Address Code and Arithmetic Optimizations for Embedded Systems

J. Ramanujam[*] Satish Krishnamurthy[*] Jinpyo Hong[*] Mahmut Kandemir[†]

Abstract

An important class of problems used widely in both the embedded systems and scientific domains perform memory intensive computations on large data sets. These data sets get to be typically stored in main memory, which means that the compiler needs to generate the address of a memory location in order to store these data elements and generate the same address again when they are subsequently retrieved. This memory address computation is quite expensive, and if it is not performed efficiently, the performance degrades significantly. In this paper, we have developed a new compiler approach for optimizing the memory performance of subscripted or array variables and their address generation in stencil problems that are common in embedded image processing and other applications. Our approach makes use of the observation that in all these stencils, most of the elements accessed are stored close to one other in memory. We try to optimize the stencil codes with a view of reducing both the arithmetic and the address computation overhead. The regularity of the access pattern and the reuse of data elements between successive iterations of the loop body means that there is a common sub-expression between any two successive iterations; these common sub-expressions are difficult to detect using state-of-the-art compiler technology. If we were to store the value of the common sub-expression in a scalar, then for the next iteration, the value in this scalar could be used instead of performing the computation all over again. This greatly reduces the arithmetic overhead. Since we store only one scalar in a register, there is almost no register pressure. Also all array accesses are now replaced by pointer dereferences, where the pointers are incremented after each iteration. This reduces the address computation overhead. Our solution is the only one so far to exploit both scalar conversion and common sub-expressions. Extensive experimental results on several codes show that our approach performs better than the other approaches.

1 Introduction

Optimizing compilers have become an essential component of embedded and high-performance computer systems. In addition to translating the input program into machine language, they analyze it and apply various transformations to reduce its running time or its size. As optimizing compilers become more efficient, programmers can become less concerned about the details of the underlying machine architecture and can apply higher-level, more succinct and more intuitive programming constructs and program organizations. Simultaneously, hardware designers are able to employ designs that yield greatly improved performance because they need to only concern themselves with the suitability of the design as a compiler target and not with its suitability as a direct programmer interface.

The emergence of embedded systems or the system-on-chip has also placed a high burden on the effectiveness of the code optimization performed by compilers. Embedded systems generally have tailor-made architectures for specific applications including special instruction sets with a goal of reducing the time/power of execution. These systems generally offer long compilation time where the original program can be analyzed deeply and then be transformed to one that performs better. The recent boom of their usage in cellular phones and DSP applications is proof of the validity of this approach.

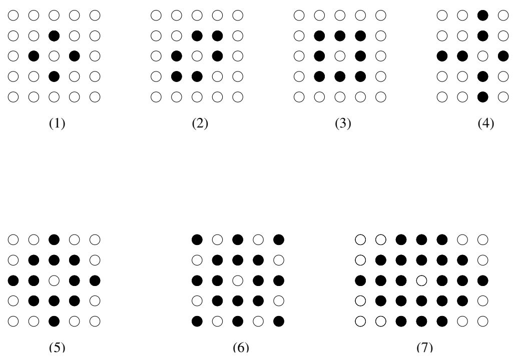

Figure 1: Commonly Observed Stencils.

An important class of problems used widely in both the embedded systems and scientific domains perform memory intensive computations on large data sets. These data sets get to be typically stored in main memory. The storage of these data sets in memory means that the compiler needs to generate the address of a memory location in order to store these data elements and generate the same address again when they are subsequently retrieved. As we shall see, this operation of memory address computation is quite resource intensive and degrades the overall performance significantly if not performed efficiently.

An important feature of the class of problems considered in this paper is the regularity exhibited by their access patterns. A regular problem can be characterized by its corresponding stencil. Figure 1 illustrates some of the commonly found stencils.

In this paper, we present an approach of optimizing the address generation of these stencil problems. Our approach makes use of the observation that in all these stencils, a significant fraction of the elements accessed are stored close to one other in memory. The main contributions of this paper is optimization technique that results in the following:

- eliminating redundant arithmetic computation by recognizing and exploiting the presence of common sub-expressions across different iterations in stencil codes; and
- conversion of as many array references to scalar accesses as possible, which leads to reduced execution time, decrease in address arithmetic overhead, access to data in registers as opposed to caches, etc.

