for j=0 to 10 for k=0 to 3 val = f(val) val += A[50i+3j+k] </pre>	<pre> int buff[34] for i=0 to 10 for m=0 to 33 buff[m]=A[50i+m] for j=0 to 10 for k=0 to 3 val = f(val) val += buff[3j+k] </pre>	<pre> int buff[1] for i=0 to 10 buff[0]=A[50i] for j=0 to 10 for k=0 to 3 if (k==3) buff[0]=A[50i+3j+3] val = f(val) val += (k%3==0)? buff[0]:A[50i+3j+k] </pre>
(a) original program (484 array accesses)	(b) applying the technique by Kandemir et al. [2002] (34 element buffer, 858 buffer accesses, 374 main memory accesses)	(c) applying our technique (1 element buffer, 374 buffer accesses, 374 main memory accesses)

Fig. 1. Comparison of several approaches for exploiting data reuse.

Our approach allows detecting and transforming a program to store reused parts of arrays in buffers that are located in scratch-pad memory. Decisions about which parts to store are made during the compile time, but data placement is made dynamically in the sense that the contents of buffers are updated at runtime by replacing the data no longer to be reused. Our technique handles any loop structure with any number of array references inside, as long as the index expressions are affine functions of the loop iterators. Data reuse opportunities are detected and exploited between different references, as well as for the same reference between different iterations of outer loops. Furthermore, our approach generates a hierarchical set of scratch-pad buffers, any of which can be selected later to be used in the transformed program. Our transformations do not change the order of loops or array accesses and allow achieving the minimal possible number of accesses to the main memory (i.e., every array element will be fetched no more than once from the main memory if all suggested buffers are implemented), and minimum buffer size, by keeping only the values that will be reused in it. This is the main advantage of our approach compared with other proposed techniques.

To show the difference in results between our and the aforementioned approaches, we apply them to the example depicted in Figure 1. The technique proposed by Panda et al. [1997] allocates the whole array *A* to the scratch-pad memory. In this case the code remains the same and we need a scratch-pad memory that can hold 534 data elements. If we use the approach of Kandemir et al. [2002], the code that can be generated is shown in Figure 1(b). The buffer size is 34 data elements, only 10 of which are used more than once. If we apply our technique, we obtain the code shown in Figure 1(c). In our approach, the number of accesses to the scratch-pad memory is two times less compared to the approach of Kandemir et al. [2002], while the number of accesses to the main memory stays the same. Moreover, the size of required scratch-pad memory is 34 times less in our case.

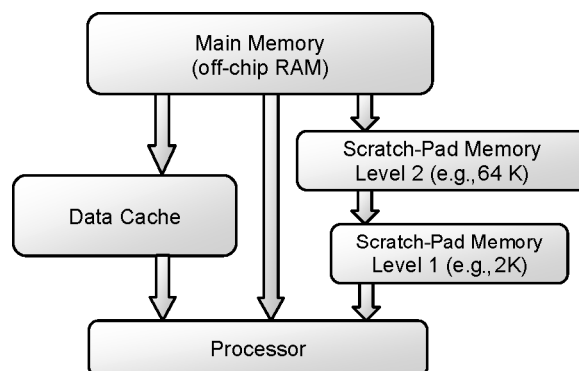


Fig. 2. Memory subsystem architecture.

3. OUR DATA REUSE DETECTION AND CODE TRANSFORMATION APPROACH

3.1 Data Memory System Architecture

Our memory system consists of several components: cache, scratch-pad memory (e.g., on-chip RAM) and main memory (usually off-chip RAM). It is possible to have several RAM modules of different sizes and delays. In this case a hierarchy of scratch-pad buffers mapped to those modules can be used. Figure 2 shows the data memory architecture with two levels of scratch-pad memory.

In our approach we identify arrays that are most heavily used with compile-time-analyzable access patterns, and exclude them from servicing by data cache. Instead, all array elements that are reused are kept in scratch-pad memories and all others are fetched from main memory directly (bypassing cache and scratch-pad memories). In the rest of the Section 3 we consider only last two categories of accesses.

The algorithm for reuse analysis uses a description of loop structure and array references as an input. It performs analysis of data reuse pattern on each nested loop level. Data reuse is detected between different array references, as well as between different iterations of the outer loops for the same array reference. If data reuse is detected, a buffer size is determined to hold the reused data. Performing these operations at each nested loop level results in a hierarchical set of buffers. Each of them can be mapped to one of the levels of the scratch-pad memory, or not used at all. After decision is made, the input code is transformed to include the appropriate data transfers to copy data to scratch-pad memory. The problem of design space exploration (selecting which of the buffers should be used and mapping them to the scratch-pad memory hierarchy) is not in the scope of this article and has been addressed previously by Brockmeyer et al. [2003]; the focus of this article is on the automation of reuse detection required to generate the exploration space of the custom memory configurations.

In the next section we give the intuition which helps in understanding the formal description of our algorithm presented in Section 3.3.

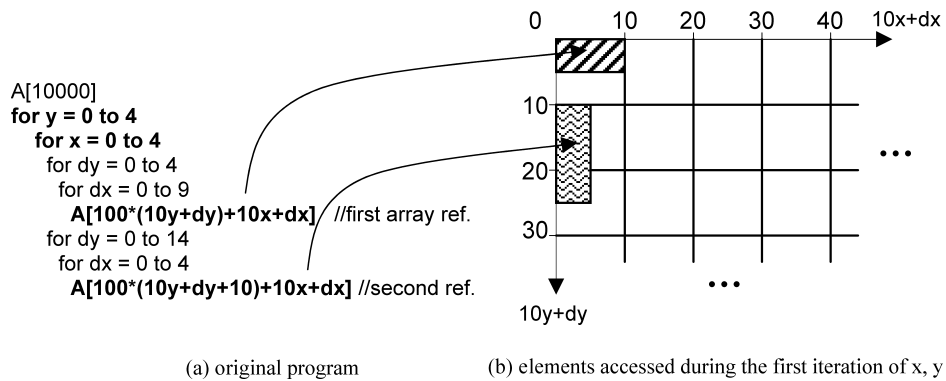


Fig. 3. Example code.

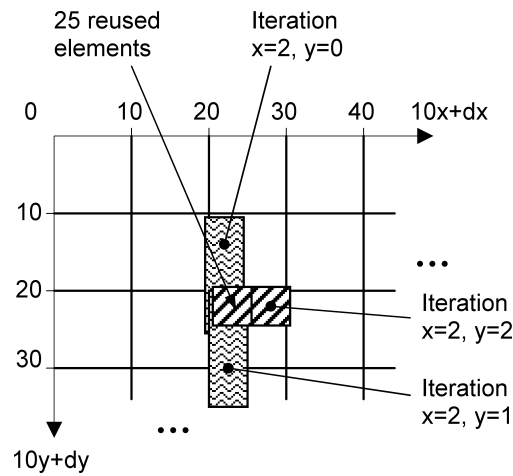


Fig. 4. Reused elements.

3.2 Basic Idea for Reuse Detection

We illustrate our basic idea for reuse detection with the program example presented in Figure 3(a). For this program the footprint of the addresses of accessed array elements when iterators y and x are set to value zero is shown in Figure 3(b). With every increment of the iterator x , the footprint shifts right by ten elements. If we increment y by one, the footprint shifts down by ten elements. If we continue iterating over x and y , we can notice that some of the elements are read more than once. For example, Figure 4 shows a set of 25 elements which is read three times during ten consecutive iterations of two outer loops. Moreover, other 25-element sets which are formed by shifting that set by integer numbers of steps behave the exactly same way: After they are read for the first time, in five iterations they are read again, and in another five iterations they are read for the last time.

All these reused datasets can be copied into the scratch-pad memory when they are accessed for the first time and can be read from there later. For this

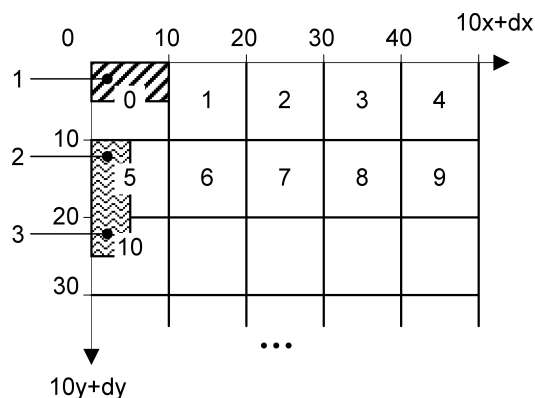


Fig. 5. Calculating the number of iterations between accesses.

particular example, we need to keep ten such sets of 25 elements each. The eleventh set can take the place of the first in the buffer, since the data is no longer reused after ten iterations. This requires a buffer size of $10 \times 25 = 250$ elements. If we don't have 250 elements available in the scratch pad, we can use a smaller buffer: We can reuse data that is read by the second array reference, while always reading data for the first array reference from the main memory. In this case, the number of iterations between the first and last accesses to the data reused is five (one iteration of y), so the buffer size needed is $5 \times 25 = 125$ elements.

