

**A Technique for Compilation to Exposed Memory
Hierarchy**

by

Benjamin Eliot Greenwald

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

© Massachusetts Institute of Technology 1999. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
September 17, 1999

Certified by
Saman Amarasinghe
Assistant Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Technique for Compilation to Exposed Memory Hierarchy

by

Benjamin Eliot Greenwald

Submitted to the Department of Electrical Engineering and Computer Science
on September 17, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

A tremendous amount of compiler research effort over the past ten years has been devoted to compensating for the deficiencies in hardware cache memory hierarchy. In this thesis, we propose a refocus of these energies towards compilation for memory hierarchy which is exposed to the compiler. With software exposure of memory hierarchy, not only can replacement policies be tailored to the application, but pollution can be minimized or eliminated, allowing programs to achieve memory reference locality with less local memory than would be needed by a cache. Additionally, prefetch latencies are fully exposed and can be hidden by the compiler, thereby improving overall performance.

We have developed a technique we call Compiler Controlled Hierarchical Reuse Management (or C-CHARM), implemented using the Stanford SUIF compiler system, which gathers compile time reuse information coupled with array dependence information, rewrites loops to use a truly minimal amount of local memory, and generates the necessary explicit data movement operations for the simple, software exposed hierarchical memory system of the M.I.T. RAW machine.

Thesis Supervisor: Saman Amarasinghe

Title: Assistant Professor

Acknowledgments

I have many people to thank for the support and encouragement that allowed me to complete this work.

I want to thank my advisors Anant Agarwal and Saman Amarasinghe for giving me the tools, the time, and that all important nudge in the right direction. Many thanks to all the members of the RAW project, and especially Matthew Frank and Andras Moritz who have acted as my sounding boards just when I needed you. In addition, without the constant support of Walter Lee, no results would have been presented here.

My family's undying love kept me afloat many times when I wasn't sure I would ever get this far. Even 3,000 miles away, I've never felt closer to you all or loved you more.

Most of all, I want to thank my loving wife Joy Elyse Greenwald. Without your love and support, this would never have been written. You are the reason that I smile.

Contents

1	Problem Statement	11
2	Background	17
2.1	An example program.	17
2.2	Iteration Space Graph	17
2.3	Memory Dependence Analysis	18
2.4	Reuse Analysis	19
2.5	Occupancy Vector Analysis	22
3	Our Algorithm	23
3.1	Precise Problem Definition	23
3.1.1	Reuse Analysis and Uniformly Generated Sets	24
3.1.2	Dependence Analysis	24
3.1.3	Find Leading and Trailing References	25
3.1.4	Occupancy Vector Analysis	25
3.1.5	Boundaries	27
3.1.6	Space Under the Occupancy Vector	28
3.1.7	The Difficulty of Mapping Between Iteration Space and Array Space	29
3.2	Program Transformations	31
3.2.1	Memory Hierarchy Interface	31
3.2.2	Rewrite Array Accesses	31
3.2.3	Generate Preload Loop	33
3.2.4	Prefetch and Writeback	34

3.2.5	Boundary Prefetches and Writebacks	36
3.2.6	Generate Postsave Loop	36
3.3	Optimizations	38
3.3.1	Eliminating the Modulus Operations	38
4	Results	41
4.1	Evaluation Environment	41
4.1.1	Benchmarks	41
4.1.2	The RAW Machine	41
4.1.3	RawCC	43
4.2	Experimental Results	43
4.2.1	Single Tile Performance	43
4.2.2	Multi-tile Performance	45
5	Related Work	49
5.1	Software Controlled Prefetching	49
5.2	Loop Transformation for Locality	50
5.3	Array Copying	50
6	Future Work and Conclusion	53

List of Figures

1-1	The CPU-Memory Performance Gap [4].	12
1-2	Memory hierarchy.	12
2-1	Our example program.	17
2-2	Iteration space graph and execution order of example.	18
2-3	A graphical representation of the dependence vector stencil for our example.	19
2-4	An iteration space vector.	20
2-5	Example vector array access functions.	20
3-1	Our example's dependence vectors.	25
3-2	The minimal OV which maximizes reuse for our example's dependence stencil.	27
3-3	The boundary array elements not accessed by the trailing reference.	27
3-4	Array reference without iteration space correspondence.	30
3-5	ISG for example without iteration space / data space correspondence.	30
3-6	The circular buffer in local memory.	31
3-7	Examples of remapping our example global accesses to local ones.	32
3-8	Our example program with remapped array accesses.	33
3-9	Example preload loop.	33
3-10	Our example loop with prefetch and writeback.	34
3-11	Our example loop with boundary operations.	35
3-12	Example postsave loop.	36
3-13	Final transformed example.	37

4-1	The RAW Microprocessor.	42
4-2	Execution Times on a Single Tile Normalized to RawCC.	44
4-3	Execution Speedup	46
4-4	Off Chip Pin Bandwidth Utilization	47

List of Tables

4.1	Comparison of Execution Times for Median Filter	48
-----	---	----

Chapter 1

Problem Statement

The performance gap between processors and main memory is a fundamental problem in high performance computing systems. According to Hennessy and Patterson [4], processor performance since 1986 has improved 55% per year while latency of memory accesses has improved only 7% per year. Figure 1-1 illustrates this nearly exponentially increasing performance disparity. The result is that with each new computing generation, the processor must effectively wait longer for each memory reference.

The solution to the processor-memory performance gap is memory hierarchy. Instead of having the processor access main memory directly, a series of intervening memories are placed between the processor and the main memory. As one moves from the main memory to the CPU, the memories get progressively smaller and faster (See Figure 1-2). By keeping as much of the data needed in the near future as close to the processor in the hierarchy as possible, the user program would receive most of the performance benefit of a small memory and the storage capability of a large one. Making memory hierarchy useful, however, requires understanding any underlying patterns in the memory accesses made by the program. Ascertaining these access patterns requires knowledge of the program's future memory behavior. The effect of these access patterns can be expressed as properties of the program in question, namely if the program data exhibits reuse and locality.

A memory stored value has the reuse property if it is used more than once in the course of the execution of the program. Locality is the property of programs that whatever data they access in the recent past, they are likely to access again in the near future. Obviously,

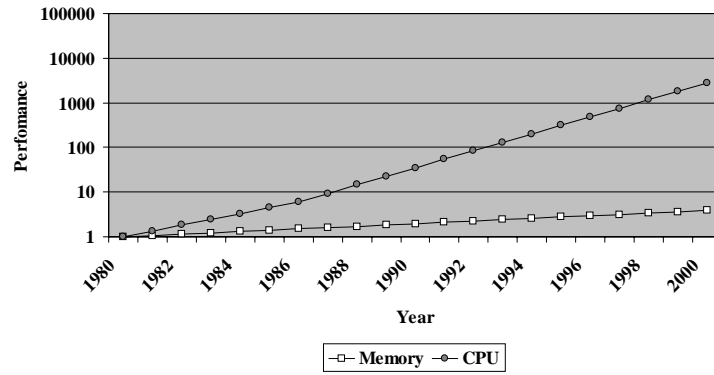


Figure 1-1: The CPU-Memory Performance Gap [4].

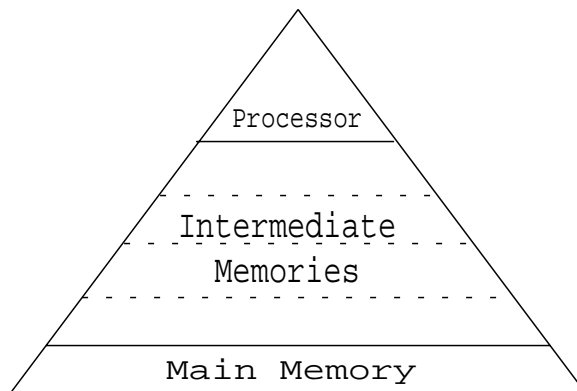


Figure 1-2: Memory hierarchy.

the set of all references to data which is reused in a program is a superset of all references which exhibit locality. A hardware based memory hierarchy management system built on the premise that all programs contain large amounts of locality would naturally perform well on such programs.

A *cache* is a hardware device which maps a subset of the main memory address space ¹ into a local memory. The cache receives address requests from the memory level above it. If the cache holds the requested address, the access is said to *hit* in the cache. Otherwise, the access is a *miss* and a lower level in the hierarchy must be consulted for the requested value. The size difference between local and main memory necessitates that each main memory address is mapped to a small set of possible local locations when it is in the current level of hierarchy. The *associativity* of the cache is the number of possible local locations for a particular main memory address to be located. The cache *replacement policy* is the method by which the cache chooses which item to evict to a lower level in the memory hierarchy in favor of another value.

Caches turn reuse into locality by attempting to hold onto a value requested by the level above for as long as possible. This is taking advantage of *temporal locality*, or the reuse of the same address over a short span of time. Caches may also fetch values nearby the requested address, assuming that if a value is used now, values near it in memory are likely to be used in the near future. This is called *spatial locality*.

