Compiler Techniques for Managing Data Motion

John S. Pieper

December 1993

CMU-CS-93-217

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:
H.T. Kung, advisor
Thomas Gross, acting advisor
Jaspal Subhlok
Hudson Ribas, AT&T Bell Laboratories

This research was sponsored by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035. John Pieper was supported in part by the Fannie and John Hertz Foundation.

Keywords: compilers, data storage representations, discrete mathematics, memory structures

Abstract

Software caching, automatic algorithm blocking, and data overlays are different names for the same problem: compiler management of data movement throughout the memory hierarchy. Modern high-performance architectures often omit hardware support for moving data between levels of the memory hierarchy: iWarp does not include a data cache, and Cray supercomputers do not have virtual memory. These systems have effectively traded a more complicated programming model for performance by replacing a hardware-controlled memory hierarchy with a simple fast memory. The simpler memories have less logic in the critical path, so the cycle time of the memories is improved.

For programs which fit in the resulting memory, the extra performance is great. Unfortunately, the driving force behind supercomputing today is a class of very large scientific problems, both in terms of computation time and in terms of the amount of data used. Many of these programs do not fit in the memory of the machines available. When architects trade hardware support for data migration to gain performance, control of the memory hierarchy is left to the programmer. Either the program size must be cut down to fit into the machine, or every loop which accesses more data than will fit into memory must be restructured by hand. This thesis describes how a compiler can relieve the programmer of this burden, and automate data motion throughout the memory hierarchy without direct hardware support.

This work develops a model of how data is accessed within a nested loop by typical scientific programs. It describes techniques which can be used by compilers faced with the task of managing data motion. The concentration is on nested loops which process large data arrays using linear array subscripts. Because the array subscripts are linear functions of the loop indices and the loop indices form an integer lattice, linear algebra can be applied to solve many compilation problems.

The approach is to tile the iteration space of the loop nest. Tiling allows the compiler to improve locality of reference. The tiling basis matrix is chosen from a set of candidate vectors which neatly divide the data set. The execution order of the tiles is selected to maximize locality between tiles. Finally, the tile sizes are chosen to minimize execution time.

The approach has been applied to several common scientific loop nests: matrix-matrix multiplication, QR-decomposition, and LU-decomposition. In addition, an illustrative example from the Livermore Loop benchmark set is examined. Although more compiler time can be required in some cases, this technique produces better code at no cost for most programs.

Acknowledgements

Many people helped greatly with this thesis. I was privileged to have a very high-quality thesis committee. My advisor, H. T. Kung, forces all of his students to do scientific research; his guidance contributed greatly to the quality of the work. Thomas Gross has inspired and guided me throughout my graduate career, and he acted as my advisor after H.T. left for Harvard. Thomas repeatedly read the thesis, and has an amazing ability to point out the weak points in a document. Jaspal Subhlok listened whenever I asked him to, and made helpful suggestions at several points. Hudson Ribas brought a keen understanding of the material and a careful reading of the thesis together to make an outstanding contribution to this document and to the ideas which are its underpinning.

I must acknowledge the support of the Computer Science Department of Carnegie-Mellon University, which has become the School of Computer Science of Carnegie (no dash) Mellon University, both for financial support and for maintaining a culture which is unique in its outstanding support of research. I also must acknowledge the support of the Fannie and John Hertz Foundation, who supported much of my graduate career with both tuition support and a very generous stipend.

I would like to explicitly thank the members of the iWarp team, both at CMU and at Intel, for exposing me to the entire design cycle of a complex parallel machine. The CMU compiler group pushed the development of the iWarp component from the top; Intel engineers pushed it from the bottom. After the system was built, the CMU team has worked on a Fortran-90 compiler; the protype work of this thesis was done in that compiler. The various members of that group listened to my ideas over and over.

Finally, I must thank Scott Hugh Robinson, inventor of SHRUNIX. Scott taught me to stalk, and to track, and came with me when I needed to get out of the city. He went through the ordeal of thesis-writing first, which gave me invaluable insight into the process. He has continued to give me support and advice on technical, academic, and personal matters since he left to begin his own career.

This thesis is dedicated to **Dawna Ellen Rooks** for her support, for putting up with my friends, for putting up with my weekly trips to the woods, for putting up with me, and for waiting seven years for this document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The data motion problem

1.1.1 Software memory management

Design of the memory hierarchy for a high-performance computer system is a difficult task. Conventional computers usually include caches and virtual memory hardware. Several high-performance architectures, however, do away with one or more of these levels. The Cray line of supercomputers has yet to include virtual memory hardware. The Intel/Carnegie Mellon iWarp system does not include a data cache, opting instead for a small static RAM with single-clock access time. These systems have effectively traded a more constrained programming model for performance, replacing a hardware-controlled memory hierarchy with a simple fast memory.

The simpler memories have less logic in the critical path, and so the cycle time of the memories is improved. For programs that fit in the resulting memory, the extra performance is great. Unfortunately, the driving force behind supercomputing today is a class of very large scientific problems, both in terms of computation time and in terms of the amount of data used. Many of these programs do not fit in the memory of the machines available to researchers. Sometimes the programs can be shrunk with some loss of accuracy, but often researchers must wait for the next generation of larger, faster machines. This thesis addresses this problem by allowing the compiler to hide the memory hierarchy from the programmer. The programmer writes his code as if there were a single large memory, and the compiler will move data into and out of the fast buffer memory to optimize performance.

Compilers have traditionally been limited in their control of the memory hierarchy. Most compilers control only the allocation of machine registers. Before the popularization of virtual memory, programmers used overlays to run programs that did not fit into main memory. Techniques for compiler generation of overlays for code were invented a little too late to become popular before virtual memory did. Code overlays may suffice for conventional programs whose data is small relative to the amount of code used. Large scientific codes use orders of magnitude more data than code. To implement data overlays for these programs, each loop that accesses more data than will fit in main memory must be restructured.

In this thesis we investigate the use of modern compiler technology to manage the memory hierarchy without hardware support (like caching or virtual memory hardware). The compiler will cut the data of a program into chunks that fit into memory. It will modify the loop structure of the program, inserting block copies of the data to move it into faster levels of the memory hierarchy as required, and to move the data back again when it is no longer needed. The compiler can effectively relieve the programmer of the burden of managing the memory hierarchy even when the hardware does not help in the process. This allows even very large programs to be run on machines whose architects opted for memory performance at the cost of hardware support for the memory hierarchy.

1.1.2 Parallelism

To meet the computational demand of scientific computing, more and more architects are turning to parallel computing. Scalable parallel architectures require the use of distributed memory, with each processor having a small local memory and communicating with other processors to get data stored in their memories. This communication can be handled by the hardware, for example by using a directory-based hierarchical caching scheme. In this case, the compiler needs only to ensure that the program has good cache locality. The other possibility is for that communication to be left to the programmer. In this case, the program must explicitly communicate with other processors when data must be exchanged. Machines with explicit communication are easier to build since no cache-snooping hardware is required and no cache control logic is required in the communication network. Unfortunately, the burden of the programmer is enormously increased.

Compilers need to be able to automatically parallelize programs for private memory machines; it is just too difficult to write parallel programs for distributed memory computers. To produce good code for parallel machines, it is not enough for the compiler to understand parallelism. The compiler must also understand the costs of data motion between processors and through the local memory hierarchy.

The goal of a parallelizing compiler is to map a program expressed in a machine-independent language into a parallel program for a distributed memory machine with a memory hierarchy, such as the one in Figure 1.3 on page 6. The compiler must manage a single global name space that is mapped into the private memories of the system. Each data item is assigned a "home" memory location in the M_2 memory of some processor. The M_1 memories are used in much the same way the register file is used by uniprocessor compilers: data items are moved from the home location in M_2 into M_1 of the processor that needs that item. If data is re-used from M_1 before it is returned to M_2, memory bandwidth (and possibly communication bandwidth) is saved.

1.1.3 Tiling

To obtain the greatest benefit from the M_1 memories, loops in the program must be restructured to optimize locality. Each loop nest defines a space of iterations to be performed. The bounds of the space are determined by the loop bounds in the program. The compiler cannot generally limit the amount of data accessed in any particular direction in this space because the loop bounds are specified by the programmer. By cutting the iteration space into tiles, the compiler can limit the amount of data accessed in a tile by choosing the tile size in each dimension. The compiler chooses the size of the tiles so that all of the data required to execute a tile fits into M_1 at the same time. The compiler will generate code which loads the data required for a tile, executes the tile, and stores back the result. All of the data accesses during the execution of a tile are M_1 accesses, so the computation can be performed very quickly.

In this thesis, the goal of tiling is to reduce the overhead of software memory management as well as to improve locality. Tiling allows the compiler to block memory references. This reduces the total memory access latency for memories which support block-access. Additionally, tiling usually increases the ratio of computation to I/O of the program. For each M_2 memory access (which can be considered an I/O operation), the number of computations that can be performed on average is increased. Since tiling does not change the computation itself, the higher computation-to-I/O ratio is achieved by lowering the number of M_2 accesses required by the loop nest.

This work investigates tiling for locality and parallelism simultaneously, by scheduling the tiles to get optimal *intertile* locality, which has not yet been addressed. Intertile locality refers to data that is used within one tile of iterations that can be kept in fast memory because it will also be used in the next tile of iterations. In cache-based uniprocessor systems, intertile locality is a second-order effect; tiling itself is the principal performance enhancer. Scheduling the tiles for intertile locality, however, further reduces the secondary memory traffic generated by a program.

1.2 Problem

In this section we discuss the limits of the problem to be solved. First, we discuss the class of programs that will be dealt with. In the following section, we discuss the kinds of machine architectures addressed in this work.

1.2.1 Input Code

Scientific programs are typified by large data sets, accessed in linear patterns. These linear patterns are exploited in this work by using linear algebra techniques to model the memory access patterns. This work is directly applicable to programs with linear array accesses and linear loop bounds. Source code in normalized form (we use Ribas's definition of "normalized"[47]) must be a set of perfectly nested loops, as shown in Figure 1.1.[1] The f_i's and g_i's in that figure are affine functions.

We make the following assumptions about the nested loops that are input to the compiler:

- We have a nest of n loops in normalized form (positive unit loop steps).

---

[1] Source code in this thesis is written in an ALGOL-like pseudo language. All code can be trivially translated into C, FORTRAN, or an equivalent language.

```
for $i_0$ = $f_0()$ to $g_0()$
 for $i_1$ = $f_1(i_0)$ to $g_1(i_0)$ do
  for $i_2$ = $f_2(i_0, i_1)$ to $g_2(i_0, i_1)$ do
   ...
    for $i_{n-1} = f_{n-1}(i_0, \cdots, i_{n-2})$ to $g_{n-1}(i_0, \cdots, i_{n-2})$ do
    begin
        ...body...
    end
```

Figure 1.1: Input code in normalized form

- Array subscript expressions are linear combinations of loop index vectors, plus possibly a constant.

1.2.2 Machine models

A simple uniprocessor with a two-level memory hierarchy, is shown in Figure 1.2. The small memory (M_1) has cycle time t and can hold M items, while the big memory (M_2) has access time Kt, $K > 1$, and can hold an infinite number of items. In the figure, the slow memory is backing store only. Data stored there cannot be operated on, only moved into fast memory: there is no direct path from M_2 to the CPU. We can relax this constraint later. If we put the CPU-M_2 path into the machine model of Figure 1.2, then the compiler should fetch any data that cannot be reused directly from M_2 and store it back directly to M_2, saving space in M_1 for data that can be reused. The development will be clearer without the added complexity of the extra data path, so without loss of generality we will assume no direct CPU-M_2 path. In Chapter 8 we will revisit this subject and sketch the changes needed to incorporate the extra data path.

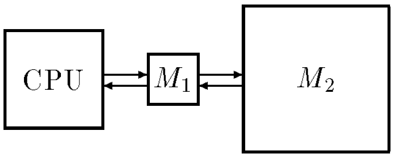


Figure 1.2: Uniprocessor machine model

Because tiling increases the computation-to-I/O ratio of a program, more efficent tiling methods are most important for small M_1 memories like register files and on-chip buffer memories. For larger memories, like off-chip caches, tiles become computation-bounded and the extra efficiency of saving a few M_2 operations is relatively unimportant; straightforward tiling techniques are sufficient. The reader should keep in mind the relatively small size of the target M_1 memories. Chapter 7 will make clearer how small M_1 must be for the extra efficiency to be important.

Figure 1.3 shows the result of using a group of these simple uniprocessors to construct a parallel machine. The important feature in this figure is that it is not possible to access data in the memories of other processors. Instead, communication primitives must be used to move the data across the network into the processor that will use the data. We will return to the parallel processor model in detail when scheduling for parallel machines is discussed in Chapter 5.

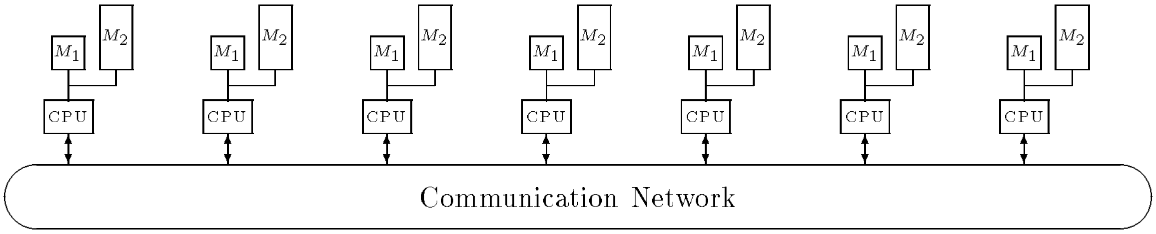


Figure 1.3: Parallel machine model

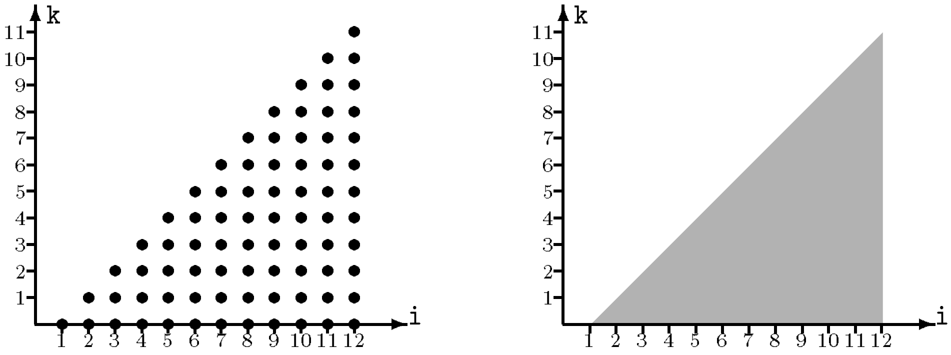
1.3 Data model

The compiler must have a model of how the program accesses data. This section describes the model used in this work. The loop nest itself is modeled as an iteration space. The data accesses are modeled using streams. Reference vectors describe the relation between the data space of an array and the iteration space of a loop nest; this allows the compiler to model the relationship between the data space and the iteration space that results after loop transformations. Ordering constraints on the iterations are modeled using generalized dependence vectors. These dependences also point out reuse of data in the iteration space.

1.3.1 Iteration spaces

In the abstract input code of Figure 1.1, the index variables of the loops are $i_1, i_2, \ldots, i_n$. The vector $\vec{\imath} = (i_1, i_2, \ldots, i_n)$ is called the *index vector* of a nested loop. It is the vector of index variables of each loop. As the loop nest is executed, $\vec{\imath}$ takes on a set of values corresponding to the iterations of the loop nest. Because the loop bounds are linear functions of outer loop bounds, the set of iterations forms a polytope in n-space. This polytope is called the iteration space $\mathcal{I}$ of the loop nest. $\mathcal{I}$ necessarily has dimensionality n.

Elementary vectors are unit-length vectors along each axis. In n-space, there are n distinct elementary vectors. The ith elementary vector, $\vec{e_i}$, is zero everywhere except in the ith position, where it has the entry 1. This vector points in the direction in which the ith loop executes, so it is also known as the *loop direction vector* for the ith loop.



```
for i = 1 to 12 do
  for k = 0 to i-1 do
    w[i] = w[i] + b[i,k]*w[i+k];
```

Figure 1.4: The iteration space of a loop nest

The set of values that $\vec{\imath}$ can take on are all integer vectors, and the iteration space is a set of integer-valued points in n-space, as shown in Figure 1.4. The code which induces the iteration space is shown on the left; the iteration space itself is in the center diagram. In this case, the iteration polytope has the shape of a triangle. It is often more convenient to think about sets of points in the iteration space as shapes rather than as sets of discrete points, as in the diagram on the right. When shapes are used, it is sometimes unclear which edges of the shapes are included in the set under consideration. Dot-diagrams will be used when it is important to be clear exactly which iterations are to be included; shape-diagrams will be used when the overall shape is important but the exact bounds are incidental.

1.3.2 Streams

Each *reference* to a variable in the loop body generates (or *induces*) a stream of *accesses* to memory as the loop nest is executed. For example, consider the first reference to `A[i]`, on the left-hand side of the assignment statement in Figure 1.5. This single reference generates the stream `<A[1], A[2], A[3], ..., A[N]>`. The second reference to `A` generates the same stream. If two references to a variable have the same subscript expressions (like the first two references to `A`), we consider the two a single reference (since they access the same data in the same order and at the same time).

```
for i = 1 to N do
        A[i] := A[i] + B[i]/A[i-1];
```

Figure 1.5: Example streams

The last reference to `A[i-1]` generates `<A[0], A[1], A[2], ..., A[N-1]>`. If two references to the same variable are uniformly generated, that is, they have the same loop index coefficients but possibly different constant offsets, the induced streams contain accesses in the same order, but skewed relative to one another. All references to `A` in the figure are uniformly generated. Uniformly generated references use the same data in the same order, just slightly earlier or later in time. We can use this observation to coalesce two or more uniformly generated references into a single stream-inducing reference (accesses made by this reference retrieve multiple items). When references are coalesced in this fashion, we call the resulting reference a uniformly generated reference. Note that since all references to the same variable need not be uniformly generated, there can be multiple uniformly generated references associated with a single variable. When data is buffered in fast memory, different uniformly generated references must use different parts of fast memory to store the associated data, but a single uniformly generated stream can store the data just once, keeping around a slightly larger window of the stream to satisfy the constant-offset references. Keeping around a few extra data items is more efficient than buffering the same data in several places if the constant offsets are small, which they usually are.

To summarize:

- An *access* is a particular memory request (read or write), represented by the memory address.

- A *reference* is an occurrence in a loop nest of an array variable.
- A *stream* is a sequence of accesses, induced by a subscripted array reference occurring inside a loop nest.

1.3.3 Reference vectors

It is assumed that array subscript expressions are linear combinations of loop index vectors, plus a constant. That is, the kth reference to a δ-dimensional array `v`

$$\mathtt{v}.k\begin{bmatrix} a_{0,0}i_0 + a_{0,1}i_1 + \cdots + a_{0,n-1}i_{n-1} + c_0 \\ a_{1,0}i_0 + a_{1,1}i_1 + \cdots + a_{1,n-1}i_{n-1} + c_1 \\ \vdots \\ a_{\delta-1,0}i_0 + a_{\delta-1,1}i_1 + \cdots + a_{\delta-1,n-1}i_{n-1} + c_{\delta-1} \end{bmatrix}$$

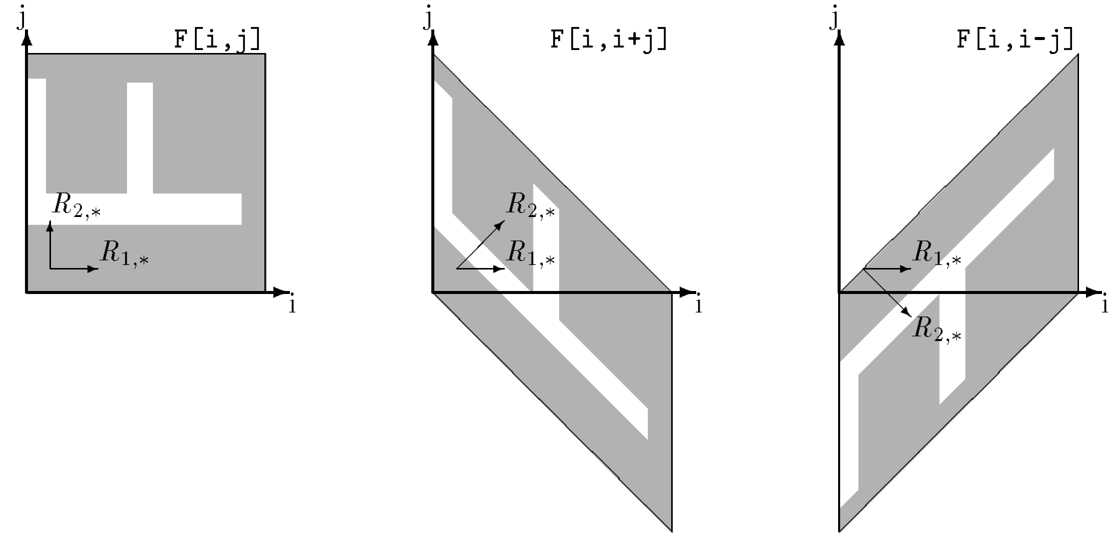
can be written as

$$\mathtt{v}.k[R_{\mathtt{v}.k} \cdot \vec{\imath} + \vec{c}]$$

by letting $R_{\mathtt{v}.k}$ be the matrix with entries $a_{i,j}$ and $\vec{c}$ be the vector with entries c_i. Since $\dim(\mathtt{v}) = \delta$, $R_{\mathtt{v}.k} \in Z^{\delta \times n}$ and $\vec{c} \in Z^{\delta}$. The rows of $R_{\mathtt{v}.k}$ are called *reference vectors* for the stream associated with the kth use of `v`. They are vectors in the iteration space that point in the direction of increasing array subscripts for each dimension of `v`, for a particular use of `v`.

We will write vectors using different notations depending on what we want to emphasize. The ith row of a reference matrix is written $R_{i,*}$. If the index vector is $\vec{\imath} = (\mathtt{i}, \mathtt{j}, \mathtt{k})$ and the ith row is $(1, -2, 7)$, the reference vector $R_{i,*}$ can be written $(1, -2, 7)$, to emphasize its nature as an integer-valued vector, or $\mathtt{i} - 2\mathtt{j} + 7\mathtt{k}$ to emphasize the relationship to the iteration space. The notation is somewhat more confusing when reference vectors are elementary vectors; if $R_{i,*} = (1, 0, 0)$, the vector $(1, 0, 0)$ may be written as just `i`. It will be clear from context when we use `i` as a vector and when it is used as a program variable.

Figure 1.6 shows examples of reference vectors relating data to the iteration space. The array reference is shown near the bottom of each diagram. Each diagram represents a different reference to a matrix `F` inside a two-deep nested loop `for i...for j...` (the exact loop bounds are unimportant here—the point is to show how the reference vectors relate the data space of the arrays to the iteration space).