We can easily compute the number of iterations between the first and last accesses to the elements that are reused, provided that with every iteration of x and y the footprint of the addresses accessed by the array shifts by a constant amount. For this, we need to take an access footprint when x and y are fixed to their initial value, as shown in Figure 5. The grid cell size in this picture represents the amount by which the footprint shifts with iterations over x and y (10×10 in our case). This picture shows that the “distance” between elements 2 and 3 is one iteration of iterator y or, since it happens at every five iterations of x , the distance is five iterations of iterator x . The distance between elements 1 and 3 is ten iterations. Taking this distance and multiplying it by the size of the reused set gives the size of buffer required.

The outline of the code that can be generated for the first configuration of the buffer (250 elements) is shown in Figure 6. Each access to the array is replaced by the code that accesses data from the buffer (if present), copies data from the memory to the buffer, or reads directly from main memory.

This simple scheme does not allow using DMA or burst transfer mode while accessing the main memory. As an alternative approach, the buffer can be initialized with data in the beginning and part of the circular buffer can be updated at every iteration of a loop. This allows the use of a DMA controller. The example of such generated code is shown later in Section 3.3.

Note that this approach for determining the distance (hence the required buffer size) and code generation is correct only for index expressions and loop boundaries that satisfy certain conditions (which are described in the next

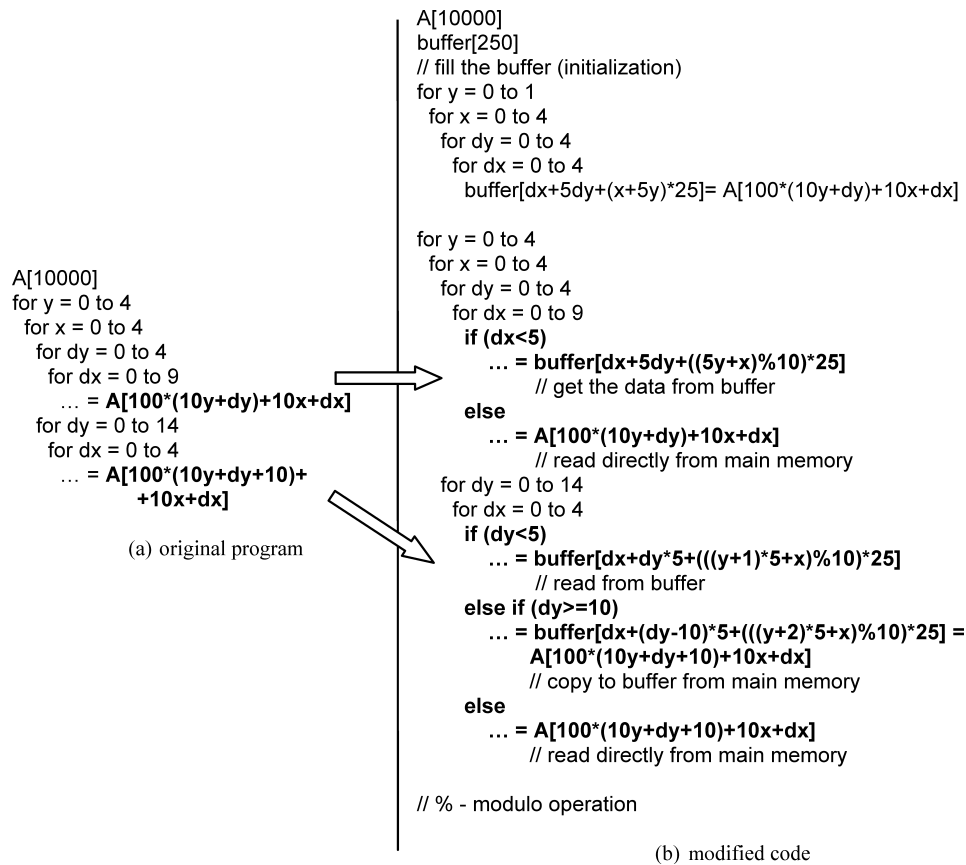


Fig. 6. Code generation example.

section). For example, if we change the upper loop bound for iterator x from 4 to 140, the reuse pattern becomes more complex and these simple rules for code generation do not work. This is the main reason for dividing the iterators into several groups (*fixed* and *moving* iterators), which is described in more detail in the next section.

3.3 DRDU: Data Reuse Analysis Algorithm and its Use for Memory Subsystem Optimization

In this section we describe our approach for building a custom scratch-pad-memory-based system using our data reuse analysis and give a formal description of all operations that have to be performed for the reuse analysis. Figure 7 outlines our overall approach.

First, in *Step 1* of Algorithm 1, we select a part of the program that may benefit from optimization, and extract its loop structure surrounding the array accesses in *Step 2*. These steps are performed manually by the designer. Please note that in order to apply our technique, the input program should conform to several requirements. First, all accesses should be expressed as indexed array

Algorithm 1: Our Approach for Memory Subsystem Customization.**Input:** program to be optimized**Output:** memory subsystem configuration and transformed program**Begin**

1. Select part of the program to be optimized
2. Extract loop and array reference information for those pieces of code
3. Perform data reuse analysis using DRDU algorithm
4. Perform design space exploration and select which of the buffers should be implemented
5. Transform the program to include selected buffers and simplify it

End

Fig. 7. Outline of the algorithm for detecting reuse and transforming the program to use a scratch-pad memory.

accesses. This is not a major requirement, as most applications exhibiting regular data reuse have statically allocated data, hence its addressing can always be expressed as an indexed function of the loop iterators. Second, this index expression must be an affine function of the outer loop iterators. For most audio and video processing and multimedia program, this is the case. However, if there are accesses to the array that are expressed using pointers, such a program can be converted to the required form by using the FORAY technique [Issenin 2005]. Regarding dealing with conditional statements, if an array reference is executed conditionally and the condition is also an affine function of outer loop iterators, such a case is handled by the proposed approach (see Step 1-2-2 later in this section). If the condition is more complex or data-dependent, it can be omitted together with the array reference, or the condition can be assumed always true. In both cases our approach can still be applied, but the results will be less accurate.

Next, in *Step 3* of Algorithm 1, we perform data reuse analysis (DRDU) for each array in the program which produces a list of all possible buffers that can be implemented at each loop level. After that, in *Step 4* of Algorithm 1, we perform design space exploration and select the buffers that should be implemented.

Finally, in *Step 5* of Algorithm 1, we modify the program to include the necessary transfers between selected buffers, main memory, and the processor. The example of generated code is shown later in this section. Possible optimizations that are not included in traditional compilers, but which can help to reduce the complexity of the code are discussed there too.

Figure 8 describes our *DRDU* algorithm that performs *data reuse detection* for uniprocessor systems.

In *Step 1-1* of the DRDU algorithm we build a *reuse tree* which resembles the loop structure of the code.

Figure 9(a) shows an example of input code. It is similar to the code from Section 3.2. Note that we use one-dimensional arrays only. This is a more general representation, since arrays with any number of dimensions can always be converted to this form (but not vice versa).

Figure 9(b) shows the corresponding reuse tree. Each reuse node in the reuse tree represents a possible position of a buffer that will be used to hold reused data. Such a reuse tree represents the hierarchical structure of the buffers.

In some cases, the reuse tree does not exactly follow the loop structure of the program. For example, in Figure 10, a program and corresponding reuse tree are

The Algorithm for Data Reuse Detection in a Uniprocessor System (DRDU).**Input:** loops and array reference information**Output:** a set of possible buffer configurations for storing reused data**Begin**

1. For each array do

1-1. Build reuse tree

1-2. For each reuse node (*current node*) of reuse tree do1-2-1. Classify iterators into three groups: *fixed*, *moving* and *filling* iterators1-2-2. Create *low cover* and *full cover* sets

1-2-3. Transform these sets to 2M-dimensional space

1-2-4. Divide transformed sets into cells and assign *distance numbers* to them1-2-5. Find *grid* and *basic pieces*

1-2-6. Determine the size of the buffers required for the current loop level

End

Fig. 8. The algorithm for data reuse analysis.