Caches are dependent on a string of past access requests in order to choose the subset of main memory to map locally. Therefore, there is a period of misses at the beginning of the execution of the program when nothing is mapped locally. These *cold start misses* are essentially the result of the cache's inability to predict any access pattern as it has no past or future access information with which to predict. The many-to-few mapping which chooses the local location for a main memory address may map multiple addresses with locality to the same local location. Accesses to such data are said to be *conflict misses* because reuse is not being turned into locality due to the happenstance of the addresses where the values in conflict reside in main memory and the inflexibility of the cache's address mapping function. Finally, there may simply be more locality than can be held by the amount of

¹The *address space* is the set of all storage locations in a memory.

memory controlled by the cache. The cache must then evict a value back to a lower level in the hierarchy, ending any possible continued locality for the evicted value in favor of the assumed locality to be found in the most recently requested address. The next access to the evicted value would be known as a *capacity miss*.

Caches have been quite successful for a long time, but they are limited in their ability to turn reuse into locality in a variety of ways. The predefined many-to-few mapping used by the cache to choose the local location for a main memory address must necessarily be an arbitrary function because it is in hardware. If the input program does not happen to distribute its often reused addresses in the way the hardware is designed to take advantage of, conflict misses may severely degrade the memory performance of programs that exhibit large amounts of short term reuse that would otherwise be easily convertible to locality. Additionally, since the cache has no information about which piece of data will be needed the furthest in the future (and therefore the ideal item to evict from the cache), the replacement policy, in the best case, will evict the item used least recently to estimate this missing information. Often, this is precisely the data item needed next. Therefore, the cache has done exactly the opposite of what is optimal, even in situations where the memory access pattern is simple and well behaved. Further, with this lack of reuse information, the cache methodology of holding on to a value as long as possible can cause the cache to hold on to many data values which do not exhibit locality in order to gain locality for those data values which do. This artifact of the speculative nature of caches is known as *cache pollution*. The result of this pollution is that for many programs, a large portion of the memory at any one level in the hierarchy is wasted, thereby exacerbating the problem of capacity misses.

A great deal of work in compiler technology over the past ten years has been devoted to the problem of making up for these deficiencies in caches. Many domains of programs have compile-time analyzable reuse patterns. This includes the multimedia and other streaming applications which have become of great importance in recent years. Software controlled prefetching [9] uses this reuse analysis to push a data value up the hierarchy. By overriding the standard replacement policy, prefetching can minimize the performance impact of poor replacement choices while improving overall performance by bringing items into the cache before they are needed. By analyzing the boundaries of array variables, array copying [12]

restructures the data at run time to prevent the replacement policy from making ultimately bad decisions. Work in loop transformations to improve locality of references [10, 14] restructures the operations in loops to limit pollution and maximize the number of reuses of a data value before it is evicted. These loop transformations can be further optimized as in [5] to further minimize the negative impact of replacement policy on performance, but at a cost of some locality.

Now that we possess the knowledge and ability to better analyze and understand programs at compile time, we have gained the capacity to do far better than caches at converting reuse into locality. Up until this point, however, this information has been used to perform complex code and data transformations to second guess and circumvent the dynamic behavior of caches. We believe compiler technology has matured to a point where a reevaluation of the interaction between the compiler and the memory hierarchy is needed. Instead of developing methods to thwart the cache or reorganize the program such that some predicted cache behavior will produce the best memory performance, we believe it is time to remove the cache altogether.

We propose an architecture where the memory in the hierarchy is exposed to software and is directly addressable in a fine grained manner. Exposure of memory hierarchy is not an entirely new idea. Multiprocessor machines such as the Intel Paragon and the Cray T3D and T3E are global address space machines where data in other processors' memories must be explicitly moved to the local portion of the address space in order to gain performance. We propose going a step further to a machine with separate address spaces for main (or global) memory and hierarchy (or local) memory and a fine-grained, compiler controlled hardware interface for explicit movement of data objects between levels of the memory hierarchy.

Before we can remove the cache entirely, however, there is one thing that caches (or distributed global address spaces) give us which static analysis does not: a safety net. Caches may not always execute the program in question with the optimal memory performance, but at least they will guarantee correctness. A compiler which explicitly manages the hierarchy but cannot analyze the input program perfectly is in danger of executing the program incorrectly. In such an event, we must have some fall-back mechanism based on runtime

program analysis and dynamic memory hierarchy control [8].

We present here a first step towards a compiler for explicit static memory hierarchy management. Through a combination of dependence analysis, array analysis and static reuse analysis, we present a method for completely static memory hierarchy management of the reuse in the restricted domain of streaming and stream-like programs. Programs can be globally transformed to statically move data from main memory into local memory just in time for it to be accessed, then retired back to main memory when it is no longer needed, all with a minimal amount of software overhead.

In Chapter 2 we will discuss the background in program analysis necessary to understand our technique. The algorithm is presented in Chapter 3. Chapter 4 demonstrates and evaluates our technique. Chapter 5 discusses some related work. Chapter 6 will conclude.

Chapter 2

Background

2.1 An example program.

Before discussing the algorithm for compiler management of the memory hierarchy, there is a significant amount of background in compiler analysis which must be understood. Throughout this chapter and Chapter 3 we will be using the program in Figure 2-1 as an example. This program takes the average of five array values forming an X shape and writes the result back into the center of the X.

2.2 Iteration Space Graph

An Iteration Space Graph (or ISG) is a graphical representation of a FOR style nested loop. The axes of the loop are represented by the axes of the graph and are labeled with the corresponding loop index variable. Each node in this integer lattice represents a particular iteration of the loop nest and can be identified by its index vector $\vec{p} = (p_1, p_2, \dots, p_n)$

```
for(i = 1; i < 64; i++)
  for(j = 1; j < 64; j++)
    A[i][j] = (A[i][j] + A[i-1][j-1] + A[i-1][j+1] +
              A[i+1][j-1] + A[i+1][j+1])*0.2;
```

Figure 2-1: Our example program.

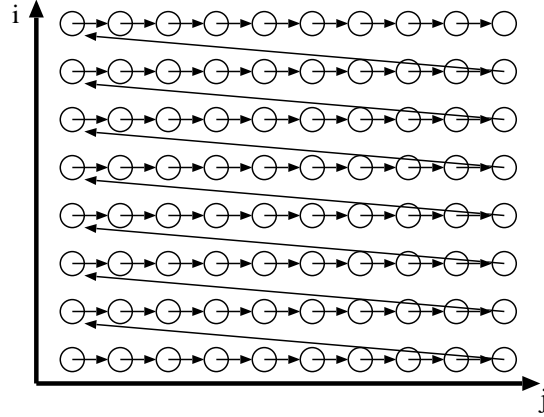


Figure 2-2: Iteration space graph and execution order of example.

where p_i is the value of the loop index for loop i in the loop nest, counting from outermost loop to innermost loop, when the iteration would be executed.

For our example in Figure 2-1, each node would be visited in turn along the horizontal j axis through a full row of the lattice before moving up to the next row along the i axis as in Figure 2-2.

2.3 Memory Dependence Analysis

A complete discussion of memory dependence analysis is beyond the scope of this thesis. The concept of iteration dependencies and the dependence vector [14], however, is critical to understanding our technique.

Given two iteration index vectors \vec{p} and \vec{q} , \vec{p} is said to be *lexicographically greater than* \vec{q} , written $\vec{p} \succ \vec{q}$ if and only if $p_1 > q_1$ or both $p_1 = q_1$ and $(p_2, \dots, p_n) \succ (q_2, \dots, q_n)$ [14]¹.

Given iterations \vec{p} and \vec{q} which both access the same memory location, if $\vec{p} \succ \vec{q}$ there is said to be a *data dependence* between the two iterations. If the data dependence represents a write at iteration \vec{q} which is read by iteration \vec{p} , it is called a true data dependence. Otherwise, it is called a storage related dependence because the dependence only exists due to the reuse of the memory location, not the value stored at that location.

A *dependence vector* is a vector in the iteration space graph of a loop which represents

¹This definition of lexicographical ordering assumes a positive step for the loops in question. If the step is negative, the comparison of p and q along those loop axes should be reversed.

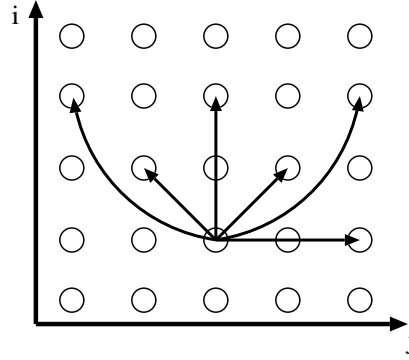


Figure 2-3: A graphical representation of the dependence vector stencil for our example.

a data dependence. In particular, if there is a data dependence between iteration \vec{p} and \vec{q} as above, the dependence vector $\vec{d} = \vec{p} - \vec{q}$. The iteration represented by the node at the tail of the vector must execute before the iteration represented by the node at the head of the vector in order to maintain program correctness.

An alternate, and in the context of this thesis more appropriate, view of a dependence is that the iteration at the head of the dependence vector will reuse a storage location used by the iteration at the tail of the vector. If we include all dependences, including the often overlooked ones where two iterations may read the same memory value² we can get a clear picture of how memory stored values are passed between iterations.

