Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies*

Ilya Issenin[1], Erik Brockmeyer[2], Miguel Miranda[2], Nikil Dutt[1]

[1]*University of California, Irvine, CA 92697*
{isse,dutt}@ics.uci.edu

[2]*IMEC, B-3001 Leuven, Belgium*
{miranda,brockmey}@imec.be

Abstract

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 paper 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 reduce energy consumption of the memory subsystem when using a scratch pad memory by a factor of two on average compared to a cache of the same size.

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 proposed for general-purpose architectures, a hardware only implementation has several drawbacks. A hardware controller approach adds additional power and area cost [2]. 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 their unpredictable latency [8].

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 paper we present an approach for performing data reuse analysis, suggesting several memory configurations for exploiting reuse that adapts the program to the selected memory configuration. 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 scratchpad organization. We show the efficiency of our approach on several multimedia and streaming benchmark kernels, generating a factor of two reduction in memory energy consumption.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes our approach for data reuse analysis. Section 4 evaluates the benefits and overheads of the proposed approach. Section 5 concludes the paper.

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. [1][12]). However, we do not address this problem since we assume that all possible loop transformations for improving locality are already performed before applying the technique presented in this paper.

In [13] an approach has been proposed to use a scratch pad memory to store scalars and some of the arrays of the application. The partitioning of data between the scratch pad and the cache is done at compile time and is fixed during the execution of the program. This leads to the non-optimal use of the scratch pad memory since during the execution of the program different parts of arrays may be reused. The same drawback has the approach presented in [15], where most frequently used data structures and basic blocks are statically placed in the scratch pad memory.

At IMEC a methodology for data transfer and storage exploration (DTSE) has been developed which includes

* This work was partially supported by NSF grants CCR-0203813 and CCR-0205712.

data reuse optimization step [5]. Up to now this step has not been fully formalized. In [17] an attempt has been made to explore tradeoffs between scratch pad memory size and power, assuming an optimal dynamic (run-time) placement of data in scratch pad memory. However, no technique was presented for implementing such optimal placement. In [18] the analysis technique has been presented. However, it has been limited to two nested loops and one array reference inside.

```
for i=0 to 10
  for j=0 to 10
    for k=0 to 3
      val = f(val)
      val +=
        A[50i+3j+k]
```

(a) Original program

```
int buf[34]
for i=0 to 10
  for m=0 to 33
    buf[m]=A[50i+m]
  for j=0 to 10
    for k=0 to 3
      val = f(val)
      val += buf[3j+k]
```

(b) Output file obtained with technique [7]

```
int buf[1]
for i=0 to 10
  buf[0]=A[50i]
  for j=0 to 10
    for k=0 to 3
      if (k==3) buf[0]=A[50i+3j+3]
      val = f(val)
      val += (k%3==0)?
                buf[0]:A[50i+3j+k]
```

(c) Output file obtained using our technique

Figure 1. Comparison of several approaches for exploiting data reuse

In [7][8] the problem of dynamic placement of data in scratch pad memory is also addressed. The solution relies on performing loop transformations first to simplify the reuse pattern. However, if loop reordering is not possible (e.g. due to dependencies) and the reused areas are not continuous, the memory requirements for scratch pad in their approach may significantly exceed the amount of actually reused data, hence leading to suboptimal results. Moreover, partial update of the buffer while keeping the data that will be reused in the future is not possible.

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 run time by replacing the data that is not going to be reused anymore by the new data. 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 both 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 actually used in the transformed program. Our transformations do not change loop or array references order.

To show the difference in results between our and the above mentioned approaches we apply those to the example depicted in Figure 1. The technique proposed in [13] allocates the whole array A to the scratch pad memory. The code of the program is not changed and the scratch pad memory required is 534 data elements. If we use the approach proposed in [7], the code that can be generated is shown in Figure 1b. The buffer size is 34 data elements, and only 10 of them are used more than once. The code obtained using our technique is shown in Figure 1c. In our approach, the number of accesses to the scratch pad is two times less compared to [7], 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.