The rest of this paper is organized as follows. In Section 2, we present the relevant background and the motivation for this paper. In Section 3, we discuss our approach to optimize the address generation for stencil codes. We also present some results that illustrate the validity of our approach. Section 4 summarizes and concludes the paper.

[*]Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA. {jxr,satish,jphong1}@ece.lsu.edu

[†]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA. kandemir@cse.psu.edu

2 Background And Motivation

Loop nests form an integral part of embedded codes. These basically consist of the same series of operations being performed on different sets of data elements. The data elements are usually declared as array data types. The main advantage of this is that they allow the programmer to concentrate on the functionality of the program instead of worrying about the storage pattern of these data in hardware and their subsequent retrieval. It is up to the compiler to provide the necessary support by retrieving the data elements from the memory locations where they may be stored. As we shall see this retrieval of data is not a trivial function. In most cases it is this part of the program that has the most bearing on its performance, especially on those programs that work with large data sets. The compiler has to perform some optimizations on the code if the performance of the program has to improve. In the following pages, we look at one such optimization that will improve the performance of memory intensive programs.

2.1 Array Mappings

An array data type is typically stored in memory as a contiguous block of memory locations. For example a single dimensional array a[N] (array 'a' contains N elements) is stored as a single contiguous block of N memory locations.If the address of the first element is denoted by BaseAddress(A) the address of the last element is BaseAddress(A) + N -1. For simplification of the discussion, we have assumed that each array element occupies one memory word, although this assumption is not necessary for our subsequent analysis.

A two-dimensional array b[ROW][COL] occupies ROW * COL memory locations. Two popular approaches to mapping two dimensional arrays to hardware are Row-Major order and Column-Major Order. In the Row-Major method of storage, elements of the first row are placed in consecutive memory locations in order of increasing column index. This is followed by elements of the second row in the same order and so on. For the Column-Major method of storage, elements of the first column are stored in consecutive memory locations in order of increasing row index. Most high level languages impose a particular type of storage order on arrays. For example languages like C impose a Row-Major order of storage on all arrays while languages like FORTRAN impose a Column-Major order of storage. Figure 2 shows the different ways in which arrays are stored in memory.

2.2 Relevant Terminology

Access to an array is denoted by its name and a subscript. For example a single dimensional array 'a' can be accessed as a[0], a[1] etc.. where '0', '1', etc are the subscripts of the array. These subscripts are actually the positions of data inside the array, measured from the beginning of the array. Thus the access a[i] is the access to a data at a distance of 'i' elements from the start of the array. The memory address where a[i] is stored can be calculated as:

$$BaseAddress(a) + i. \tag{1}$$

A two-dimensional array has both a length and a breadth and its access is denoted by b[i][j], where 'i' is the row subscript and 'j' is the column subscript. Again the memory address where b[i][j] is stored can be calculated as:

$$BaseAddress(b) + (number\ of\ columns\ in\ b * i) + j. \tag{2}$$

This method of address calculation assumes a row major order of memory storage. This assumption is followed throughout this paper unless mentioned otherwise.

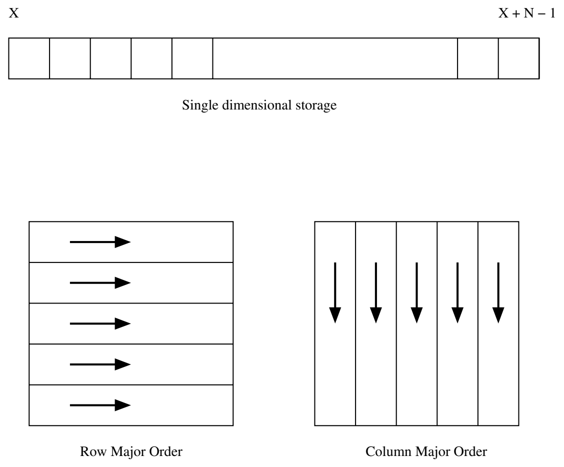

Figure 2: Storage pattern of Arrays in Memory.