For most regular loops, the dependencies flowing out of a particular iteration form a fixed pattern that repeats throughout the iteration space. This *stencil* of dependencies represents the regular and predictable interaction between any given iteration and the iterations which depend on it. See figure 2-3 for our example’s dependence vector stencil.

2.4 Reuse Analysis

Array reuse analysis [9, 14] is a means of statically determining how and when array elements will be reused in a loop nest. The basis of reuse analysis is conversion of affine array accesses³ into vector expressions. By solving for properties of these vector expressions,

²Dependence analysis does not usually talk about these “Read After Read” dependences because they do not inhibit the reordering of the loop iterations in any way.

³An *affine array access* is one which is a linear function of the loop index variables, plus some constant.

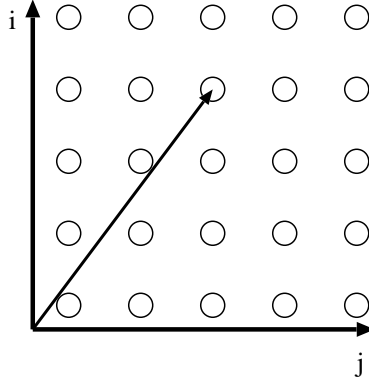


Figure 2-4: An iteration space vector.

$$\begin{aligned}
 A[i][j] &= A \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \\
 A[i][0] &= A \left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \\
 A[2*i][j-1] &= A \left(\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right)
 \end{aligned}$$

Figure 2-5: Example vector array access functions.

we can gain understanding about the reuse patterns for the corresponding array access.

An *iteration space vector* \vec{i} is a vector in iteration space which points from the beginning of the loop at the iteration space origin to a node in the ISG. It has dimensionality n equal to the the number of axes in the loop and its elements consist of the values of the loop index variables, in order of nesting, at the node at the head of the vector. See figure 2-4 for an example.

An affine array access can be thought of as a function $\vec{f}(\vec{i})$ which maps the n dimensions of the loop into the d dimensions of the array being accessed. Any affine array access function can be written as

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{c}$$

where H is a $d \times n$ linear transformation matrix formed from the coefficients in the index expressions, \vec{i} is again an iteration space vector, and \vec{c} is a vector of d dimensionality consisting of the constants. See figure 2-5 for some examples.

In cache parlance, a reuse is associated with a dynamic instance of a load. This dynamic load is said to exhibit locality in one of two ways:

- *temporal* : A dynamic load uses a location recently used by some other dynamic load. These loads may or may not be instances of the same static instruction.
- *spatial* : A load uses a location very near another recently used location.

Reuse analysis defines four types of reuse and gives information about which category each array reference is in. Unlike when one is talking about caches, these classifications apply to the static reference, not just some dynamic instance. The four types of reuse are

- *self-temporal* : The static array reference reuses the same location in a successive loop iteration.
- *self-spatial* : The static array reference accesses successive locations in successive loop iterations.
- *group-temporal* : A set of static array references all reuse the same location.
- *group-spatial* : A set of static array references access successive locations.

A full discussion of self reuse can be found in [14]. Only group reuse is of significance to the technique presented here.

Both forms of group reuse can only be exploited for a set of references which are uniformly generated [3]. A *uniformly generated set* of references are a group of references for which only the constant terms in the access expressions differ. In reuse analysis terms, two array accesses are in the same uniformly generated set and therefore exhibit exploitable group reuse if and only if

$$\exists \vec{r} : H\vec{r} = \vec{c}_1 - \vec{c}_2$$

For example, given the references $A[i][j]$ and $A[i-1][j-1]$ from our example program,

$$A[i][j] = A \left(\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \right), \text{ and}$$

$$A[i-1][j-1] = A \left(\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right) \right).$$

These references belong to the same uniformly generated set if they access the same array and there exists some integer solution (i_r, j_r) to

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_r \\ j_r \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Since the vector $(1, 1)$ is such a solution, these references are in the same uniformly generated group reuse set. In fact, all six array accesses in our example program belong to this same set.

Since, by definition, each element of a group reuse set accesses the same array elements in turn, there must be some member of the set which accesses each data object first. This access is known as the *leading reference*. Its dual is the *trailing reference* or the last access in the set to access each particular array element.

2.5 Occupancy Vector Analysis

Once all the iterations dependent on a value initially accessed or produced in a given iteration \vec{q} have executed, there is no need to continue storing said value in local memory as it will not be accessed again for the remainder of the loop. An iteration \vec{p} occurring at or after this point in the program can therefore use the same local storage. If we were to overlap the storage for these two iterations, their vector difference $O\vec{V} = \vec{p} - \vec{q}$ is called the *occupancy vector*. Occupancy vector analysis [11] can be used to separate iterations into storage equivalence classes. Members of the same storage equivalence class are iterations separated by the occupancy vector and their storage can be overlapped in memory by introducing a storage related dependence based on $O\vec{V}$.

Chapter 3

Our Algorithm

In this chapter, we will present our algorithm for transforming group reuse programs to do explicit management of the memory hierarchy.

3.1 Precise Problem Definition

Explicit management of memory hierarchy means that the compiler controls all aspects of data movement between local and global memory.

Envision a machine with the following properties:

1. A large global memory connected to the processor. Latency of accesses to this memory are considerably longer than to any processor on-chip resource.
2. A small local memory. This memory is insufficient in size to hold the data for any reasonably sized program. It has an address space completely independent of the global memory address space.
3. A set of architected primitive memory hierarchy control instructions. These primitives would allow the compiler to explicitly copy a data item from the global memory address space to an arbitrary local memory location, and vice versa.

Our problem consists of finding a methodology by which the compiler can understand the input program well enough to transform it for execution on just such a machine. Simultaneously, the compiler must choose a tailored replacement policy for the program which

maximizes memory locality. The compiler must decide when a value must be moved to local memory, inserting the necessary instructions to accomplish this. It must then decide how long it can afford to hold that value in local memory in order to maximize locality. When the value can be held locally no longer and the value has changed since it was brought to the local address space, the compiler generates instructions to copy the value back to the global memory.

3.1.1 Reuse Analysis and Uniformly Generated Sets

The array references in the loop must be broken down into uniformly generated sets, each of which will have group reuse. Recall from Section 2.4 a *uniformly generated set* is a set of array references which differ only in the constant parts of the access expressions.

The abstract syntax tree for the body of the loop nest is traversed and all array references are noted. They are then sorted by which array they reference. The linear transformation matrix H and the constants vector \vec{c} are then calculated for the vector form of each array access. The accesses are further classified by equality of their H matrices. A solution for the uniformly generated set qualification, namely

$$\exists \vec{r} : H\vec{r} = \vec{c}_1 - \vec{c}_2$$

is then sought for each pairing of elements in the H equality sets, segregating the accesses into the final uniformly generated sets. As the reuse in these sets is the only kind this technique considers, we refer to these sets at this point in the algorithm as *reuse groups*. Recall from Section 2.4 that our example program contains a single reuse group consisting of all six accesses to the array A.

3.1.2 Dependence Analysis

Reuse analysis gives us the reuse groups, but it gives us little information about the memory behavior within the group and how the members interact. Dependence analysis gives us this information.

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \end{bmatrix}$$

Figure 3-1: Our example's dependence vectors.

For each reuse group, we must find the set of all dependences. The elements of each reuse group are dependence tested in a pairwise fashion and the results from all the tests are collected. Any duplicates are removed. The result is the so-called stencil of the dependences for each iteration of the loop.

For our example program, this set of dependence vectors can be seen graphically in Figure 2-3 or in Figure 3-1 in column vector notation.

3.1.3 Find Leading and Trailing References

In order to do fetches and replacements to the correct locations, we must figure out which members of the reuse group correspond to the leading and trailing reference. By definition, a reuse group forms a uniformly generated set of references. Therefore, the group members differ only in the constant portion \vec{c} of the access function $\vec{f}(\vec{i})$. The reference which will access array elements first is the one whose \vec{c} vector is last in lexicographic order. Likewise, the trailing reference is the one whose \vec{c} vector is first in lexicographic order. Lexicographic ordering is discussed in Section 2.3.

By sorting the \vec{c} vectors for each reuse group into lexicographic order, we have found the leading and trailing references. In our example program in Figure 2-1 the trailing reference is $A[i-1][j-1]$ and the leading reference in $A[i+1][j+1]$.

3.1.4 Occupancy Vector Analysis

For each iteration of the loop, there is some minimum number of array elements that must be in local memory to correctly execute the loop body. It is clear that this number is equal to the number of independent array references in the loop body; every separate location accessed by the loop body must be brought into local memory before the loop can be executed. For our example, this would be five locations. While choosing such a mapping

would result in a correctly executed program, this storage mapping would not take advantage of group reuse. Every locally allocated location would be reloaded every iteration. Therefore, we want to choose a storage mapping which converts the maximum amount of reuse into locality while at the same time using as little local memory as possible.

We want to create a set of storage equivalence classes for local memory. We use the term *storage equivalence class* to mean a set of iterations which can have their local storage overlapped. A storage dependence based on an occupancy vector over the entire reuse group is precisely what we need. We choose an OV which indicates an eviction for the element brought in at the tail of the OV exactly when that element will no longer be needed for the remainder of the loop. This maximizes the reuse of that element while guaranteeing that local memory will never be polluted.