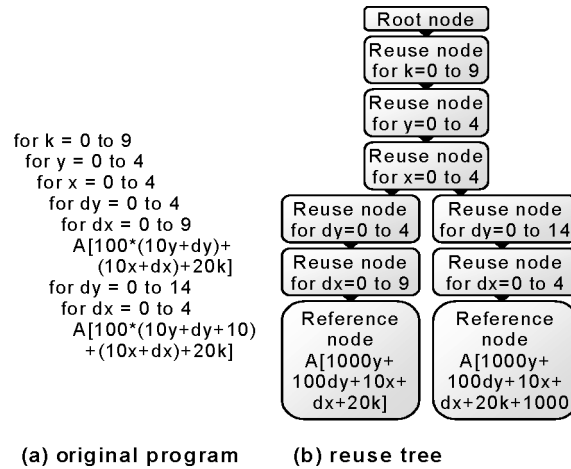


Fig. 9. Creating a reuse tree.

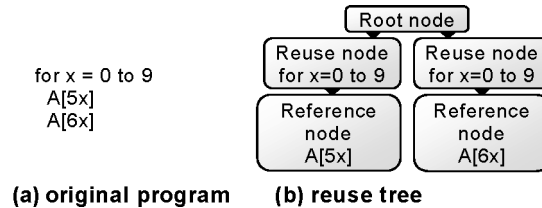


Fig. 10. An example of a reuse tree with split nodes.

presented. Since index expressions of the references have different coefficients affecting iterator x , there are few reuse opportunities between accesses of the two array references and it is not efficient to allocate the same reuse buffer for both of them. Therefore the reuse node corresponding to loop x must be duplicated.

In general, the property that should be enforced in a reuse tree is that for each reuse node N , all descendant reference nodes should have the same coefficients of the iterators declared in the node N and all its ancestors.

After obtaining the reuse tree we try to determine which data is reused at each loop level (in each reuse node). For each reuse node L (let's call it a *current node*) of the reuse tree, we perform the following operations.

In *Step 1-2-1* of the DRDU algorithm, we classify all loop iterators that are descendants or ancestors of L , as well as L itself, into three groups: *fixed iterators*, *moving iterators*, and *filling iterators*. We call filling iterators all loop iterators defined in reuse nodes that are descendants of L . All the iterators defined in nodes from *current iterator* to root node are split into two groups: moving iterators (from node L to some node K) and fixed iterators (from the ancestor of K to the root node). To determine node K , iterators should be added one-by-one to a set \mathbf{M} of moving iterators as long as the following property remains true.

$$\begin{aligned} & \forall (N, C) \in \mathbf{M}, \text{coef}(C) > \text{coef}(N): \\ & (\text{upper_bound}(N) - \text{lower_bound}(N) + 1) * \text{coef}(N) \leq \text{coef}(C) \quad (1) \\ & \wedge \text{coef}(C) \% \text{coef}(N) = 0 \end{aligned}$$

Here $\text{upper_bound}(N)$ and $\text{lower_bound}(N)$ are bounds of the loop of iterator N ; $\text{coef}(N)$ is the coefficient of iterator N in the index expression; and $\%$ is the modulo operation.

This classification for loop iterators is necessary in our approach. It allows easy determination of repeating addresses in the address footprint of accessed array elements between iterations of moving iterators when the fixed iterators do not change their values and the filling iterators are iterating over their complete loop bounds. This allows keeping the data reused efficiently during iteration over moving iterators.

For the code shown in Figure 9(a) and for the current iterator x , iterators dx and dy are filling iterators, x and y are moving iterators, and k is a fixed iterator.

The preceding algorithm described for selecting moving iterators always returns the maximum number of moving iterators possible for the current loop level. However, it is often useful to consider the buffer configurations obtained by assigning less than a maximum possible number of iterators to moving iterators. Usually, this results in smaller buffers with more accesses to the next level of memory. We illustrate this on the example depicted in Figure 11. On j -level (which means that j is the current iterator) iterators i , di and dj are filling iterators and j is a moving iterator. This defines the buffer at level j (if we perform all necessary steps to obtain the buffer) to be $5 \times 64 = 320$ elements with 4096 misses to the main memory. On i -level, iterators di and dj are filling iterators and i and j are moving iterators. In this case we obtain a smaller buffer of $1 \times 261 = 261$ elements with the same number of accesses to the main memory. The price for the reduced size is the frequency of updates: Now we have to update the buffer at every iteration of i loop with one element, instead of every iteration of j loop with a row of 64 elements, as in the previous case. However, if we consider reduced to one the number of *moving* iterators for i -level (j : fixed iterator; i : moving iterator; di and dj : filling iterators), we get a much smaller $5 \times 5 = 25$ element buffer with 19,200 accesses to main memory. Of course, it is also possible to combine buffers (b) and (d) (of Figure 11) to reduce the number of accesses to main memory to 4,096 while keeping the size of

<pre> a[6400] for j=0 to 59 for i=0 to 59 for dj=0 to 4 for di=0 to 4 val += f(a[100j+100dj+i+di]) </pre>	<pre> buff[5][64] for n=0 to 3 // buf init for m=0 to 63 buff[n][m] = a[100n+m] for j=0 to 59 { for m=0 to 63 // buf update buff[(j+4)%5][m] = a[100*(j+4)+m] for i=0 to 59 for dj=0 to 4 for di=0 to 4 val += f(buff[(j+dj)%5][i+di]) } </pre>	<pre> buff[261] for n=0 to 255 // buf init buff[n] = a[100*(n/64)+n%64]; for j=0 to 59 { for n=0 to 3 // buf update buff[(64*j+n+256)%261]= a[100*(j+4)+n] for i=0 to 59 { buff[(64*j+i+260)%261] = a[100*(j+4)+i+4]//update for dj=0 to 4 for di=0 to 4 val += f(buff[(64*(j+dj)+i+di)%261]) } } </pre>	<pre> buff[5][5] for j=0 to 59 { for n=0 to 4 // buffer init for m=0 to 3 buff[n][m] = a[100*(n+j)+m]; for i=0 to 59 { for n=0 to 4 // buf update buff[n][(i+4)%5]= a[100*(n+j)+i+4] for dj=0 to 4 for di=0 to 4 val += f(buff[dj][(i+di)%5]); } } </pre>
(a) original program # of memory accesses: 90,000	(b) transformed program moving iterator: j buffer size: 5x64 = 320 # of memory accesses: 4,096	(c) transformed program moving iterators: i, j buffer size: 1x261 # of memory accesses: 4,096	(d) transformed program moving iterator: i buffer size: 5x5 # of memory accesses: 19,200

Fig. 11. The effect of selection of moving iterators on buffer configuration.

25 elements of the first scratch-pad memory buffer if two levels of scratch-pad memories are available.

In the following steps we introduce the concept of a *cover set*. A cover set is a set of the one-dimensional values of array index expressions when the fixed iterators are set to their first value, filling iterators are iterating within the corresponding loop bounds, and the moving iterators are iterating within specified bounds. Index expressions are taken from reference nodes that are descendants of the current reuse node in the reuse tree.

In *Step 1-2-2* of the DRDU algorithm, both a *low cover set LC* and a *full cover set FC* are calculated. A low cover set is a cover set when moving iterators are set to their first value. A full cover set is a cover set when the moving iterators are iterating within the corresponding loop bounds.

We illustrate this using the code shown in Figure 9(a). Assuming that x is the current iterator, the low and full cover sets are

$$\begin{aligned}
 \mathbf{LC} = [V] : \{ \exists dx, dy : (0 \leq dy \leq 4 \wedge 0 \leq dx \leq 9 \wedge V = 100dy + dx) \\
 \vee (0 \leq dy \leq 14 \wedge 0 \leq dx \leq 4 \wedge V = 100dy + dx + 1000) \}
 \end{aligned} \quad (2)$$

$$\begin{aligned}
 \mathbf{FC} = [V] : \{ \exists x, y, dx, dy : (0 \leq y \leq 4 \wedge 0 \leq x \leq 4) \wedge \\
 ((0 \leq dy \leq 4 \wedge 0 \leq dx \leq 9 \wedge V = 1000y + 10x + 100dy + dx) \vee \\
 (0 \leq dy \leq 14 \wedge 0 \leq dx \leq 4 \wedge V = 1000y + 10x + 100dy + dx + 1000)) \}.
 \end{aligned} \quad (3)$$

Please note that for the cases when array reference “if” is conditionally executed and the condition is an affine function, that condition is also included into low and full cover set equations.

Next, in *Step 1-2-3* of the DRDU algorithm we transform the cover sets of one-dimensional points to the *transformed cover sets* of $2M$ -dimensional points, where M is the number of moving iterators.