2.3 Performance Issues: Overview

Most memory intensive applications are characterized by some common features such as loop nests and the usage of large data blocks that are usually stored in memory as arrays. The presence of these traits mean that the performance is dependent on the speed of retrieval and subsequent storage of data elements in memory. Most modern processors feature one or more levels of cache memory. These cache memories speed up data retrieval by storing frequently accessed data closer to the processor where they may be accessed at high speeds. Another important performance bottleneck is the problem of memory address computation. As we have already seen, mapping a logical array to hardware means that the data values are stored at some pre-determined memory location. When ever this data has to be retrieved, the same address has to be generated. This is not a trivial problem and is quite computation intensive. For each access to the elements of the array the compiler has to generate an address according to Equation 2. It then has to generate a load instruction with this address. As the memory subsystem is slow when compared to the processor, it takes a significant amount of time before this load instruction may be serviced.

Modern compilers perform optimizations on the original code to improve their performance. Tiling is a well-known example. Tiling reduces the iteration space over which reuse of data occurs and help in utilizing the cache memory to the best possible extent. Loop unrolling is another optimization that improves the performance by reducing the loop overhead, increasing the instruction level parallelism and improving the register and data cache locality.

In the following subsection we examine an example memory intensive code segment and discuss the implications of this segment with respect to memory address computation.

2.4 Motivating Example

Consider the loop nest given below and its memory access pattern. Here N is the size of the array along its length.

```
for( i = 3; i <N - 3; i++) {
    b[i] = ( a[i-3] + a[i-2] + a[i-1] + a[i] + a[i+1] + a[i+2] + a[i+3]) / 7 ;
}
```

Such loop nests are called stencil codes because they compute values using neighboring array elements in a fixed stencil pattern. This stencil pattern of data accesses is then repeated for each element of the array. For instance, the above loop nest consists of

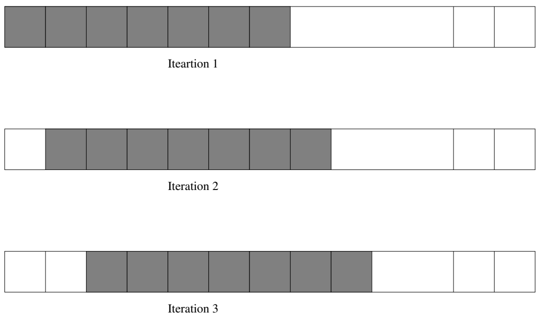

Figure 3: Access pattern for the motivating example

a simple 7 point stencil in one dimensions as shown in figure 3. On each loop iteration, seven neighboring elements of the array are accessed and their sum calculated. As the computation progresses, the stencil pattern is repeatedly applied to array elements in the row, sweeping through the array. Such kind of loop nests are very popular in image processing applications.

At first glance, it is immediately obvious that we are trying to access 7 elements of the single dimensional array 'a' , all at different locations, in successive iterations. Thus we need to perform 7 address computations to retrieve the data from the memory. Most traditional compilers tend to store the base address of array 'a' in a register. Access to different elements of this array means the computation of the memory addresses using Equation 1. The computation of the sum of these 7 elements is also not a trivial operation. Since these operations need to be performed for each iteration of the loop, the number of total computations to be performed for even small values of N is quite exhaustive and could degrade performance if not performed efficiently.

In the next subsection, we take a close look at some popular approaches to make the above example more efficient. In Section 3, we present our approach of using pointers to access the data elements. Lastly we present a summary of the results for our new technique that demonstrates the effectiveness of it.

2.5 Related Work

Specialized hardware has been used for reducing address arithmetic overheads [8], but this leads to increased design complexity and cost. In addition, several authors have addressed the problem of laying out scalar variables to make effective use of address generation units in embedded processors [10]. The IMEC group has used several transformations and advocated the use of special hardware for reducing the effect of address arithmetic overhead [4, 6, 13, 14]. Gupta etl al. [7] have used induction variable analysis and optimization to improve the performance of address arithmetic.