We can look to the stencil of dependencies for the reuse group in order to ascertain on which iteration \vec{p} a value produced on iteration \vec{q} is no longer needed by the loop. If we think of dependences as representing communication of a value between iterations, a value produced on iteration \vec{q} is dead once it has been communicated to all the iterations which will receive it. This is a generally complicated relationship between iterations which is vastly simplified by two assumptions of our algorithm:

1. The dependencies form a repeated, regular stencil.
2. No loop transformations are performed. The loop executes in its original iteration order.

By the first assumption, we know that whichever iteration \vec{p} is, the vector difference $\vec{p} - \vec{q}$ will be a constant over the entire execution of the loop. By the second assumption, we know the dependencies in the stencil will be satisfied in lexicographic order because the iterations at the other end of the dependencies will be executed in lexicographic order. Therefore, communication of \vec{q} 's value will be complete immediately after the execution of the iteration which satisfies the last lexicographically ordered stencil dependence.

Consequently, we choose the shortest vector which is longer than the longest dependence in each reuse group to act as the minimal occupancy vector. This longest dependence corresponds to the dependence between the trailing and leading references in the set. The

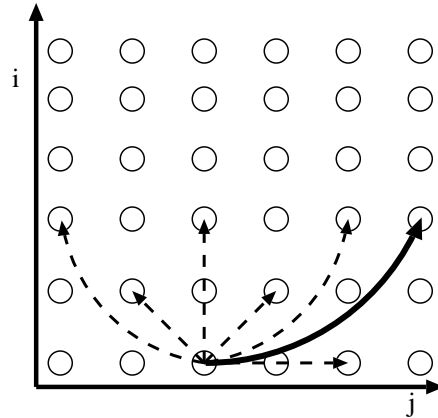


Figure 3-2: The minimal OV which maximizes reuse for our example's dependence stencil.

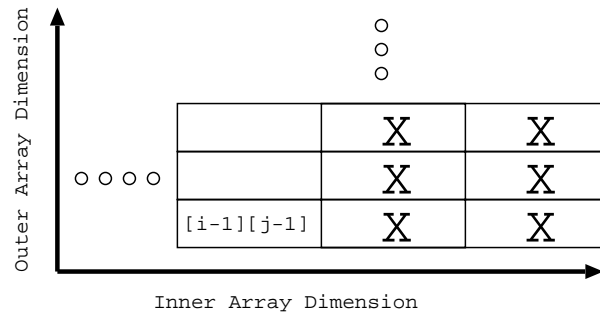


Figure 3-3: The boundary array elements not accessed by the trailing reference.

chosen vector for our example is pictorially demonstrated in Figure 3-2. The solid vector is the occupancy vector and it implies that the iterations at the head and tail are local storage equivalent. As one can see, all possible dependences and reuses will be incorporated into the space under this occupancy vector.

Since the longest dependence for the example in Figure 2-1 is the vector $(2, 2)$, the minimal occupancy vector is $(2, 3)$.

3.1.5 Boundaries

The definition of group reuse implies that every element of the group accesses the same memory locations in turn. This is not precisely accurate. If we again consider our example program from Figure 2-1, at the end of an iteration of the j loop the trailing reference will be accessing the array element shown in Figure 3-3. Therefore, by virtue of the reference's position in the reuse group, the elements marked with X's will be accessed by some member

of the group, but never by the trailing reference. This is generally true of any element of any reuse group along the loop boundaries. Therefore, we must make sure that the entire rectilinear space encompassing all array elements accessed by the group is prefetched and written back to main memory correctly. One can observe that the rectilinear space in question is the space which has the longest dependence in the group mapped from iteration space into array data space as its corresponding longest diagonal. We will refer to this boundary vector as $B\vec{V}$.

The boundary vector in our example is $(2, 2)$.

3.1.6 Space Under the Occupancy Vector

Allocating enough local storage for each reuse group is equivalent to the problem of finding the amount of space required by all iterations under the group's occupancy vector. The *iterations under the occupancy vector* are all iterations executed by the loop between the iteration at the tail of the vector and the iteration at the head.

Given that we are considering regular, FOR style loop nests, the access pattern of any particular member of a reuse group must be strictly monotonic. This means that each reference accesses a different array element each iteration. By the definition of group reuse, ignoring boundary conditions for now, all members of the reuse group are accessing array elements already accessed by the leading reference. Therefore, the group as a whole accesses one new array element on each iteration of the loop and that value is accessed at the leading reference. Likewise, the trailing reference accesses a different element each iteration and that element, by the properties of group reuse, is being accessed for the last time in the loop. Therefore, a reuse group also retires one array element each iteration and that element is retired at the trailing reference. If a reuse group must fetch a new value each iteration and retires a value each iteration, then the space required to map the group is a constant equal to the number of iterations under the OV times the size of each array element.

To compute the number of iterations executed under any iteration space vector, we introduce the concept of the dimension vector. The *dimension vector* of a loop is a vector

of dimensionality n where n is the depth of the loop, and whose elements consist of the number of times the body is executed for each execution of the respective loop. More precisely, if we number the loops from outer most to inner most and we use L_i to indicate the lower bound of loop i and U_i to indicate the upper bound of loop i , the dimension vector is

$$\vec{DV} = ((dv_0, \dots, dv_n) | dv_i = U_i - L_i + BV_{i+1})$$

Note that we have added in elements of the boundary vector \vec{BV} discussed in Section 3.1.5, offset by one loop axis, to compensate for the boundary elements not accessed by the trailing reference and thereby rectilinearizing the space. Once \vec{DV} has been computed for the loop, finding the number of iterations under an iteration space vector is trivial. The number of iterations I under a vector \vec{p} is

$$I = \vec{p} \cdot \vec{DV}$$

where “ \cdot ” represents standard Cartesian dot product.

For our example program, $\vec{DV} = (65, 1)$.

3.1.7 The Difficulty of Mapping Between Iteration Space and Array Space

The above algorithm assumes a one-to-one correspondence between points in iteration space and array elements in each reuse group. This assumption makes mapping iterations to array elements trivial. The mapping function is the original array access function before it is rewritten by the algorithm. This assumption is also the basis for finding the space under the occupancy vector in Section 3.1.6.

One is still safe as long as there is a one-to-one correspondence between the iterations of some innermost subset of the loops in the nest and the array elements. This represents a self reuse which occurs along the boundary between the innermost subset and the rest of the loop.

If the mapping is not one-to-one for any loop subset, however, deciding what data is

```

for(i = 0; i < 64; i++)
  for(j = 0; j < 64; j++)
    ... A[i+j] ...

```

Figure 3-4: Array reference without iteration space correspondence.

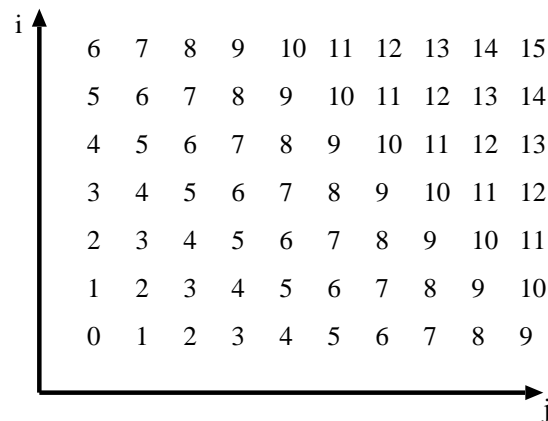


Figure 3-5: ISG for example without iteration space / data space correspondence.

needed when and how much space to reserve can become extremely complicated. The program in Figure 3-4 is represented in Figure 3-5 as an ISG where the iteration points are marked with the element of the array A accessed on that iteration. As one can see, the reference exhibits self reuse, but it reuses a sliding subarray of A ; the set of elements reaccessed by the reference in Figure 3-4 changes in every iteration of every loop in the nest. To correctly manage this reuse, the iteration space would need to be projected along the direction of the reuse, the vector $(-1, 1)$. Generating a mapping function from iteration space to data space, therefore, requires solving a complex linear algebra problem to define the space over which the one-to-one correspondence does exist, and then understanding the original array accesses in terms of this new vector space. Solving this problem is left for future work.

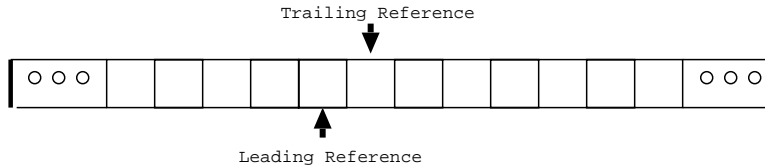


Figure 3-6: The circular buffer in local memory.

3.2 Program Transformations

3.2.1 Memory Hierarchy Interface

Before discussing how the compiler goes about transforming our example program to explicitly manage the memory hierarchy, an interface to the memory hierarchy must be outlined. We assume there are only two levels in the hierarchy, global memory and local memory. The primitive operations for moving a value in the hierarchy are

- $laddr = \text{CopyFromGlobal}(gaddr)$: `CopyFromGlobal` assigns the value currently stored in the global memory location $gaddr$ to the local address $laddr$.
- $\text{CopyToGlobal}(gaddr, value)$: `CopyToGlobal` assigns $value$ to the global memory location $gaddr$.