3. Our Data Reuse Detection and Code Transformation approach

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 memory and all the others are fetched from main memory directly (bypass).

The algorithm for reuse analysis uses a description of loop structure and array references as an input. The output of our algorithm is a hierarchical set of buffers, any of which can be placed in the scratch pad memory, and the input code transformed to include the appropriate data transfers. The problem of design space exploration (selecting which of the buffers should be used) is not in the scope of this paper and has been addressed previously in [3]; the focus of this paper is on the automation of reuse detection required to generate the exploration space of the custom memory configurations.

Our algorithm performs analysis of data reuse pattern on each nested loop level. Data reuse is detected between different array references as well as between the 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 those operations at each nested loop level results in a hierarchical set of buffers.

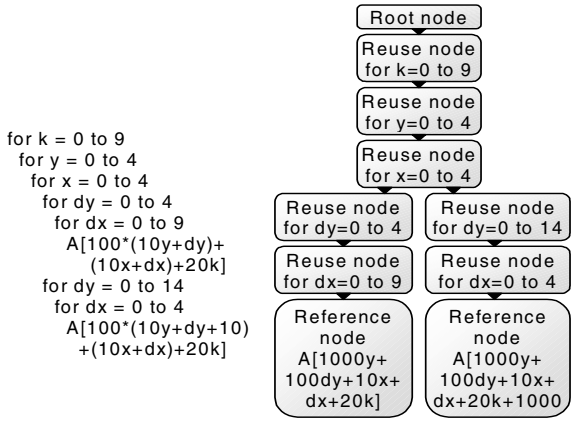
Figure 2 gives an outline of our algorithm. First, in Step 1 we select a part of the program that may be beneficial to optimize and extract its loop structure surrounding array access. This step is performed manually by the designer. Then we build a *reuse tree*, which resembles the loop structure of the code. Figure 3a,b shows an example of input code and corresponding reuse tree. Each reuse node in the reuse tree represents 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 4, a program and corresponding reuse tree is presented. Since index expressions of the references have different coefficients affecting iterator x, there is little reuse opportunities 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 $\mathcal{N}$ all descendant reference nodes should have the same coefficients of the iterators declared in the node $\mathcal{N}$ and all its ancestors.

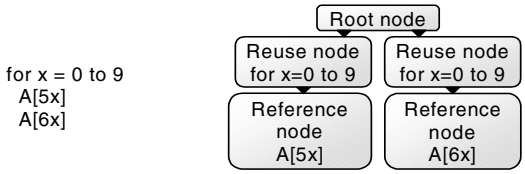
Input: description of loops and array references
Output: memory configuration and transformed program
1: Select part of the program to be optimized
2: Extract loop and array reference information for those pieces of code
3: **For** each array **do**
  3-1: Build reuse tree
  3-2: **For** each reuse node (*current node*) of reuse tree **do**
    3-2-1: Classify iterators into three groups: *fixed, moving* and *filling iterators*
    3-2-2: Create low cover and full cover sets
    3-2-3: Transform sets to 2M-dimensional space
    3-2-4: Divide transformed sets into cells and assign distance numbers to them
    3-2-5: Find grid and basic pieces
    3-2-6: Determine the size of the buffers required for the current loop level
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

Figure 2. Outline of the algorithm for detecting reuse and transforming the program to use a scratch pad memory