Rivera and Tseng present algorithms for loop tiling for 3D stencil codes. They argue that since the working set is small for 2D stencil codes, tiling in not necessary as the data fits within the data cache. However for 3D stencil codes, the larger working set justify loop tiling. They then look at algorithms to find the tile sizes that would utilize the data cache to the optimal extent. They also look at inter array padding to reduce the cross interference and self interference misses that might occur. This approach helps in improving the performance by speeding up data retrieval. This is because most machines take a long time to service a secondary memory load operation. So tiling loops to reuse data in the cache reduces the time spent on this operation because in most machines the data cache operates at processor speed. However this approach does not address the actual arithmetic computation, which we have seen has a large impact on performance.

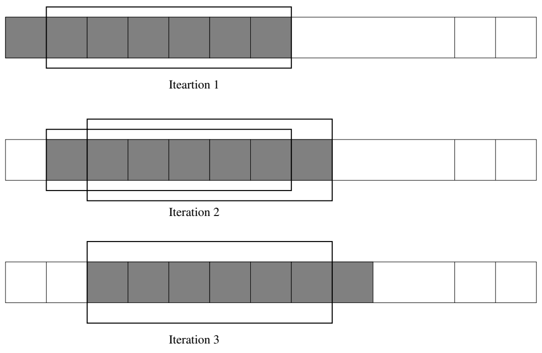

Figure 4: Access pattern for the motivating example.

Callahan et al. [3] (and Liu's group [11, 12]) present a technique for register allocation of subscripted variables. They argue that most compilers do not allocate array elements to registers because standard data-flow analysis make it difficult to analyze the definitions and uses of individual array elements. They then discuss an approach of replacing subscripted variables by scalars to effect reuse. The subsequent code is then modified to use the data elements stored in these scalars. The advantage of this approach is that all array variables are replaced by scalars that are then mapped to registers. In successive iterations, those variables which are reused can be accessed from registers directly. This improves performance because it eliminates the address calculation overhead. Replacing array variables by scalars means that the cache configuration does not degrade performance significantly. This is because all variables that are reused are present in registers and reused data is no longer accessed via the cache mechanism. This approach does improve the performance to a large extent. However the arithmetic overhead involving the actual data elements still remains. Another problem is that of register pressure. By mapping the scalars to registers, we use up a lot of registers. If the amount of reuse is large, the register pressure builds up and can significantly degrade the performance.

In contrast to these works, we present a technique that (i) eliminates redundant arithmetic computation by recognizing and exploiting the presence of common sub-expressions across different iterations in stencil codes; and (ii) converts as many array references to scalar accesses as possible, which leads to a significant improvement.

3 A Pointer Approach to Array Accesses

As seen in Section 2, the overheads involved in stencil codes involve both the pure arithmetic overhead and the address overhead. The pure arithmetic overhead involves the actual computation using the data elements. The address overhead involves computing the memory location where each data element is stored. In this section, we take a look at our approach to improve the performance of stencil codes. We take a look at how our approach decreases the overheads and finally we present some results to justify the validity of our approach.

3.1 Memory Access Patterns

Let us again consider the loop nest and the memory access pattern described in the previous section. Figure 4 shows the access pattern. From the figure it is clear that there are a number of data elements that are reused. Callahan et al. [3] make use of this feature to replace these array accesses by scalars. This means that in successive iterations, we can directly use the values stored in these scalars instead of accessing the array again. However for new data elements that are needed, we still use array accesses. The effective goal of the code segment described in the previous section is to calculate the sum of seven data elements, the elements moving along the array across successive iterations. From Figure 4, another thing that can be seen is that as many as five data elements are reused across any two iterations. This means that the sum of these five elements is a common sub-expression in the arithmetic computation across any two successive iterations. If we were to store the value of the common sub-expression in a register, then across iterations the amount of arithmetic computation needed would be greatly decreased. Storing the common sub-expression in a register also decreases the number of scalars that we operate with. This correspondingly means that there is almost no register pressure unlike the other approaches that we described. In each iteration, we also access the new data elements that we need using pointers instead of array accesses. This helps a lot in improving the performance because the memory address computation overhead that we described has now been eliminated.