Reference		$R_{1,*}$		$R_{2,*}$	
code form	matrix form	code form	vector	code form	vector
`F[i,j]`	`F`$\left[\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}\texttt{i}\\ \texttt{j}\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}\right]$	`i`	(1,0)	`j`	(0,1)
`F[i,i+j]`	`F`$\left[\begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix}\begin{bmatrix}\texttt{i}\\ \texttt{j}\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}\right]$	`i`	(1,0)	`i+j`	(1,1)
`F[i,i-j]`	`F`$\left[\begin{bmatrix}1 & 0\\ 1 & -1\end{bmatrix}\begin{bmatrix}\texttt{i}\\ \texttt{j}\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}\right]$	`i`	(1,0)	`i-j`	(1,-1)

Figure 1.6: Reference vectors relate data spaces to the iteration space

The white letter `F` in each figure represents how the array is oriented in the iteration space. The letter is oriented so that the vertical line which forms the left side of the letter is aligned with a column of the matrix, the horizontal "flag" parts are aligned with rows of the matrix, the top of the `F` is near low-numbered rows, and the vertical line is near low-numbered columns (the letter `F` was chosen because it is notably asymmetric both vertically and horizontally; this is particularly important in the rightmost diagram where the matrix is reflected upside-down).

The reference vectors in each diagram point in the directions of increasing array subscripts in each dimension. This means that $R_{1,*}$, the row reference vector, points *across* rows, and $R_{2,*}$, the column reference vector, points across columns. When a variable reference has all its reference vectors perpendicular to one another, it is easy to think that reference vectors point *along* rows or columns, but this is not the case.

The constant-offset vector $\vec{c}$ has the effect of shifting the data relative to the origin of the iteration space. It does not affect the orientation of the data. Figure 1.7 shows an example. In the figure, two streams are being referenced in a 2-dimensional loop nest, with loops in variables `i` and `j`. The darker-shaded area corresponds to the layout of the stream `F[i,j]`, while the lighter-shaded area corresponds to the layout of the stream `F[i-7,j-2]`. The constant offset vector of the first stream is $\vec{0}$, the zero-vector. The constant offset vector of the second stream is (-7,-2). This has the effect of shifting the elements used by an iteration 7 units in the first dimension of the array and 2 units in the second dimension.

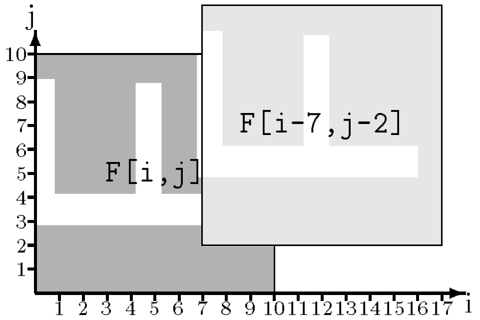


Figure 1.7: How constant offsets affect array layout in the iteration space

The set of pairs $[R_{\mathtt{v}.i}, \vec{c}]$ for all $\vec{c}$ and i corresponds to the set of all streams for a given variable `v`. The set of matrices $\{R_{\mathtt{v}.i}\}$ for all i corresponds to the set of uniformly generated streams for `v`. $R_{\mathtt{v}.i}$ is the reference matrix for a particular stream $\mathtt{v}.i$, associated with a particular use (or set of uses, in the case of a uniformly generated stream) of a variable. If two distinct variable references $\mathtt{v}.i$ and $\mathtt{w}.j$ have the same reference matrices, they still represent different streams because they are accessing different arrays so the reference matrices would be $R_{\mathtt{v}.i}$ and $R_{\mathtt{w}.j}$.

The space spanned by the union of all reference vectors is also important. Since we must divide the iteration space into chunks that reference a data set that fits into M_1, we must be able to limit how much of each stream must be stored to execute a chunk. This implies that we must be able to cut the space spanned by the union of all reference vectors into finite-sized pieces. We denote the space spanned by the union of all reference vectors $\mathcal{D}$. The iteration space $\mathcal{I}$ necessarily has dimensionality n. Let λ be the dimensionality of $\mathcal{D}$. Note that we have $1 \leq \lambda \leq n$.

1.3.4 Dependences

Reference matrices allow us to describe the relationship between array elements and the iteration space. Dependences are relations between iterations that access the same array elements. Dependences precisely capture reuse in the iteration space, and they are the primary tool of a compiler seeking to manage data motion efficiently. Dependences also describe the limitations on what reorderings the compiler can perform without changing the semantics of the program.

Traditionally, dependences are relations between memory accesses. A dependence exists between two memory accesses m_1 and m_2 if they both refer to the same memory location and m_1 occurs before m_2 in the ordering specified by the source code. We will write this dependence between memory accesses $m_1 \xrightarrow{M} m_2$.

In this thesis we assume that the sets of memory locations used by different arrays are completely disjoint, so that dependences exist between two iterations if and only if the iterations access the same element of the same array.[2] The dependence relation can be written with the name of the array to emphasize this fact. If $m_1 \xrightarrow{M} m_2$ because both accesses refer to an array variable `v`, the dependence is written $m_1 \xrightarrow{\mathtt{v}} m_2$.

A compiler which deals with iteration spaces needs a generalization of this kind of dependence. A dependence exists between two distinct iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ if there is a memory reference m_1 to `v` which occurs in $\vec{\imath}_1$ and a memory reference m_2 to `v` which occurs in $\vec{\imath}_2$, and $m_1 \xrightarrow{\mathtt{v}} m_2$. This dependence is written $\vec{\imath}_1 \xrightarrow{\mathtt{v}} \vec{\imath}_2$.

This definition introduces a slight complication. The dependence relation on memory accesses is transitive, because if there is a dependence $m_1 \xrightarrow{\mathtt{v}} m_2$ and a dependence $m_2 \xrightarrow{\mathtt{v}} m_3$, there is necessarily a dependence $m_1 \xrightarrow{\mathtt{v}} m_3$ because all of the accesses reference the same array. This is not true of iteration dependences, because given three iterations $\vec{\imath}_1, \vec{\imath}_2, \vec{\imath}_3$, it is possible that $m_1 \xrightarrow{\mathtt{v}_1} m_2$, and $m_3 \xrightarrow{\mathtt{v}_2} m_4$, but $\mathtt{v}_1 \neq \mathtt{v}_2$. The dependence relation relation on iterations is therefore defined as follows: a dependence exists between iteration $\vec{\imath}_1$ and iteration $\vec{\imath}_2$, written $\vec{\imath}_1 \rightarrow \vec{\imath}_2$, if and only if there is some chain of dependences

$$\vec{\imath}_1 \xrightarrow{\mathtt{v}_1} \vec{\imath}_a \xrightarrow{\mathtt{v}_a} \vec{\imath}_b \xrightarrow{\mathtt{v}_b} \ldots \vec{\imath}_z \xrightarrow{\mathtt{v}_z} \vec{\imath}_2$$

---

[2] Many programming languages allow arrays to be accessed with different names. This "feature" forces the compiler to consider the possibility that two different names might refer to the same memory location. This is commonly called the *aliasing problem.* The solution of this problem is beyond the scope of this work.

Kinds of dependences

A single memory reference can be a read or a write. Dependences can be classified according to the type of references, as shown in Table 1.1. These labels apply directly to dependences between memory accesses; the labels will be generalized to iteration dependences later.

Of the four kinds, input dependences are often omitted from standard works on dependence analysis, because reordering two reads cannot change the semantics of the program. Input dependences do not restrict the reorderings that can be applied; the other three types do. All four kinds signal reuse, however.

m_1	m_2	kind
read	read	input dependence
read	write	anti dependence
write	read	flow dependence
write	write	output dependence

Table 1.1: Dependence types

Types of dependences

All dependences point out reuse in the iteration space, but some dependences point out more reuse than others. Many dependences point out a single reuse, while others point out a number of reuses proportional to the size of the iteration space.

Consider the program of Figure 1.8. The iteration space diagram shows a number of dependences drawn as arrows between iterations that depend on one another. Although the number of dependences is proportional to the size of the iteration space, each dependence is a marker for a single reuse. Consider the element `A[3,3]`. It is written by iteration $\vec{\imath} = (3,3)$ and read by iteration $(4,5)$, and otherwise is not accessed.

Figure 1.9 shows a program where dependences point out a number of reuses proportional to the size of the iteration space. Consider the iteration (1,2). This iteration accesses `B[2]`. So do the iterations (2,2), (3,2), (4,2), (5,2), and (6,2). So there are five dependences with their tails at (1,2), of length (1,0), (2,0), (3,0), (4,0) and (5,0). Such dependence relations are usually abstracted to just their signs, and written (+,0); this notation will be more fully explained in the discussion of dependence representation on page 16. In this case, because of the transitivity of the dependence relation, the compiler can represent the

```
for i = 1 to 6
  for j = 1 to 6
    A[i,j] = A[i-1, j-2];
```

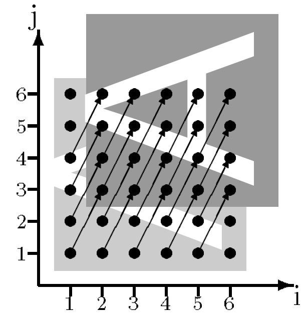


Figure 1.8: Dependences which point out only 1 reuse

full set of dependences with only the vector (1,0). This vector also applies at every point in the iteration space, but since it is an abstraction of the set of dependences $(c, 0)$, it marks reuse proportional to the size of the iteration space.

```
for i = 1 to 6
  for j = 1 to 6
    A[i,j] = A[i,j] + B[j];
```

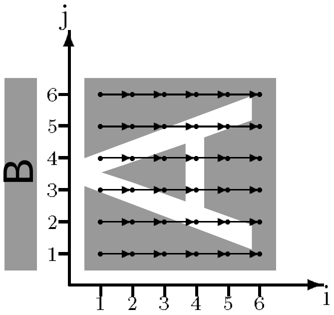


Figure 1.9: Dependences marking a number of reuses proportional to the loop bounds

Dependences as vectors

An iteration dependence $\vec{\imath}_1 \rightarrow \vec{\imath}_2$ can be represented by a vector with its tail at $\vec{\imath}_1$ and its head at $\vec{\imath}_2$. Compilers often assume that if such a dependence exists anywhere in the iteration space, a vector of the same length, pointing in the same direction, exists everywhere in the iteration space. This is justified for two reasons: first, the dependences often are replicated everywhere in this fashion; and second, the kinds of transformations the compiler considers are either prevented or not by a single dependence, so if the dependence exists between one pair of iterations, it may as well exist between all pairs with similar relative geometry.

Replicating the vectors everywhere allows the compiler to simplify its representation, by retaining only the vectors themselves and assuming they apply at every iteration point. A vector exists between two iterations whenever the subscript functions are equal for two array accesses.

Given the array references $\mathtt{v}[A_1\vec{\imath}+\vec{c}_1]$ and $\mathtt{v}[A_2\vec{\imath}+\vec{c}_2]$, the compiler must find values for $\vec{\imath}_1$ and $\vec{\imath}_2$ which satisfy

$$A_1\vec{\imath}_1 + \vec{c}_1 = A_2\vec{\imath}_2 + \vec{c}_2$$

or, equivalently,

$$A_1\vec{\imath}_1 - A_2\vec{\imath}_2 = \vec{c}_2 - \vec{c}_1$$

The vector from iteration $\vec{\imath}_1$ to $\vec{\imath}_2$ is given by $\vec{d} = \vec{\imath}_2 - \vec{\imath}_1$. Substituting $\vec{\imath}_2 - \vec{d}$ for $\vec{\imath}_1$, this equation becomes

$$A_1(\vec{\imath}_2 - \vec{d}) - A_2\vec{\imath}_2 = \vec{c}_2 - \vec{c}_1$$

and solving for $\vec{d}$,

$$A_1\vec{d} = (A_1 - A_2)\vec{\imath}_2 + (\vec{c}_1 - \vec{c}_2)$$

From this equation, it is easy to see that if $A_1 = A_2$, the value of $\vec{d}$ does not depend on where in the iteration space the vector is. The $\vec{\imath}_2$ term drops out, resulting in the simplified equation

$$A_1\vec{d} = (\vec{c}_1 - \vec{c}_2)$$

Now it can be seen that if $\mathrm{rank(A_1)} = n$, A_1 is invertible and $\vec{d} = A_1^{-1}(\vec{c}_1 - \vec{c}_2)$. This situation ($A_1 = A_2$ and $\mathrm{rank(A_1)} = n$) results in dependences which mark a single reuse.

If $A_1 = A_2$ and $\mathrm{rank(A_1)} < n$, $\vec{d}$ takes on a set of the values of the form $\vec{d} = \vec{\nu} + \vec{c}_*$ where $\vec{c}_*$ is the preimage of $(\vec{c}_1 - \vec{c}_2)$ relative to A_1, and $\vec{\nu}$ is any vector in the null space of A_1. In this case, there is reuse proportional to the size of the null space. The size of the null space is determined by the loop bounds, so the vectors represent much more reuse.

If $A_1 \neq A_2$, $\vec{d}$ takes on a set of values which depend on $\vec{\imath}_2$; that is, the dependences are different depending on which iteration they point to (it is easy to show that the dependences differ depending on which iteration they point from by substituting for $\vec{\imath}_2$ instead of $\vec{\imath}_1$). In this case, if the space spanned by the rows of A_1 is different from the space spanned by the rows of A_2, there is reuse proportional to the size of the iteration space. If A_1 and A_2 span the same space, there is only a single reuse.

Dependence representation

The compiler must choose some method for representing dependence vectors. In this thesis, we use Wolf's generalized dependence vector representation ([57], page 17):

> ...Each component d_i of a dependence vector $\vec{d}$ is a possibly infinite range of integers, represented by $[d_i^{min}, d_i^{max}]$, where
>
> $$d_i^{min} \in \mathbb{Z} \cup \{-\infty\}, d_i^{max} \in \mathbb{Z} \cup \{\infty\} \text{ and } d_i^{min} \leq d_i^{max}.$$
>
> The dependence vector $\vec{d}$ is also a distance vector if each of its components is a degenerate range containing a singleton value, meaning $d_i^{min} = d_i^{max}$. We use the notation '+' as shorthand for $[1, \infty]$, '-' as shorthand for $[\infty, -1]$, and '±' as shorthand for $[-\infty, \infty]$. They correspond to Wolfe's directions '<','>', and '$*$' respectively...

The dependence vector matrix is denoted D; each column $D_{*,j}$ of D is a dependence vector. Other dependence models, specifically dependence cones[29], could be used; the critical property of the dependence model is that it permits testing for legal execution directions (see section 1.4.2).

Ordering vectors

The statement "$\vec{z}$ is an ordering vector" for any integer-valued vector $\vec{z}$ means that given two iterations $\vec{x}$ and $\vec{y}$, $\vec{x}$ precedes $\vec{y}$ (written $\vec{x} \prec \vec{y}$) if $\vec{z} \cdot (\vec{y} - \vec{x}) > 0$. A vector $\vec{z}$ is a *legal* ordering vector if $D^{\mathrm{T}} \vec{z} > 0$, that is, if no dependences are violated. For example, in matrix-matrix multiply (Figure 3.1), there is a single dependence carried by the `k`-loop. We will write this as either $D = [\mathtt{k}]$ or $D = (0, 0, 1)^{\mathrm{T}}$. The first representation is used to show how the vectors relate to the loops, and the second notation is used to emphasize the relationship to the space of iterations induced by the loop nest.

There are some legal orderings of the iteration space that cannot be modeled by a single ordering vector (specifically, when dependence vectors have integer divisors other than one, limited re-ordering is often possible, but is not allowed under our model). However, ordering vectors define schedulings that lend themselves to automatic manipulation. Gaining a few extra operations that could be re-ordered is not as important as observing the general trend of data access patterns, which *are* captured by ordering vectors.

1.3.5 Perpendicular vectors

When input dependences are included, dependence vectors capture all reuse available in a loop nest. Unfortunately, dependence vectors are sometimes difficult to compute, and they are not necessarily constant integer vectors. For these reasons, it is sometimes useful to generate vectors representing locality which are known to be constant integer vectors.

One method to do this is to choose linearly independent subsets of $n-1$ reference vectors, and solve for a vector perpendicular to all these. The solution vector is perpendicular to $n-1$ reference vectors, and so represents a direction of locality for any streams whose reference vectors are a subset of the $n-1$ vectors chosen.

Since the set of solutions is a line, there are two rays which are perpendicular to the $n-1$ vectors, one along the line in each direction from the origin. The compiler includes the ray that is positive with respect to the dependence set, if there is one (if both rays are positive, only one is included, and the choice is made arbitrarily).[3] The ray is scaled to be as small as possible while still having all integer entries. Note that given n linearly independent vectors, the compiler can find n perpendicular vectors simultaneously by putting the vectors in a matrix Q and solving for Q^{-1}. The ith vector of Q^{-1} is perpendicular to all but the ith vector of Q (the inner product of the ith vector of Q with the i vector of Q^{-1} is one; the inner product with any other vector is zero). The vectors forming the inverse matrix are then scaled to make them integral.

The vectors constructed with this method form the set of perpendicular vectors for each stream, $V^{\perp}$. Because they span the null space of the reference matrix, $V^{\perp}$ spans the space of dependences which point out a number of reuses proportional to the size of the iteration space. These vectors are constructed to be used as normal vectors to tiling hyperplanes (tiling is discussed in Section 1.4), however, and not as constraints on the ordering of the iterations. The vectors of $V^{\perp}$ are always integer-valued, while dependences are not.

---

[3] A vector $\vec{v}$ is positive with respect to the dependence set if and only if every element of $\vec{v}D$ is nonnegative.

1.3.6 Cones

For any matrix M, the set of vectors $\mathcal{C}(M) = \{\vec{x} \in \mathbb{R}^n | M^{\mathrm{T}}\vec{x} > 0\}$ is called the cone of M. This set is the intersection of the half-spaces defined by hyperplanes passing through the origin and oriented perpendicular to each (column) vector of M. In the case of dependence vectors, $\mathcal{C}(D)$ is exactly the set of possible legal ordering vectors. For a vector $\vec{x}$ to be a legal ordering vector, $D^{\mathrm{T}}\vec{x} > 0$ must hold, and $\mathcal{C}(D)$ is the set of vectors satisfying this requirement.

The union of a cone and its boundary is called the *closure* of the cone. The closure of $\mathcal{C}(M)$ is $\mathcal{C}^*(M) = \{\vec{x} \in \mathbb{R}^n | M^{\mathrm{T}}\vec{x} \geq 0\}$. The difference between $\mathcal{C}(D)$ and $\mathcal{C}^*(D)$ is explained in Section 1.4.2.

For a matrix M of full rank, a *ray* of a cone is a vector $\vec{\rho} \in \mathbb{Z}^n$ that is on the boundary of the cone and is the intersection of at least $k - 1$ of the hyperplanes $M_{*,j} \cdot \vec{\rho} = 0$. The set of rays of the cone of a matrix M is denoted by rays(M). Figure 1.10 graphically shows two cones. The left side of the figure shows a two-dimensional cone. Each vector is perpendicular to a hyperplane; the side of this hyperplane away from the vector is not in the cone (points not in the cone are shown shaded in the figure). In three dimensions, a cone can have an infinite number of rays. In the right side of Figure 1.10, eight hyperplanes are shown, each perpendicular to one of eight lines. In this case the set of points in the cone are the set of points inside what looks like an ice cream cone.

When M is not of full rank, there is a non-trivial null space $\mathcal{N}$. In this case, we follow the method of Schreiber and Dongarra[48], who define the set of rays to be a basis for the null space of M, plus the set of rays for the space spanned by M. The particular set of vectors in the basis for the null space is determined using QR factorization. The details are unimportant to the development here.

1.4 Introduction to tiling

1.4.1 Hyperplane tiling

All of the data referenced by a program is too large to fit into M_1 at once (otherwise there would be nothing for the compiler to do). The compiler must find a way to chop the iteration space of the program into pieces that do fit into M_1. A variant of hyperplane

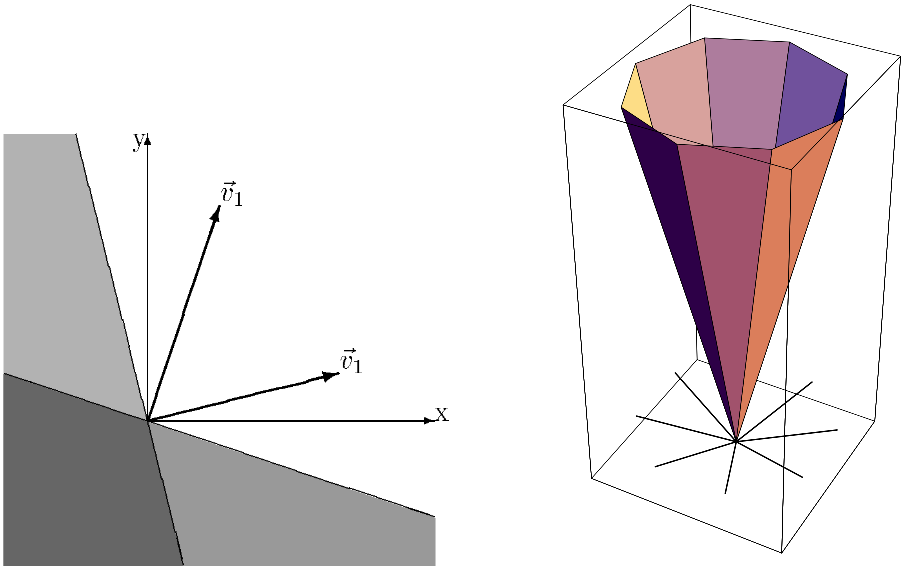


Figure 1.10: Rays of a cone in 2-D and 3-D.

tiling[29] is used.

A vector $\vec{v}$ in the iteration space $\mathcal{I}$ can be used to split the computation by dividing the computation along hyperplanes perpendicular to $\vec{v}$, as in Figure 1.11. In this figure, hyperplanes perpendicular to each of the two vectors (4,1) and (2,6) are spaced evenly by the length of the vectors. The hyperplanes can be spaced by the length of the defining vector, or we can use the vector for direction only and give a separate spacing distance along each vector. This would be the case, for example, if we used the vector (1,3) instead of (2,6); we would then have to specify that the planes are to be spaced with distance $\sqrt{40}$ measured along the normal vector ($2^2 + 6^2 = 40$). We will find it more convenient to scale the dividing vectors to have unit length and use explicit scaling factors.

We will not necessarily tile the full iteration space. We use the term *dividing* to mean tiling a subspace of the iteration space. A *dividing* (of the iteration space) is generated by a set of λ linearly independent unit-length vectors $B_{1,*}, \cdots, B_{\lambda,*}$, and a set of spacing factors along those vectors, $\beta_1, \cdots, \beta_\lambda$. The vectors form the rows of a dividing basis, denoted by the matrix B. B is called a *dividing basis* because B must form a *basis* for the tiled space.