We assume these are represented as function calls which are later converted to the corresponding architectural primitives in the backend. We also assume some cooperation between the compiler and the assembly level linker to map the local arrays into the local memory address space.

3.2.2 Rewrite Array Accesses

All array accesses must be rewritten to access the local memory space allocated for each reuse group. The local memory is managed as a circular buffer (see Figure 3-6) where on each iteration the value at the trailing reference is replaced with the next value to be read by the leading reference. This complicates the array accesses because it requires a modulus operation to be performed with a base equal to the space requirement of the group. It is this modulus operation that introduces the storage related dependence which implements

$$\begin{aligned}
A[i][j] &\rightarrow A_l[(65 * i + j) \% 133] \\
A[i - 1][j - 1] &\rightarrow A_l[(65 * (i - 1) + (j - 1)) \% 133] \\
A[i + 1][j + 1] &\rightarrow A_l[(65 * (i + 1) + (j + 1)) \% 133]
\end{aligned}$$

Figure 3-7: Examples of remapping our example global accesses to local ones.

the storage equivalence classes. Two addresses separated by the storage requirement of the group will be mapped to the same location in local memory by the mod. In Section 3.3.1 we discuss a method for removing many of these mods in order to make the local array accesses affine.

Consider the standard method by which a multidimensional array access generates an address in one dimensional memory space. The array access function $\vec{f}(\vec{i})$ can be seen as the dot product of the iteration space vector \vec{i} and a vector \vec{s} which is a function of the sizes of each array dimension. If the array has dimensionality d , each array dimension size is denoted by the constant s_j , and the array is accessed in row major order, then

$$\vec{s} = ((s_2, s_3, \dots, s_d), (s_3, s_4, \dots, s_d), \dots, (s_d), 1)$$

numbering array dimensions from 1. The address a for an access to array A with access function $\vec{f}(\vec{i})$ on iteration \vec{j} is therefore

$$a = A + \vec{j} \cdot \vec{s}.$$

Linearizing the accesses into the locally allocated storage, therefore, is simply a matter of choosing a new \vec{s} . Given that this storage is allocated and laid out based on the loop dimension vector $D\vec{V}$ discussed in Section 3.1.6, it is clear that this vector is the ideal choice to replace \vec{s} in the above calculation. Therefore, the local linearized address a_l into a given reuse group's local memory allocation A_l on iteration \vec{j} is

$$a_l = A_l + \vec{j} \cdot D\vec{V}.$$

To remap the global accesses to local ones, arrays are allocated in local memory for each group and the array access base addresses are changed to be that of the local storage.


```

for(i = 1; i < 64; i++)
  for(j = 1; j < 64; j++)
    A_local[(65*i + j) % 133] =
      (A_local[(65*i + j) % 133] +
       A_local[(65*(i-1) + (j-1)) % 133] +
       A_local[(65*(i-1) + (j+1)) % 133] +
       A_local[(65*(i+1) + (j-1)) % 133] +
       A_local[(65*(i+1) + (j+1)) % 133])*0.2;

```

Figure 3-8: Our example program with remapped array accesses.

```

for(temp0 = 0; temp0 < 133; temp0++)
  A_local[(65*(i-1) + (j-1)) + temp0 % 133] =
    CopyFromGlobal(&A[i-1][j-1] + temp0 * sizeof(A[0][0]));

```

Figure 3-9: Example preload loop.

Then the above transformation is applied to all the array accesses in each reuse group in the loop. The modulus operations with base equal to the space allocated to the group are added to the access functions in the new array accesses, completing the remap. See Figure 3-7 for some examples from our sample program and Figure 3-8 for the text of our example loop after remapping has been done.

3.2.3 Generate Preload Loop

Before execution of the loop originally in question can begin, the values under the occupancy vector for each group starting from the origin must be loaded into local memory. To do so, a preload loop is generated just before the original loop, which preloads the required number of elements (call this number *SR* for space requirement) under the occupancy vector. If we assume row major order, the elements of the global array in question are simply the first *SR* elements after and including the trailing reference.

The preload loop body consists of a request for the global address of the trailing reference, plus the current offset into the circular buffer, and writes that value into the local trailing reference location, plus the offset, in the local memory. Figure 3-9 demonstrates

```

for(i = 1; i < 64; i++)
  for(j = 1; j < 64; j++)
  {
    A_local[(65*i + j) % 133] =
      (A_local[(65*i + j) % 133] +
       A_local[(65*(i-1) + (j-1)) % 133] +
       A_local[(65*(i-1) + (j+1)) % 133] +
       A_local[(65*(i+1) + (j-1)) % 133] +
       A_local[(65*(i+1) + (j+1)) % 133])*0.2;
    CopyToGlobal(&A[i-1][j-1],
                 A_local[(65*(i-1) + (j-1)) % 133]);
    A_local[(65*(i-1) + (j-1)) % 133] =
      CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
  }

```

Figure 3-10: Our example loop with prefetch and writeback.

the preload loop.

3.2.4 Prefetch and Writeback

During the execution of the actual loop body, we must prefetch the value needed by the next iteration and possibly commit a no longer needed local value back to global memory. The value to be written back is obviously the value at the local trailing reference location at the end of each iteration. One might be tempted to say that the address to be prefetched is the global leading reference plus one, but this is incorrect if we choose an occupancy vector which is not minimal. One might choose an occupancy vector which is not space minimal if some other property of the program needed optimizing. Such a situation is discussed in Section 3.3.1. Therefore, we instead prefetch the global address corresponding to the trailing reference plus the occupancy vector. The prefetched value overwrites the no longer needed value at the local trailing reference. The transformed example program body, including prefetches and writebacks, can be seen in Figure 3-10.

```

for(i = 1; i < 64; i++)
{
    for(j = 1; j < 64; j++)
    {
        A_local[(65*i + j) % 133] =
            (A_local[(65*i + j) % 133] +
             A_local[(65*(i-1) + (j-1)) % 133] +
             A_local[(65*(i-1) + (j+1)) % 133] +
             A_local[(65*(i+1) + (j-1)) % 133] +
             A_local[(65*(i+1) + (j+1)) % 133])*0.2;
        CopyToGlobal(&A[i-1][j-1],
                    A_local[(65*(i-1) + (j-1)) % 133]);
        A_local[(65*(i-1) + (j-1)) % 133] =
            CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
    }
    CopyToGlobal((&A[i-1][j-1]) + (1 * sizeof(A[0][0]),
                    A_local[(65*(i-1) + (j-1)) + 1 % 133]);
    CopyToGlobal((&A[i-1][j-1]) + (2 * sizeof(A[0][0]),
                    A_local[(65*(i-1) + (j-1)) + 2 % 133]);
    A_local[(65*(i-1) + (j-1)) + 1 % 133] =
        CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
    A_local[(65*(i-1) + (j-1)) + 1% 133] =
        CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
}

```

Figure 3-11: Our example loop with boundary operations.

```

for(temp1 = 0; temp1 < 133; temp1++)
    CopyToGlobal(&A[i-1][j-1] + temp1 * sizeof(A[0][0]),
                A_local[(65*(i-1) + (j-1)) + temp1 % 133]);

```

Figure 3-12: Example postsave loop.

3.2.5 Boundary Prefetches and Writebacks

As discussed in Section 3.1.5, the trailing reference, and for that matter the trailing reference plus $\vec{O}V$, does not access all the elements needed by the group. Therefore along the loop boundaries in the nest, additional prefetches must be added to guarantee the entire rectilinear space under the $\vec{O}V$ is in memory before the next iteration is executed. These prefetches are inserted at the end of the respective loop in the nest and the number of prefetches needed is equal to BV_{i+1} where i is the current nesting depth¹. If the group contains a write, these boundary elements must also be written back at the same time.

In our example, the second element of $\vec{B}V$ is 2, indicating we must do two additional prefetches and writebacks at the end of the i loop before returning to the next execution of the j loop. These correspond to the elements marked with the X's in Figure 3-3. These additional prefetches and writebacks can be seen in Figure 3-11.

3.2.6 Generate Postsave Loop

The final code transformation that must be made is the dual to the preload loop. If the group contains a write, the last SR data objects must be written back to global memory. This postsave loop is constructed precisely the same way as the preload loop. Our example postsave loop can be seen in Figure 3-12.

The completely transformed program appears in Figure 3-13.