(a) Original program **(b) Reuse tree**

Figure 3. Creating reuse tree



(a) Original program **(b) Reuse tree**

Figure 4. An example of reuse tree with split node

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

In step 3-2-1 we classify all loop iterators that are descendants or ancestors of $\mathcal{L}$ and $\mathcal{L}$ 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 $\mathcal{L}$. All the iterators defined in nodes from current to root node are split into two groups: moving iterators (from node $\mathcal{L}$ to some node $\mathcal{K}$) and fixed iterators. To determine node $\mathcal{K}$, iterators should be added one by one to a set **M** of moving iterators as long as the following property remains true:

$$\forall(N,C) \in \mathbf{M}, \text{coef}(C) > \text{coef}(N):$$
$$(\text{upper\_bound}(N)-\text{lower\_bound}(N) + 1)*\text{coef}(N) \le \text{coef}(C) \cup$$
$$\text{coef}(C) \bmod \text{coef}(N) = 0,$$

where upper_bound(N), lower_bound(N) are bounds of the loop of iterator N; coef(N) is the coefficient of iterator N in index expression; mod is the modulo operation.

In other words, all moving iterators should have coefficients that evenly divide each other and for any two iterators the greatest coefficient should not be less than the value on which the loop with iterator with lesser coefficient can change the value of the index expression.

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

For the code shown in Figure 3a 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.

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 the next step 3-2-2, 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.

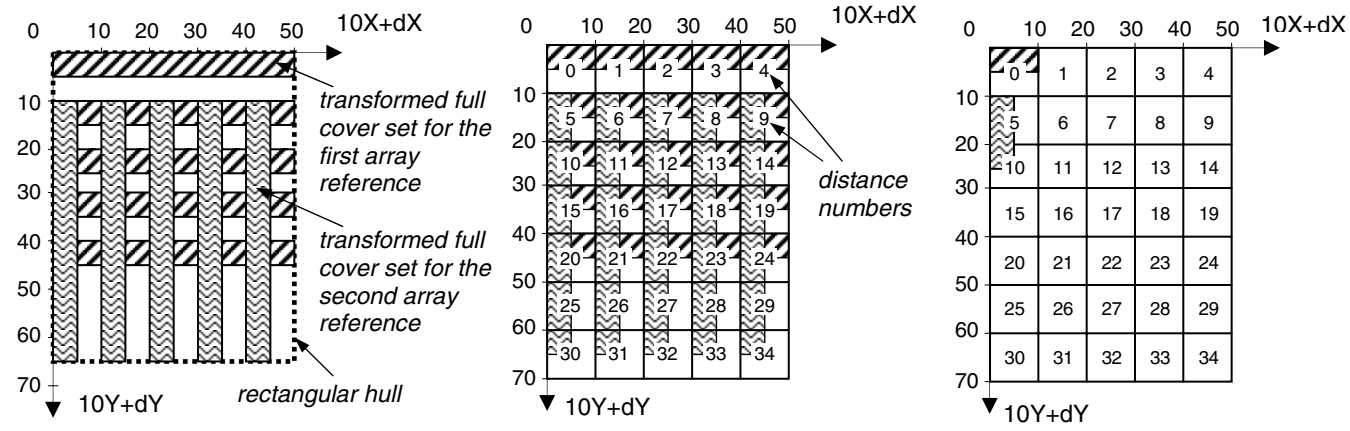
We illustrate our algorithm using the code shown in Figure 3a. Assuming that x is the current iterator, the low cover set is:

$$LC = [V]:\{\exists dx,dy: ((0 \le dy \le 4 \cap 0 \le dx \le 9 \cap v=100dy+dx) \cup (0 \le dy \le 14 \cap 0 \le dx \le 4 \cap V=100dy+dx+1000))\}$$

Next, in Step 3-2-3 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. This is needed to obtain smaller buffers with simpler address calculations in the transformed program. It is done by applying the following relation R to the cover sets:

$$R = [V] \rightarrow [V_1..V_M, dV_1..dV_M]:$$
$$\{0 \le dV_1 \le dV_{1max} \cap .. \cap 0 \le dV_M \le dV_{Mmax} \cap$$
$$0 \le V_1 \le V_{1max} \cap .. \cap 0 \le V_M \le V_{Mmax} \cap$$
$$V = \text{coef}(V_1)/\text{step}_1*(\text{step}_1*V_1+dV_1) + .. +$$
$$\text{coef}(V_M)/\text{step}_M*(\text{step}_M*V_M+dV_M) + \min(V)\},$$