The compiler's goal is to tile the space $\mathcal{D}$. This guarantees that the compiler can limit the data required by a tile to a compiler-selected amount. Linear independence guarantees that $\lambda \leq n$, the dimensionality of the iteration space $\mathcal{I}$. The iteration subspaces that result from a dividing are called *divisions* of the iteration space. In the case $\lambda = n$, the vectors

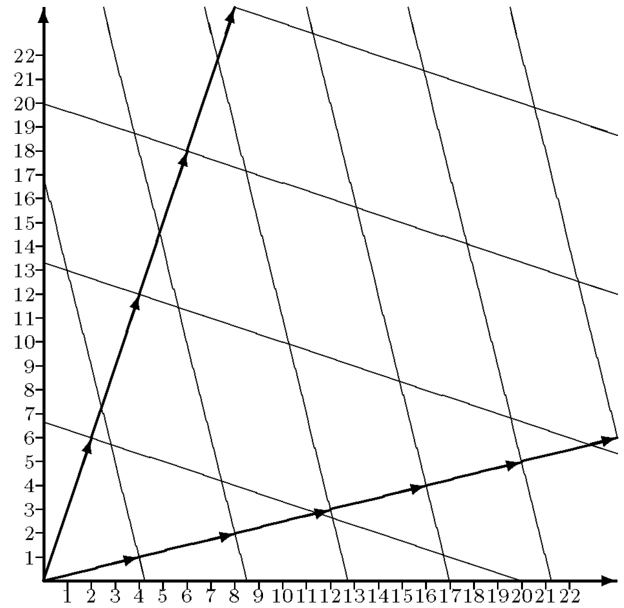


Figure 1.11: Using vectors to define cutting hyperplanes

form a basis for the iteration space, and the resulting dividing is a *tiling* of the iteration space.

In general, however, it is not necessary to have a tiling of $\mathcal{I}$ if we are only interested in data motion; a dividing will suffice, so long as we have a tiling of $\mathcal{D}$. This is because a division of the iteration space with unlimited length in some dimension is acceptable so long as the data requirements for localized streams of the division are limited to some controllable amount. In Figure 1.12, a one-dimensional stream `A[j]` is referenced in a two-dimensional iteration space, consisting of an `i`-loop and a `j`-loop. Tiling the `j` loop is sufficient to limit the data required for each tile, at least for this stream. Tiling the `i` loop does not help at all.

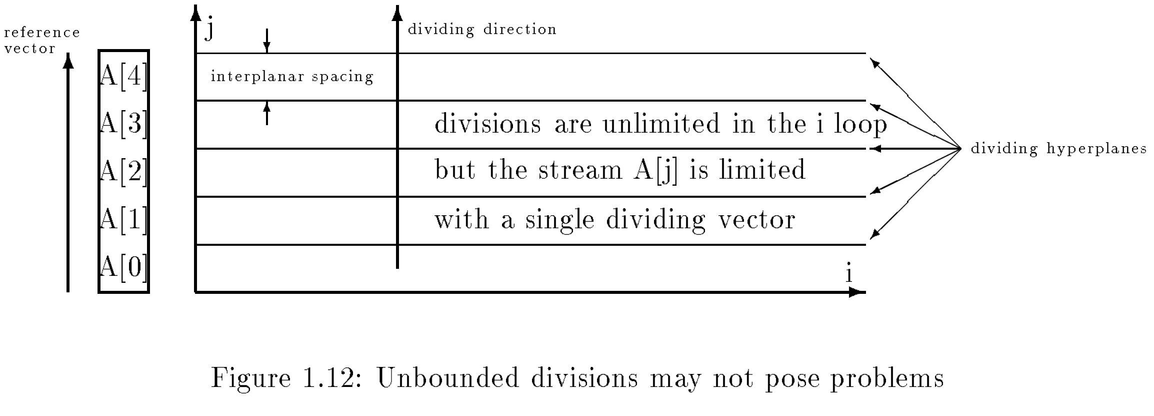


Figure 1.12: Unbounded divisions may not pose problems

The amount of data referenced by the iterations of a division must be limited to a finite (compiler-controllable) size. This will be the case if no reference vector is perpendicular to all dividing vectors, that is, if $\forall i \; \exists j R_{i,*} \cdot B_{j,*} \neq 0$ (the reference matrix $R_{\mathtt{v}.k}$ is abbreviated R here for notational purposes).

To tile a subspace of $\mathcal{I}$, the compiler must first transform the loop nest so that $\mathcal{D}$ is spanned by λ loops, and $n-\lambda$ loops have their direction vectors orthogonal to $\mathcal{D}$. If possible, the orthogonal loops should be moved innermost, increasing the amount of computation performed in each tile. Because this is not always possible, and because it is notationally more convenient, in the rest of this thesis the entire iteration space is tiled. From now on, B is a $n \times n$ matrix.

1.4.2 Dependence constraints

A division of the iteration space is completely executed before another division is worked on; so for this (sequential) case we have an atomicity constraint on the divisions: each division must be such that once all its inputs are ready, it can be executed start-to-finish without interruption. We can ensure this by requiring that dividing basis vectors $B_{i,*}$ satisfy the filtering equation:[4]

$$\forall j\colon B_{i,*} \cdot D_{*,j} \geq 0 \tag{1.1}$$

We can re-write this as $BD \geq 0$. We are choosing B from the set of legal sequential ordering vectors. This ensures that we could safely execute along each basis vector $B_{i,*}$, because every entry of every dependence vector will be positive in the new basis (the transformed dependences are given by BD). The loops of the new basis are therefore fully permutable (and thus tilable). Thus we want to choose dividing vectors from the set $\mathcal{C}^*(D)$.

$\mathcal{C}^*(D)$ differs from $\mathcal{C}(D)$ in that while we cannot choose legal *sequential* ordering vectors from $\mathcal{C}^*(D) - \mathcal{C}(D)$, we *can* choose partitioning directions from that set. If the partitioning direction is in $\mathcal{C}^*(D)$ and not in $\mathcal{C}(D)$, there will be a dependence along the border of a partition as in Figure 1.13. The partitioning direction is (1,-1), which means the tile boundaries lie along (1,1). There are dependence vectors also along (1,1). These dependences do not cross the partition boundaries, but run along it. This does not preclude a linear

---

[4] The condition given is sufficient to prevent dependence violations but is not strictly necessary; there are some dividings that are valid and yet do not meet this constraint [29].

scheduling of either the iterations within a partition or of the partitions themselves.

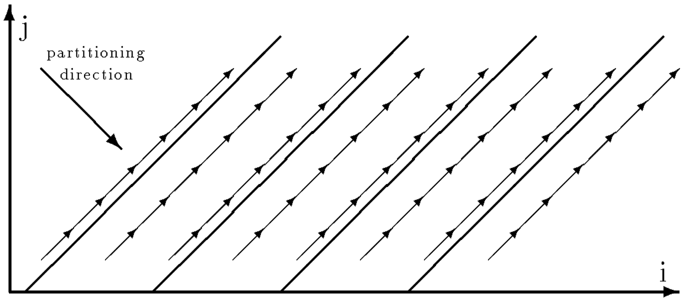


Figure 1.13: Dependence vectors parallel to partitioning hyperplanes

1.5 Approach

The goal of the compiler is to transform a loop nest written for an infinitely large memory into a new loop nest that uses the memory hierarchy to greatest advantage, by copying data from one level to another when needed, but using locality to reduce the total number of copies required.

The basic tool used in this thesis is tiling. Substantial effort is spent to find the tiling that minimizes execution time. First, a set of candidate tiling basis vectors is formed. For every possible basis that can be formed from this set, the best schedule is selected, the best tile shape is computed (i.e., the hyperplane spacing factors are selected to minimize execution time), and the execution time is estimated. The tiling with the smallest cost is selected from all possibilities given the candidate set.

A prototype compiler was implemented, which automates most of the work involved. Specifically, the prototype generates the candidate set, selects each possible basis, finds a schedule for the basis, and builds the cost model from which the tile size factors are chosen. Building the cost model requires then compiler to transforms the loop nest, finding new loop bounds in the new basis.

Due to time constraints, the cost model solver was not implemented, nor was the final mechanical step of strip-mining the loop bounds given the tile size factors (which are the blocking factors for the loops). In later chapters, we discuss the numerical stability of the cost model, showing that if the compiler has complete information, an optimal solution is easily obtained. If the compiler cannot determine the loop bounds at compile time, the cost model can be solved using approximations of the loop bounds without significant degradation of solution quality.

Care is taken in several areas to ensure that execution time is minimized. The buffering schemes for moving the data back and forth between M_1 and M_2 are chosen to be as efficient as possible in terms of storage required. The order in which the tiles are executed is chosen to maximize the amount of data that can stay resident in M_1, thereby minimizing the slow memory bandwidth required. Finally, the relative fraction of M_1 used for buffering each stream is not fixed, but is decided by the compiler to provide the maximal amount of computation per slow memory access.

1.6 Outline of the thesis

This chapter described the problem to be solved, and laid a foundation for its solution. The reader should have a fundamental grasp of iteration spaces, reference matrices, dependence vectors, and tiling. This theory is more or less common to all works in data motion management using tiling.

Chapter 2 discusses earlier work at solving similar problems. Some general work in compiler theory is discussed first. Approaches to managing data motion not based on tiling are discussed, and finally earlier work using tiling is described.

Chapter 3 develops the basic cost model used throughout the thesis. The cost of moving data is simply the amount of data to be moved for each tile times the number of times that amount of data must be moved. Both of these parameters are expressed in terms of the vector of tile sizes.

Chapter 4 develops the first part of the cost model: how much data must be moved for a tile. This includes developing the address translation from the memory space of the full data set in M_2 to the buffer memory space in M_1.

Chapter 5 develops the other part of the cost model: how many times the data must be moved. It addresses scheduling the tiles to minimize data motion taking advantage of the locality between tiles. Finally, it describes how to find a formula for the number of times data will be moved in terms of the tile size vector.

Chapter 6 describes how the cost model is evaluated to find the optimal value for the tile size vector. This requires detailing how the loop bounds are transformed from the source space to the new basis space. Once the loop bounds are transformed, polynomial arithmetic needed to evaluate the loop bounds is discussed. A complete example is given showing how the cost model is developed from source code to finished transformed code, and the chapter concludes with some discussion of the optimality of the techniques used.

Chapter 7 evaluates the techniques used by applying them to several well-known scientific loop bounds. Specific comparisons between this work and previous work is given, showing specifically what problems the new techniques address that the old techniques did not.

Chapter 8 reiterates the contributions made by this work, and the conclusions that can be drawn from it. It also points out several new areas of research that have been identified as a result of the work of this thesis.

Chapter 2

Related work

The related work is divided into three parts. The first section describes general work in compiler theory. The second part describes approaches to compiler management of data other than tiling, and the last section describes the body of related tiling work.

2.1 Compiler theory

The first two parts of this section describe work in dependence analysis and code generation techniques that are used later in the thesis. The last part describes several approaches to parallelization that have been taken by different researchers, for contrast to the method of parallelization by tiling used in this work.

This work is done in the context of optimizing compilers for imperative languages. The reader who is not familiar with optimizing compilers should become familiar with them before proceeding. Wolfe's book[58] is a good place to start. In particular, the reader should be familiar with loop transformations such as unrolling, jamming, strip-mining, and interchanging. The reader should also be familiar with standard data flow analysis (Aho, Sethi, and Ullman's book[3] is good) and data dependence analysis (see below).

2.1.1 Dependence analysis

Tiling cannot be accomplished without effective dependence analysis. The standard reference for dependence analysis is the book by Bannerjee[7]. This standard baseline has been improved in different ways by other researchers. Ribas[47] describes adding *rebounding* *facets* to turn non-constant dependences into constant ones in some loops. Wolf and Lam represent dependences as lexicographically positive vectors, which simplifies transformation theory for non-constant vectors.

Pugh[40, 41] developed an algorithm called the Omega test for solving the integer linear programming problem that is at the core of dependence analysis. This test serves as the basis for the dependence analyzer of the Fx compiler, of which this work is a part.

2.1.2 Code generation

When a tiling basis is chosen, we need to transform the source iteration space into the new iteration space, and then applying strip-mining to the resulting nest. Li and Pingali[35] describe exactly the transformation required. In Section 6.1 we describe the results of this paper in detail. Ancourt and Irigoin[4] describe techniques for scanning the integer points in a polyhedra using `DO` loops, which could be used to perform the same task. For generating loop bounds in the tiled code, Ancourt and Irigoin's method is inferior to Li and Pingali's, because Li and Pingali scan exactly the points required, while Ancourt and Irigoin scan the convex hull of the points required. When generating fetch and store loops, however, copying the convex hull of the data may be be cheaper, because Li and Pingali's method visits each *iteration* point once, while Ancourt and Irigoin's method visits each *data* point once. When the same data is referenced several times by different iterations, fetching the convex hull may be preferable.

Part of the loop transformation process is performing Fourier-Motzkin elimination[13]. We use a slightly modified version of Duffin's methods for eliminating extra inequalities[14].

2.1.3 Parallelization

Tseng[52] automates mapping of programs to distributed memory machines, by using programmer hints to the compiler in the form of distributed arrays called `DARRAY`s. He also uses programmer hints to simplify dependence analysis, but this could be automated as well. The programming language shows a strong resemblance to Fortran D[23].

Ribas[47] demonstrated the feasibility of automatically generating code for systolic arrays from nested loop algorithms. The mathematical approach to compilation in that work was the inspiration for this work.

Kung[32] describes nine different computational models for linear processor arrays. Some of them, such as the pipeline model, are well-suited to intratile parallelism. Using such a model, the entire array is used as a single powerful processor. All the processors work on the same tile simultaneously. Because systolic arrays can move data directly from the communication hardware (the "systolic pathway") into the arithmetic units, systolic parallelism can increase the net data bandwidth into the arithmetic units, turning a program whose execution time is limited by memory bandwidth into one that is limited by computation bandwidth. Removing the memory bottleneck in this way is a powerful tool. One important criterion for the techniques developed in this thesis is that they do not prohibit the use of systolic parallelism within a tile.

Moldovan and Fortes[16] discuss other methods of generating systolic algorithms from nested loops. The transformation techniques they use are incorporated into, and surpassed by, the code generation techniques of Li and Pingali discussed in Section 2.1.2.

Sussman[51] describes techniques that allow a compiler to choose among several execution models for mapping programs onto distributed memory machines. He shows that a compiler can choose among data partitioning techniques and computation partitioning techniques, including block and interleaved data partitioning, and loop body pipelining. Since these techniques cannot be exactly modeled on a complex machine, he uses an upper bound and a lower bound function for modeled execution time.

Pingali and Rogers[38] use programmer-supplied data decompositions to drive parallelization. They try to compile the program so that computation is executed on the processor where the data is resident. Their compiler supports data distributions using wrapped rows, wrapped columns, and square blocks. They use compile-time information when possible, and rely on run-time resolution when necessary.

2.2 Other approaches

This section describes approaches to compiler management of the memory hierarchy other than tiling. First some general array-handling techniques are discussed. The next section examines work on compiler cache management: first cache bypass and then software prefetching techniques.

2.2.1 Array management

Callahan and Kennedy describe *scalar replacement,* a method that allows register allocators that do not handle arrays to keep some array elements in registers. They also describe using loop unroll-and-jam to improve the effectiveness of their method. They hint that tiling to improve locality may surpass the performance of their method. Scalar replacement is only necessary when the compiler's flow analysis is insufficient to perform register allocation of subscripted variables. Maydan *et al.*[36] describe a method for improving standard data-flow analysis that is more general.

Gupta and Kajiya[21] describe techniques for laying out data in memory so that executing the code results in accessing sequential addresses in memory. They provide evidence that the compiler can usually determine which axis of an array is scanned fastest. Organizing data to match the scanning order of loops increases spatial locality. This method does not improve temporal locality for loops whose data does not fit into the lowest level of the memory hierarchy, because the accesses themselves are not reordered, just the mapping of addresses is changed.

Wholey investigates trade-offs between parallelism and locality in mapping data onto parallel machines. Array axes that are aligned in the iteration space are bundled together at compile time; at run time, a search is performed that computes the best distribution of data elements to processors. A cost model that takes into account both parallelism and communication costs is used. The techniques for finding tiles sizes presented in this thesis are more exact since they do not rely on data sizes being powers of two. The cost model used in this thesis also takes into account locality within each processor. Data mapping is the primary goal addressed by Wholey's work. In our work, data mapping is done by scheduling the tiles onto the processors, and by choosing tile sizes.

Balasundaram *et al*[6] describe an interactive system for partitioning and distributing data. This approach does not address data locality within a processor, but could be extended with tiling for locality. The general approach of interactively advising the user to make changes in his program is a fine idea for tuning a program to a particular architecture, but makes the code less portable. Fully automatic techniques are necessary for portability.

Jalby *et al*[18, 17, 15] describe a method for computing the number of elements that would have to be held in fast memory for re-use to occur. This could be used to compute the number of unrolls needed in Callahan and Kennedy's method. The term "uniformly generated", used to describe array accesses with the same coefficients but possibly different constant terms, originates in the work described by these papers.

2.2.2 Cache work

This section describes work on cache bypass and cache prefetching strategies for compilers. Cache bypass keeps the cache from being flushed by large arrays. This forces accesses to arrays to operate at slow memory speeds, but has the advantage of leaving in the cache those data items that do exhibit locality. Software prefetching attempts to hide slow memory latency by prefetching data items before they are needed. Prefetching does not reduce the slow memory bandwidth requirement of a loop nest. If the slow memory is a bottleneck, software prefetching will not be effective.

Chi and Dietz[11] describe the generation of cache-bypass information. Some processors allow various control over cachability: pages can be marked uncachable, address spaces can be marked uncachable, or individual references can be marked uncachable.[1] A compiler can scan through instruction traces generating cache/don't cache information for each reference. Bypassing the cache for references that are known to be poor candidates for caching can greatly improve performance. Bypassing avoids pollution of the cache. This keeps cachable references present, and it increases the effective size of the cache since many references never go into it.

Porterfield *et al*[39, 9] discuss using predecessors of tiling (*peel-and-jam* and *strip-mine, skew and interchange*) for reducing the number of cache misses, and software prefetching for reducing the effective cost of cache misses that are not eliminated. The tiling part of this work is improved on by that of Wolf and Lam (see below).

Gornish, Granston, and Veidenbaum[19] investigate prefetching in shared-memory processors. In particular, they compute the earliest point at which a data item can be prefetched. They also give simulation results to evaluate the effectiveness of their method.

---

[1] No current processors are known to provide cachability on a per reference basis, but there is enough instruction encoding space to implement it on Hewlett-Packard's Precision Architecture, version 1.1[22].

2.3 Tiling

Tiling is a loop restructuring transformation. Usually it is aimed at increasing the locality of a loop nest. Kung and Hong[24] show that the computation-to-I/O rate of a program can be bounded. Kung later uses this theory to show that increasing the computation rate of a processor array without increasing its I/O rate requires more memory per processor to maintain full utilization[31]. The bounds on the computation-to-I/O rate are a fundamental limit on the effectiveness of tiling for locality. In particular, in evaluating the new tiling techniques in Chapter 7, these bounds help to explain why the new techniques succeed when they do. The bounds can also be used to explain why in some cases, only a constant factor improvement is possible.

2.3.1 General tiling work

Tiling for locality has been extensively developed in optimizing compilers. The origins are found in Abu-Sufah's work to increase locality in paging systems[1]. This work split and fused loops to minimize the number of page frames required to execute a program with only a few page faults. Strip-mining was applied to loops so that once a page was brought into memory, as much computation as possible was done on that page before it was returned to disk.

The next advance in tiling work was Irigoin and Triolet's use of hyperplanes to partition the iteration space[29]. This changed a loop transformation problem into a geometric one: choosing a basis for the iteration space such that all basis vectors are positive with respect to each dependence vector. This lead to a concept called a *dependence cone,* which is the set of all legal scheduling vectors. Wolfe[59] describes roughly equivalent functionality in terms of loop transformations instead of the more theoretical approach of Irigoin and Triolet.

Carr and Kennedy[10] studied tiling (they call it blocking) loops for linear algebra algorithms. The key insight of this work is that many linear algebra algorithms that use pivoting have dependences that prevent adequate tiling. Blocking these algorithms requires more than simple loop transformations.

Schreiber and Dongarra[48] advanced tiling by suggesting a new method of choosing the loop transformation: they pick a basis for the transformed iteration space from vectors lying inside the dependence cone. More specifically, they start with a subset of the rays of the dependence cone, and modify this basis to make it orthogonal. While their argument for orthogonality is convincing, and it certainly holds for matrix multiply, orthogonality of the tiling basis is not generally optimal; in the next chapter we will show that having *scheduling* vectors perpendicular to the basis vectors is the right abstraction. We note, however, that for many common linear algebra programs, the two ideas coincide.

Note that Schreiber and Dongarra's method of choosing the rays of the dependence cone as a new basis for the iteration space requires dependences which are distance vectors; the method cannot be directly applied to loops with direction vectors. They choose the basis to maximize reuse based on a simple model of the program. In their model, the amount of data accessed by a tile is proportional to the surface area of a tile. This is certainly true of $(n-1)$-dimensional arrays in n-dimensional loops, such as are found in matrix multiply (their primary example), but it does not hold in general. They do choose non-square tile shapes using a method similar to the one we present. They also discuss locality between tiles. Their work is largely restricted to uniprocessors, although they do discuss wavefronting tiles for parallelism.

2.3.2 Tiling for cache locality

Wolf and Lam have done considerable work on the problem of tiling nested loops for machines with caches[33, 54, 55, 56, 57]. The best reference is Wolf's thesis[57]; although long, it contains everything that the papers contain, plus more space is devoted to clarification and examples. An important theoretical contribution of this work is an advance in dependence representation. Dependences are represented as a combination of distance and direction vectors, and are required to be lexicographically positive. They use unimodular matrices to model loop transformations. Loop nests are transformed to get sequences of fully permutable loops. Fully permutable loops can be freely interchanged because all dependences are satisfied regardless of the nesting order of the loops (because the dependences are positive in every loop, not just in the outermost loop).

They strip-mine fully permutable nests to form tiles. They choose the tile size so that there is no cache interference within a tile. This typically results in using a small fraction of the cache space. They always use square tiles. Square tiles are not generally the optimal choice, but changing the loop nest from one that usually misses in the cache to one that almost always hits is such a significant speedup that a suboptimal shape choice is not a critical consideration: tiling for locality with square tiles can reduce execute time by orders of magnitude; relative to square tiles, optimally-shaped tiles result in a slight performance increase of a small constant factor.

Wolf and Lam execute tiles in parallel using DO-ACROSS parallelism[12]. They do not consider scheduling tiles to optimize locality between tiles.

In contrast, in this work we target RAM memories instead of caches: local (private) memories in a distributed memory machine, or on-chip RAMs in machines like the Transputer. There is no cache interference possible. We therefore choose tile sizes as large as possible subject to the size of local memory. We choose tile shapes to minimize the number of non-local accesses.

Wolf and Lam suggest copying data into a linear buffer to reduce cache interference. Skewed rectangular buffering, discussed in section 4.5, is closely related to this problem; fetching a skewed buffer is essentially a gather operation, copying data into consecutive locations in fast memory.

This work also addresses scheduling tiles for intertile locality, which Wolf and Lam do not. Intertile locality is a secondary effect compared to intratile locality.

2.3.3 Tiling for minimal communication

Ramanujam and Sadayappan[42, 43, 44, 45, 46] tile to reduce communication in distributed memory parallel computers. They target machines with high communication latency, as opposed to systolic arrays, which have low communication costs. They choose a subset of the rays of the dependence cone as tiling vectors (they call the rays *extreme vectors*), which requires constant dependences. They use simple wavefronting for parallelism. They offer a formula for determining the size of tiles in 2-dimensional iteration spaces. They also develop a test for determining if there is a communication-free partition of data to processors.