To obtain transformed cover sets, we apply the following relation \mathbf{R} to the cover sets.

$$\begin{aligned} \mathbf{R} &= [V] \rightarrow [V_1..V_M, dV_1..dV_M]: \\ &\{0 \leq dV_1 \leq dV_{1max} \wedge \dots \wedge 0 \leq dV_M \leq dV_{Mmax} \wedge \\ &\quad 0 \leq V_1 \leq V_{1max} \wedge \dots \wedge 0 \leq V_M \leq V_{Mmax} \wedge \\ &\quad V = coef(V_1)/step_1 * (step_1 * V_1 + dV_1) + \dots + \\ &\quad coef(V_M)/step_M * (step_M * V_M + dV_M) + min(V)\} \end{aligned} \quad (4)$$

Here $(V_1..V_M, dV_1..dV_M)$ is a new $2M$ -dimensional element of the set, $min(V)$ the value of the minimal element in an input set of one-dimensional elements, and $coef(V_K)$ the coefficient of the K^{th} moving iterator in the index expression. Constants $dV_{1max}..dV_{Mmax}$, $V_{1max}..V_{Mmax}$, and $step_1..step_M$ should be selected so that any input element V could be represented by some output element $(V_1..V_M, dV_1..dV_M)$ only in one way.

This transformation of cover sets is necessary for determining the “distance” (difference in number of iterations between accesses to the same data; intuitive explanation of this was given in Section 3.2 and the formal definition is given later in this section). To calculate the distance, we need to know to which rectangular cell each of the array elements belongs (see Figure 5 in Section 3.2). We call these cells *grid cells*, which are defined as following. If we apply relation \mathbf{R} to one of the elements from the cover set, the first M dimensions of the result $(V_1..V_M)$ in Eq. (4) give us coordinates of the grid cell. This allows to calculate the distance. The rest of the M dimensions $(dV_1..dV_M)$ in Eq. (4) give the offset of the element within its cell. Constants $step_1..step_M$ determine the size of the M -dimensional cell.

More than one possible choice may exist for constants $dV_{1max}..dV_{Mmax}$, $V_{1max}..V_{Mmax}$, and $step_1..step_M$. They affect both the complexity of address calculations in the transformed code as well as the buffer sizes. One of the ways (the simplest) to set these constants is the following. Assuming that the moving variables are numbered here in order of their coefficients ($\forall i \in 1..M-1$: $coef(V_i) \leq coef(V_{i+1})$), the constants can be calculated as

$$\begin{aligned} step_1 &= coef(V_1) \\ \forall i \in 1..M-1 : \{step_{i+1} &= 1; V_{imax} = coef(V_{i+1})/coef(V_i)/step_{i+1} - 1\} \\ V_{Mmax} &= \infty \\ \forall i \in 1..M : dV_{imax} &= step_i - 1 \end{aligned} \quad (5)$$

Other variants of constant assignments are different from the one shown previously by selecting different $step_{i+1}$ for $i \geq 1$. For these assignments, $step_{i+1}$ can be set to a number for which the following conditions are met.

$$\begin{aligned} &step_{i+1} \text{ evenly divides } coef(V_{i+1})/coef(V_i) \text{ and} \\ &\mathbf{RFC}(V \cap V_i = V_{i,low}) \cap test_cov = \emptyset \end{aligned} \quad (6)$$

Here $test_cov = [V_1..V_M, dV_1..dV_M]: \{V_{imax} - (V_{i_hi} - V_{i_low}) + 1 \leq V_i \leq V_{imax}\}$; V_{i_hi} and V_{i_low} are the upper and lower loop bounds, respectively, of the loop for iterator V_i ; and

$\mathbf{FC}(V \cap V_i = V_{i_low})$ is the full cover set with moving iterator V_i set to its initial value.

Optimal step sizes for different moving variables can be selected independently. For evaluating each of them, the rest of the steps of the DRDU algorithm need to be performed to obtain *basic pieces* (see Step 1-2-5) and calculate buffer sizes (Step 1-2-6), assuming that step values for other moving variables are set to one, or any other correct value. Good metrics for comparing the results are total buffer size and complexity of address calculations. The latter corresponds to the number of equalities and inequalities in a polyhedral representation of the basic piece sets.

For our example (x and y are the moving variables, $coef(V_1) = coef(x) = 10$, $coef(V_2) = coef(y) = 1000$), $step_1 = coef(V_1) = 10$. We should select $step_2$ to be a divider of $coef(V_2)/coef(V_1) = 100$, for example, 10. In this case the relation is

$$\begin{aligned} \mathbf{R} = [V] \rightarrow [V_1, V_2, dV_1, dV_2] : \{ & 0 \leq dV_1 \leq 9 \wedge 0 \leq dV_2 \leq 9 \wedge \\ & 0 \leq V_2 \leq \infty \wedge 0 \leq V_1 \leq 9 \wedge \\ & V = 1000V_2 + 100dV_2 + 10V_1 + dV_1 \}. \end{aligned} \quad (7)$$

Now we should check if the second condition for the selection of $step_2$ is also satisfied.

$$\begin{aligned} test_cov = [x, y, dx, dy] : \{ & 9 - (4 - 0) + 1 \leq x \leq 9 \} = [x, y, dx, dy] : \{ 6 \leq x \leq 9 \} \\ \mathbf{FC}(V \cap x = 0) = [V] : \{ & \exists dx, dy, y : ((0 \leq y \leq 4 \wedge 0 \leq dy \leq 4 \wedge 0 \leq dx \leq 9 \wedge \\ & V = 100dy + dx + 1000y) \vee (0 \leq y \leq 4 \wedge 0 \leq dy \leq 14 \wedge 0 \leq dx \leq 4 \wedge \\ & V = 100dy + dx + 1000y + 1000)) \} \\ \mathbf{R}\mathbf{FC}(V \cap x = 0) = [x, y, dx, dy] : \{ & (0 \leq y \leq 4 \wedge x = 0 \wedge 0 \leq dy \leq 4 \wedge \\ & 0 \leq dx \leq 9) \vee (1 \leq y \leq 6 \wedge x = 0 \wedge 0 \leq dy \leq 9 \wedge 0 \leq dx \leq 4) \} \end{aligned} \quad (8)$$

Hence, $\mathbf{R}(\mathbf{FC}(V \cap V_i = V_{i_low})) \cap test_cov = \emptyset$, which shows that our selection of $step_2 = 10$ is acceptable.

To transform the low and full cover sets, we apply the relation \mathbf{R} to the sets. The transformed sets for our example are depicted in Figures 12(a) and (c).

Next, in *Step 1-2-4* of the DRDU algorithm, we divide the transformed full cover set into M -dimensional rectangular cells and define *distance numbers*. The way in which transformed full cover set is divided into cells is that in each cell, the values of $V_1..V_M$ are constant. All cells that intersect with the rectangular hull of a full cover set are numbered sequentially starting from zero. We call those numbers distance numbers (Figures 12(a) and (b)).

In *Step 1-2-5* of the DRDU algorithm we determine *grid* and *basic pieces*. They are sets of M -dimensional points. To find grid pieces, we intersect the transformed low cover set with the grid cells obtained earlier. In other words, for each value of $V_1..V_M$, we obtain a grid piece $\mathbf{G}(\mathbf{R}(low\ cover\ set))$, where