3.2 Algorithm

The algorithm consists of the following steps:

1. From the access pattern of the loop, find the common sub-expression (CSE) across any two successive iterations.
2. Initialize the CSE at the beginning of the loop body.
3. Modify the loop body to use the value of this CSE.
4. In each iteration, update the value of the CSE.
5. Replace all array accesses by pointer dereferences.

Given a loop nest, the algorithm first looks at the access pattern of the array elements involved. From this access pattern, we pick out the common sub-expression that exists between any two successive iterations. This common sub-expression is effectively a scalar variable that has been mapped to a register. This scalar is initialized at the beginning of the code segment. The rest of the code is then modified to use the value of the common sub-expression. Using this value means that the amount of arithmetic computation involved is decreased significantly. The common sub-expression is also updated in each iteration. New data elements that are needed in each iteration are accessed by pointer dereferences. Using pointers to access data elements improves the performance by reducing the memory address computation overhead. This approach to stencil codes makes full utilization of the spatial locality exhibited by these stencil codes.

3.3 Experimental Results

We made the modifications as determined by our algorithm on some popular loop nests. These included some scientific problems exhibiting regular computational patterns. Most of these patterns are given in section 1. We also modified the loop nests by using Callahan et al.'s approach [3].

The original and modified codes were then timed over 100 runs and averaged. We increased the array dimensions over a wide range of values and measured performance over this range. We plotted graphs for the original code and the modified codes using both Callahan et al.'s approach and ours. This helps us to compare the performance improvements on the original code as a result of the optimizations using both the approaches. On the X-axis we plotted the array dimension and on the Y-axis we plotted the time taken in seconds to fully execute the code segment for that value of array dimension. In the following pages, we show the performance graphs that we plotted. On the top of each page, we also show the stencil pattern over which the performance was evaluated. This shows the performance improvements attained by using Callahan et al.'s approach [3] and our approach over the original code segment.

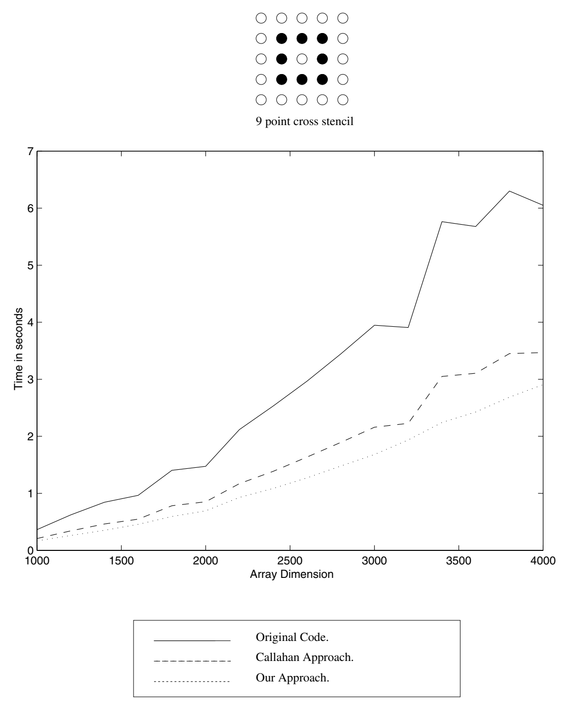

Figure 5: Results for the 9 point cross stencil.

Figure 5 shows the performance of both the original and the modified 9 point 2D stencil codes over a wide range of array dimensions. The performance improvements for both Callahan et al.'s optimizations [3] and ours are about 100 percent for most array dimensions. When the code is compiled using the default compiler options, the compiler performs almost no optimizations on its own. The arithmetic and the address computation overheads are quite huge and have a large impact on the performance. The optimized codes on the other hand, perform much better because the overheads have been reduced to a large extent. We can see that our approach at optimizing the code results in an improvement of about 10% to 20% for most array dimensions. This is because, we have reduced the pure arithmetic overhead to a larger extent than that was done by the other approach.