Since their objective is solely to minimize communication, they use a much more abstract model of data motion: they measure communication by taking the dot product of the dependence vectors and the tiling vectors. Because they assume constant dependences, this is a good approximation. In our work, we do not attempt to choose tiling vectors directly to reduce communication. Since we are targeting multiprocessor systems with memory hierarchies, we find tiles small enough to fit in the fast memory of a single processor. We reduce communication by scheduling these tiles onto the processors in the way that maximizes locality. Since we make our final selection based on the total execution cost, our method is always at least as good as theirs. Because we also include slow memory fetch costs in our model, our results should surpass theirs in cases where memory locality within a processor is more important than minimizing communication.

2.3.4 Tiling for locality given a data distribution

Li and Pingali[34] restructure loops for locality in Fortran D. The user describes how to decompose data among processors. Rather than directly applying the "owner computes" rule, the compiler restructures loops so that executing the outermost loop in parallel results in maximal locality within each processor. The inner loops are tiled if necessary so that non-local accesses are block accesses. The transformed iteration space is chosen directly from the *data access matrix,* that is, from the set of filtered reference vectors.

In this thesis, we tile for data locality even for streams that are held entirely within a single processor. Li and Pingali's work does not address this, pointing out instead Wolf and Lam's work on data locality within a processor. Their emphasis is on problem decomposition.

In this work, both inter-processor and intra-processor locality are addressed, simultaneously, using the tiling as the mechanism for achieving both. We therefore keep a more detailed model of the reference stream: rather than simply generating the set of all reference vectors, we keep a reference matrix associated with each array reference. This allows us to compute the number of nonlocal accesses required by a transformed loop nest exactly.

Li and Pingali describe a method for completing a tiling basis given a partial basis. The work described in this thesis avoids this problem by tiling the data space rather than the iteration space. We are guaranteed that there are enough reference vectors to span the data space, so completion is not required.

2.4 Contributions of this work

This thesis investigates compiler techniques for managing data motion through memory hierarchies without support (or interference) from the hardware. This problem is more difficult than simply tiling to improve locality, because the compiler must also perform all the duties of a cache: it must decide what data to bring into fast memory, where to put it, and when it should be returned to slow memory.

The problem is also more difficult than that of standard overlays, because it requires loops in the program to be restructured. Furthermore, this restructuring should be done in a way that maximizes locality of reference. Standard overlaying techniques do not address these issues.

The tradeoff between parallelism and locality is also investigated. In our work, both parallelism and locality contribute to reduction of execution time. By using a cost model that incorporates both, and by selecting a tiling basis to minimize this cost function, the tradeoff between parallelism and locality can be neatly addressed.

This work also solves the problem of automatically choosing optimal tile sizes in each dimension. This alleviates the problem of deciding which loops to tile, because all loops can be tiled, and the tile dimensions will be set so that loops which need not have have been tiled can be returned to their source from with a simple post-tiling optimization step.

To manage data motion, data accesses must be carefully modeled by the compiler writer. The model of data streams generated by different array references in loops is simple and powerful. Reference matrices are an effective way to capture exactly the locality information needed by a compiler. The following chapters show how this powerful model of how data spaces relate to the iteration space can be used in a methodology for managing data motion in machines with software-controllable memory hierarchies.

Chapter 3

Cost model fundamentals

The goal of this thesis is to produce a set of compiler techniques for managing data motion through the memory hierarchy. This chapter and the next three are devoted to the development of the techniques. This chapter gives an overview of the approach. The first section discusses the approach in general terms. The next section describes the execution model describing where data resides, how it is moved, and how the computation is performed. Based on this execution model, cost criteria are developed for comparing different tiled loop nests. Finally, a specific cost model is developed.

3.1 Overview

The goal of an optimizing compiler is to generate the best code possible without spending an unreasonable amount of time. Any tiling will result in intratile locality. A simple compiler can choose any legal tiling basis, and choose the tile size to be the largest rectangular tile that fits in M_1. An optimizing compiler should expend a little extra effort to choose the best tiling basis and then to choose the best tile size in each dimension.

The space of all possible legal tiling bases is infinite, so we cannot possibly search the entire space. The criterion used to evaluate the basis choice, execution time, is not simple enough to allow analytical choice of a basis from this infinite space. The compiler instead constructs a set of candidate tiling vectors, and evaluates each possible combination of those vectors.

3.1.1 Candidate tiling vectors

There are several obvious choices for candidate vectors. Loop index vectors are the simplest possibility. Tiling on loop index vectors corresponds to strip-mining each loop in the original nest, and interchanging the controlling loops outwards. The set of loop index vectors is always the set of rows of the identity matrix, and so is written I.

Dependence vectors are another possible source of tiling basis vectors. For the purposes of choosing candidate vectors, the compiler includes input dependences as well as flow-, anti-, and output-dependences. The set of dependence vectors of the latter three types is written D. When augmented with input dependences, the set is written D^+. Dependences point out reuse in the iteration space. Choosing dependences will allow the maximum possible locality between different tiles, as will be shown in Chapter 5.

There are two problems with using dependences as candidate basis vectors. The first problem is that the rank of the dependence set is often less than n, even when input dependences are included. This does not prevent us from using dependences in the candidate set, but the dependence vectors are not sufficient by themselves for tiling, even though they do point out all the available reuse.

The second problem with using dependences as candidate vectors is that tiling basis vectors must be integer-valued. When dependences are distance vectors (i.e., every entry is a range consisting of a single integer), or can be represented using distance vectors, the dependences can be used directly as candidate basis vectors. Non-constant dependences (direction vectors) cannot be used as candidate vectors unless they can be converted into integer-valued vectors.

Reference vectors are another good choice for candidate vectors. Since they point in the direction of increasing subscripts for each dimension of an array, cutting with hyperplanes perpendicular to them results in tiles that reference rectangular subarrays. The set of all reference vectors is called V.

The class of *extreme vectors,* or rays of the dependence cone, are also guaranteed to be legal. We denote this set of vectors E. Ramanujam and Sadayappan[42, 43, 44, 45, 46] use these vectors for tiling, and Schreiber and Dongarra begin with a subset of these vectors, modifying them to get an orthogonal basis. Including these vectors in the candidate set allows the extremes of the space of legal orderings to be searched.

The last class of vectors to consider as candidate vectors is $V^\perp$. These are vectors which are perpendicular to all the reference vectors of a stream. The vectors of $V^\perp$ are essentially (augmented) dependence vectors, but they are much easier to derive, and are always constant integer vectors.

Unfortunately, of the above classes $(I, D^+, V, E, V^\perp)$, only I and E are necessarily *legal* vectors; the vectors of D^+, V, and $V^\perp$ must be filtered against the dependence set.

3.1.2 An example

Since the compiler will evaluate every linearly independent subset of the candidate vector set, including so many different kinds of vectors may seem costly. Fortunately, in most real programs several of the sets discussed above overlap considerably. Consider the example of matrix-matrix multiply, shown in Figure 3.1.

```
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            c[i,j] = c[i,j] + a[i,k] * b[k,j];
```

Figure 3.1: Matrix-matrix multiply

The index vector set in any program is I, the identity matrix. For matrix multiply, I consists of the three vectors `i`, `j`, `k`. The dependence vector set of matrix multiply contains the single vector $(0, 0, 1)$, a flow-dependence in the `k`-loop on `c`. The augmented dependence set is $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$, because there are input dependences in the `i` direction for `b[k,j]`, in the `j` directions for `a[i,k]`, and in the `k` direction for `c[i,j]`. The reference vector set consists of the three vectors `i`, `j`, `k`. Since the dependence matrix is not of full rank, the rays of the dependence cone include a basis for the space spanned by D (in this case, `k`), and a basis for the null space of D (in this case, `i` and `j`). The vectors of $V^\perp = I$ because $V = I$. All of these are legal vectors; the union of all the sets is just I. There is only one tiling basis choice in this case, I.

3.2 Execution model

The final code must iterate over the all the iterations in the original iteration space. The iterations are divided into groups called tiles using the methods described in the previous chapter. A set of outer loops (called controlling loops) will iterate from tile to tile. The loop body of these outer loops will first fetch from M_2 into M_1 the data required to execute the division (or a superset of the data required). Inner loops perform the computations of the tile. Finally, the data that has changed is stored back. Data motion that is loop-invariant in the innermost controlling loop is moved outward to the first controlling loop in which it is not invariant.

When there is more than one write-reference to a single variable, a coherency problem becomes apparent. Since each stream is buffered separately, we must ensure that writes to a particular array element are copied into each buffer that currently holds that element. The dependence vectors give us precisely the information that we need. We must be certain that any dependences from one stream to another stream are satisfied either by the ordering of the computations, so that there is never any element in common between any two buffers for the same variable, or else we must insert code to perform the necessary updates to each stream when they do overlap. For the time being, we will ignore this problem, since it seems easily solvable; instead we concentrate on the costs of execution that lead us to choose our tiling basis.

Once the tiling basis is chosen, we can generate code in terms of symbolic interplane spacing factors. The tiled code for matrix-multiply is shown in Figure 3.2.

```
for i = 1 to n by $\beta_i$ do
  for j = 1 to n by $\beta_j$ do
    for k = 1 to n by $\beta_k$ do
      for ii = i to min (n, ii+$\beta_i$-1) do
        for jj = j to min (n, jj+$\beta_j$-1) do
          for kk = k to min (n, kk+$\beta_k$-1) do
            c[ii,jj] = c[ii,jj] + a[ii,kk] * b[kk,jj];
```

Figure 3.2: Tiled matrix-matrix multiply

3.3 Cost criteria

Execution time, or at least estimated execution time, is always the final arbiter in compiler decisions. In tiling, however, the computation time remains the same, since the same amount of computation will be performed by any tiled loop nest. Furthermore, since all the loops will be tiled regardless of basis choice, the overhead of each tiled loop nest is the same. Different tiling bases can therefore be compared solely on the amount of time spent doing slow memory accesses.

Furthermore, in the machine model of Figure 1.2, data cannot be operated on in M_2, so every data item accessed in a loop must be brought into M_1 *at least once.* This cost will be the same for any tiled loop also. The difference between two different tiling bases is then the sum of two terms: the number of times a data item is brought into fast memory *after* the first time, and the number of times a data item is brought into M_1 and not used at all. The first kind of access is called a *refetch.* The latter is called an *overfetch.*

Refetching is often necessary for at least some data items. In matrix-matrix multiply, for example, each row of the `a` matrix must eventually be co-resident with every column of `b`, and every column of `b` must eventually be co-resident with every row of `a`. If M_1 is too small to contain an entire matrix, some refetching must occur.

Overfetching results when the compiler for some reason fetches data that is not used. Data might be overfetched by the hardware that requires accesses to have a minimum size (like a cache line, or a disk sector), or it might be overfetched because of code generation tradeoffs. Since the source code loops can be very complex, using the same loop structure as the computation loops to fetch or store data may be inefficient. We can instead construct a new set of loops, which have exactly as many loops as the data has dimensions, to fetch the data. In doing so, some information is of course lost. Following Irigoin's method [4], we will fetch the convex hull of the data elements referenced. This can introduce overfetch. If data is fetched that will never be used in the division for which it was fetched, that data is said to have been overfetched. This can happen, for instance, if the convex hull of data used in a division contains data items that are not used in that division (see Figure 4.5). The compiler has the choice of fetching the convex hull of the data required using a simple loop guaranteed to fetch each data item only once, or using the source loop. The source loop will fetch exactly the data needed, but may fetch the same element several times. The cost of the savings in not fetching the same data multiple times is that some data may be fetched that are never used.

One last cost that must be incorporated is an indirect cost: the cost of overallocation. When space is allocated in fast memory that is not used for data used in a tile, that space is *overallocated.* The cost of overallocated space is the number of cycles of slow memory access that must be performed that would not have to be performed given a more efficient allocation.

3.4 Cost model

Since the cost of the computation is the same for any tiling, the relative cost of each tiling can be compared by measuring the time spent accessing slow memory. Each stream contributes its own portion to the total cost. The cost of a stream s is the number of times a block of data must be fetched (or stored) for that stream, denoted ρ_s, times the amount of time spent fetching the block. The fetch time per block is modeled as a fixed access time per block, plus a linear cost per word. Letting c_b be the cost per block, c_w the cost per word, and μ_s the number of words allocated in M_1 for s, the total cost is

$$\sum_{\mathrm{s} \in \mathrm{streams}} \rho_s * (c_b + c_w * \mu_s) \tag{3.1}$$

The size of the tiles is dependent on the buffering mechanism used by the compiler, and on the relative fraction of fast memory dedicated to each stream. The buffering mechanism determines the efficiency with which data is packed into M_1. The relative fraction of M_1 spent on each tile determines the size of each block. The number of block fetches per stream depends primarily on how the tiles are scheduled for execution.

Chapter 4 describes buffering techniques, and defines μ_s as a function of the tile size vector $\vec{\beta}$. Scheduling is discussed in Chapter 5; the schedule determines ρ_s, also as a function of $\vec{\beta}$. Chapter 6 describes how the cost model is evaluated once μ_s and ρ_s are known in terms of the tile sizes.

Chapter 4

Buffering schemes

In this chapter we describe the techniques used to store data in M_1. This requires deciding, for each tile in the iteration space, which data is used by that tile and must be fetched, and where in M_1 that data will be stored. Two facets of the cost model are explained in this chapter: the amount of data fetched for a tile and the amount of M_1 space dedicated to each stream. Both are computed in terms of the symbolic tile size vector $\vec{\beta}$. The subscript functions needed for code generation are also described. The compiler needs a subscript function for the full-size arrays in M_2, and another subscript function for the temporary arrays in M_1.

The first section of this chapter describes how the iteration space is transformed prior to generate tiles. The second section introduces the machinery required to discuss buffering techniques. In the third section, the mechanism for choosing among buffering methods is discussed, and finally there is a section on each method.

4.1 Transformation theory

The compiler has to generate code to execute a tiled loop nest from the source loop nest. Rather than trying to directly solve for the loop bounds given arbitrary normal vectors to the cutting hyperplanes, the compiler transforms the iteration space so that the normal vectors are elementary vectors (they point along the axes). Each loop can then be strip-mined, and the inner loops of each new pair are interchanged inwards to form tile loops.

The n-dimensional vector $\vec{p}_i$ denotes a point in the source iteration space. A point $\vec{p}_t$

in the transformed space is related to the original iteration space by

$$\vec{p_t} = B\vec{p_i} \tag{4.1}$$

which can be rewritten $\vec{p_i} = B^{-1}\vec{p_t}$. It is clearer for expository purposes if a new set of loop index variables is used in the transformed loop, at least in the general case. The vector of loop index variables in the source code is denoted $\vec{\imath}$; in the transformed space the loop index variable vector is $\vec{t}$. Note that $\vec{t} = B\vec{\imath}$ must hold for these vectors.

4.2 Buffering theory

Buffering techniques are applied independently to each stream. For each stream, a size requirement in terms of $\vec{\beta}$ is computed. These size requirements are used together to compute the relative fraction of M_1 to be dedicated to each stream after scheduling is performed. The rest of this chapter deals with size functions and address generation for a particular stream.[1] The stream is generated by the kth reference to `v` in the loop nest, and is denoted `v`.k. The reference matrix associated with this stream is normally denoted $R_{\texttt{v}.k}$, but since buffering deals with a single stream, $R_{\texttt{v}.k}$ is abbreviated for the rest of the chapter as R.

The matrix R is a $\delta \times n$-dimensional matrix, where δ is the number of dimensions of `v`. R maps iterations in the source loop nest to points in the data space of the referenced array. The element of `v` used by an iteration point $\vec{p_i}$ is $\vec{s}$, a δ-dimensional vector. The relationship between $\vec{p_i}$ and $\vec{s}$ is given by

$$\vec{s} = R\vec{p_i} \tag{4.2}$$

Combining 4.1 and 4.2 results in

$$\vec{s} = RB^{-1}\vec{p_t} \tag{4.3}$$

The matrix R maps source iterations to array elements. The iteration-to-array element transformation in the transformed space is given by the matrix $S = RB^{-1}$, which is also a

---

[1] If the target M_1 memory is a register file which cannot be referenced via indexing, loop unrolling mustbe applied since each register must be specifically named. Wolf[57] describes this process in detail.

$\delta \times n$-dimensional matrix. In the transformed code, references to $\mathtt{v}.k[R\vec{\imath} + \vec{c}]$ are replaced with $\mathtt{v}.k[S\vec{t} + \vec{c}]$. The matrix S is called a *subscript matrix.* Rows of S are called *subscript vectors.* Subscript matrices and subscript vectors are the transformed-space counterparts of reference matrices and reference vectors.

While subscripting functions for the source arrays are easily given in terms of the iteration space vector $\vec{p}_i$ ($\vec{s} = R\vec{p}_i$) or the transformed iteration space vector $\vec{p}_t$ ($\vec{s} = S\vec{p}_t$), it is much more convenient to give the buffer-array subscripts in terms of the iteration space vector *within a tile,* which is denoted $\vec{r}$. An example will help to illustrate the point. Figure 4.1 shows a two-dimensional loop nest. The grey area represents the extent of the `b` matrix; the iteration space is limited to the lower triangle. The n-vector $\vec{\imath}$ is the source iteration space vector; for the code in the figure, $\vec{\imath} = \begin{bmatrix} \mathtt{i} \\ \mathtt{k} \end{bmatrix}$.

```
for i = 1 to 12 do
  for k = 0 to i-1 do
    w[i] = w[i] + b[i,k]*w[i+k];
```

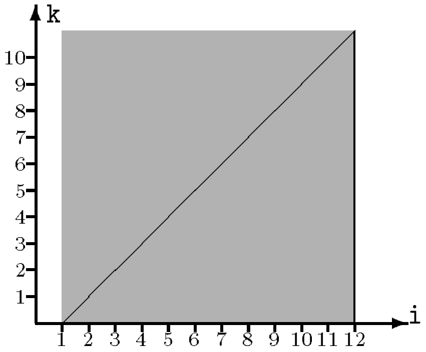


Figure 4.1: A two-dimensional loop nest

When transformed by B, $\vec{\imath}$ becomes $\vec{t}$, which is also an n-vector. Imagine that the compiler will skew the inner loop prior to tiling so that all references to the variable `w` are expressions involving only a single loop index variable. The new basis is $B = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$. For this choice of B, $\vec{t} = \begin{bmatrix} \mathtt{u} \\ \mathtt{v} \end{bmatrix} = \begin{bmatrix} \mathtt{i} \\ -\mathtt{i} + \mathtt{k} \end{bmatrix}$. The transformed code is shown in Figure 4.2. The grey area represents the `b` matrix as before, now skewed along with the iteration space.

After transformation, the loop nest is strip-mined, and the controlling loops are interchanged so that they are outermost. The inner n loop indices then form the tile-space iteration vector $\vec{r}$. The tiled code for the example program is shown in Figure 4.3. Each tile can be considered to have its own coordinate system, with its origin in the lower left corner of the tile (in n dimensions, in the corner of the tile closest to the origin in the transformed

```
for u = 1 to 12 do
  for v = 0 to 12-u do
    w[u] = w[u] + b[u,u-v]*w[v];
```

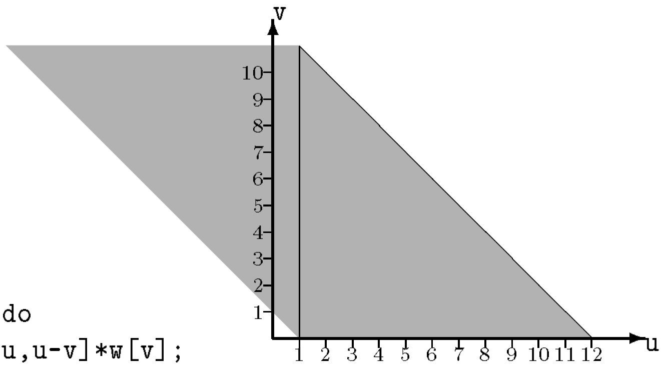


Figure 4.2: The same loop nest skewed

space). In the tile code, $\vec{t}$ is the vector $(\mathtt{uu}, \mathtt{vv})$, and $\vec{\tau}$ is the vector $(\mathtt{uu} - \mathtt{u}, \mathtt{vv} - \mathtt{v})$. As an example, the iteration $\vec{p_i} = (9, 5)$ in the original source loop is the iteration $\vec{p_t} = (5, 5)$ in the transformed space. After tiling, this iteration lies in the tile that has its origin at $\vec{p_t} = (4, 3)$. Within that tile, it has coordinates $\vec{p_\tau} = (1, 2)$.

```
for u = 1 to 12 by $\beta_u$ do
  for v = 0 to 12-u by $\beta_v$ do
    for uu = u to min(12, u+$\beta_u$-1)
      for vv = v to min(12-u, v+$\beta_v$-1)
        w[u] = w[u] + b[u,u-v]*w[v];
```

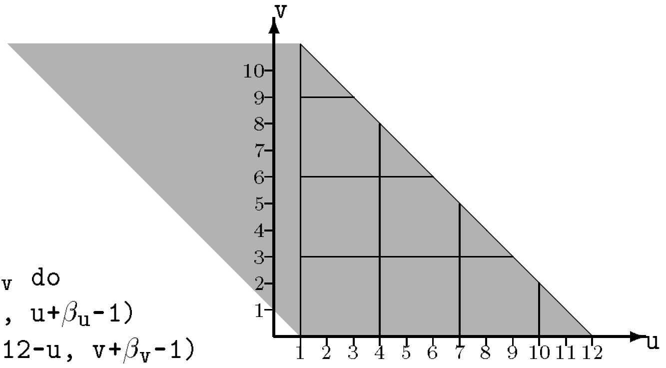


Figure 4.3: The example loop nest skewed and then tiled

4.3 Two buffering methods

Buffering is handled by cases. If rows of R are in B, the corresponding rows of S will be elementary vectors, that is, rows of I. In this case, a method called rectangular buffering is applied. If $S = I$, buffering is trivial. Each tile uses a rectangular submatrix of `v`. The space requirement is the product of the tile dimensions. The data can be buffered in an array in M_1 that has the same dimensionality as the tile. The buffer subscript is $\vec{\tau}$, and the source array subscript is $\vec{p_t}$.

If S has less than full rank, but the rows of S are rows of I, the same basic method applies. If the jth row of S is the kth row of I, the size requirement in dimension j of the buffer array is β_k. The buffer array subscript function in the jth dimension is given by τ_k.

When S has a special form described in Section 4.5, the tile references a skewed n-rectangle (a parallelepiped) of data. While rectangular buffering could be applied in this situation, it is much more space-efficient to use a method called skewed rectangular buffering. In this method, a skewed n-rectangle of data is copied into M_1. The data is unskewed as it is copied in, so that a rectangular array is used as buffer space.

In fact, skewed rectangular buffering is a generalization of rectangular buffering. However, rectangular buffering is still useful for two reasons. First, it serves as a gentle introduction to the kind of addressing mechanism required for the more general case. Second, rectangular buffering is more or less forced on the compiler when the M_2 memory is strongly block-oriented. If the M_2 memory intrinsically deals with 1Kbyte blocks, for example, the compiler may be forced to deal with such blocks (because, for example, the memory may not support modifying a partial block, but may require that the entire block be written back if any portion is modified).