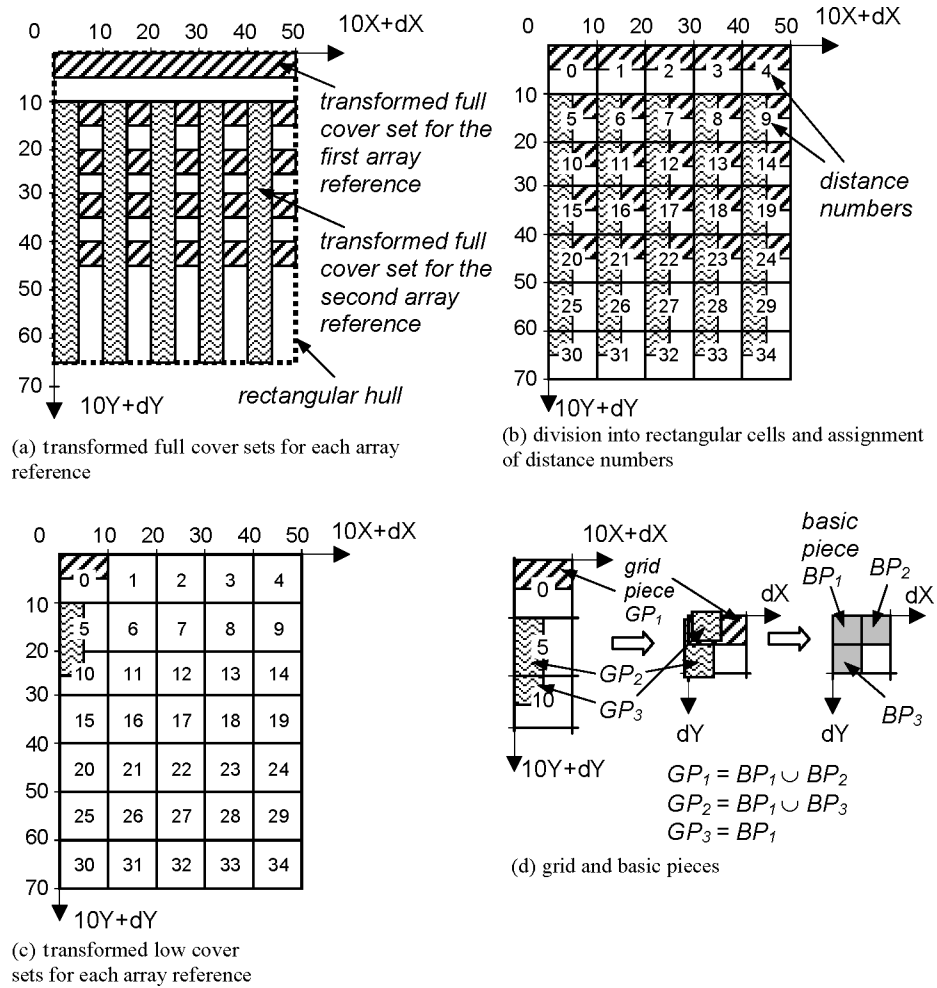


Fig. 12. Finding buffers configuration.

transformation G is

$$\begin{aligned}
 G &= [V_1..V_M, dV_1..dV_M] \rightarrow [ddV_1..ddV_M]: \\
 &\{ddV_1 = dV_1 \wedge \dots \wedge ddV_M = dV_M \wedge \\
 &\quad V_1..V_M \text{ are set to given values}\}.
 \end{aligned}$$

Grid pieces may overlap with each other. However, it is possible to represent every grid piece as a union of nonoverlapping basic pieces. The set of basic pieces is common and different combinations of its elements are used to represent all grid pieces. Figure 12(d) shows the grid and basic pieces for the example discussed earlier.

Finally, in *Step 1-2-6* of the DRDU algorithm we determine the size of buffers required for the current level of the memory hierarchy. This is done by analyzing the basic pieces. Each basic piece represents an opportunity for a buffer. If a

basic piece belongs to only one grid piece, there is no data reuse and no buffer is necessary. If a basic piece is a part of two or more different grid pieces, a buffer can be introduced. The size of the buffer is computed by multiplying the basic piece size (number of elements in the basic piece) by the maximum difference in distance numbers of the grid pieces that contain the basic piece.

To further illustrate how the buffer size and configuration are determined, consider a fragment of our example as shown in Figure 12(d), where only the basic piece BP_1 is reused. The size of the BP_1 is 25 (5×5). The grid pieces containing BP_1 are GP_1 , GP_2 , and GP_3 with distance numbers 0, 5, and 10, respectively. The maximum difference in distance numbers is 10. Hence, the size of the buffer needed to hold all reused data is $25 \times 10 = 250$.

As mentioned in Section 3.2, sometimes it is beneficial to have a smaller buffer and to select some of the other distances instead of maximum distance. In our case, we can also select the distance difference to be 5 with a buffer size of 125. This buffer can hold data that is reused between grid pieces GP_1 and GP_2 (or GP_2 and GP_3), which increases the number of accesses to main memory but decreases the cost of scratch-pad memory.

After determining possible buffer configurations and deciding which buffers to implement (*Step 4* of Algorithm 1), the original code has to be modified to include necessary transfers to the selected buffers (*Step 5* of Algorithm 1). Next, we describe a scheme that transfers blocks of data to the scratch-pad buffer (unlike the example shown in Figure 6) and thus can use DMA to speed-up memory accesses.

The modified code for the example discussed before is shown in Figure 13. The original loop structure is highlighted in bold. Array A represents the original data residing in the main memory, and array buf is the buffer that is allocated to the scratch-pad memory.

The program in Figure 13 consists of several highlighted parts. The first part is the declaration of the scratch-pad buffer buf . Note that the buffer has the size of 275 elements, which means that it can hold 11 basic pieces BP_1 . One additional piece is needed because of the order in which the buffer is updated: Update precedes reading data from the buffer.

The second highlighted part performs initialization of the scratch-pad buffer. It is placed before that loop with the outermost moving iterator. The *cell* loop iterates over all copies of the basic piece BP_1 that are stored in the buffer (except the last, which is updated in part 4 of the code). The innermost loops *tmp_dy* and *tmp_dx* allow copying of one basic piece from the main memory. The loop bounds of these loops are set to the minimum and maximum values of the points in each dimension in the basic piece. This way, the smallest M -dimensional rectangular hull of the basic piece is prefetched. While this may require more scratch-pad memory when the basic pieces are not of rectangular shape, it simplifies the scratch-pad memory addressing and thus reduces additional computational overhead of address calculations.

The next two parts perform the scratch-pad buffer update after each iteration of the loops with moving iterators. While in the general case both of them may be needed, in our example the first update (*Update 1*) is not needed and never executed. The update in the innermost moving iterator loop (*Update 2* in our

<pre>int buf[275];</pre>	Scratch pad buffer declaration	1
<pre>for (k=0; k <= 9; k++) { for (cell=0; cell < 10; cell++) for (tmp_d_y=0; tmp_d_y <= 4; tmp_d_y++) for (tmp_d_x=0; tmp_d_x <= 4; tmp_d_x++) buf[25*cell + 1*(tmp_d_x-0) + 5*(tmp_d_y-0)] = A[0+20*(k-0)+100*tmp_d_y+1*tmp_d_x+0+(cell%5)/1*10+cell/5*1000]; }</pre>	Initialization (prefetching) of the scratch pad buffer	Part 2
<pre>for (y=0; y <= 4; y++) { for (cell=((0*7+y-0)*5)+11-1; cell <= ((0*7+y-0)*5)+11-2; cell++) for (tmp_d_y=0; tmp_d_y <= 4; tmp_d_y++) for (tmp_d_x=0; tmp_d_x <= 4; tmp_d_x++) { int cell_md = cell % 11; buf[25*cell_md + 1*(tmp_d_x-0) + 5*(tmp_d_y-0)] = A[0+20*(k-0)+100*tmp_d_y+1*tmp_d_x+0+(cell % 5)/1*10+cell/5*1000]; } }</pre>	Update 1 of the scratch pad buffer	Part 3
<pre>for (x=0; x <= 4; x++) { for (cell=((0*7+y-0)*5+x-0)+11-1; cell <= ((0*7+y-0)*5+x-0)+11-1; cell++) for (tmp_d_y=0; tmp_d_y <= 4; tmp_d_y++) for (tmp_d_x=0; tmp_d_x <= 4; tmp_d_x++) { int cell_md = cell % 11; buf[25*cell_md + 1*(tmp_d_x-0) + 5*(tmp_d_y-0)] = A[0+20*(k-0)+100*tmp_d_y+1*tmp_d_x+0+(cell % 5)/1*10+cell/5*1000]; } }</pre>	Update 2 of the scratch pad buffer	Part 4
<pre>{ int _val_index = 0+20*0+1000*y+10*x+100*dy+1*dx - 0; int _val_y = (_val_index/1000); int _d_y = (_val_index % 1000)/100; int _val_x = (_val_index/10) % 10; // x=(index/coef(x)) % x_{MAX} int _d_x = (_val_index % 10)/1; // dx=(index % coef(x))/(coef(x)/step(x)) int _distance = 0; _distance = _distance*7+_val_y; _distance = _distance*5+_val_x; }</pre>	Finding x , dx , y , dy and the distance number of the current access address	Part 5
<pre>if ((_d_y >= 0 && _d_y <= 4) && (_d_x >= 0 && _d_x <= 4)) _value_ = buf[(_distance) % 11] * 25 + 1*(_d_x-0) + 5*(_d_y-0); else _value_ = A[20*k+1000*y+10*x+100*dy+1*dx]; }</pre>	Accessing the data	Part 6
<pre>for (dy=0; dy <= 14; dy++) for (dx=0; dx <= 4; dx++) { int _val_index = 1000+20*0+1000*y+10*x+100*dy+1*dx - 0; int _val_y = (_val_index/1000); int _d_y = (_val_index % 1000)/100; int _val_x = (_val_index/10) % 10; int _d_x = (_val_index % 10)/1; int _distance = 0; _distance = _distance*7+_val_y; _distance = _distance*5+_val_x; }</pre>	Finding x , dx , y , dy and the distance number of the current access address	Part 7
<pre>if ((_d_y >= 0 && _d_y <= 4) && (_d_x >= 0 && _d_x <= 4)) _value_ = buf[(_distance) % 11] * 25 + 1*(_d_x-0) + 5*(_d_y-0); else _value_ = A[1000+20*k+1000*y+10*x+100*dy+1*dx]; }</pre>	Accessing the data	Part 8
<pre>}}}</pre>		

Fig. 13. Output code before optimizations.