In the rest of the pages, we show the results of our approach on most of the regular stencil patterns shown in Section 1. Again for all these stencil patterns too, we analyze the performance using both the default and the best compiler options available. The graphs show clearly that our approach at optimization is better than than the one proposed by Callahan et al. [3].

3.4 Summary of the Experimental Results

The graphs plotted in the previous pages show the performance of the naive code using the default and the best compiler options. These options are selected by the compiler so that the resulting application performs at its best. The graph of the naive code provides

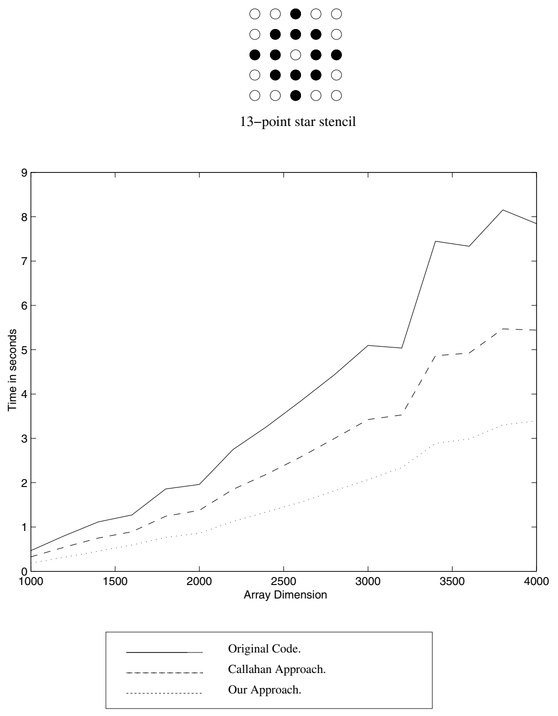

Figure 6: Results for the 13 point star stencil.

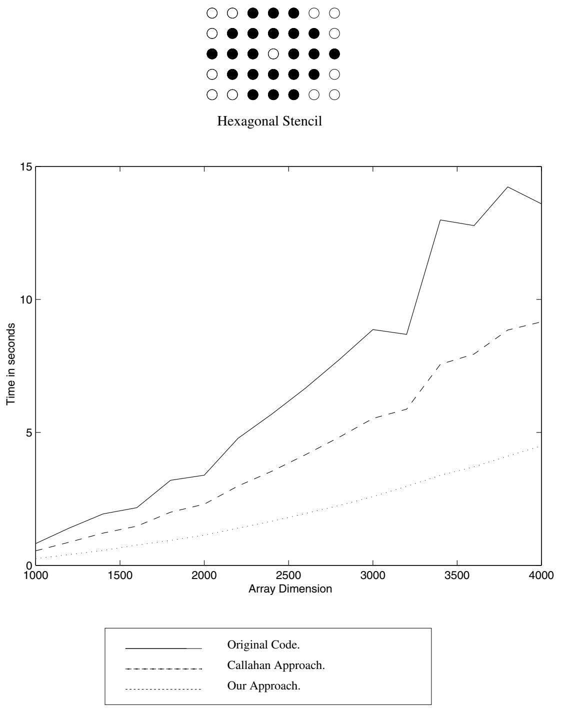

Figure 7: Results for the hexagonal stencil.

us with a benchmark to compare results. We then used Callahan et al.'s approach [3] to optimize the naive code further. This again was compiled using the default and the best compiler options. The resulting code performed much better than the naive code. This was because it succeeded in reducing some of the overheads that we had described. We then used our approach to optimize the naive code , again with a view of reducing the overheads. We also plotted the graphs for our approach. Plotting the graphs for all the approaches on the same page gives us an effective way of comparing the performance improvements. As expected the optimized code performs much better than the naive code. Also we can see that out approach at optimizing the code performs much better than the approach proposed by Callahan et al. The performance improvement between these two approaches is as much as 10-20 percent for most array dimensions.