¹No prefetches need be done after the loop has finished executing, therefore BV_{i+1} is always a valid member of the boundary vector.

```

for(temp0 = 0; temp0 < 133; temp0++)
    A_local[(65*(i-1) + (j-1)) + temp0 % 133] =
        CopyFromGlobal(&A[i-1][j-1] + temp0 * sizeof(A[0][0]));

for(i = 1; i < 64; i++)
{
    for(j = 1; j < 64; j++)
    {
        A_local[(65*i + j) % 133] =
            (A_local[(65*i + j) % 133] +
             A_local[(65*(i-1) + (j-1)) % 133] +
             A_local[(65*(i-1) + (j+1)) % 133] +
             A_local[(65*(i+1) + (j-1)) % 133] +
             A_local[(65*(i+1) + (j+1)) % 133])*0.2;
        CopyToGlobal(&A[i-1][j-1],
                    A_local[(65*(i-1) + (j-1)) % 133]);
        A_local[(65*(i-1) + (j-1)) % 133] =
            CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
    }
    CopyToGlobal((&A[i-1][j-1]) + (1 * sizeof(A[0][0]),
                A_local[(65*(i-1) + (j-1)) + 1 % 133]);
    CopyToGlobal((&A[i-1][j-1]) + (2 * sizeof(A[0][0]),
                A_local[(65*(i-1) + (j-1)) + 2 % 133]);
    A_local[(65*(i-1) + (j-1)) + 1 % 133] =
        CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
    A_local[(65*(i-1) + (j-1)) + 1% 133] =
        CopyFromGlobal(&A[(i-1) + 2][(j-1) + 3]);
}

for(temp1 = 0; temp1 < 133; temp1++)
    CopyToGlobal(&A[i-1][j-1] + temp1 * sizeof(A[0][0]),
                A_local[(65*(i-1) + (j-1)) + temp1 % 133]);

```

Figure 3-13: Final transformed example.

3.3 Optimizations

3.3.1 Eliminating the Modulus Operations

Removing the integer mod operations from the local array accesses is desirable primarily for two reasons:

1. Integer mod operations are often expensive, vastly increasing the cost of the address computation.
2. Array accesses containing mods are not affine. Other array-based optimizations in the compiler most likely cannot analyze non-affine array accesses.

To remove the mods, we make heavy use of the optimizations for mods presented in [1]. The optimizations presented use a combination of number theory and loop transformations to remove integer division and modulus operations. The number theory axioms already understood by the above system were limited in their understanding of expressions with multiple independent variables, so we added the following.

Given an integer expression of the form $(ai + f())\%b$, where a and b are integers and $f()$ is some arbitrary valued function

$$(ai + f())\%b \equiv (a(i + x))\%b + f()$$

if and only if $a \setminus b$ and $ax \leq f() < a(x + 1)$ for some integer value x .

Simply put, variables whose coefficient divides the base of the mod can be factored out of the mod expression if the above criteria are met. This allows the system in [1] to simplify the expression into a form which its other axioms can operate on.

The a in the above corresponds to the element of the dimension vector for the loop whose index is i . We cannot change any of the elements of the dimension vector as $\vec{D}V$ is a property of the loop². On the other hand, b corresponds to the space requirement SR of the reuse group in question. Since SR is a function of the occupancy vector, and we have control over the choice of $\vec{O}V$, b can be arbitrarily increased. The magical value for SR is

²One could transform the loop, but we do not consider that option at this time.

the smallest multiple of the least common multiple of the elements of $D\vec{V}$ which is larger than the minimum value of SR required by the dependences in the group. In our example, the LCM of the elements of $D\vec{V} = (65, 1)$ is, of course, 65. The smallest multiple of 65 greater than the minimum value of SR , which is 133, is 195. Therefore, if we were to lengthen the $O\vec{V}$ in our example from $(2, 3)$ to $(3, 0)$, we would consume 62 additional elements of local memory but the mods would become candidates for removal.

Chapter 4

Results

4.1 Evaluation Environment

Before discussing any experimental results, we need to introduce the environment we are using for evaluation.

4.1.1 Benchmarks

The benchmarks we have used to evaluate our technique are simple kernels with clear group temporal reuse patterns.

Convolution: A sixteen tap convolution.

Jacobi: Jacobi relaxation.

Median Filter: Filter which finds the median of nine nearby points.

SOR: A five point stencil relaxation.

4.1.2 The RAW Machine

Our target is the RAW microprocessor [13]. Figure 4-1 depicts the features and layout of the RAW architecture. The basic computational unit of a RAW processor is called a tile. Each tile consists of a simple RISC pipeline, local instruction and data memories,

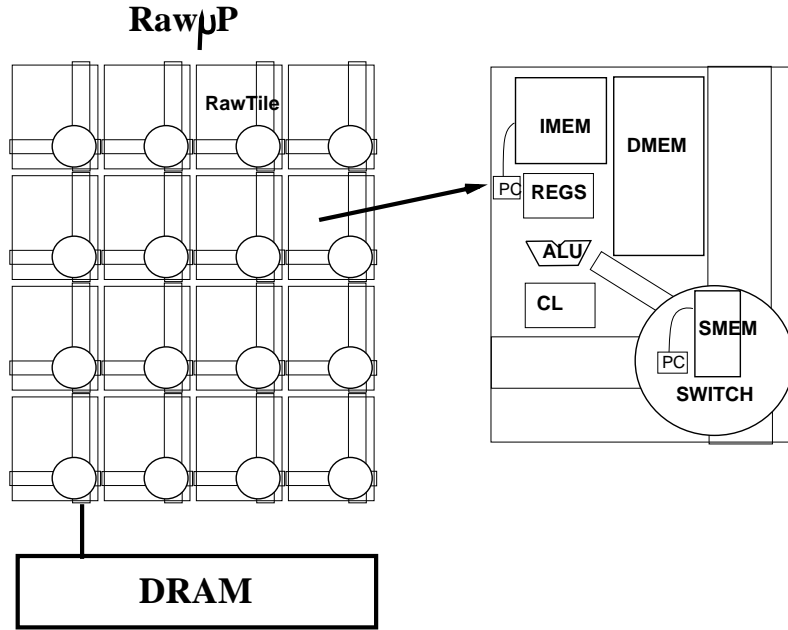


Figure 4-1: The RAW Microprocessor.

and a compiler controlled programmable routing switch. The memories are completely independent and can only be accessed by the tile on which they reside; the hardware does not provide a global view of the on chip memory. The chip consists of some number of these tiles arranged in a two-dimensional mesh, each tile being connected to each of its four nearest neighbors. The switches in the mesh form a compiler controlled, statically scheduled network. For communication which cannot be fully analyzed at compile time, or for communication patterns better suited to dynamic communication, a dimension ordered, dynamic wormhole routed network also interconnects the tiles.

The programmable switches allow for high throughput, low latency communication between the tiles if the compiler can deduce all information about the communication and can statically schedule it on this static network of switches. Connections to off-chip resources, such as global memory, are also made through the static network. Main memory control protocol messages are routed off the chip periphery and the responses routed back on chip by a switch along the chip boundary. This organization completely exposes control of any external main memory to the compiler. It is the combination of this exposure of main memory and the separately addressable local data memories on each tile that makes RAW an excellent candidate to evaluate our technique.

The programmable switches also provide the interface to the off chip global memory. Off chip pins are connected directly to the static network wires running off the edge of the chip. In this way, the switch can route to any off chip device just as if it were routing to another tile.

The global memory we modeled is a fully pipelined DRAM with a latency of three processor cycles. This model is consistent with the DRAMs we expect to use in our RAW machine prototype. Currently, we are only modeling a single off chip DRAM. In the future, we plan to model multiple DRAMs and study the affects of global data distribution in conjunction with our technique.

The results presented here were taken from execution of our benchmarks on a simulator of the RAW machine.

4.1.3 RawCC

Our standard compiler for the RAW architecture is known as RawCC [2, 7]. RawCC is an automatically parallelizing compiler, implemented with the Stanford SUIF compiler system, which takes sequential C or Fortran code as input, analyzes it for parallelism and static communication, and maps and schedules the resulting program for a RAW microprocessor. The parallelism sought by RawCC is instruction level. Loops are unrolled to generate large basic blocks and then each basic block is scheduled across the tiles of the RAW machine. However, RawCC makes an important simplifying assumption, namely that the RAW tiles have infinitely large memories. By using RawCC to parallelize and schedule the output of our compiler, as we have in our experiments, we gain the more realistic model of local memory from our technique and the power of an autoparallelizing compiler.