<pre>for (cell=0; cell < 10; cell++) A[(2*cell) % 5]*10;</pre> <p>(a) original code</p>	<pre>tmp = 0; for (cell=0; cell < 10; cell++) { A[tmp*10]; tmp = tmp > 2 ? tmp - 3 : tmp + 2; }</pre> <p>(b) optimized code</p>
--	---

Fig. 15. Modulo operation optimization.

Furthermore, the overhead can be reduced by control flow transformations such as code hoisting and removing conditional statements from innermost loops by loop splitting [Falk et al. 2003], and by replacing computationally expensive modulo and integer division operations by less expensive simple arithmetic operations with predication [Ghez et al. 2000] for the addressing computation. The simple case of the latter optimization is illustrated in Figure 15, where the modulo operation of the affine function of the outer loop iterator is replaced by addition and subtraction with predication.

Note that the described output code can be modified to perform buffer updates one iteration ahead-of-time (in this case the size of the buffer should also be increased to hold one more copy of the basic piece). With the use of DMA hardware to perform the updates, it is possible to overlap these memory accesses with the execution of the previous loop iteration. This way, the memory access latency can be completely hidden. In this case it is necessary to have buffers for all basic pieces, including those that have no reuse.

3.4 Complexity Analysis of the DRDU Algorithm

The tool we have built uses a polyhedral model for representing sets used in the algorithm. All operations on sets are described with Presburger formulas, which have superexponential worst-case complexity in terms of the length of the formula [Fischer et al. 1974]. Later in this section we analyze the complexity of our algorithm in terms of the number of polyhedral operations required for executing it. All other operations that do not use polyhedral sets are less computationally expensive and not taken into account.

The maximum number of iterations of Steps 1-2-1 to 1-2-6 of the DRDU algorithm (due to loops in Step 1 and Step 1-2) is $R * N$, where R is the number of array references in the program and N is the maximum loop nest level.

Step 1-2-1 is implemented without using the polyhedral library.

Steps 1-2-2 and 1-2-4 require a constant number of polyhedral operations.

In Step 1-2-3 we need a constant number of polyhedral operations to transform sets for given values of $step_i$ (see Eq. (6)). However, there may be many possible values of $step_i$ for each i to consider if $coef(V_{i+1}) / coef(V_i)$ (from Eq. (6)) has many dividers. If more than one value of $step_i$ has to be evaluated, Steps 1-2-4 to 1-2-6 have to be performed for each value of $step_i$ as well. To limit the complexity of the algorithm, the number of evaluations of different values of the latter step can be limited by some constant L .

To calculate grid pieces in Step 1-2-5, we need to call the polyhedral library as many times as there are nonempty grid pieces. It is possible to find the upper

bound on this number in at most N polyhedral library calls. We need to make one library call to find each grid piece. Again, if this number is too large, it can be limited by some constant K . All grid pieces that were not calculated can be assumed to be equal to the union of all (even those that were not calculated) grid pieces. It is possible to calculate that union in a constant number of polyhedral operations. In many cases this will lead to the exact results. Even in the cases when it will not, the wrong prediction will lead to increased buffer size with the same number of misses to the higher level of memory, but the code generated will still be functionally correct.

The maximum number of basic pieces which are possible to get from G different grid pieces is $2^G - 1$. We need a maximum of $2G$ polyhedral operations for the calculation of one basic piece. To determine which grid pieces are different, we need less than $K * K$ operations. Similar to the case with grid pieces, we can limit the maximum number of different grid pieces by some constant B by taking the union of all other different grid pieces and representing them as one grid piece.

To calculate the size of a basic piece in Step 1-2-6, we have used the PolyLib library [Wilde 1993]. The author does not specify the complexity of performing this operation. Alternatively, we can use the maximum of $N + 1$ polyhedral operations to find the area of the smallest rectangular hull which contains the basic piece.

After adding all the aforementioned calculated costs, we can estimate that the upper bound on the number of polyhedral operations is in the order of $R * N * L * (K * K + N * B * 2^B)$, where R is the number of array references in the program, N the maximum loop nest level, and L , K , and B are some selected constants that limit the running time of the algorithm.

In our experience it was not necessary to limit the algorithm execution time for the real benchmarks. The running time of the tool was in the order of seconds for the benchmark kernels we used without setting any limits with constants L , K , or B .

4. EXPERIMENTAL RESULTS

One of the goals of our experiments is to compare our customized scratch-pad memory against a cache-based memory subsystem for their relative abilities to exploit temporal locality of the data accesses in a number of multimedia and streaming applications. We also study the overheads incurred in using a customized scratch-pad memory.

We have created a tool that implements the described technique. It uses a polyhedral model for representing sets used in the algorithm. We have used the Omega [Kelly et al. 1995] and PolyLib [Wilde 1993] polyhedral libraries for performing operations on these sets.

A version of the SimpleScalar simulator [Burge and Austin 1997] configured to use the ARM instruction set has been used for obtaining the number of misses for cache and getting performance estimates for XScale architecture. We have used the CACTI model [Shivakumar and Jouppi 2001] for energy and performance estimation of the cache at 130 nm technology. The estimations

Table I. Experiment Results: Memory Subsystem

Benchmark and size of the buffer when using scratch-pad memory	The number of accesses to the main memory		Scratch-pad memory/cache			Main data memory	
	Reduction in comparison with a system without the cache	Reduction in comparison with a system with the cache of the same size	Size	Energy per access, nJ		Size	Energy per access, nJ
				Memory	Cache		
H.263 with a buffer 36K	80%	-9%	64K	0.27	0.52	256K	0.66
H.263 with a buffer 36K+1.6K	83%	9%	64K	0.27	0.52	256K	0.66
H.263 with a buffer 6K	58%	54%	8K	0.11	0.2	256K	0.66
QSDPCM with a buffer 1K	89%	13%	1K	0.044	0.12	128K	0.39
Laplace with a buffer 4K	89%	2%	4K	0.071	0.22	1M	2.1
Laplace with a buffer 2.5K	89%	2%	4K	0.071	0.22	1M	2.1
Susan with a buffer 1K	92%	22%	1K	0.044	0.12	128K	0.39

Benchmark and size of the buffer when using scratch-pad memory	Benchmark data memory footprint	Memory subsystem energy reduction in comparison with a system with a cache of the same size	Reduction in time spent in memory subsystem	Ratio scratch-pad memory (or cache) latency/main memory latency
H.263 with a buffer 36K	146K	37%	-3%	1:2
H.263 with a buffer 36K+1.6K	146K	40%	2%	1:2
H.263 with a buffer 6K	146K	51%	39%	1:3
QSDPCM with a buffer 1K	99K	49%	4%	1:3
Laplace with a buffer 4K	890K	34%	1%	1:6
Laplace with a buffer 2.5K	890K	34%	1%	1:6
Susan with a buffer 1K	30K	53%	5%	1:3

for the scratch-pad memory have been obtained by isolating the data array subsystem out of the CACTI cache model. Main memory has been also modeled using CACTI, since the memory footprint of all our applications allow these to be implemented in SRAM. The values we have obtained are included in Table I.

For the benchmarks, we have used kernels extracted from an H.263 video encoder [Lillevold et al. 1995], QSDPCM encoder [Stobach 1988], the Laplace algorithm performing edge enhancements in images, and the image recognition application Susan [Guthaus et al. 2001]. By running our tool we have obtained reuse trees with all possible buffer configurations. For each of the benchmarks, we have selected several buffer configurations. We have not performed full design space exploration to determine Pareto optimal configurations (since this is outside of the scope of this article), but instead have selected configurations that consist of one buffer (with the exception of line 2 in Table I) to limit transformed code complexity. The tool has provided us with the code versions that implement necessary data transfers between scratch-pad buffers and the main memory for the selected buffers. The resulting code was optimized afterwards.