4 Conclusion and Future Work

In this paper, we focused on optimizing codes that exhibit regular access patterns. These codes are called stencil codes. These code segments have two major overheads: (i) the pure arithmetic computation overhead and (ii) the memory address computation overhead. It it these overheads that determine the performance of these code segments. Callahan et al. [3] propose an approach to optimize these stencil codes and thereby improve their performance. They replace all subscripted array variables by scalars, thereby effecting reuse of these scalar variables. These scalar variables are then mapped to registers. Subsequent reuse of these data elements means that they can be directly accessed from registers instead of through the cache mechanism. This means that loads of all reused data elements can be serviced at processor speed instead of having to deal with cache conflicts and subsequent loads from secondary memory. This approach results in a good improvement in performance because the memory address computation overhead has been reduced. However the major disadvantage with this approach is that because of the large number of data elements that might be reused, the number of scalars that will be needed is also large. This creates a lot of register pressure which then starts to degrade performance. Also this approach does not seek to reduce the pure arithmetic computation overhead.

We have presented an approach to optimize stencil codes with a view of reducing both the arithmetic and the address computation overhead. The regularity of the access pattern and the reuse of data elements between successive iterations of the loop body means that there is a common sub-expression between any two successive iterations. If we were to store the value of the common sub-expression in a scalar, then for the successive iteration, the value in this scalar could be used instead of performing the computation all over again. This greatly reduces the arithmetic overhead. Since we store only one scalar in a register, there is almost no register pressure. Also all array accesses are now replaced by pointer dereferences. This reduces the address computation overhead. These optimizations helped to improve the performance of the stencil codes to a large extent. We also compared the performance with some other popular approaches. The results in Section 3 prove conclusively that our approach was better than the one proposed by Callahan et al. [3].

Work is in progress in integrating the effects of induction variable analysis and optimization, as well as in reducing storage overhead in non-stencil codes, and in combining these with data layout optimization techniques.

Acknowledgments J. Ramanujam has been supported in part by NSF Young Investigator Award 9457768 and NSF grants 0073800 and 0121706. Mahmut Kandemir is supported in part by NSF CAREER award 0093082.

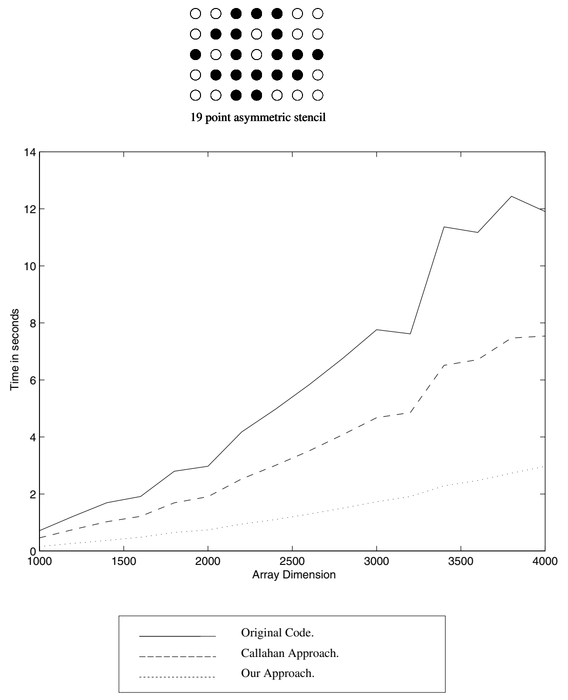

Figure 8: Results for the 19 point asymmetric stencil.