4.2 Experimental Results

4.2.1 Single Tile Performance

Figure 4-2 shows the execution time of our benchmarks, normalized to the execution time using RawCC alone, running on a single RAW tile. This graph demonstrates the cost of

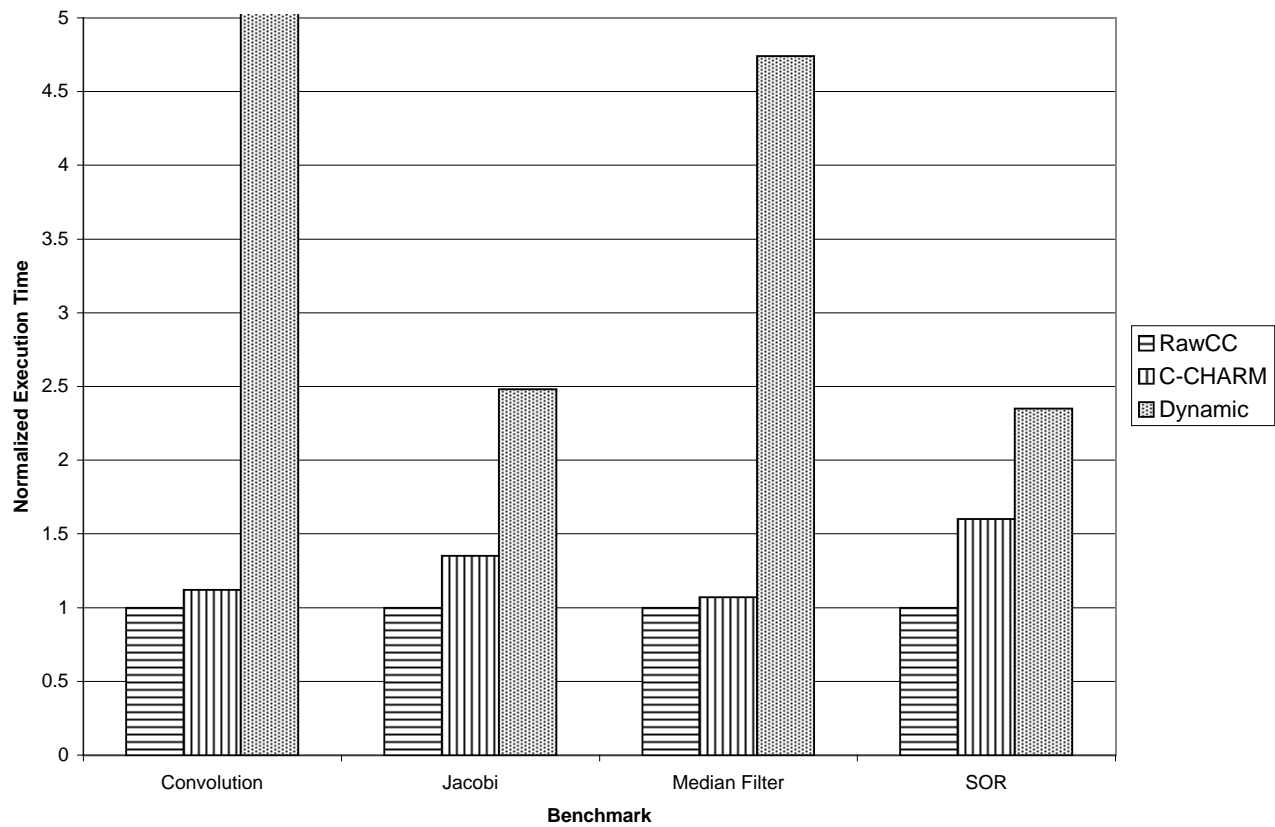


Figure 4-2: Execution Times on a Single Tile Normalized to RawCC.

our technique relative to the infinite memory case and versus a dynamic, software based, direct-mapped caching scheme [8]. Due to the static mapping we have done, our software overhead is significantly lower than that of the dynamic scheme. With the addition of our tailored replacement policy, our scheme always does better than the dynamic one on our benchmarks.

The cost of having accesses to a memory which is off chip are still significant when we compare our technique to one which assumes infinite local memory. The overhead is quite respectable in comparison to this unrealistic model, ranging from 7% for median filter, up to 60% for SOR.

4.2.2 Multi-tile Performance

We analyzed the scalability of our benchmarks in an attempt to better understand the source of the overhead. Figure 4-3 shows the speedup of each benchmark with infinite local memory and with our C-CHARM technique, ranging from one to sixteen RAW tiles. Each point in each curve is the execution time of the benchmark on the respective number of tiles normalized to that benchmark's execution time on a single tile. For convolution, jacobi, and SOR, the speedup is quite comparable up to four processing tiles. Above four processors however, the C-CHARM speedup quickly reaches a plateau while RawCC continues to improve. Median filter performs as well as RawCC alone, but there is little speedup because RawCC fails to find instruction level parallelism.

Our initial assumption was that this plateau affect was due to the new sharing of the off chip bandwidth to main memory. Figure 4-4 shows this not to be the case. We define *pin utilization* to be the ratio of cycles in which the pins are used to the total number of execution cycles. The top portion of the ratio (the number of times the pins are utilized) is a constant for each program, only the number of execution cycles changes. For our benchmarks convolution, jacobi, and SOR, the utilization of the pins running from on chip to the DRAM ¹ is well below 15%. The pin utilization for median filter is so low it is

¹The pins running from on chip to the DRAM are far more heavily utilized than the corresponding pins running from the DRAM onto the chip. This is because of the memory interface protocol request messages. The DRAM to on chip path is only used by the data being returned by main memory.