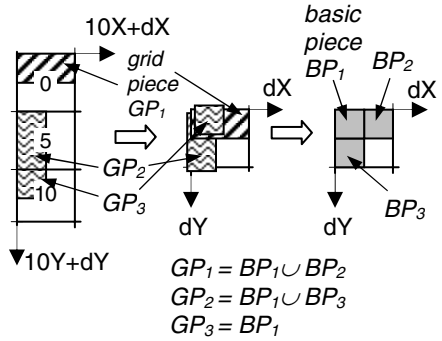
1530-1591/04 $20.00 © 2004 **IEEE**



(a) Transformed full cover sets for each array reference

(b) Division into rectangular cells and assignment of distance numbers

(c) Transformed low cover sets for each array reference



(d) Grid and basic pieces

Figure 5. Finding buffers configuration

where $V_1..V_M$ is a new M-dimensional element of the set; min(V) – value of the minimal element in input set of one-dimensional elements; coef(V_K) – coefficient of K^{th} moving iterator in 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 ($V_1..V_M$) only in one way. More than one possible choice may exist for those constants. They affect both the complexity of the address calculations in the transformed code as well as the buffer sizes.

To illustrate selection of the constants, let's look at the index expression from our example: index = 1000y+100dy+10x+dx+k. It can be rewritten as index = 100(10y + dy) + (10x+dx) + k. The relation for this example is

$$R = [V] \rightarrow [V_1,V_2, dV_1,dV_2]:\{0 \le dV_1 \le 9 \cap 0 \le dV_2 \le 9 \cap 0 \le V_1 \le \infty \cap 0 \le V_2 \le 9 \cap V = 100*(10V_1+dV_1) + 1*(10V_2+dV_2) + 0\}.$$

To transform the low and full cover sets we apply the relation R to the sets. The transformed sets for the example are depicted in Figure 5a and Figure 5c.

Next, in Step 3-2-4 we divide the transformed full cover set into M-dimensional rectangular cells. 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 0. We call those numbers *distance numbers* (Figure 5a,b).

In Step 3-2-5 we determine the *grid* and the *basic pieces*. *Grid* and *basic pieces* are sets of M-dimensional points. To find grid pieces, we intersect the transformed low cover set with the rectangular cells obtained earlier. In other words, for each value of $V_1..V_M$, we obtain a set (which is called a grid piece) G(R(low cover set)), where

$$G = [V_1..V_M, dV_1..dV_M] \rightarrow [ddV_1..ddV_M]: \{ddV_1= dV_1 \cap .. \cap ddV_M= dV_M \cap V_1..V_M \text{ are set to given values}\}$$

Grid pieces may overlap with each other. However, it is possible to represent every grid piece as a union of non-overlapping *basic pieces*. The set of basic pieces is common and different combinations of its elements are used to represent all grid pieces. Figure 5d shows the grid and basic pieces for the example discussed above.

Finally in Step 3-2-6 we determine the size of the buffers required for the current level of the memory hierarchy. It 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 5d, 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. The maximum difference in distance numbers is 10. Hence, the size of the buffer needed to hold all reused data is 25*10=250.

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) that increases the number of accesses to main memory but decreases the cost of scratch pad memory.

After obtaining all information about buffers that can be used at each loop level, in step 4 we perform design space exploration and select the buffers that should be implemented.

Finally, in Step 5, we transform the program to include selected buffers holding reused data. Each array reference is replaced by some conditional statements that determine where the data should be read from and whether it should be copied to the buffer. This simple scheme does not allow to use DMA or burst transfer mode while accessing the main memory though.

As an alternative approach, the buffer can be filled with data every iteration of *fixed iterators* and parts of the buffer can be updated in the beginning of iteration of the loops with *moving iterators*. This allows the use of a DMA controller.