4.4 Rectangular buffering

To simplify code generation, the compiler could require the blocks allocated in, and possibly moved into, M_1 to be rectangular subarrays of the data stored in M_2. It was pointed out earlier that when $S = I$, space requirements and subscripts functions are trivial. In this section, rectangular buffering for general S is discussed. When $S \neq I$, overallocation results, and overfetching becomes a possibility (recall that overallocation means space is allocated in M_1 for data that is not required for a tile, and overfetching means data is copied from M_2 into M_1 that will not be used in a tile; space can be overallocated even if no data is ever copied into that space). For this reason, the prototype compiler uses skewed rectangular buffering when necessary. Rectangular buffering may be a reasonable choice for other compilers (for example, if M_2 access time significantly rewards rectangular blocks), and so some general analysis is included here.

Since the range of data needed by a division can be other than rectangular, the smallest rectangular superset of the data required by a division will be allocated in M_1. The convex hull [5, pageref?] of the data required will be fetched prior to execution of the division. This can lead to overallocation and overfetching. In Figure 4.4, the iteration space is divided by vectors lying at ±45°, that is, $B = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$. A data stream with reference matrix $R = I$ (i.e., a two-dimensional array oriented in the obvious way, with the first dimension parallel to the i axis and the second parallel to the j axis) will require extra data to be allocated per tile. Two tiles are highlighted in light grey, marked "A" and "B". The data space allocated for these tiles include the tiles themselves and also the darker grey regions. Note that not only is data allocated that is not used for a particular tile (like the lower left corner of "A"), but data can be allocated that is not even referenced by the iteration set (the upper left corner of the allocation for "A").

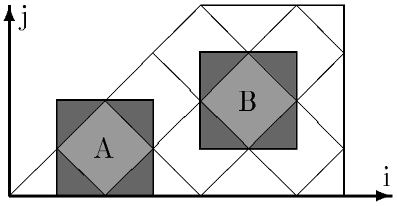
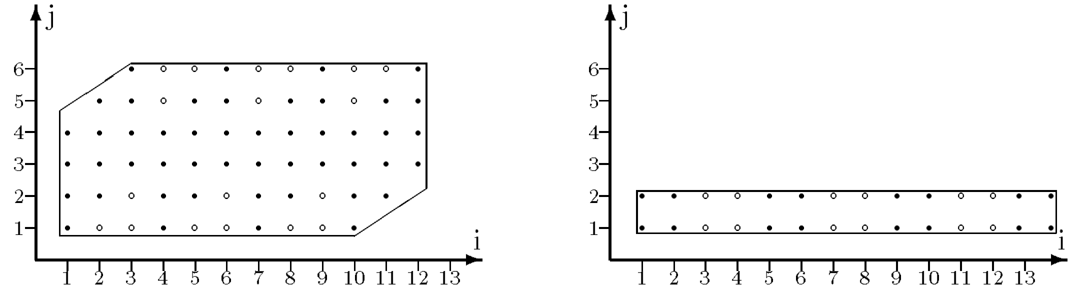


Figure 4.4: Overallocation of data required for rectangular buffering

Figure 4.5 shows how overfetching can occur. On the left side of the figure, a division of a larger iteration space is shown. This division consists of four iterations of the `k`-loop. The convex hull of the data used in the division include several points that need not be fetched (open circles). On the right hand side of the figure, we can see that this problem can be made arbitrarily bad; by increasing the coefficient of `k` in the loop bound expressions for the `i` loop, the ratio of data fetched to data used can be made arbitrarily high. In real programs, access patterns are often dense, but it is not uncommon for the iteration space to be of higher dimensionality than the data space. In both examples in the figure, the iteration space is three dimensional (the "`k`" dimension extends outward from the page), while the data space is two-dimensional. Projecting a higher-dimensional iteration space onto a lower-dimensional data space is often a cause of non-dense access patterns.

Reference vectors show directions of increasing subscripts for each dimension of an array. If we use reference vectors as dividing vectors, we are dividing the iteration space into partitions that reference rectangular sub-blocks of the data. This eliminates overallocation



```
for k = 0 to 3 do
    for i = 3k+1 to 3k+3 do
        for j = i-3k to i-3k+3
                ...A[i,j]...
```

```
for k = 0 to 3 do
    for i = 4k+1 to 4k+2 do
        for j = 1 to 2 do
                ...A[i,j]...
```

Figure 4.5: Overfetch of data using convex hull method

for those streams that have all of their reference vectors in the dividing basis, assuming dense data access. If there are exactly n distinct reference vectors (and they are linearly independent), we can use the reference vectors as tiling basis vectors and the resulting tiles will have no overallocation.[2]

Note that it is not necessary to choose *exactly* the reference vectors of a stream to lower the memory requirements: if vector $\vec{a}$ is closer to a reference vector than vector $\vec{b}$ (the relative angle between $\vec{a}$ and the reference vector is smaller than the angle between $\vec{b}$ and the reference vector), then choosing $\vec{a}$ reduces the overallocation in comparison to choosing $\vec{b}$, as shown in Figure 4.6. In the figure, both tiles encompass the same area, so both have the same number of computations and the same number of data elements (because of the orthogonality of $R_{1,*}$ and $R_{2.*}$). However, the overallocated areas (shown by the dashed regions) are much larger when the partitioning vector is moved farther from the reference vector.

---

[2] Recall that we made the assumption in Chapter 1 that the dimensionality of the data space equals the dimensionality of the iteration space. If a dividing of the data space is being sought, rather than a tiling of the entire iteration space, only λ linearly independent reference vectors are required, where λ is the dimensionality of the data space.

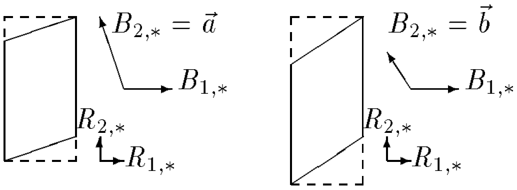


Figure 4.6: Overfetch varies inversely with the closeness of dividing vectors to reference vectors

4.4.1 Selection of basis vectors

Rectangular buffering leads us to choose reference vectors (elements of V) as dividing vectors, because this results in no overallocation for the streams whose reference vectors are in the dividing basis. If there are many reference vectors to choose from, priority goes to the vectors of high-dimensioned arrays, since much more fast memory space is wasted when they are overallocated.

4.4.2 Space allocation

The memory requirement $\mu_{\mathbf{v}_s}$ for a stream $\mathbf{v}_s$ can be found by taking the product of the memory requirements for each of the dimensions of the associated variable. As an example, a one-dimensional array will require enough memory to store every element from the smallest-indexed element referenced to the largest. A two-dimensional array will require the product of its x range and its y range. Let $\vec{s}_i^{min}$ be the smallest subscript value of data referenced by a tile in dimension i. Similarly, let $\vec{s}_i^{max}$ be the largest referenced subscript value along dimension i. The formula for memory required can be written:

$$\prod_{i=1}^{\delta} \left(\vec{s}_i^{max} - \vec{s}_i^{min} + 1 \right)$$

We now need a formula for the number of points lying in a particular division of the iteration space. Consider a generic tile. We can consider the corner that occurs first in the sequential order to be the origin. In order for a point $\vec{p}_i$ in the original iteration space to lie within the tile, it must be that $\forall j : \vec{p}_i \cdot B_{j,*} \geq 0$ and also $\forall j : \vec{p}_i \cdot B_{j,*} \leq \vec{\beta}_j$. That is, for

$\vec{p_i}$ to lie within the tile, we must have

$$0 \leq B\vec{p_i} \leq \vec{\beta}$$

This set of inequalities gives a bound on the points that can lie in a tile. This in turn allows us to compute the parts of an array that can be referenced by that division. Each dimension i of variable `v` has an associated subscript vector in the new basis space $S_{i,*}$. The maximum and minimum referenced elements are given by $\vec{s}_i^{max} = \max(S_{i,*} \cdot \vec{p_t})$ and $\vec{s}_i^{min} = \min(S_{i,*} \cdot \vec{p_t})$ over the division $0 \leq \vec{p_t} < \vec{\beta}$. Thus, our earlier expression for the amount of memory required by the stream per division becomes:

$$\mu_{\mathtt{v}.k} = \prod_{i=1}^{\delta} \left[\left(\begin{array}{l} \max S_{i,*} \cdot \vec{p_t} \\ \text{subject to } 0 \leq \vec{p_t} < \vec{\beta} \end{array} \right) - \left(\begin{array}{l} \min S_{i,*} \cdot \vec{p_t} \\ \text{subject to } 0 \leq \vec{p_t} < \vec{\beta} \end{array} \right) + 1 \right] \tag{4.4}$$

Note that every corner of a division has the form $(\vec{p}_{t1}, \vec{p}_{t2}, \ldots, \vec{p}_{tn})$ where either $\vec{p}_{ti} = 0$ or $\vec{p}_{ti} = \beta_i - 1$. We can find the maximum value by summing $S_{i,j}(\beta_j - 1)$ if $S_{i,j}$ is positive, and zero if it is not. Conversely, we can find the minimum by summing $S_{i,j}(\beta_j - 1)$ if $S_{i,j}$ is negative, and zero if it is not. This allows us to find the maximum and minimum values in linear time in n. The memory requirements are expressed symbolically, as a polynomial in the unknown β_i's. A more general discussion of finding bounds of a linear function in an iteration space can be found in Chapter 4 of Banerjee's book on dependence analysis[7].

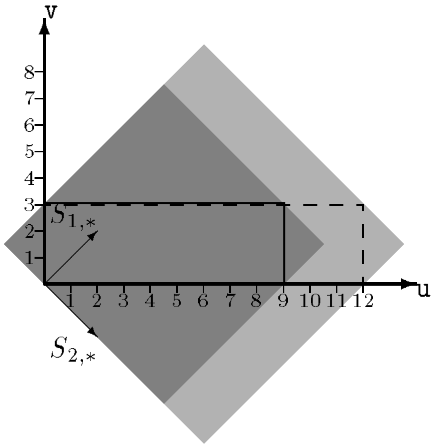


Figure 4.7: Nonlinearity in the tile size expression

If the vectors of R are in B, and the vectors of B which aren't in R are perpendicular to the vectors of R, then S is a permutation of δ rows of the identity matrix. In this case, $\mu_{\mathtt{v}.k}$ is a simple product of the tile size factors (i.e., $\mu_{\mathtt{v}.k} = \beta_{\mathtt{i}}\beta_{\mathtt{j}}\beta_{\mathtt{k}}$). This is not true in general, however, as is shown in Figure 4.7. A tile with dimension $\vec{\beta} = (9, 3)$ is shown as a solid box. An array is referenced with subscript vectors $S_{1,*}$ and $S_{2,*}$. The dark grey area corresponds to the size of data that must be allocated for the tile using rectangular buffering. When the tile size is increased to $\vec{\beta} = (12, 3)$, the increase in $\beta_{\mathtt{u}}$ causes an increase in the size requirement for both dimensions of the array (the allocation requirement is shown as a lighter grey area). Increasing the single tile dimension increases the data requirement for the tile in both the $R_{1,*}$ and $R_{2,*}$ directions. The size requirement is proportional to the square of $\beta_{\mathtt{u}}$ in this case.

4.5 Skewed rectangular buffering

When S has a particular form, the range of data accessed by a tile has the shape of a skewed rectangle. Rather than allocating a buffer that corresponds to a rectangular part of the source array, the compiler can allocate a buffer that corresponds to a skewed-rectangular section of the source array. This results in little or no overallocation (overallocation is only possible for partial tiles at the edges of the iteration space).

Skewed rectangular buffering is a linear transformation from a k-dimensional parallelepiped (a skewed k-rectangle) to an orthogonal k-rectangle. Section 4.5.1 defines precisely what is meant by a parallelepiped. Section 4.5.2 describes the form the subscript matrix S must have in order for the set of data referenced by the iterations of a tile to form a parallelepiped in the data space. Section 4.5.3 derives the transformation itself, using the idea of repeatable kernels. Finally, Section 4.5.4 describes the allocation used for skewed rectangular buffering, and derives the M_1 size requirement in terms of the tile size vector $\vec{\beta}$.

4.5.1 Skewed rectangles

Skewed rectangles can be used for two different purposes by the compiler. Tiles in the iteration space can be represented as skewed rectangles, and sometimes the data accessed by the iterations of a tile can be represented as a skewed rectangle. When the data accessed does form a skewed rectangle, the compiler can use a linear transformation to unskew the data, so that it can be stored with no waste. We now define precisely what is meant by a skewed rectangle.

Definition 1 *Given a integer basis* $K = (\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots, \vec{e}_k)$ *for* k*-space, the* unit parallelepiped $P(K)$ *is the set of vectors which are convex linear combinations of the basis vectors. Specifically,*

$$P(K) = \left\{ \vec{v} | \vec{v} \in \mathbb{R}^k \text{ and } \vec{v} = v_1\vec{e}_1 + v_2\vec{e}_2 + \ldots + v_k\vec{e}_k \right\}$$

where $0 \leq v_i \leq 1$ *for all* i*.*

Geometrically, a unit parallelepiped is a region of real space (i.e., the k-dimensional space of real numbers). There is a corresponding structure in the integers:

Definition 2 *Given an integer basis* $K = (\vec{e}_1, \vec{e}_2, \vec{e}_3, \ldots, \vec{e}_k)$ *for* k*-space, the* integer parallelepiped $\pi(K)$ *is the set of integer-valued vectors which are convex linear combinations of the basis vectors. Specifically,*

$$\pi(K) = \left\{ \vec{v} | \vec{v} \in \mathbb{Z}^k \text{ and } \vec{v} = a_1\vec{e}_1 + a_2\vec{e}_2 + \ldots + a_k\vec{e}_k \right\}$$

where $0 \leq a_i < 1$ *for all* i*.*

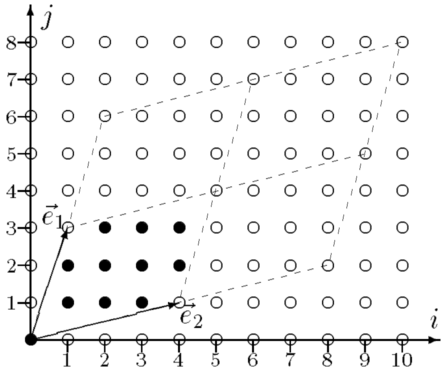


Figure 4.8: Example of unit parallelepiped borders

An important difference in the definitions is that $\pi(K)$ does not contain all of its border. The border hyperplanes which do not pass through the origin are excluded. In this way, when k-space is tessellated with copies of an integer parallelepiped, the excluded borders will be included in other parallelepipeds. Each point belongs to exactly one copy. As an example, examine the parallelepipeds of Figure 4.8. The set of points belonging to the parallelepiped at the origin are shown with solid dots. Iterations not belonging to that tile are shown using open circles. As the set of iterations is copied to fill 2-space, each integer point in 2-space will belong to exactly one copy.

The set of iterations executed by a tile can now be described as simply the set of points belonging to some copy of an integer parallelepiped. The left side of Figure 4.9 shows two basis vectors in a 2-dimensional iteration space. There is a pair of hyperplanes perpendicular to each basis vector; one defines the lower extent of the tile with respect to the basis vector, and the other defines the upper extent of the tile with respect to the basis vector. The result is a parallelepiped with one corner at the origin. The parallelepiped is subtended by a set of vectors denoted Ψ, as shown on the right side of the figure. The set of iterations in the tile is then $\pi(\Psi)$.

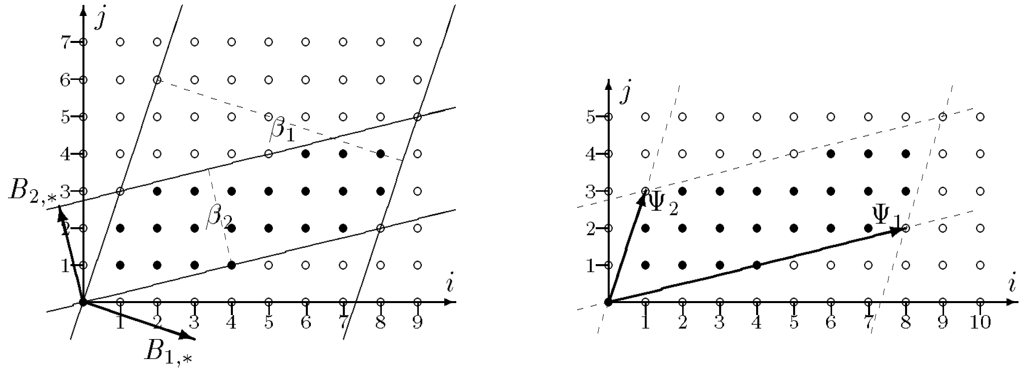


Figure 4.9: Tiles are unit parallelepipeds

Once the loops are transformed to the new basis space, they are simply strip-mined to form tiles. In the transformed space, tiles are therefore n-dimensional rectangles with edges parallel to the axes (n-orthorectangles). The set of rays Ψ form an n-orthorectangle in the transformed space. The rays which subtend this n-orthorectangle form the diagonal matrix $\Psi = \text{diag}(\vec{\beta})$.[3] In the original iteration space, the tiles form parallelepipeds subtended by the vectors of $B^{-1}\Psi$. This is the matrix B^{-1} with row i scaled by β_i.

---

[3] The diagonal array $\text{diag}(\vec{\beta})$ has zeroes everywhere except the diagonal; the ith diagonal element is β_i.

The subscript matrix S maps iterations in the transformed space into elements in the data space of an array. When the image under S of the orthorectangle $P(\Psi)$ is $P(\Delta)$ for some Δ, the data accessed by the iterations in $\pi(\Psi)$ form a unit parallelepiped in the data space of the referenced array. This parallelepiped is what the compiler must store efficiently in M_1. Section 4.5.2 describes the conditions necessary for this to be the case.

4.5.2 What S must be

The following theorems describe the form S must have in order for $S(P(\Psi))$ to be $P(\Delta)$ for some Δ, that is, in order for S to map an n-orthorectangle into a δ-parallelepiped. The key insight is that the set of iterations in a tile is a convex combination of the vectors of Ψ. Since S is a linear transformation, the image of a tile under S is the set of vectors which are a convex combination of the images of each vector of Ψ under S.

Theorem 1 *The set V_C of vectors formed by convex linear combinations of a set V_L of m vectors in k-space, $k \leq m$, forms a parallelepiped with one corner at the origin if there is a subset V_B of V_L which forms a basis for k-space, and all vectors in $V_L - V_B$ are either zero or a positive multiple of some vector in V_B.*

Proof: We must show that if V_L can be partitioned into V_B and $V_L - V_B$ as described, every k-vector which is a convex combination of the vectors of V_L is a convex combination of the elementary vectors of some basis Δ. $P(\Delta)$ is the parallelepiped formed.

Here is how to construct Δ: order the set V_L so that the basis vectors (members of V_B) are numbered $V_1, V_2, \ldots, V_k$; the vectors not in the basis are numbered $V_{k+1}, V_{k+2}, \ldots, V_m$. Define

$$m(i,j) = \begin{cases} a & \text{if } V_j = aV_i \\ 0 & \text{otherwise} \end{cases}$$

so that $m(i,j) = 0$ if $V_j = 0$ or if V_j is not a positive multiple of V_i. Note that it is not possible that $a < 0$, because then there are points along the ray V_i (and V_j) on both sides of the origin; the parallelepiped formed would have to include the origin, which would preclude its being a corner. (The theorem can be extended to allow negative a's; we would then want $m(i,j) = |a|$ or 0. See footnote 4 on page 55 for a sketch of the extension.)

The set of column vectors $\Delta_i = (\sum_{j=1}^{m} m(i,j))V_i$ for $i = 1, 2, \ldots, k$ form a basis for k-space. Any vector $\vec{v}$ which is a convex linear combination of the vectors of V_L is a convex linear combination of the columns of Δ, and vice versa. Thus $P(\Delta)$ is a parallelepiped which is exactly the set of vectors formed by a set of convex linear combinations of V_L. ■

Theorem 2 *The set V_C of vectors formed by convex linear combinations of a set V_L of m vectors in k-space, $k \leq m$, forms a parallelepiped with one corner at the origin only if there is a subset V_B of V_L which forms a basis for k-space, and all vectors in $V_L - V_B$ are either zero or a positive multiple of some vector in V_B.*

Proof: If V_L has rank less than k, the space spanned by V_L is lower-dimensional than k-space, so V can't be a k-parallelepiped; therefore, V_L has rank k, and there is a set of k vectors $V_B \subset V_L$ which forms a basis for k-space. All that is left is to show that the vectors of $V_L - V_B$ are either zero vectors, or positive multiples of some vector in V_B.

Let basis which induces the k-parallelepiped be the set of vectors Δ. All the vectors in V_L must have elements in the range [0,1] when written in basis Δ, because each vector of V_L is trivially a convex combination of all the vectors in V_L (so each vector in V_L must lie in the parallelepiped formed). From now on, consider all vectors in this new basis. There are k corners of the parallelepiped which are elementary vectors Δ_i. Δ_i is a convex combination of some of the vectors in V_L, and since the V_L are everywhere non-negative, Δ_i can be expressed as the sum of all the vectors in V_L with non-zero entries in the ith position. But since Δ_i is an elementary vector, it is zero everywhere else, which implies that the vectors that were summed to form it are also zero everywhere else. Thus there are exactly $k+1$ equivalence classes of vectors in V_L: one class consists of the zero vectors, and the other k classes consist of positive multiples of Δ_i, for $i = 1, 2, \ldots, k$. Each class must have at least one entry. Any basis for k-space must necessarily contain at least one entry from each of the classes other than the zero class. Any set of k vectors, one chosen from each class, can form V_B as required. ■

Theorem 3 *Given a basis Ψ for n-space and a linear transformation S of rank δ from n-space to δ-space, $\delta \leq n$, $S(P(\Psi))$ is a unit parallelepiped $P(\Delta)$ for some basis Δ of δ-space if and only if the columns of S can be separated into two set of vectors $S_B = \{S_1, S_2, \ldots, S_\delta\}$ and $S_A = \{S_{\delta+1}, S_{\delta+2}, \ldots, S_n\}$, where S_B forms a basis for δ-space and the vectors of S_A are either zero vectors or positive multiples of some vector in S_B.*

Proof: Follows immediately from the previous two theorems, by noting that since $P(\Psi)$ is formed by taking convex combinations of the vectors in B', $S(P(\Psi))$ will also be formed by taking convex combinations of a set of vectors (in this case, the images under S of the vectors in Ψ). ■

Now we can state the form S must have in order for the data used by a tile to be a skewed rectangle. The columns of S must be partitionable into two sets; one set of δ vectors forms a basis for the data space. The other columns must be zero, or positive multiples of one of the basis columns.[4]

If the tiling basis B is chosen from the set of reference vectors, the subscript matrix $S = RB^{-1}$ of any stream whose reference vectors are in B will be a permutation of vectors from the identity matrix. Reference vectors therefore are good choices for candidate vectors.