The different buffers obtained for our benchmarks are shown in Figure 16. For each benchmark several buffers are possible. They differ by the iterators that were considered as moving iterators. Buffers can be combined if the loop levels of the buffers (which mean nesting levels of loops with moving iterators) do not overlap. Loop levels of each of the buffers are marked inside the boxes in Figure 16; level 1 is the outermost loop. The buffers are vertically positioned

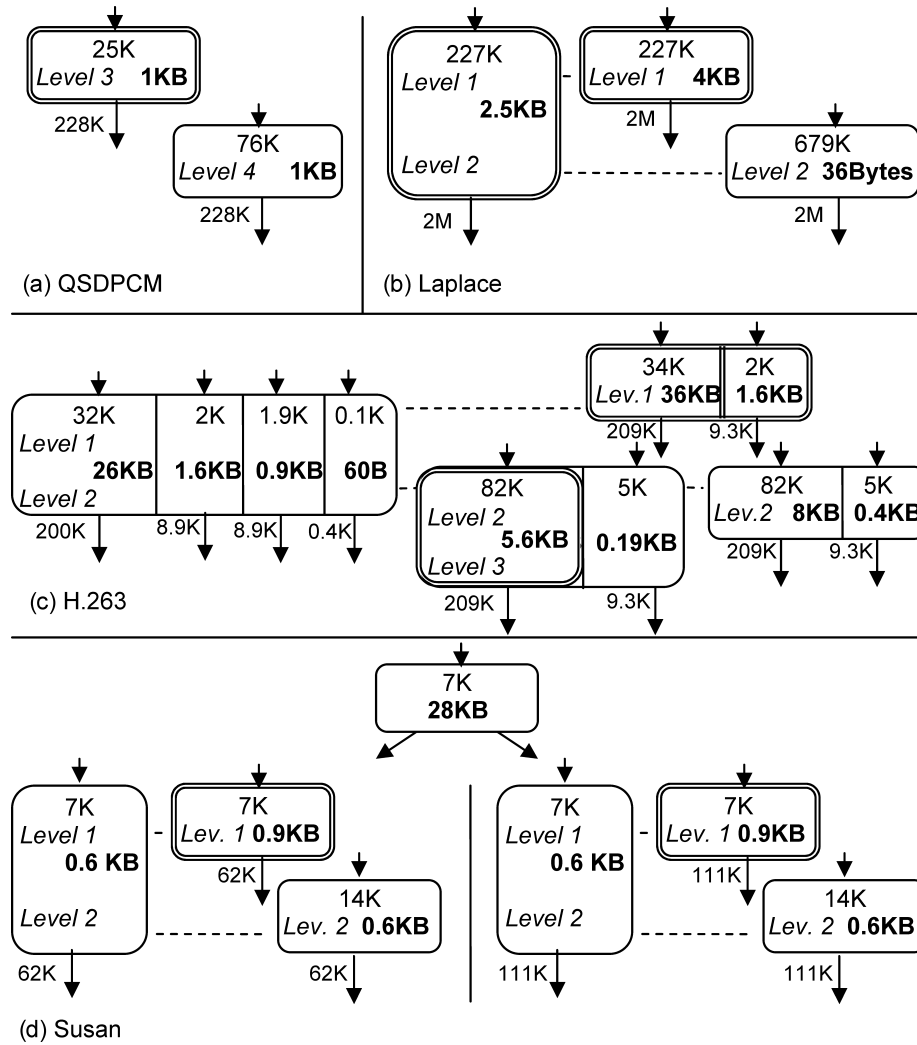


Fig. 16. Different possible buffers generated for the benchmarks.

in the picture according to their loop levels. The number of accesses (misses) to the next buffer in the hierarchy (or to the main memory), along with the size of the buffer, is also shown inside the buffers. Below the buffer is shown the number of accesses to it from the processor, provided that the buffer is the only one in the memory subsystem. If (for particular buffer configuration) two buffers are combined (e.g., for Laplace benchmark the buffers 4KB and 36 bytes), the number of accesses to the outer buffer (4KB) is equivalent to the number of misses of the inner buffer (679K accesses). This may not be true in, general case, but it is for our examples. The buffers in Figure 16 that have more than one size specified for them consist of several basic pieces (in the H.263 benchmark). In this case, all sizes, misses, and access counts are shown separately for each basic piece. The last benchmark (Susan) has two sequential

loop nests and the outermost 7K buffer holds elements that are used in both. In this benchmark the same scratch-pad memory space can be reused for buffers from both loop nests. The buffers that were used for the experiments described in the next sections are highlighted by double-line borders.

4.1 Effect on the Memory Subsystem

In this section we examine the effect when using a scratch-pad memory on the reduction of traffic to main memory, as well as on the energy and time spent in the memory subsystem. The comparison of the scratch-pad-based memory architecture has been made against a system having a data cache of the same size. The goal is to evaluate the energy and performance efficiency of our software-steered data replacement approach compared to that implemented by a hardware cache controller. The sizes of the scratch-pad memory and cache have been selected to be the closest values that are powers of two, while being greater than or equal to the buffer size required in the scratch-pad configuration. The associativity of the caches was selected to produce the lowest total energy consumption of the cache-based memory subsystem so that the fairest comparison could be made. Otherwise, the energy penalty per hit access of a cache memory having a high associativity will be far too large compared to that of the scratch pad, making the cache prohibitive from an energy viewpoint. We found that for all benchmarks except Laplace, the most energy-efficient was a direct mapped cache, while for Laplace we have used a twoway associative cache. The cache line size has been selected to be the minimal allowed by the simulator [Burge and Austin 1997] (8 bytes, which is two data elements in all benchmarks). In this way, we solely compare how well the temporal locality of the data is exploited, without considering spatial locality issues. These issues are largely orthogonal and can be optimized by data layout transformations [Kulkarni et al. 2001] which fall outside the scope of this article.

The energy savings when using a scratch pad in comparison with a cache come from two sources.

First, a scratch-pad memory consumes less energy than a cache of the same size per single access (about two times less for direct mapped cache [Shivakumar and Jouppi 2001]).

In addition, the more optimal data replacement decisions for scratch-pad memory (when compared to those made according to the replacement policy of the cache controller) result in less accesses to the main memory for all cases except the first (see Table I). This happens because the first configuration uses a data reuse buffer that holds only one basic piece out of the two required for the full data reuse, and the cache is big enough to have a relatively small number of conflicts. If we add the second basic piece (line 2 in Table I), our approach outperforms the cache in the number of main memory accesses. A reduced number of accesses to the main memory results in improved performance as well.

As we can see from Table I, the scratch-pad-based memory subsystem consumes 35% to 55% less energy than the system with a cache of the same size. The time spent in memory accesses is also reduced in all cases where the number of

accesses to main memory is reduced (see Table I). The figures presented there were calculated using the ratio of latencies of the scratch pad/cache and main memory from the approximation of the delays reported by CACTI, assuming that the on-chip memory is used as the main memory.

We have also compared the effectiveness of our technique with the approaches proposed by Panda et al. [1997] and Kandemir et al. [2002] for the QSDPCM benchmark. The Panda et al. [1997] approach requires a 100 times bigger scratch-pad memory, but eliminates all accesses to main memory. However, it consumes 4.5 times more energy in the memory subsystem than the energy obtained with our approach. By using the Kandemir et al. [2002] approach (without loop transformations, as we do in our approaches) and implementing a buffer at the same level as we have done for our reported results, it requires a 10 times bigger buffer. The amount of accesses to the main memory is also about 10 times greater than in our approach due to the loading of unused data. This even exceeds the number of accesses in the original program, giving negative energy savings in comparison with the original program.

4.2 Evaluation of the Address and Control Code Overhead

Adding buffers for keeping frequently used data implies the addition of code that both determines whether the current fetch should be performed from the buffer or main memory and calculates the position of requested data in the buffer. This adds time and energy overhead. In this section we estimate the amount of additional cycles needed to perform the described calculations and increase in the code size.

To estimate the time overhead, the original and transformed programs have been run on a number of workstations and simulated on the SimpleScalar simulator, configured for XScale architecture. In order to measure only the overhead caused by additional calculations and not the data memory latencies, we modified the program to remove all (or most) scratch-pad memory accesses while keeping the program structure and address calculations intact. Explicit transfers between the main memory and scratch-pad buffers were also removed (except the address calculations).

The increase in number of cycles ranges from 1% to 655% depending on the benchmark, buffer configuration, and platform (see Table II). However, when including the main memory latency effect, these numbers range between -24% and 400% for XScale architecture. For this experiment we assume the use of off-chip DRAM main memory with 70-cycle latency per cache line (and the same throughput for scratch-pad to main memory accesses). This number approximately corresponds to the cache miss latency (70–80 cycles) measured on the ADI Engineering 80200EVB evaluation board with an Intel 80200 processor with XScale architecture. The scratch-pad memory and cache hit latencies had been set to one cycle. The experiments show that for half of the benchmarks, the performance is better with scratch-pad memory than with the cache.