After obtaining the transformed program it is necessary simplify it to reduce computation overhead due to the need of addressing scratch pad memory. Even though conventional compilers do perform some optimizations that help in this case, other optimizations (e.g. address optimizations [6], etc) should be applied due to special structure of the code.

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 ability 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 [9][19]. The running time of the tool is in order of seconds for the benchmarks we used.

We have used a Pentium 4, Sun UltraSparcIII, PA-8500 (HP) workstations for profiling purposes. The SimpleScalar simulator [4] has been used for obtaining the number of misses for cache. We have used the CACTI model [14] for energy estimation of both the cache and scratch pad memories and also for the 128Kb off-chip memory all at 130nm technology.

For the benchmarks we have used kernels extracted from a H.263 video encoder [11], QSDPCM encoder [16], and the Laplace algorithm performing edge enhancements in images. 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. Each of them consists of one buffer. The tool has provided us with the code versions that implement necessary data transfers between the scratch pad buffers and the main memory for the selected buffers. All transfers in the code are performed by the processor without the use of DMA support.

4.1. Effect on the memory subsystem

In this section we have examined the effect when using a scratch pad memory on the reduction of traffic to main memory as well as on the energy spent in the memory subsystem. The comparison of the scratch pad based memory architecture has been made against a system having a direct mapped data cache of the same size. The goal is to evaluate the energy efficiency of our software steered data replacement approach compared to that one implemented by a hardware cache controller i.e. with the LRU replacement policy. The sizes of the scratch pad memory and the cache have been selected to be the closest values that are powers of two while being greater or equal than the buffer size required in the scratch pad configuration. The cache line size has been selected to be the minimal allowed by the simulator [4] (8 bytes, which is 2 data elements in all benchmarks). In this way, we compare solely how well is the temporal locality of the data exploited without considering spatial locality issues. These issues are largely orthogonal and can be optimized by data layout transformations [10] which fall outside the scope of this paper.

The energy savings when using scratch pad in comparison with cache are coming from two sources.

First, a scratch pad memory consumes less energy than a cache of the same size per one access (about two times less for direct mapped cache [14]).

But especially also the more optimal data replacement decisions for the scratch pad memory when compared to that ones made according to the LRU policy of the cache controller result in less accesses to the main memory for all cases except the first one (see Table 1) which uses only one data reuse buffer out of two required for full data reuse. In our experiments we have used a relatively small off-chip memory and have not accounted for the energy dissipation in the off-chip buses due to limitations of the used energy model [14]. It is clear that the memory system energy savings improves if we account for buses energy or use larger main memory (for all cases except the first one).

As we can see from Table 1, the scratch pad based memory subsystem consumes 40% to 60% less energy than the system with a cache of the same size.

We have also compared the effectiveness of our technique with the approaches proposed in [13] and [7] for the QSDPCM benchmark. Using [13], it requires 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 [7] and implementing a buffer at the same level as we have done for our reported results, it requires 10 times bigger buffer. The amount of accesses to the main memory is also about 10 times greater than in our approach mainly due to the loading of unused data. This even exceeds the number of accesses in the original

Table 1. Experiment results

Benchmark and size of the buffer when using scratch pad memory	The number of accesses to the main memory		Scratch pad memory size	Memory subsystem energy reduction in comparison with a system with the cache of the same size	Increase in the execution time (without the effect of main memory latency)			Code size increase
	Reduction in comparison with a system without the cache	Reduction in comparison with a system with the cache of the same size			Pentium 4	Sun UltraSparcIII	PA-8500 (HP)	
H.263 with a buffer 36K	80%	-9%	64K	41%	1.40	1.85	1.40	3.2
H.263 with a buffer 6K	58%	54%	8K	50%	1.87	2.33	1.59	2.4
QSDPCM with a buffer 1K	89%	13%	1K	49%	1.28	1.12	1.77	1.78
Laplace with a buffer 4K	89%	65%	4K	58%	1.29	1.39	1.36	1.85
Laplace with a buffer 2.5K	89%	65%	4K	58%	1.68	1.54	2.21	1.68

program, giving negative energy savings in comparison with the original program.