To see that the projection of a parallelepiped onto a lower-dimensional space does not always yield a parallelepiped, imagine a three-dimensional cube, as it is often depicted in a two-dimensional medium (see the left side of Figure 4.10). The projection along the viewer's central axis yields a six-sided figure (on the right side of the same figure).

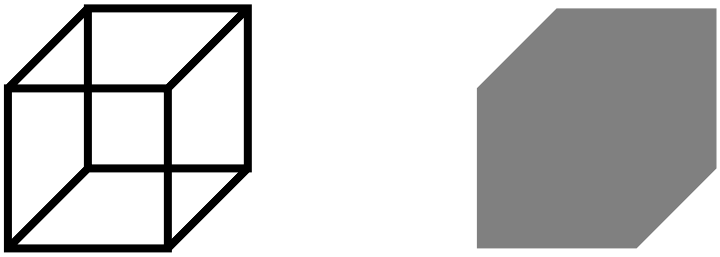


Figure 4.10: A cube and its projection into 2-space

4.5.3 Transforming a parallelepiped to a cube

In the transformed iteration space, a tile is an n-orthorectangle, subtended by the rows of the matrix $\Psi = \mathrm{diag}(\vec{\beta})$. The compiler can easily check whether S maps the iterations of a tile into a k-parallelepiped; if it does, Theorem 1 shows how to compute Δ, the set of rays which subtend the parallelepiped. The matrix Δ^{-1} maps the k-parallelepiped into a unit orthorectangle in δ-space. In this section, we describe how to modify Δ^{-1} into a unimodular matrix, so that the number of integer vectors in the k-parallelepiped is the same as the

---

[4] The form given is for data spaces which are parallelepipeds with one corner at the origin. Negative multiples of basis vectors can be allowed also; allowing negative multiples results in parallelepipeds which can contain the origin. Note that any parallelepiped containing the origin can be decomposed into a set of parallelepipeds, each of which having one corner at the origin; the ray vectors of these parallelepipeds must be multiples of one another.

number of integer vectors in the k-cube it is mapped into.

If S is not unimodular, the accesses of the stream aren't dense: data items which lie in the parallelepiped may not actually be accessed by any iteration in the tile. In particular, if $S_i, S_j, S_k, \ldots$ are multiples of one another, data is not densely accessed unless the multiples are relatively prime (ignoring the possibility of multiple streams referencing the same arrays resulting in a dense pattern overall). In this work, no effort is made to coalesce possible non-dense accesses; the entire parallelepiped of data is stored, assuming every data element in the parallelepiped is used.

4.5.4 Allocation and space requirements

In order to understand the linear transformation from a parallelepiped to a cube of the same size, the reader must first understand how the hyperplanes which are used to index into the data are spaced and numbered. The next section describes hyperplane spacing. The section after that describes an allocation procedure for the data. The final section describes the linear transformation that results.

Hyperplane spacing

This section describes how hyperplanes perpendicular to normal vectors are numbered, and shows how to ensure that enough hyperplanes are used, but not too many.

An illustrative example will be helpful in the discussion. The left side of Figure 4.11 shows two normal vectors is a two-dimensional space. In the middle diagram, hyperplanes normal to the first vector are shown. In the rightmost diagram, hyperplanes perpendicular to the second normal vector are shown. The hyperplanes are regularly spaced so that every integer lattice point intersects some hyperplane. It is possible that a few hyperplanes will not intersect lattice points if the tile is very small, but every lattice point must lie on some hyperplane.

Definition 3 *Let $\vec{n}$ be a normal vector. The vector $\hat{n}$ is the smallest multiple of $\vec{n}$ with all integer entries. The vector $\hat{n}$ is called a* GCD-1 vector, *because the greatest common divisor of the elements of $\hat{n}$ is 1, and $\hat{n}$ has all integer entries.*

The hyperplanes are numbered so that the plane through the origin is numbered 0, and numbers increase in the direction of $\hat{n}$. The number of the plane in which a point $\vec{p}$ lies is

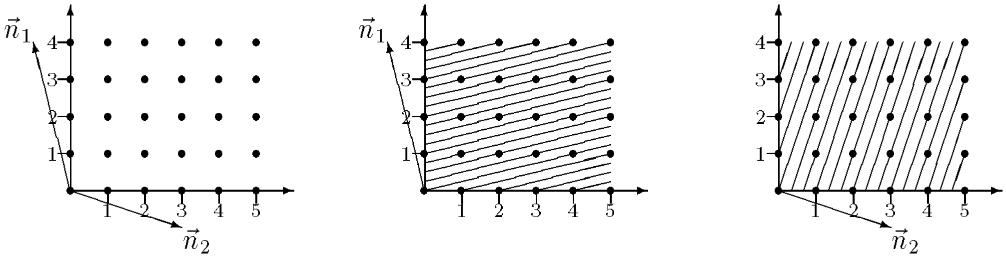


Figure 4.11: How data relates to the transformed iteration space

then given by $\hat{n} \cdot \vec{p}$. The set of solutions $\vec{x}$ to $\hat{n} \cdot \vec{x} = k$ for integer k correspond to the set of hyperplanes perpendicular to $\hat{n}$ with $\hat{n} \cdot \vec{x} = 0$ being the hyperplane passing through the origin. There is clearly a hyperplane that passes through any given integer-valued point $\vec{p}$, because $\hat{n}$ and $\vec{p}$ are integer-valued (so $\hat{n} \cdot \vec{p}$ must be integer-valued). The following theorem shows that there is always at least one integer-valued point in plane k:

Theorem 4 *For any GCD-1 vector $\hat{n}$, $\hat{n} \cdot \vec{p} = k$ has at least one integer-valued solution $\vec{p}_0$.*

Proof: Let $\vec{a}$ be the vector of non-zero elements of $\vec{n}$. There is at least one non-zero element. Consider the set of integers $\mathcal{T}$ that can expressed as the dot product of some vector $\vec{x}$ with $\vec{a}$, that is, $\mathcal{T} = \{c | c = \vec{x} \cdot \vec{a}\}$. Let $\mathcal{T}^+$ be the positive part of $\mathcal{T}$, i.e., $\mathcal{T}^+ = \{c | c = \vec{x} \cdot \vec{a} \text{ and } c > 0\}$. We know $\mathcal{T}^+$ is non-empty because $(1, 0, 0, ..., 0) \cdot \vec{a} \in \mathcal{T}$ and also $(-1, 0, 0, ..., 0) \cdot \vec{a} \in \mathcal{T}$. By the Well-Ordering Theorem of modern algebra[49], any non-empty set of positive integers has a non-zero least element l. So $l > 0$ and $l = \vec{x}' \cdot \vec{a}$ for some $\vec{x}'$. Dividing each element a_i of $\vec{a}$ by l, we have $a_i = lq_i + r_i$ (q_i is the quotient and r_i the remainder), and $0 \leq r_i < l$ for all i. Then

$$\begin{aligned}
r_i &= a_i - lq_i \\
&= a_i - q_i(\vec{x}' \cdot \vec{a}) \\
&= a_i + \sum_{j=1}^{k} -q_i x_j' a_j \\
&= a_i(1 - q_i {x_i}') + \sum_{j=1, j \neq i}^{k} -q_i x_j' a_j
\end{aligned}$$

Thus, r_i can be written as the dot product of an integer vector with $\vec{a}$, so $r_i \in \mathcal{T}^+$ for all i. Since $r_i < l$ and l is the least element in $\mathcal{T}^+$, it follows that $r_i = 0$ for all i, so $l = \vec{a} \cdot \vec{x}'$ divides a_i for all i. Since the greatest common divisor of the elements of $\vec{a}$ is 1, it follows that $l = 1$ so $\vec{a} \cdot \vec{x}' = 1$. Obviously then $\vec{n} \cdot \vec{p} = 1$ has a solution where $p_i = 0$ if the corresponding element of $\vec{n}$ is zero, and an element of $\vec{x}'$ otherwise. If $\vec{p}'$ is a solution to $\vec{p} \cdot \vec{n} = 1$, then $\vec{p}_0 = k\vec{p}'$ is a solution to $\vec{n} \cdot \vec{p} = k$. ■

Theorem 4 states that there is at least one integer point in every plane when the normal vectors are GCD-1 vectors. It has already been shown that every integer point lies on a plane. Thus without *a priori* knowledge of the size of the parallelepiped, this spacing of normal hyperplanes is both necessary and sufficient.

A set of k linearly independent normal vectors form a basis for k-space. Any integral point $\vec{p}$ in k-space can be written in the new basis as $\vec{p}_{new} = N\vec{p}$ where N is the basis matrix formed from the normal vectors. The position of point $\vec{p}$ in dimension i of the new basis is simply $\vec{n} \cdot \vec{p}$. If $\vec{n}$ is a GCD-1 vector, this position is also the number of the hyperplane which passes through the point in the system of hyperplanes described earlier. For our parallelepiped subtended by the columns of Δ, the normal vectors are given by the rows of Δ^{-1}. The number of planes in direction i is given by $\Delta_{*,i} \cdot \text{GCD-1}(\Delta^{-T}_{*,i})$, where $\text{GCD-1}(\vec{n}) = \hat{n}$ is a function mapping any integer vector to its GCD-1 multiple.

As parallelepipeds are used to tessellate n-space, each parallelepiped abutting the next, each parallelepiped has an integer-valued point for each corner. It is easy to see that each parallelepiped must therefore contain the same number of integer-valued points, and that the number of points must equal the volume of the parallelepiped. The volume of the parallelepiped subtended by the vectors of Δ is simply $|\det \Delta|$.

A similar argument can be used to show that the number of integer points within a tile lying on each hyperplane perpendicular to a given normal vector must be the same, and that in fact, each hyperplane must have the same number of points on it. The number of points in the parallelepiped which lie on any hyperplane perpendicular to the ith normal vector is simply the number of points in the parallelepiped divided by the number of planes, or $\frac{|\det \Delta|}{\Delta_{*,i} \cdot \text{GCD-1}(\Delta^{-T}_{*,i})}$.

This observation is key to the allocation procedure, but it is not sufficient. The same integral point lies on a different plane in each dimension, but the intersection of different planes does not generally lie at an integer point (see Figure 4.13). Using GCD-1 spacing in each direction overallocates space because it allocates an entry for every intersecting point whether it is integral or not. An efficient allocation allocates only enough space for the integral-valued points, as described below.

Allocation

We need to assign to each dimension *i* of the parallelepiped a number $alloc_i$ that evenly divides the total number of points in the parallelepiped and the number of planes in direction *i*. The product of the $alloc_i$'s must be equal to the volume of the repeatable kernel. One way to accomplish this is demonstrated by the pseudo-code of Figure 4.12.

```
v = $|\det \Delta|$;
for i = 1 to $\delta$ do
{
        allocation[i] = GCD ($\Delta_{*,i} \cdot$ GCD-1($\Delta^{-T}_{*,i}$), v);
        v = v/allocation[i];
}
```

Figure 4.12: Pseudo-code for performing allocation

Figure 4.13 shows an example of allocation. The parallelepiped is subtended by $\Delta_1 = (12, 3)$ and $\Delta_2 = (2, 6)$. In this case, there are 33 planes in the Δ_1 direction and 22 planes in the Δ_2 direction (taking the inverse matrix and performing the inner products). The volume of the parallelepiped is 66 integral points. This can either be allocated using a 33×2 array, as shown on the right, or using a 22×3 array, as shown on the left (each line represents a new row or column). The code above would generate the 33×2 solution.

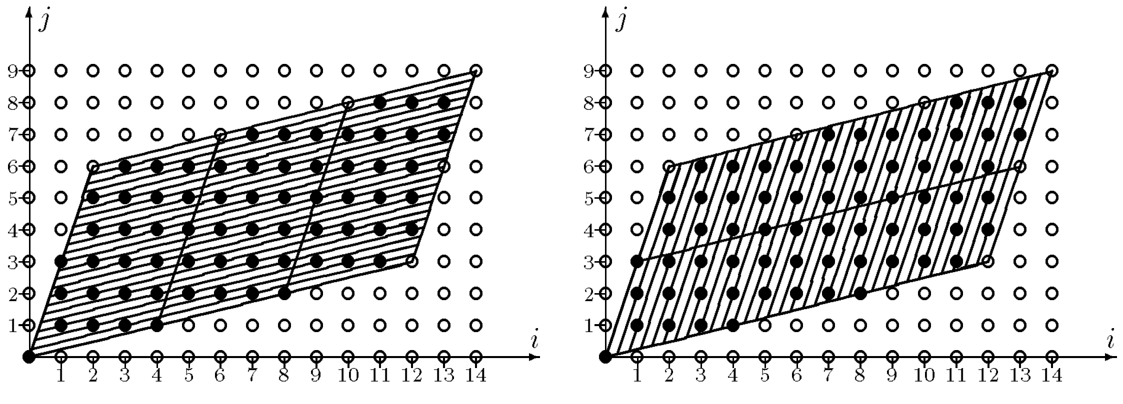


Figure 4.13: Examples of allocations

In general, we need $\prod_{i=1}^{\delta} alloc_i$ equal to the volume of the parallelepiped, and we need $alloc_i$ to divide $\Delta_{*,i} \cdot \text{GCD-1}(\Delta^{-T}_{*,i})$ for all *i*.

Transforming array subscripts to buffer subscripts

The matrix Δ^{-1} maps the parallelepiped of data referenced by a tile into a unit k-cube. The allocation mechanism effectively changes this mapping into a k-cube of some specified dimension. Let T be the transformation from the parallelepiped to the desired rectangular allocation. The matrix T satisfies $\mathrm{diag}(\mathrm{alloc}) = T\Delta$. This can be re-written $T = \mathrm{diag}(\mathrm{alloc})\Delta^{-1}$. In effect, the vectors of the inverse matrix are scaled by the allocation in each dimension.

In the example of 4.13, the parallelepiped is subtended by the columns of

$$\Delta = \begin{bmatrix} 12 & 2 \\ 3 & 6 \end{bmatrix}$$

The inverse matrix is

$$\Delta^{-1} = \begin{bmatrix} 6/66 & -2/66 \\ -3/66 & 12/66 \end{bmatrix}$$

The GCD-1 vectors are $(3, -1)$ and $(-1, 4)$. There are $(12, 3) \cdot (3, -1) = 33$ hyperplanes in dimension 1 and $(2, 6) \cdot (-1, 4) = 22$ hyperplanes in dimension 2. Using the 33×2 allocation, the transformation matrix T is given by

$$T = \begin{bmatrix} 33 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 6/66 & -2/66 \\ -3/66 & 12/66 \end{bmatrix} = \begin{bmatrix} 3 & -1 \\ -1/11 & 4/11 \end{bmatrix}$$

The transformation T has been developed for parallelepipeds at the origin. When generating the loops to copy data into or out of buffers, the compiler can easily find the transformed iteration vector $\vec{t}_0$ which is the origin of the tile (this is trivial in the generated code; $\vec{t}_0$ is just the index vector of the controlling loops). A reference `v`$[R\vec{\imath} + \vec{r}]$ in the source code is `v`$[S\vec{t} + \vec{r}]$ in the transformed space. This can be re-written `v`$[S\vec{t}_0 + S\vec{\tau} + \vec{r}]$ to emphasize the origin of the tile in the data space.

In general, the compiler can generate loops to copy data into and out of the buffer space by looping over each element of the buffer and copying in or out the appropriate element. The loop for copying data from M_2 into M_1 is of the form

```
for $\vec{k}$ = $\vec{0}$ to $\vec{\alpha}$
  buffer[$\vec{k} + \vec{r}$] = source-array[$S\vec{t} + T^{-1}\vec{k} + \vec{r}$]
```

where $\vec{\alpha} = (alloc_1 - 1, alloc_2 - 1, \ldots)$. Recall that $\vec{t}$ is the origin of the tile in the transformed iteration space (at least outside of the innermost computation loop), so $S\vec{t}$ is the origin of the parallelepiped of data referenced by the tile. If the array is modified (written), the buffer must be copied back into M_2. The loop for this takes the form

```
for $\vec{k}$ = $\vec{0}$ to $\vec{\alpha}$
  source-array[$S\vec{t} + T^{-1}\vec{k} + \vec{r}$] = buffer[$\vec{k} + \vec{r}$]
```

In our running example, the following code would be generated to copy the data in the parallelepiped (see Figure 4.13) into a buffer in M_1 memory:

```
Temporary Array buf[33,2];

for a = 0 to 33-1
    for b = 0 to 2-1
        buf[a,b] = source-array[4a/11+b,-a/11+3b];
```

The matrix T has determinant 1 but is not unimodular, because the entries are not generally integer ("unimodular" is used to describe integer matrices with determinants of +1 or -1). The entries of T can be directly used as subscript coefficients assuming that integer division performs truncation.

In the execution loop itself, references to the source array are replaced with references to the buffer array. Since an element `v`$[S\vec{t}_0 + \vec{r}]$ in the source array corresponds to an element `v`$[T\vec{r}]$ in the buffer array, the final variable reference is `v_buf`$[TS\vec{r} + \vec{r}]$. In our example, references to `source-variable`$[i,j]$ would be replaced with references to `buf[3i-j,-i/11+4j/11]`.

4.6 Conclusions

This chapter has developed the machinery necessary to implement buffering techniques required for software management of the memory hierarchy. The easiest mechanism to use is rectangular buffering. It can always be applied, but can result in overallocation (wasted memory space) when the subscript matrix S is not a permutation of the vectors of the identity matrix. If the basis transformation B is chosen from the vectors of the reference vector set, S will be a permutation of the identity matrix for those streams whose reference vectors are in B.

In cases where rectangular buffering leads to overallocation, the compiler can apply skewed rectangular buffering. In either case, the M_1 memory buffer size for each stream is given by a formula in the tile size factors β_i. Using skewed rectangular buffering, the formula is linear with respect to any one β_i value. With rectangular buffering, the space requirement can be nonlinear in the β_i values.

It is important to note that using either buffering method, a rectangular buffer array is referenced in the source code, using subscripts which are linear in the loop indices. This kind of subscript is easily handled by optimizing compilers which use strength reduction and similar optimizations to improve execution time.

Chapter 5

Scheduling the tiles

Chapter 4 showed that the number of array elements allocated in M_1 per stream for each tile is a simple function of the tile size vector $\vec{\beta}$. The next step is to find a formula (in terms of $\vec{\beta}$) for the number of times the data in these buffers must be moved from one memory to the other. The compiler can then find the total cost of data motion for a tiling in terms of $\vec{\beta}$. This will allow the compiler to find the value of $\vec{\beta}$ that minimizes data motion.

A naïve compiler would simply copy the data from M_2 into M_1 before each tile is executed, and copy it back afterwards. The number of times a buffer is moved is then twice the number of tiles in the iteration space. A simple optimization is to eliminate the copy back into M_2 for read-only data. This makes the number of times a buffer is copied either the number of tiles (for read-only data) or twice that (for writable data).

A more complex optimization takes advantage of the fact that sometimes data resident in an M_1 buffer for one tile may also need to be resident for the next consecutively executed tile.[1] Shared data need not be moved, but can stay resident in M_1 until different data is required. This can eliminate a substantial amount of data motion. To determine when data may be shared between consecutively-executed tiles, the compiler must be able to determine the execution order of the tiles. Furthermore, an optimizing compiler should *choose* the execution order to maximize the amount of data being reused. Choosing the execution order is called *scheduling* the tiles, and that is the subject of this chapter.

---

[1] It is possible for only *some* of the data used by a tile to be used in the next tile. The techniques used in this thesis address only full sharing on a stream basis (that is, if all of a stream's data used in one tile is used by the same stream in the next tile). Partial sharing, in which, for example, half of the data could remain resident, requires more complex addressing techniques to be used for accessing the data in M_1.

In the next section, some basic theory and notation is introduced. Scheduling for uniprocessors is then covered. For uniprocessors, locality between tiles is the only significant scheduling goal. Section 5.3 discusses various approaches to integrating tiling for memory management and parallelism using a simple form of parallelism.

5.1 Scheduling issues

In the first part of this section, intertile locality is described in detail. The second part of this section addresses the question of whether the compiler should schedule before finding a tile size vector or afterwards, since the two problems are closely interrelated.

5.1.1 Intertile locality

Recall that the input program is an n-deep nested loop. After tiling, there are $2n$ loops. The outer n loops select a tile to execute, and the inner n loops execute the iterations within that tile. The outer loops are called *controlling loops*, a term coined by Wolf and Lam[33, 55, 56]. Recall the earlier example of matrix-matrix multiply. The source code is a 3-deep nested loop (Figure 3.1). The tiled code is a 6-deep nested loop (Figure 3.2).

The tiling basis B defines the shape of the tiles. It also defines a possible execution order of the tiles: the iteration space can be transformed to the new basis, tiled, and the tiles executed in the resulting order. The innermost controlling loop would then be executing along the direction of the last basis vector $B_{n,*}$. It is possible, however, that reordering the execution of the tiles would lead to additional locality, in the form of *intertile locality.* Intertile locality results when two tiles share data, and those tiles are executed directly after one another on the same processor, so that the shared data need not be moved.

Intertile locality is independent of the execution order within a tile: the inner n loops can be executed in any order allowed by dependences. The problem addressed in this chapter is how to select the ordering of controlling loops to minimize data motion.

A stream s, generated by array reference $\mathtt{v}.k$ and having subscript matrix S, is said to be *perpendicular* to a loop direction $\vec{b}$ if $S\vec{b} = 0$, that is, if every subscript vector is perpendicular to $\vec{b}$ (since basis vectors are used loop execution directions, S is perpendicular to the basis vector $B_{i,*}$ if and only if the ith column of S, $S_{*,i}$, is zero). If s is perpendicular to the innermost controlling loop, then the data brought into M_1 for s can be kept in M_1

over the entire innermost loop. If s is perpendicular to the next loop as well, then it can be held locally throughout both loops, and so on.

Figure 5.1 illustrates this point. The code on the left is the source loop. The code on the right is the tiled code before buffer-copying loops have been inserted. The reference to `A` has all its subscript vectors (the only one is $(1,0)^T$ corresponding to the subscript `i`) perpendicular to the innermost controlling loop (the `j` loop in the code on the right; its direction vector is (0,1)). As iterations of the `j` loop are executed, the same data stays resident in M_1 for the `A` stream. Figure 5.2 depicts the iteration space geometrically. The `A` matrix lies along the `i`-axis, so it is perpendicular to the `j` loop. The shaded region of the `A` matrix must be copied in for the first tile in the second column, but it need not be moved again until the entire column has been executed.

```
for i = 1 to n do
  for j = 1 to n do
    A[i]=A[i]+W[i,j]*B[j];
```

```
for i = 1 to n by $\beta_i$ do
  for j = 1 to n by $\beta_j$ do
    for ii = 1 to min($\beta_i$,n) do
      for jj = 1 to min($\beta_j$,n) do
        A[ii]=A[ii]+W[ii,jj]*B[jj];
```

Figure 5.1: Example of stream perpendicularity

The reference to `B` is not perpendicular to the `j` loop, because the `B` matrix lies parallel to the `j`-axis. As the tiles are executed up columns, the `B` matrix must be copied into M_1 over and over. One of the `W`-stream's subscript vectors is perpendicular to the `j`-axis, but since both are not, there is no intertile locality available.