For comparison, the last rows in Table II show the overhead of the transformed programs when no address optimizations on the generated code before compilation are performed. An order of magnitude higher overhead shows the

Table II. Experiment Results: Processor Overhead

Benchmark and size of the buffer when using scratch-pad memory	Increase in the execution time (without the effect of main memory latency)				XScale: Increase in the execution time considering main memory latency	Code size increase
	Intel Pentium 4	Sun Ultra-SparcIII	HP PA-8500	XScale		
H.263 with a buffer 36K	1.44	1.22	1.02	7.55	4.97	3.2
H.263 with a buffer 6K	1.60	1.12	4.07	5.12	1.35	2.4
QSDPCM with a buffer 1K	1.23	2.33	1.76	2.76	2.43	1.78
Laplace with a buffer 4K	2.40	2.62	2.40	2.52	0.99	1.85
Laplace with a buffer 2.5K	1.05	1.25	2.18	1.73	0.76	1.68
Susan with a buffer 1K	1.22	1.10	1.06	1.01	0.91	1.52
Overhead of the code without optimizations						
H.263 buf. 36K not optimized	17	17	26	38		
H.263 buf. 6K not optimized	5.9	6.6	18	16		
QSDPCM buf. 1K not optimized	14	15	6.3	8.5		
Laplace buf. 2.5K not optimized	3.7	16	5.8	24		

insufficiency of the traditional optimizations used in modern compilers for this kind of code. Some of the buffer configurations use simple addressing in the innermost loops. No optimizations have been applied to them and they are not included in these rows.

The code size overhead (expressed as the ratio in number of assembly lines) has been measured using the gcc compiler for the Sun workstation. The increase in code size of the benchmark kernels we have used is about a factor of two. However, when compared to the size of the whole application, the overhead in code size is much smaller and not likely to cause significant increase in the instruction cache miss rate nor consequently increase the energy consumption in the instruction memory hierarchy.

Using scratch-pad memory incurs overheads which depend on the nature of the benchmark. This shows the necessity of a design space exploration which permits selection of the desired tradeoff between execution time and power savings.

4.3 Complexity of the DRDU Algorithm

Despite of the high worst-case complexity estimated in Section 3.4, the running time of the tool that is shown in the last column of Table III is in the order of seconds for all benchmarks analyzed. The times (and number of suggested buffers) are shown for the cases when the number of moving variables is set to the maximum possible value (Figure 16 also lists the buffers obtained by limiting the number of moving variables to one). The running time does not grow as fast as predicted by worst-case estimation with the increase of the

Table III. Running Time of the Tool

Benchmark	Number of array references in the benchmark kernel	Maximum loop nest level	Total number of buffers suggested	Running time, s
H.263	1	4	6	5
QSDPCM	1	8	2	16
Laplace	9	2	2	4
Susan	26	2	5	5

number of array references or loop nest levels, which are shown in the same table. We have run our analysis tool on a Pentium 4 2.4 GHz workstation.

The total number of buffers suggested by the program varies depending on the benchmark. However, we have found that not all of them are equally useful. Some of the buffers have the same size and amount of traffic reduction to the main memory, but different configurations and thus different processor overheads. Some of the buffers suggested are small and only used on the small portion of the processed image, but nonetheless incur processor overhead. That is why we have selected only those few most promising buffer configurations for the experiments described earlier.

5. CONCLUSION

In this article we present a compiler technique for data reuse analysis of data-transfer-dominated programs. The results of our analysis are vital for the design of a software-controlled memory hierarchy implemented as a customized scratch-pad memory organized as hierarchical set of buffers. We have demonstrated, on average, a 43% reduction in memory subsystem energy and also memory performance improvement when applying our approach in comparison with a cache-based implementation, and we outperform previously reported scratch-pad approaches.

Future work will address the problem of reducing the control code overhead, which can be done by clustering some of the data reuse basic pieces together. This may significantly reduce the complexity of conditional statements at a price of slightly increased buffer sizes in the scratch-pad memory.

REFERENCES

- AVISSAR, O., BARUA, R., AND STEWART, D. 2002. An optimal memory allocation scheme for scratch-pad based embedded systems. *IEEE Trans. Embedded Comput. Syst.* 1, 1, 6–26.
- BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Computing Surv.* 26, 4.
- BANAKAR, R., STEINKE, S., LEE, B. S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign (CODES)*.
- BROCKMEYER, E., MIRANDA, M., CATTHOOR, F., AND CORPORAAL, H. 2003. Layer assignment techniques for low energy in multi-layered memory organisations. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE)*, Germany.
- BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. Tech. Rep. 1342, Department of Computer Science, University of Wisconsin-Madison, WI

- COOPER, K., AND HARVEY, T. 1998. Compiler-Controlled memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA). 2–11.
- DIGUET, J., WUYTACK, S., CATTLOOR, F., AND DE MAN, H. 1997. Formalized methodology for data reuse exploration in hierarchical memory mappings. In *Proceedings of the IEEE International Symposium on Low Power Design* (Monterey, CA). 30–35.
- FALK, H. AND MARWEDEL, P. 2003. Control flow driven splitting of loop nests at the source code level. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE)* (Munich, Germany).
- FISCHER, M. J. AND RABIN, M. O. 1974. Super-Exponential complexity of Presburger arithmetic. In *Proceedings of the American Mathematical Society Symposium on the Complexity of Computational Processes*. 27–41.
- GHEZ, C., MIRANDA, M., VANDECAPPELLE, A., CATTLOOR, F., AND VERKEST, D. 2000. Systematic high-level address code transformations for piece-wise linear indexing: Illustration on a medical imaging algorithm. In *Workshop on Signal Processing Systems* (Lafayette, LA).
- GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, I., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *4th Annual Workshop on Workload Characterization*.
- ISSEIN, I., BROCKMEYER, E., MIRANDA, M., AND DUTT, N. 2004. Data reuse analysis technique for software-controlled memory hierarchies. In *Proceedings of the Design, Automation and Test in Europe Conference*.
- ISSEIN, I. AND DUTT, N. 2005. FORAY-GEN: Automatic generation of affine functions for memory optimizations. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE)* (Munich, Germany).
- KANDEMIR, M. AND CHOUDHARY, A. 2002. Compiler-Directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th Design Automation Conference* (New Orleans, LA).
- KANDEMIR, M., RAMANUJAM, J., IRWIN, M. J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Design Automation Conference* (Las Vegas, NV).
- KELLY, W., MASLOV, V., PUGH, W., ROSSER, E., SHIPEISMAN, T., AND WONNACOTT, D. 1995. The Omega library interface guide. Tech. Rep. CS-TR-3445, Department of Computer Science, University of Maryland, College Park, MD.
- KULKARNI, C., MIRANDA, M., GHEZ, C., CATTLOOR, F., AND DE MAN, H. 2001. Cache conscious data layout organization for embedded multimedia applications. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)* (Germany).
- LILLEVOLD, K. ET AL. 1995. Telenor R&D, H.263 test model simulation software.
- MCKINLEY, K., CARR, S., AND TSENG, C.-W. 1996. Improving data locality with loop transformations. In *ACM Trans. Program. Lang. Syst.* 18, 4.
- OZTURK, O., KANDEMIR, M., DEMIKIRAN, I., CHEN, G., AND IRWIN, M. J. 2004. Data compression for improving SPM behavior. In *Proceedings of the Design Automation Conference* (San Diego, CA).
- PANDA, P., DUTT, N., AND NICOLAU, A. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)* (Paris).
- SHIVAKUMAR, P. AND JOUPPI, N. 2001. CACTI 3.0: An integrated cache timing, power, and area model. WRL Tech. Rep. 2001/2.
- SJÖDIN, J. AND VON PLATEN, C. 2001. Storage allocation for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Atlanta, GA). 15–23.
- STEINKE, S., WEHMEYER, L., LEE, B., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)* (Paris).
- STOBACH, P. 1988. A new technique in scene adaptive coding. In *Proceedings of EUSIPCO* (Grenoble, France).
- UDAYAKUMARAN, S. AND BARUA, R. 2003. Compiler-Decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (San Jose, CA).

- VAN ACHTEREN, T., CATHOOR, F., LAUWEREINS, R., AND DECONINCK, G. 2003. Search space definition and exploration for nonuniform data reuse opportunities in data-dominant applications. *ACM Trans. Desi. Autom. Electron. Syst.* 8, 1.
- VAN ACHTEREN, T., CATHOOR, F., LAUWEREINS, R., AND DECONINCK, G. 2002. Data reuse exploration techniques for loop-dominated applications. In *IEEE/ACM Design Automation and Test Conference (Paris)*.
- VERMA, M., STEINKE, S., AND MARWEDEL, P. 2003. Data partitioning for maximal scratchpad usage. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)* (KitaKyushu, Japan).
- VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the International Workshop on Hardware/Software Code-sign (CODES)* (Stockholm, Sweden).
- WILDE, D. 1993. A library for doing polyhedral operations. Tech. Rep. 785, IRISA Rennes, France.

Received June 2004; revised June 2006; accepted December 2006