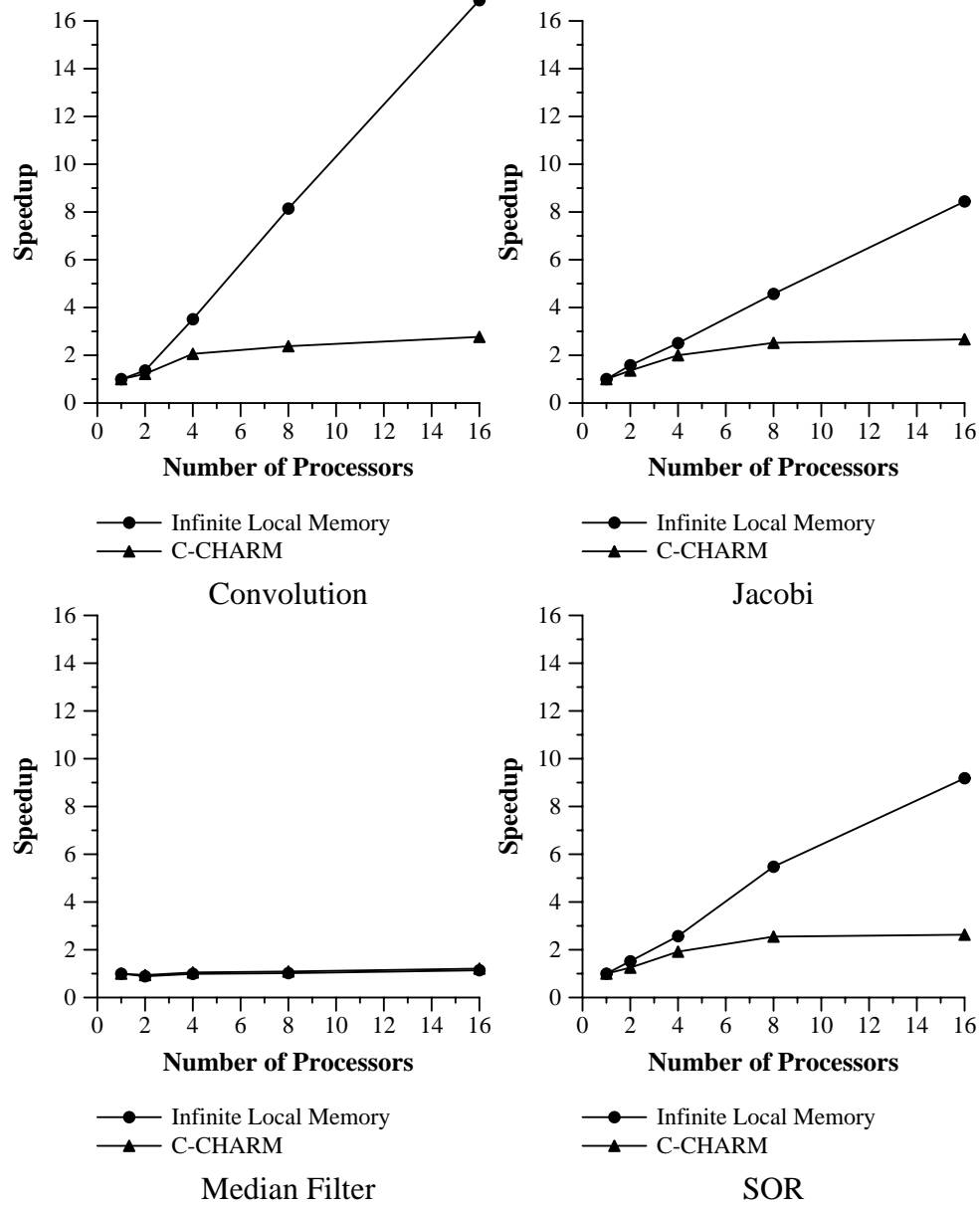


Figure 4-3: Execution Speedup

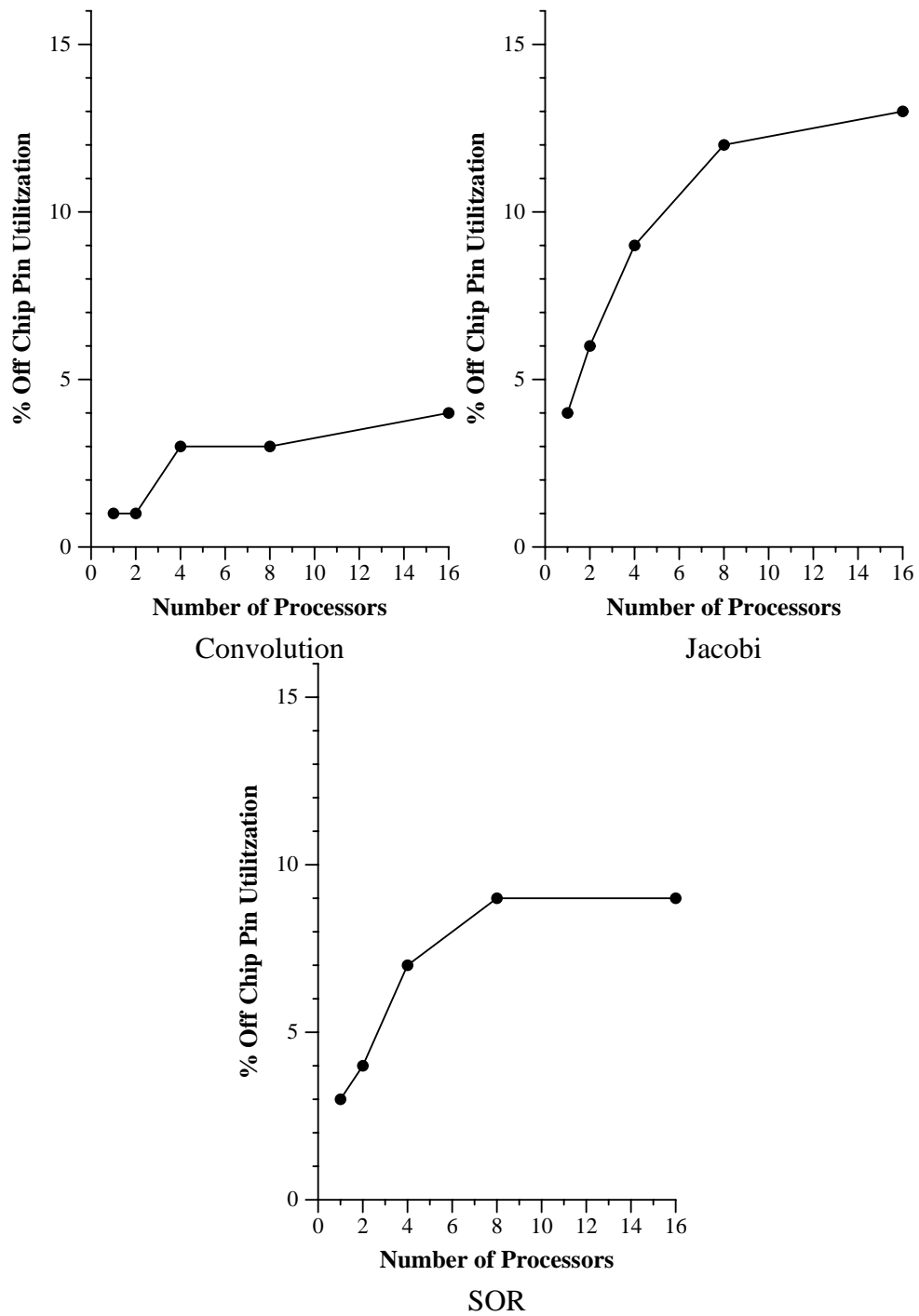


Figure 4-4: Off Chip Pin Bandwidth Utilization

Processors	RawCC cycles	C-CHARM cycles	C-CHARM / RawCC
1	16887490	18128421	1.07
2	18872244	19129609	1.01
4	17050038	17240383	1.01
8	16425922	16536251	1
16	14757724	14893168	1

Table 4.1: Comparison of Execution Times for Median Filter

almost immeasurable. Even if there were network contention resulting from the fan-in of requests to the single set of pins used to communicate with DRAM, those pins should still be far more heavily utilized than 15%.

The answer lies in understanding the results for median filter. The primary difference with regards to our technique between median filter and the other three benchmarks is that median filter has significantly more work in the inner loop. The inner loop of median filter is dominated by a bubble sort of the nine element stencil while the other codes mostly consist of a small series of multiplies and adds. Even though RawCC fails to find parallelism in median filter, it is able to cover the latency of the DRAM operations with this extra work. Table 4.1 shows the absolute execution time in cycles of median filter with infinite local memory and with C-CHARM. The difference is negligible, indicating that DRAM latency is not impacting performance. This means the techniques employed by RawCC for finding instruction level parallelism are not sufficient to hide the long latency of DRAM accesses. However, when the latencies are hidden, our technique can potentially perform extremely well.

Chapter 5

Related Work

5.1 Software Controlled Prefetching

No work is perhaps more closely related to the work presented in this thesis than that of Todd Mowry's work in software controlled prefetching [9]. Mowry uses reuse analysis to ascertain when an affine array access in a nested FOR loop is likely to miss the cache. An architected *prefetch* instruction is used to inform the cache of the likely miss, modifying the cache's default fetch and replace behavior in favor of knowledge gathered at compile time. This work also makes attempts to schedule these prefetches sufficiently in advance of the need for the data object to overlap the latency of memory with useful computation.

Our technique also assumes an architected method for affecting the use of local memory, but all aspects of the data movement is exposed as opposed to providing a simple communication channel to the cache. Instead of altering the existing cache replacement policy with the information gathered at compile time, our compiler uses similar information to set the policy directly. We also schedule the prefetch of a data item before it is needed by the computation, but not on a previous iteration.

5.2 Loop Transformation for Locality

Extensive work has been done to alter loops and loop schedules ¹ in order to maximize data locality. Allan Porterfield used an analysis similar to ours as a metric for locality improving loop transformations [10]. He was looking for what he called an *overflow iteration*, or the iteration at which locations in the cache would likely need to be reassigned new main memory locations because the cache was full. To find this overflow iteration, he looked at how iterations communicated values and attempted to put a bound on the number of new values brought into local memory on each iteration. He was using a variant on reuse analysis plus dependence analysis to minimize capacity misses. We use this information to generate a mapping to local memory which is conflict free.

Michael Wolf's formulation of dependence and reuse analysis form the core of our analysis [14]. He used the information found at compile time to reorder the loop schedule, including use of the tiling transformation, to maximize the locality achieved by scientific codes running in cache memory hierarchy.

5.3 Array Copying

Lam, Rothberg and Wolf observed in [5] that although loop transformations such as tiling work well to limit capacity misses, conflict misses are still a significant problem. In fact, small factors can have major affects on the performance of tiled codes. Temam, Granston and Jalby developed a methodology for selectively copying array tiles at run time to prevent conflict misses [12]. By remapping array elements to new global locations, they could be assured that the cache replacement policy would not map multiple reused values to the same local memory location.

Our technique takes an approach with similar results to [12]. One could look at our local memory mapping methodology as copying the portion of the global memory array currently being actively reused into consecutive, non-interfering local memory locations. By exposing the local memory, this task has been made significantly easier. We do not

¹The *loop schedule* is the order in which the iterations of the loop nest are executed.

have to calculate what the cache will decide to do and choose global addresses which will not conflict at run time. We know the global addresses do not conflict because we chose a replacement policy which maps them into different local locations.

Chapter 6

Future Work and Conclusion

The most important future work which needs to be undertaken is solving the problem of finding complex mapping relationships between iteration space and data space as alluded to in Section 3.1.7. Applying the solution to this linear algebra problem, once the solution is found, will almost certainly involve transformations of the input loop schedule. Such loop transformations are a problem that has not yet been studied in conjunction with our work to date. Solving these problems will significantly increase the range of programs over which our ideas can be applied.

Once the above problems have been solved, methods for further covering the latency to DRAM should be studied. We have considered two possible means to this end. One could unroll the loops some number of times before performing the C-CHARM analyses and transformations. This would place more work between the prerequisite prefetch at the beginning of each iteration and the receive at the end. The significant disadvantage to this approach is that by putting more references in each iteration, the reuse groups can become much larger, requiring more local memory to be devoted to the group. Software pipelining [6] the loop after the C-CHARM transformations would allow us to place the prefetch some number of iterations before the receive. This would allow us to overlap the latency of the global memory fetch without increasing local memory pressure, but has the disadvantage of additional compiler complexity.

In conclusion, we have detailed and demonstrated a technique for explicit compiler management of the memory hierarchy. Our compiler takes programs with group reuse

and explicitly maps the currently active subset of addresses into local memory. All main memory fetches and writebacks are compiler generated. We rely on software exposure of the memory hierarchy by the underlying architecture and some simple architected primitives to handle the data movement operations. We demonstrated that our technique one achieves excellent performance on a uniprocessor. A combination of our completely static technique and a runtime controlled dynamic scheme to fall back on may well perform on equal footing with modern caches for many programs. On machines with a large number of processors, however, overlapping the latency to main memory requires more sophistication than we currently have.

Bibliography

- [1] Saman Prabhath Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford University, 1997.
- [2] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.
- [3] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.
- [4] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.
- [5] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 63–74, Santa Clara, CA, April 1991.
- [6] Monica Sin-Ling Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie-Mellon University, 1987.
- [7] Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.
- [8] Csaba Andras Moritz, Matt Frank, Walter Lee, and Saman Amarasinghe. Hot pages: Software caching for raw microprocessors. Technical Report LCS-TM-599, Laboratory for Computer Science, Massachusetts Institute of Technology, Sept 1999.
- [9] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.
- [10] Allan Kennedy Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, 1989.
- [11] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 24–33, San Jose, CA, October 1998.

- [12] Oliver Temam, Elana D. Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, November 1993.
- [13] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997. Also available as MIT-LCS-TR-709.
- [14] Michael Edward Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, 1992.