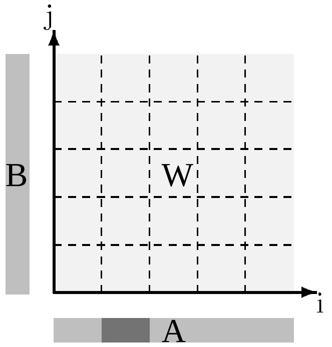


Figure 5.2: Graphic representation of stream perpendicularity

As a further example, imagine that the compiler skewed the code in this example. The result is shown in Figure 5.3. Now there is no locality available at all. The reference vector for the `A` matrix is now $(1, -1)$ (or `i`$-$`j`), which is clearly not perpendicular to the innermost controlling loop.

```
for i = 1 to 2n do
  for j = max(1,i-n+1) to min(i,n) do
    A[i-j+1]=A[i-j+1]+W[i-j+1,j]*B[j];
```

Figure 5.3: The previous example skewed

Figure 5.4 shows this graphically. Executing up the second column (the darker column) requires fetching the shaded portion of the `A` matrix. Each tile in the column requires a slightly different portion of the A matrix.[2] New data must be fetched for every tile, so no locality is available.

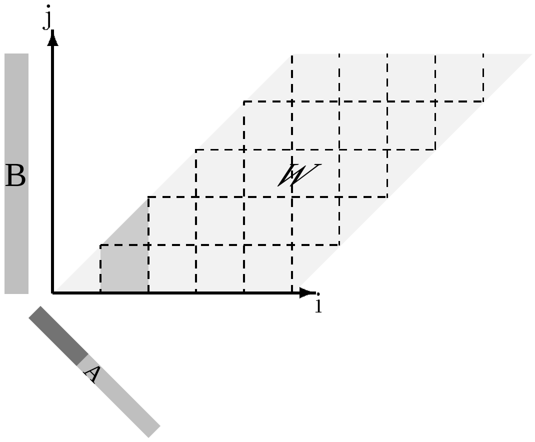


Figure 5.4: Example of stream perpendicularity

These two examples also serve to illustrate (albeit in the negative) how the compiler can control locality. By choosing the proper basis B, the compiler aligns the iteration space so that, as much as possible, streams are perpendicular to the innermost controlling loop. Using rectangular buffering, the compiler chooses the tiling basis so that streams are aligned with the iteration axes, and then chooses the best axis for the innermost controlling loop. Skewed rectangular buffering (see Chapter 4 allows the compiler to choose the tiling basis from directions which result in locality for the most streams directly. This will be illustrated in later sections.

---

[2] This is an example where there is some re-use of a stream, but not full re-use. Half of the elements of `A` used in one tile are also used in the next tile.

5.1.2 Scheduling versus computing tile sizes

In finding a tiling that results in minimal execution time, the compiler can vary several parameters: the slope of each face of the tile (determined by the basis B), the tile size in each dimension (determined by the elements of $\vec{\beta}$), and the order in which the tiles are executed (determined by the schedule). In this thesis, the compiler evaluates every possible basis. For a given basis, the compiler finds a tile size vector and a schedule to minimize execution time. This process is illustrated in Figure 5.5.

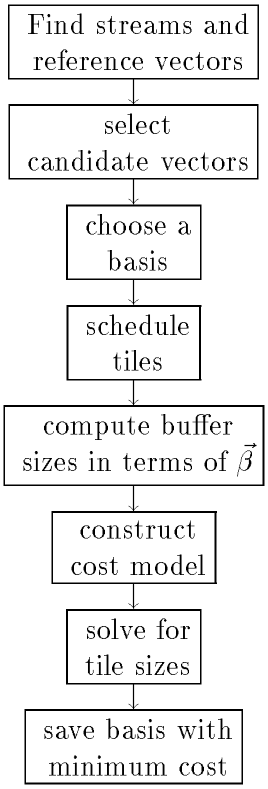


Figure 5.5: The order of compiler phases

The compiler must not choose the tile size vector before scheduling. Before a schedule exists streams with locality and streams without locality receive the same consideration in distributing valuable M_1 space. After locality is taken into account, the number of M_2–M_1 copies drops significantly for streams with intertile locality. This change in the cost model allows the compiler to be much smarter in choosing the tile size vector. For this reason, the compiler performs scheduling first, and chooses the tile size vector given the schedule.

5.2 Scheduling

When scheduling is performed first, the compiler has incomplete information about the eventual tile sizes. This prevents it from making perfect decisions. The primary goal of scheduling is to minimize execution time, but since tile sizes are not known at scheduling time, the compiler cannot compute the actual execution time that would result from different schedules. The scheduler must substitute the goal of maximizing potential intertile locality.

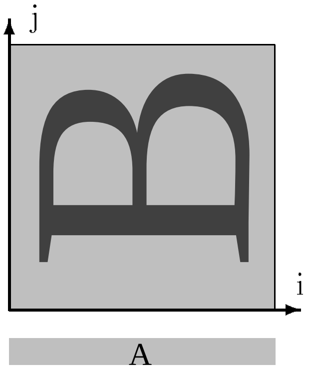
When tiles have the same size in each dimension (e.g., when $\vec{\beta} = (\alpha, \alpha, \ldots, \alpha)$), higher-dimensional streams take up much more space in M_1 than lower-dimensional streams. For example, a typical two-dimensional stream would require an $(\alpha \times \alpha)$-word buffer, while a typical one-dimensional stream would only require α words. The number of words used in M_1 by each stream is the number of words that must be moved into M_1 prior to tile execution (and the number of words that must be moved back after tile execution). Without *a priori* knowledge of the final tile sizes, the compiler maximizes potential intertile locality by keeping the higher-dimensional streams local in preference to lower-dimensional streams.

Intertile locality can be increased by increasing the number of streams held locally, or by increasing the dimensionality of the streams held locally. Because the compiler cannot know tile sizes before a schedule is chosen, the compiler maximizes the number of $(n-1)$-dimensional streams held locally. (Note that an n-dimensional stream inside an n-dimensional loop can never be held locally, because each iteration uses a different element.) Among all schedules with the maximal number of $(n-1)$-dimensional local streams, the compiler selects the schedule that has the most $(n-2)$-dimensional streams held locally, and so on.

5.2.1 Scheduling examples

A few examples of how the scheduler works will help to illustrate the important points. In Figure 5.6, two arrays are accessed in a two-dimensional loop nest. Since the `B` stream is the same dimensionality as the iteration space, each iteration uses a different element of the `B` matrix, and no locality is possible. There is locality available for the `A` matrix in the `j` direction. The `j` direction would therefore be scheduled innermost.

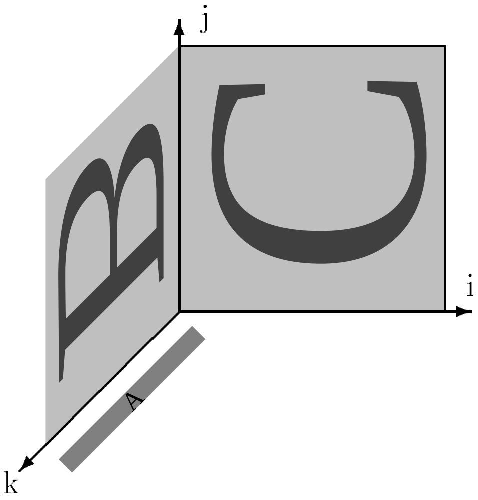
In Figure 5.7, three matrices are referenced in a three-dimensional loop nest. There



```
for i = 1 to n do
  for j = 1 to n do
    B[i,j] = B[i,j] * A[i];
```

Figure 5.6: A simple scheduling example

are two two-dimensional streams, `B` and `C`. The stream `B` is left local when `k` is innermost, and the stream `C` is left local when `i` is left local. Since `B` is read and written, twice as many references are saved by keeping it local, so the `i`-direction is chosen innermost. The `C` stream cannot be held locally once the `i` loop is chosen innermost. The `A` stream is left local, however. The compiler must therefore chose whether `k` or `j`will be the next innermost loop; choosing `j`leaves the `A` stream local, while choosing `k` does not. The final schedule is `k` outermost, `j` in the middle, and `i` innermost.



```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      B[k,j] = B[k,j] + C[i,j] * A[k];
```

Figure 5.7: A complex scheduling example

5.2.2 Calculating the number of refreshes

The controlling loops define a schedule of the tiles. The number of refreshes for each stream is computed once these outer loops are chosen.[3]

For each stream, we scan outward from the innermost loop searching for the first loop in which a stream is not local. Because the compiler has not yet determined the tile sizes, it cannot determine the exact number of times a controlling loop will execute. The total number of refreshes is approximated by summing over transformed loop bounds before tiling, starting at the innermost non-local controlling loop $\vec{e}_{nl}$ and moving outward:

$$\rho = \frac{1}{\beta_1} \sum_{i_1=l_1}^{u_1} \frac{1}{\beta_2} \sum_{i_2=l_2}^{u_2} \cdots \frac{1}{\beta_{nl}} \sum_{i_{nl}=l_{nl}}^{u_{nl}} 1 \tag{5.1}$$

where l_i and u_i are the lower and upper transformed loop bounds, respectively, and β_k is the spacing factor along $B_{k,*}$. Note that the formula above is for the number of times a buffer is filled with data from M_2; if the stream is written, that number must be doubled to account for the write-back.

The formula assumes that each loop will be executed enough times that fragmentation can be ignored. That is, if the loop bounds are `i=1 to n`, the formula yields $\mathtt{n}/\beta_{\mathtt{i}}$; but in fact the number of refreshes required is $\lceil \mathtt{n}/\beta_{\mathtt{i}} \rceil$. Loops which execute only a single iteration require a refresh even if β_i is greater than one. We implicitly assume that the compiler can tell loops which may execute only a few iterations from loops which execute many iterations. The rest of the thesis depends on the assumption that all loops in the loop nest execute a large number of iterations. Section 8.2 will outline a technique for removing this assumption.

As an example, recall the program of Figure 5.1 on page 65. The `A` matrix is local to the innermost controlling loop, so the number of refreshes for the `A` matrix is approximated by computing

$$\rho_{\mathtt{A}} = \frac{1}{\beta_{\mathtt{i}}} \sum_{\mathtt{i}=1}^{n} 1 = \frac{n}{\beta_{\mathtt{i}}}$$

---

[3] A *refresh* occurs any time data is moved from M_2 into M_1 as defined in Chapter 4; the term *refresh* may seem to imply that the data have already been moved in once, but this is not the intended meaning. A refresh operation empties M_1 of modified data, and fills it with data for the next tile. The very first tile requires a refresh operation prior to its execution, just as all the other tiles do.

The number of refreshes for the `B` stream is approximated by

$$\rho_{\mathtt{B}} = \frac{1}{\beta_{\mathtt{i}}} \sum_{\mathtt{i}=1}^{n} \frac{1}{\beta_{\mathtt{j}}} \sum_{\mathtt{j}=1}^{n} 1 = \frac{n^2}{\beta_{\mathtt{i}}\beta_{\mathtt{j}}}$$

5.2.3 Evaluating nested summations

Finding the number of refreshes requires the compiler to evaluate sums of polynomial expressions. Because the inner loop bounds may depend on outer loop bounds but not vice versa, the summations can be evaluated using simple rules for the value of polynomials. In evaluating a sum, the compiler works from the inside out. At each stage the summations bounds are represented as polynomials in the outer summation variables. The first step is to normalize the summation bounds to start at 0. Then sums are split using additive associativity rules, so that each summation is a constant times the summation variable raised to some exponent. Constants are moved outside the sums, and finally the sums of powers are replaced directly using power-coefficient rules. The following rules are used to simplify the sums down to the form $\sum_{v=0}^{h} v^p$:

$$\sum_{v=l}^{h} x = \sum_{v=0}^{h} x - \sum_{v=0}^{l-1} x$$

$$\sum_{v=0}^{h} (e_1 + e_2) = \sum_{v=0}^{h} e_1 + \sum_{v=0}^{h} e_2$$

$$\sum_{v=0}^{h} c = c(h+1)$$

$$\sum_{v=0}^{h} cx = c \sum_{v=0}^{h} x$$

In these rules, x, e_1, and e_2 stand for arbitrary expressions, while c stands for an expression not involving v, the summation variable.

The rules for evaluating $\sum_{v=0}^{h} v^p$ are not finite, but only the first four rules have been used in the prototype:

$$\sum_{v=0}^{h} 1 = h + 1$$

$$\sum_{v=0}^{h} v = \frac{h^2}{2} + \frac{h}{2}$$

$$\sum_{v=0}^{h} v^2 = \frac{h}{6} + \frac{h^2}{2} + \frac{h^3}{3}$$

$$\sum_{v=0}^{h} v^3 = \frac{h^2\,(1+h)^2}{4}$$

A general rule for generating coefficients can be found in [8]; in the prototype, this rule is incorporated into a recursive procedure for computing the coefficients.

The evaluation of a summation yields an expression in the loop bound variables. The number of M_2 memory accesses for a stream is therefore generally of the form

$$\frac{C}{\beta_0\beta_1\ldots\beta_{nl-1}} \tag{5.2}$$

where C is either constant (for constant loop bounds), or some expression of the loop bound variables.

5.2.4 Scheduling with parallelism

When tiles are executed in parallel, there is an additional complication. Dependences often prevent the processor array from simultaneously starting on tiles. Instead, the second tile cannot start until results of the first tile's execution are available, and so on. The tiles can be executed along a wavefront using DO-ACROSS-style parallelism, but there is a latency between the start-time of two tiles which is dependent of tile size.

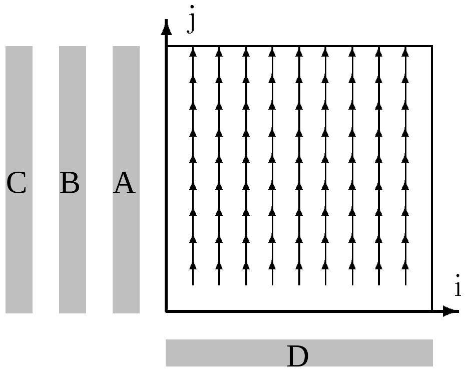
The compiler cannot compute the latency of tile start-up because the tile sizes are not known. This means the scheduler cannot compute the execution order which minimizes the total execution time, because the start-up latency contributes to execution time.

The compiler must settle for picking the execution order which maximizes potential intertile locality. If the problem is large enough, the intertile start-up latency will be much smaller than the total execution time (because intertile start-up latency is proportional to the number of processors and the tile size, but is independent of problem size), while data motion costs are often proportional to the problem size.

5.3 Approaches to parallelism and locality

For our purposes, the multitude of methods of parallelization across multiple processors (as opposed to, say, instruction-level parallelism within a single processor) may be classified into two basic types of parallelism: intertile parallelism, where the iteration space is tiled, and tiles are doled out to single processors to be executed; and intratile parallelism, where all the processors work on a single tile at the same time.[4]

Tiling can be performed with the goal of increasing locality, or it can be performed with the goal of creating parallelism between tiles. For parallelism, the compiler's goal is to generate enough tiles to guarantee all processors can be kept busy without introducing too much overhead. For locality, the compiler's goal is to generate tiles which fit into a faster level of the memory hierarchy. The tiles should maximize the ratio of computation to secondary memory bandwidth consumed. These goals can conflict, since the directions in the iteration space in which there is reuse may also be the directions in which there is parallelism. As an example, consider Figure 5.8. This two-dimensional loop references four matrices. The matrices `A`, `B`, and `C` are read-only, while the matrix `D` is both read and written. This results in dependences in the `j`-direction. The `i`-direction can be executed in parallel, but the most data locality exists in the `i`-direction.



```
for i = 1 to n do
  for j = 1 to n do
    D[i] = D[i] + A[j]+B[j]*i+C[j]*i*i;
```

Figure 5.8: Parallelism can exist in the preferred locality direction.

There are several possible approaches to the problem: the compiler could tile twice, tiling the loop nest once to obtain parallelism, and then tiling the tiled loops a second

---

[4] Intertile parallelism and intratile parallelism can be combined, resulting in a scheme where groups of processors cooperate to execute single tiles, and multiple groups work in parallel. This thesis considers only the two simpler cases.

time, scheduling the second set of tiles for intertile locality. The second approach is to tile only once, and schedule the tiles to obtain both parallelism and intertile locality. The third approach is to tile and schedule for intertile locality assuming that the tiles will be parallelized for the entire processor set using intratile parallelism. This is the only way to integrate tiling for data-motion management with parallelizing methods which introduce complex communication patterns: bidirectional communication among tiles cannot be allowed unless a new scheduling methodology is developed, but any method can be used to execute the iterations within a tile (at least as far as the tile scheduler is concerned).

5.3.1 Tiling twice: Wolf's method

In his thesis[57], Wolf discusses combining parallelism and locality by first tiling the iteration space to find coarse-grain parallelism and then tiling the coarse-grain tiles to obtain sub-tiles which fit into the memory hierarchy level of interest. This technique has the advantage that it allows up to $n-1$ degrees of parallelism to be extracted from an n-deep nested loop. If the iteration space is so large that the problem will not fit in the machine, however, a single degree of parallelism may well be sufficient. Further, tiling for coarse-grain parallelism limits intertile locality to that available within a coarse-grain tile as opposed to that available within the iteration space.

5.3.2 Scheduling for intertile parallelism

This thesis investigates the possibility of tiling once, and scheduling the tiles to obtain parallelism in some directions and intertile locality in other directions. In this approach, it is assumed that there is enough parallelism in any one parallel direction in the iteration space to keep all the processors busy. The concentration is not on producing all the parallelism possible, but rather on producing enough parallelism to keep the processors busy; the rest of the directions in the iteration space are used to schedule for intertile locality.

Scheduling for intertile locality and intertile parallelism requires the scheduler to separate the loop nest into a set of loops to be executed in parallel, and a set to be executed sequentially. Iterations of a parallel loop may be executed on any processor, but all iterations of a sequential loop are executed on the same processor.

The compiler first marks every loop as potentially parallel or necessarily sequential. If there is only a single parallel loop, it is always placed outermost. Next, each loop is marked with how many streams are left local if that loop is innermost. The loop that leaves the most streams local is placed innermost. The next outermost loop is chosen by determining how many streams are local to both the inner most loop and the next outermost, and so on.

If there are multiple parallel loops, each one is a candidate for being outermost. In this case, the loop with the most intertile locality is selected as the innermost loop. The compiler proceeds as before, adding loops to the outside of the loop nest, except before a loop is added, the compiler checks how many potential parallel loops are left unassigned. When there is only one parallel loop, it is placed outermost.

In the matrix multiply example, the `i` and `j` loops are fully parallel, so they are both candidates for the outermost position. The `k` loop is not considered parallel, since the dependence analyzer does not take the associativity of addition into account. Each loop leaves one stream local, but the `k` loop leaves the writable stream `c[i,j]` local, which represents twice as many memory transactions. The compiler puts the `k` loop innermost. Since there are still two choices for a parallel outermost loop, the compiler tries to choose another loop with intertile locality. At this point, there is no locality left. The compiler can arbitrarily pick either the `i` or `j` loop to be outermost.

Finding a transformed loop with no parallelism is rare; there are no such loops in our examples in Chapter 7. In the case that there is no parallel loop in the loop nest, the controlling loops are first skewed until there is a parallel loop (an n-deep tilable loop nest can be transformed to code containing at least $n-1$ degrees of parallelism[29]). This complicates the expression for the number of refetches. The expression in Equation 5.1 is based on the fact that the number of tiles in loop direction x is simply the size of the loop divided by the size of the tile. Skewing changes the number of iterations executed in a direction.

Note that when loops are executed in parallel, the refresh operations happen in parallel (because writable data is not shared, so everything can happen locally). The number of refreshes computed by Equation 5.1 is therefore divided by the number of processors.

5.3.3 Scheduling for intratile parallelism

The compiler can tile for locality and intratile parallelism; the resulting tiles are to be executed by the entire array working as a unit. After tiling, the tiled loops are passed on to a parallelization phase. Arbitrary communication between processors is allowed during the execution of a tile; the tiling software places a barrier synchronization before and after the execution of each tile. This allows maximum flexibility in choosing tiles, since forms of parallelism using communication can be used. The scheduler operates in almost the same way for intratile parallelism as it does for a uniprocessor, so optimal intertile locality is available.

Some cooperation between the tiler and the parallelization phase are required, however. The scheduler must be able to obtain cost metrics for executing a parametric-sized tile on the entire set of processors. The scheduler models the parallel machine as a single processor with a single fast memory, but which may have nonlinear execution cost measures for different tiles sizes or shapes.

Determining what fits in a distributed memory isn't quite the same as determining what fits in a single memory—it may be better to trade data replication for communication, and this would decrease the effective memory size. The compiler can handle this in two different ways: it can target a fraction of the available memory, assuming that the resulting tile will fit even after data replication; or it can complicate the expressions giving tile size.

When the cost model is evaluated, the compiler is attempting to minimize the number of slow memory accesses, expressed in terms of the tile sizes, subject to a memory bound constraint: the sum of the M_1 memory allocations can not exceed the physical size of M_1. When intratile parallelism is applied, however, the real constraint is that the M_1 memory allocation *in each processor* must not exceed the M_1 size *of that processor.* Data placement for intratile parallelism is done by the intratile parallelizer, which may choose to replicate some data across the memories of each processor, effectively reducing the aggregate M_1 size. This requires replacing our memory constraint with a new constraint smaller than M_1, based on how much data is replicated.

5.3.4 Comparison of approaches

Tiling for intertile parallelism can easily be combined with tiling for locality by tiling twice. An alternative approach is to tile only once, and schedule the resulting tiles for parallelism in one dimension while scheduling for intertile locality in other directions.

Tiling for intratile parallelism is somewhat more complex, in that the compiler must target the full processor array. Data replication has the effect of shrinking the available memory, but the compiler cannot determine the exact degree of replication until after it has decided on the tile sizes.

Chapter 6

Cost model evaluation

In previous chapters, various pieces of the cost model were described in some detail. This chapter explains the details of evaluating the cost model, solving for the value of the tile size vector $\vec{\beta}$, and computing the total cost of the tiling.

The development thus far has shown how to compute the size of the buffer used by each stream in terms of $\vec{\beta}$, and roughly how to compute the number of times these buffers are filled and emptied, also in terms of $\vec{\beta}$. To complete the cost model, the number of refreshes required must be computed. This requires the loop bounds in the transformed space. The next section describes the techniques used to generate the new loop bounds. First, the techniques of Li and Pingali for generating the new loop bounds are reviewed. Next, several improvements to standard Fourier-Motzkin elimination are discussed; this is the process used to solve the transformed loop bounds into expressions acceptable in a standard imperative language.

The two pieces of the cost model are then combined into a single optimization problem. The solution to this optimization problem is the correct tile size vector $\vec{\beta}$. The second section of this chapter is devoted to examining different approaches to solving this optimization problem.

The last section of this chapter is devoted to a complete example loop, showing how it is transformed at each stage, so that the reader can get a feel for how all the pieces of theory fit together.

6.1 Code generation

The transformation required has already been discussed in general terms: the iteration space is first transformed to have the new basis B, and then the loops are strip-mined to produce tiles. The new basis B is chosen so that tiling is always legal. Chapter 4 described the transformation applied to subscript expressions; transforming expressions using the old loop indices to equivalent expressions in the new basis is straightforward. Generating new loop bounds, however, is complex; that is the subject of this section.