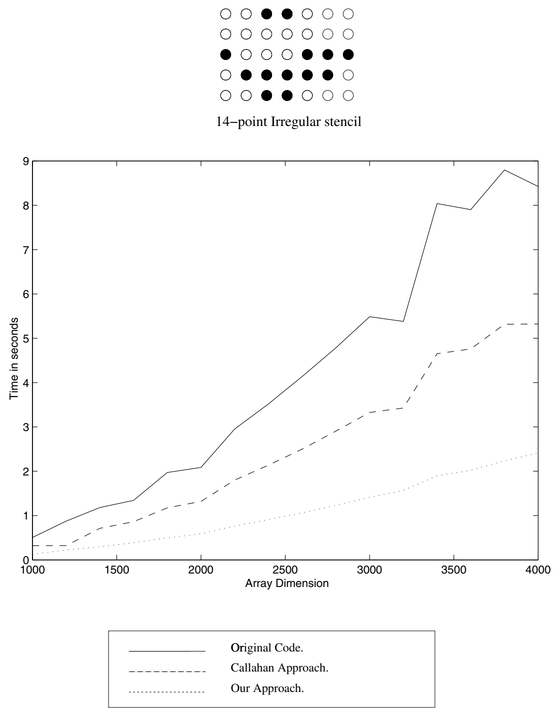

Figure 9: Results for the 14 point irregular stencil.

References

[1] D. Bacon, S. Graham and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys,* 1994.

[2] D. Bailey. Unfavourable Strides in Cache Memory Systems. Technical Report RNR-92-015, NASA Ames Research Center, May 1992.

[3] D. Callahan, S. Carr and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proc. ACM SIGPLAN '90 conference on Programming Language Design and Implementation,* 1990.

[4] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle. *Custom Memory Management Methodology,* Kluwer Academic Publishers, 1998.

[5] S. Coleman and K. McKinley. Tile size Selection Using Cache Organization and Data Layout. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation,* June 1995.

[6] K. Danckaert, F. Catthoor, and H. De Man. System-level memory management for weakly connected image processing. In *Proc. Euro-Par'96*, 1996.

[7] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta. Analysis of high-level address code transformations for programmable processors. In *Proc. Design, Automation and Test in Europe (DATE),* Paris, March 2000.

[8] K. Kitagaki, T. Oto, T. Demura, Y. Araki, and T. Takada. A new address generation unit architecture for video signal processing. *Proc. Visual Communications and Image Processing,* 1991.

[9] M. Lam, E. Rothberg and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. Fourth International Conference on Architerctural Support for Programming Languages and Operating Systems,* April 1991.

[10] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. *Proc. Int. Conference on Computer-AidedDesign (ICCAD),* 1996

[11] Y. Liu and S. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE International Conference on Computer Languages,* May 1998.

[12] Y. Liu, S. Stoller and T. Teitelbaum. Static Caching for Incremental Computation. *ACM transactions on Programming Languages and Systems,* Vol. 20,No. 3, May 1998.

[13] M. Miranda, F. Catthoor, M. Janssen, and H. De Man. High-level address optimization and synthesis techniques for data-transfer intensive applications. *IEEE Trans. VLSI Systems,* vol. 4, no. 6, 1998.

[14] M. Miranda, F. Catthoor, M. Janssen, and H. De Man. ADOPT: Efficient hardware address generation in distributed memory architectures. In *Proc. International Symposium on System Synthesis,* 1996.

[15] P. Panda and N. Dutt. Reducing Address Bus Transitions for Low Power Memory Mapping. In *Proc. European Design and Test Conference,* Paris, March 1996.

[16] P. Panda and N. Dutt. Memory Data Organization for Improved Cache Performance in Embedded Processor Applications. *ACM Transactions on Design Automation of Electronic Systems,* October 1997.

[17] P. Panda, N. Dutt and A. Nicolau. Local Memory Exploration and Optimization in Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* January 1999.

[18] P. Panda, H. Nakamura, N. Dutt and A. Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Transactions on Computers,* February 1999.

[19] G. Rivera and C. Tseng. Tiling Optimizations for 3D Scientific Computations. In Proceedings of SC'00, Dallas, TX, November 2000.

[20] The SPARC Architectural Manual, Version 9. SPARC International Inc. Santa Clara, California.

[21] M. J. Wolfe. More Iteration Space Tiling. InProceedings of Supercomputing '89, Reno, NV, November 1989.