4.2. Control code overhead estimation

Adding buffers for keeping frequently used data implies the addition of code that determines if the current fetch should be performed from the buffer or main memory and it 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 transfers and increase in the code size.

To estimate the overhead, the original and transformed programs have been run on a number of workstations. Since on a workstation a data cache is large enough to hold the whole data set, we can ignore the influence of the cache configuration for these platforms.

The code size overhead (expressed as the ratio in the number of assembly lines) has been measured using gcc compiler for the Sun workstation. The results are shown in Table 1. On average the increase in the number of cycles is about 1.6. The increase in the code size is about a factor of two. However, when compared to the size of the whole application, the overhead in the code size is much smaller and is not likely to cause significant increase in instruction cache miss rate and consequently increase in energy consumption in the instruction memory hierarchy.

5. Conclusion

In this paper we present a 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 scratchpad memory organized as hierarchical set of buffers. Our approach considers the data reuse opportunities both between different references as well as for the same reference between different iterations of the outer loop. We have demonstrated about a factor of two reduction in memory subsystem energy when applying our approach when compared to a cache based implementation, and we outperform previously reported scratchpad approaches.

References

[1] D. F. Bacon, S. L. Graham et al. Compiler Transformations for High-Performance Computing. ACM Computing Surv., 26(4), 1994.

[2] R. Banakar, S. Steinke et al. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. CODES, 2002.

[3] E. Brockmeyer, M. Miranda, F. Catthoor, and H. Corporaal. Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organisations. DATE, Germany, 2003.

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

[5] J. Diguet, S. Wuytack, F. Catthoor, and H. De Man. Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings. In Proceedings of the IEEE International Symposium on Low Power Design, pages 30-35, Monterey, CA, August 1997.

[6] C. Ghez, M. Miranda, A. Vandecappelle, F. Catthoor, D. Verkest. Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm. Workshop on Signal Processing Systems, Lafayette LA, Oct. 2000.

[7] M. Kandemir and A. Choudhary. Compiler-Directed Scratch Pad Memory Hierarchy Design and Management. DAC, New Orleans, Louisiana, June 2002.

[8] M. Kandemir, J. Ramanujam, M. J. 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.

[9] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shipeisman, and D. Wonnacott. The Omega Library interface guide. Tech. Report CS-TR-3445, CS Dept., Univ. of Maryland, College Park, March 1995.

[10] C. Kulkarni, M. Miranda, C. Ghez, F. Catthoor, H. De Man. Cache Conscious Data Layout Organization For Embedded Multimedia Applications. DATE, Germany, March 2001.

[11] K. Lillevold et al., Telenor R&D, H.263 test model simulation software. Dec. 1995.

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

[13] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. DATE, Paris, March 1997.

[14] P. Shivakumar, N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Technical Report 2001/2, Aug. 2001.

[15] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. DATE, Paris, France, March 2002.

[16] P. Stobach. A new technique in scene adaptive coding. In Proc. Of EUSIPCO, Grenoble, 1988.

[17] T. Van Achteren, F. Catthoor, R. Lauwereins, G. Deconinck. Search Space Definition and Exploration for Nonuniform Data Reuse Opportunities in Data-Dominant Applications. ACM Trans. on Design Automation of Electronic Systems, Vol. 8, No. 1, Jan. 2003.

[18] T. Van Achteren, F. Catthoor, R. Lauwereins, and G. Deconinck. Data Reuse Exploration Techniques for Loop-Dominated Applications. In IEEE/ACM Design Automation and Test Conference, Paris, France, 2002.

[19] D. K. Wilde. A Library for Doing Polyhedral Operations. Technical Report 785, IRISA Rennes, France, 1993.