The code generation algorithm is based on the work of Li and Pingali[35]. The problem is to transform a source loop nest to a target loop nest that executes an equivalent set of iterations in a new basis B. The index variable set in the source loop nest is given by $\vec{\imath}$; in the target space we will use $\vec{\jmath}$. Each index point in the source space is related a point in the target space by the equation $\vec{\jmath} = B\vec{\imath}$. The compiler transforms subscript expressions by replacing expressions in $\vec{\imath}$ with an equivalent expression in $\vec{\jmath}$. A variable reference `v[`$R\vec{\imath} + \vec{r}$`]` is replaced by `v[`$RB^{-1}\vec{\jmath} + \vec{r}$`]`.

The loop bounds expressions cannot be as easily replaced, because while the inner loop bounds are allowed to be functions of the outer loop indices, the reverse is not true. The loop bounds are therefore transformed in two stages. First, the bounds are translated to an auxiliary space and simplified. The second step translates the loop nest in auxiliary space into a loop nest in the target space.

6.1.1 Transforming original loop bounds to auxiliary space

We re-write the loop nest bounds in the original space as matrices:

`for` $\vec{\imath} = L\vec{\imath} + \vec{l}$ `to` $M\vec{\imath} + \vec{m}$ `do` $\{\ldots\}$

This corresponds to the inequalities

$$L\vec{\imath} + \vec{l} \leq \vec{\imath} \leq M\vec{\imath} + \vec{m}$$

which can be re-written as

$$\begin{bmatrix} L - I \\ I - M \end{bmatrix} \vec{\imath} \;\leq\; \begin{bmatrix} -\vec{l} \\ \vec{m} \end{bmatrix}$$

$$\vec{\imath} \quad \text{integer}$$

Letting $A = \begin{bmatrix} L - I \\ I - M \end{bmatrix}$ and $\vec{b} = \begin{bmatrix} -\vec{l} \\ \vec{m} \end{bmatrix}$, we can write the system of inequalities as

$$\begin{array}{ll} A\vec{\imath} \leq \vec{b} \\ \vec{\imath} & \text{integer} \end{array}$$

Our job is to find the loop bounds in the space transformed by the new basis matrix B. The new index vector will be $\vec{\jmath}$. Since B has full rank, it represents a one-to-one mapping of points in the old iteration space into points in the new iteration space. We thus have $\vec{\jmath} = B\vec{\imath}$. We can find a solution by letting $B = HU$ where U is unimodular, and H is lower-triangular with positive diagonal elements. Let $\vec{k} = U\vec{\imath}$ so that $\vec{\jmath} = H\vec{k}$. We can re-write these as

$$\vec{\imath} = U^{-1}\vec{k}$$

and

$$\vec{k} = H^{-1}\vec{\jmath}$$

Letting $A' = AU^{-1}$, we can re-write our system of inequalities as

$$\begin{array}{ll} A'\vec{k} \leq \vec{b} \\ \vec{k} & \text{integer} \end{array}$$

Any solution $\vec{k}_o$ to the new system is a solution to the original system because U is unimodular (and so is U^{-1}). We can solve the system in A' by using Fourier-Motzkin pairwise elimination. We then need to translate the loop bounds for $\vec{k}$ into loop bounds for $\vec{\jmath}$.

6.1.2 Transforming auxiliary space loop bounds to target space

After Fourier-Motzkin elimination, the loop for variable k_m is of the form

`for` k_m `= max(`$\bar{p}^1\bar{k}, \bar{p}^2\bar{k}, \ldots, \bar{p}^x\bar{k}$`) to min(`$\bar{q}^1\bar{k}, \bar{q}^2\bar{k}, \ldots, \bar{q}^y\bar{k}$`) do` $\{\ldots\}$

where $\bar{k} = (k_1, k_2, \ldots, k_{m-1})$, $\bar{p}^z = (p^z_1, p^z_2, \ldots, p^z_{m-1})$, and $\bar{q}^z = (q^z_1, q^z_2, \ldots, q^z_{m-1})$. We can find bounds for j_m by replacing $\bar{k}$ by its equivalent in terms of $\bar{\jmath}$.

Since H is non-singular and lower-triangular, H^{-1} is also lower triangular. This allows us to replace $\bar{k}$ with a linear combination of $\bar{\jmath}$ and vice-versa. Since $\vec{\jmath} = H\vec{k}$, it must be that $\bar{\jmath} = \bar{H}\bar{k}$ and $\bar{k} = \bar{H}^{-1}\bar{\jmath}$.

Now $j_m = H_{m,m}k_m + \sum_{i=1}^{m-1} H_{m,i}k_i$. Letting $v = \bar{H}_m\bar{k} = \bar{H}_m\bar{H}^{-1}\bar{\jmath}$, where H_m is the mth row of H, we have $j_m = v + H_{m,m}k_m$. Here v remains constant; so we need to determine how j_m relates to k_m. Since k_m steps from $\max(\bar{p}^1\bar{k}, \bar{p}^2\bar{k}, \ldots, \bar{p}^x\bar{k})$ to $\min(\bar{q}^1\bar{k}, \bar{q}^2\bar{k}, \ldots, \bar{q}^y\bar{k})$, $H_{m,m}k_m$ should step from $H_{m,m}\max(\bar{p}^1\bar{k}, \bar{p}^2\bar{k}, \ldots, \bar{p}^x\bar{k})$ to $H_{m,m}\min(\bar{q}^1\bar{k}, \bar{q}^2\bar{k}, \ldots, \bar{q}^y\bar{k})$ by steps of $H_{m,m}$. Once again, we replace $\bar{k}$ with $\bar{H}^{-1}\bar{\jmath}$; this is accomplished by multiplying $\bar{q}^z$ and $\bar{p}^z$ by $\bar{H}^{-1}$ for all z.

The loop for variable j_m is of the form

```
for $j_m$ = $\bar{H}_m\bar{H}^{-1}\bar{\jmath} + H_{m,m}\max(\lceil\bar{p}^1 H^{-1}\bar{\jmath}\rceil, \lceil\bar{p}^2 H^{-1}\bar{\jmath}\rceil, \ldots, \lceil\bar{p}^x H^{-1}\bar{\jmath}\rceil)$
        to $\bar{H}_m\bar{H}^{-1}\bar{\jmath} + H_{m,m}\min(\lfloor\bar{q}^1 H^{-1}\bar{\jmath}\rfloor, \lfloor\bar{q}^2 H^{-1}\bar{\jmath}\rfloor, \ldots, \lfloor\bar{q}^y H^{-1}\bar{\jmath}\rfloor)$ step $H_{m,m}$
        do $\{\ldots\}$
```

Note that $\bar{H}_m\bar{H}^{-1}$ is not zero. The $(m-1)$-element vector $\bar{H}_m$ is the mth row of H *minus its diagonal element.*

6.1.3 Improvements to Fourier-Motzkin pairwise elimination

Fourier-Motzkin pairwise elimination is a general method of solving systems of inequalities. The basic idea is to eliminate variables one at a time until only a single variable is left. Duffin's paper[14] is the best introduction to the subject. A more rigorous approach is taken by Dantzig and Eaves[13].

Eliminating redundant inequalities

Pairwise elimination can cause the number of inequalities to grow exponentially. Duffin[14] described how to minimize the number of inequalities by applying some simple rules for choosing which variable to eliminate and for eliminating redundant inequalities. Since the outer loop bounds cannot be expressed in a procedural language as a function of the inner loop indices, the compiler must eliminate variables starting with the innermost loop index and proceeding outwards. The compiler can apply Duffin's rules for elimination of redundancy introduced by pairwise elimination.

In Duffin's work, each original inequality has its own *parametric term* λ_i (his notation). Inequalities are eliminated by adding positive multiples of inequalities together, so every resulting inequality will have all positive parametric terms. Duffin's Rule (b) states that any inequality can be eliminated that has more than $t+2$ positive parametric terms after t variables have been "actively eliminated." The phrase "active elimination" refers to eliminating a variable for which there is at least one inequality with a non-zero coefficient; variables that have zero coefficients in all inequalities can be "passively eliminated" at any time, preferably as soon as possible.

In generating loop bounds, the compiler is not dealing with parametric inequalities, but Duffin's method can be easily adopted: before pairwise elimination is begun, the original inequalities are numbered 1 to $2n$ (there are $2n$ inequalities, an upper and lower bound for each loop). A $2n$-element vector is kept for each inequality. Initially, these vectors are zero everywhere except the ith position is set to 1 for the ith inequality. As inequalities are multiplied and added, the extra vectors are multiplied and added the same way. These vectors simulate the parametric terms in Duffin's method. After t variables have been actively eliminated, the compiler can eliminate any inequalities with $t+2$ non-zero entries in its "parameter" vector. In the prototype compiler, this is made especially fast by the observation that there are only two important states in the parameter vector: zero and non-zero. Once a parameter is made non-zero, it will be non-zero in every inequality it is added into. Since the maximum loop nest depth is always less than 16, there are always less than 32 initial inequalities. The prototype therefore represents the parameter vectors as bit vectors using a single word of storage per inequality.

Duffin's Rule(c) for eliminating dominance can also be easily applied using the bit-vector representation. An inequality a dominates another inequality b unless there is at least one i for which the ith element of a's parameter vector is set while b's ith element is clear. Rule (c) states that any dominated inequality can be eliminated; in the prototype this is implemented with bitwise arithmetic: the parameter vector of a is bit-wise and-ed with the one's complement of b's parameter vector; a dominates b if and only if the result is zero (in all bits). The symmetric operation is used to compute whether b dominates a.

Back-propagation

Classic Fourier-Motzkin elimination is a forward-only process; once the outer loop bounds are determined, the process stops. Duffin's methods help eliminate redundancy introduced by pairwise elimination, but it certainly does not eliminate all redundancy in the system of inequalities. This can lead to unnecessarily complex expressions for inner loop bounds. Since the compiler uses these loop bound expressions to compute the cost model, it is worthwhile to simplify these expressions by propagating constraints on the outer loop indices inward to eliminate redundant constraints on the inner loop bounds. This process of using outer loop bounds to simplify inner loops bounds propagates information inwards, whereas Fourier-Motzkin elimination propagates information outwards; for this reason the process is referred to as *back-propagating* the constraints.

Back-propagating constraints is difficult in the general case. Fortunately, the practical situations that cause the need for back-propagation are simple enough that a simple, fast algorithm can handle the common cases easily. Rectangular loops never require back-propagation. Triangular loops can cause the need for back propagation, as will be seen in the QR decomposition example below. This covers nearly all the loops used in scientific codes that have linear loop bounds.

In the prototype compiler, the loop bounds are scanned from outermost to innermost. When a loop is found with multiple upper (or lower) bound inequalities, the compiler tries to prove that all but one of the inequalities is redundant, given the outer loop bounds. The coefficients in each loop bound expression are searched for a non-zero coefficient in an outer loop index. A non-zero coefficient means that the outer loop bounds affect the inequality, so propagating information inward may help.

Once an opportunity for back-propagation has been found, the compiler must find an outer loop inequality to add in. Either the upper or lower loop bound inequality for the non-zero coefficient will be added in. Which it will be is determined by the sign of the coefficient and by whether an upper or lower bound is being eliminated. When an upper loop bound is being eliminated, the lower loop bound of the outer loop is added in if the coefficient is *negative,* otherwise the upper loop bound of the outer loop is added in. When eliminating a lower loop bound, the upper outer loop bound is added for a negative sign, and the lower for a positive sign.

As an example, consider the QR-decomposition code of Figure 6.1. In this case, $\vec{\imath} =$ (`k,i,j`), $L = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$, $\vec{l} = (0,0,0)$, $M = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, and $\vec{m} = (119,119,119)$. The

```
for k = 0 to 119
  for i = k to 119
    for j = k to 119
      if (i == k) then
        r[k,j] = a[i,j];
      else
        if (j == k) then
          c = r[k,j]/SQRT(r[k,j]*r[k,j]+a[i,j]*a[i,j]);
          s = a[i,j]/SQRT(r[k,j]*r[k,j]+a[i,j]*a[i,j]);
          r[k,j] = c*r[k,j] + s*a[i,j];
        else
          rt = r[k,j];
          r[k,j] = s*a[i,j] + c*rt;
          a[i,j] = c*a[i,j] - s*rt;
        endif
      endif
```

Figure 6.1: QR-decomposition

system of inequalities $A\vec{\imath} \leq \vec{b}$ is given by

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{k} \\ \mathtt{i} \\ \mathtt{j} \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 119 \\ 119 \\ 119 \end{bmatrix}$$

The dependences in QR decomposition are $(\leq, 0, 0)$ and $(0, \leq, 0)$. Suppose the compiler wishes to move the `i`-loop outermost, by interchanging the `k` loop inwards over the `i` loop. For clarity, the transformed loop index variables will be called `u`, `v`, and `w`. The desired transformation corresponds to choosing a new basis $B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The transformed inequality set is:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{u} \\ \mathtt{v} \\ \mathtt{w} \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 119 \\ 119 \\ 119 \end{bmatrix}$$

To apply Fourier-Motzkin elimination, the first step is to solve for the innermost loop bounds. This involves reading off the inequalities that have non-zero `w`-coefficients. The result is $\mathtt{v} \leq \mathtt{w} \leq 119$. Once the bounds for `w` have been saved, `w` is eliminated from the inequalities. All variables must be actively eliminated; passive elimination implies there are no loop bound inequalities for an index variable. This cannot occur in practice. Actively eliminating a variable means every equation with a positive coefficient for the variable to be eliminated must be added to every inequality with a negative coefficient. At each step, if the number of inequalities before elimination is N, after elimination there can be as many as $(N-1)^2$ inequalities. This could theoretically lead to an exponential increase in the number of inequalities; in practice this rarely occurs.

Eliminating `w` from the previous system of inequalities results in

$$\begin{bmatrix} 0 & -1 \\ -1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{u} \\ \mathtt{v} \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 119 \\ 119 \\ 119 \end{bmatrix}$$

Note that the last row has the same coefficients as the third row; when two inequalities have the same coefficients, the tighter bound is kept and the looser bound can be eliminated (assuming the compiler can detect which bound is tighter). In this case, they are equivalent, so either inequality can be dropped. This is called *equal-coefficient redundancy elimination;* Duffin does not discuss this form of redundancy elimination but it is straightforward and easy to implement in a compiler.

After dropping the bottom inequality, there is one inequality with a negative coefficient for `v`; that is, there is one lower bound inequality. There are two upper bound inequalities. Both upper bounds must hold, so the upper bound for `v` is the minimum of the two. The next set of loop bounds is therefore $0 \le \mathtt{v} \le \min(\mathtt{u}, 119)$. Eliminating `v` from the new set of inequalities leaves

$$\begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \begin{bmatrix} \mathtt{u} \end{bmatrix} \le \begin{bmatrix} 119 \\ 0 \\ 119 \end{bmatrix}$$

The third row has all zero coefficients; in a general Fourier-Motzkin elimination procedure, such rows would be checked for non-negative right-hand sides; negative right-hand sides indicate that the original system of inequalities has no solutions. This corresponds to loop nests whose bodies are never executed, as in "`for i = 1 to 0 do` *body*". The right-hand side may be an expression that cannot be evaluated at compile-time; in this case the check is skipped (the compiler has to assume the loop nest will be executed if it cannot prove otherwise).

The outer loop bounds can be found by inspection: $0 \le \mathtt{u} \le 119$. Traditional Fourier-Motzkin elimination ends at this point. By back-propagating constraints, however, the compiler can show that the loop bounds for the `v` loop can be simplified to $0 \le \mathtt{v} \le \mathtt{u}$.

6.1.4 Computing the number of refreshes

The output of code generation requires us to sum outwards where the upper bound is the minimum of several linear functions, and the lower bounds are maximums of several linear functions. If we can determine where these minimums and maximums occur at compile time, we can split the summations into piece-wise summations; in effect, we generate several consecutive loops to execute the code, and each consecutive loop can be summed.

In general, however, the problem of computing the number of iterations of a loop nest is equivalent to finding the number of solutions to an integer linear programming problem. This problem is very difficult; specifically, it is in a class of problems called $P\sharp$ ("P sharp"). This class is more difficult to solve than NP-class problems.

If a compiler encounters a program in which the transformed loop bounds are only piecewise linear, and the loop bounds cannot be computed at compile time, that basis can be dropped from consideration. In practice, loop bounds are nearly always purely linear, because array subscripts are almost always simple loop indices, and the compiler chooses transformations from the reference vector set.

6.2 Evaluating the cost model

In Chapter 4 the number of fetches per tile for each stream was computed as a function of the tile size vector $\vec{\beta}$. In Chapter 5, the number of times that data will be fetched (or stored) was computed, also in terms of $\vec{\beta}$. The total cost in terms of fetches and stores can now be expressed in terms of $\vec{\beta}$. The compiler's task is to minimize this cost by choosing values for $\vec{\beta}$ subject to the constraint that the resulting stream buffers must all fit together in M_1.

6.2.1 An example

As an example, take the matrix multiply example of Figure 3.1. The tiling basis chosen is I, and the controlling loops are kept in the original order `i`, `j`, `k`. The amount of data that must be fetched per tile for each stream is given by

$$\begin{aligned} \mu_{\texttt{c[i,j]}} &= \beta_i \beta_j \\ \mu_{\texttt{a[i,k]}} &= \beta_i \beta_k \\ \mu_{\texttt{b[k,j]}} &= \beta_k \beta_j \end{aligned}$$

The number of times each stream will be refreshed is given by

$$\begin{aligned} \rho_{\texttt{c[i,j]}} &= \frac{\mathtt{n}^2}{\beta_i \beta_j} \\ \rho_{\texttt{a[i,k]}} &= \frac{\mathtt{n}^3}{\beta_i \beta_j \beta_k} \\ \rho_{\texttt{b[k,j]}} &= \frac{\mathtt{n}^3}{\beta_i \beta_j \beta_k} \end{aligned}$$

The total number of memory operations for matrix multiply using this basis is then

$$X = \frac{2\mathtt{n}^2 \beta_i \beta_j}{\beta_i \beta_j} + \frac{\mathtt{n}^3 \beta_i \beta_k}{\beta_i \beta_j \beta_k} + \frac{\mathtt{n}^3 \beta_k \beta_j}{\beta_i \beta_j \beta_k}$$

which simplifies to

$$2\mathrm{n}^2 + \frac{\mathrm{n}^3}{\beta_j} + \frac{\mathrm{n}^3}{\beta_i} \tag{6.1}$$

The compiler now needs to find the minimum of 6.1 subject to the memory constraint

$$\beta_i\beta_j + \beta_i\beta_k + \beta_k\beta_j \leq \mathrm{size}(M_1)$$

The tile sizes cannot be zero or negative, so the compiler implicitly has the constraints

$$\begin{aligned} 1 &\leq \beta_i \\ 1 &\leq \beta_j \\ 1 &\leq \beta_k \end{aligned}$$

The problem is a nonlinear optimization problem. Fortunately, because of symmetry it is clear the optimal solution has $\beta_i = \beta_j$. Since β_k does not appear in the cost formula, the optimal solution must have β_i and β_j as large as possible, and $\beta_k = 1$. Taking $\beta_k = 1$ and $\beta_i = \beta_j$, from the memory constraint it is clear that $\beta_i = \beta_j = \sqrt{\mathrm{size}(M_1) + 1} - 1$.

6.2.2 The general problem

In general the total cost of a tiling for a set of streams V is

$$X = \sum_{v \in V} \rho_v(c_b + c_w * \mu_v) \tag{6.2}$$

where c_b is the overhead of a block move and c_w is the cost per word of a block move. The general form of the memory constraint is

$$\sum_{v \in V} \mu_v \leq \mathrm{size}(M_1) \tag{6.3}$$

that is, the sum of the buffer sizes must be less than the available memory size. There are n other constraints which force each element of $\vec{\beta}$ to be positive:

$$\vec{1} \leq \vec{\beta} \tag{6.4}$$

The number of refreshes ρ_v is an expression in $\vec{\beta}$ of the form $\frac{c}{\beta_0\beta_1\ldots\beta_k}$ where k is the innermost nonlocal loop, and c here is a constant term reflecting the total number of iterations in the loop nest.

When all the reference vectors are included in the tiling basis, a more specialized form of the general problem results. Since this is the case whenever there is only a single candidate basis (which is true for all the scientific loops described in Chapter 7), this case deserves special consideration.

When all the reference vectors of a stream s are included in the basis, S will be a permutation of the rows of the identity matrix, so the buffer size μ_v is an expression of the form $\mu_v = \beta_x\beta_y\ldots\beta_z$. Note that in particular, if μ_v contains β_x, β_y, etc., these vectors must appear in ρ_v as well, because there cannot be locality in directions of increasing subscripts.

In this special case, the elements of $\vec{\beta}$ in the numerator cancel out the corresponding elements in the denominator. All of the β_i's in the numerator cancel out, although some may be left in the denominator. The stream's contribution to the cost formula is therefore of the form

$$X = \frac{c}{\beta_x\beta_y\ldots\beta_z} \tag{6.5}$$

that is, a constant (or an expression in program variables) divided by several of the elements of $\vec{\beta}$. The partial derivatives of this special form are strictly non-positive whenever $\vec{\beta} \geq \vec{1}$; this guarantees that there are no local minima which are not global minima, so numerical techniques can easily find the global minimum using a simple gradient search, modified to search along the boundary of the feasible region.

6.2.3 Numerical techniques

If the loop bounds are known (or can be estimated) at compile-time, various numerical techniques can be applied to find the optimal $\vec{\beta}$. The general problem is to optimize (6.2) subject to the constraints (6.3) and (6.4). This is an integer nonlinear programming problem (INLP). It can be approximated by a real-valued nonlinear programming problem (NLP). Techniques for solving NLP's are much more well-developed. The mathematics software package MATLAB contains an off-the-shelf NLP solver. The next section describes it use. The section after that discusses properties of the problem under consideration that may affect the choice of a solution technique if an off-the-shelf solver is not available.

Using MATLAB to solve constrained optimization problems

MATLAB is a program for doing mathematics. As part of its Optimization Toolbox[20], MATLAB contains the function `constr`, which implements a constrained optimization solver. This solver is based on a Sequential Quadratic Programming method. The interface is straightforward. The caller must specify the function to be minimized (in this case, X), the constraints (bounds are easier to specify explicitly as bounds than as constraints), and an initial guess. The easiest initial guess to formulate is $\vec{\beta} = \vec{1}$, but $\vec{\beta} = (b, b, \ldots, b)$ is probably closer to the optimal value. The memory constraint can be solved for the value of b (a single equation in one unknown).

The solver requires constraints to be of the form $g(\vec{x}) \leq 0$, so (6.3) is re-written

$$\left(\sum_{v \in V} \mu_v\right) - \text{size}(M_1) \leq 0$$

The left side of this inequality is the constraint function $g(\vec{\beta})$.

Applying other techniques

A full description of techniques for solving NLP's is beyond the scope of this thesis; the book by Wismer and Chattergy[53] is a good reference for readers needing background material. This section discusses particular facets of the problem to be solved which allow selection of an appropriate technique.

Both the first and second derivatives of the cost function are continuous in the feasible region, so several types of gradient descent techniques can be applied, including "Newton" techniques which use the second derivative to achieve faster convergence. Since there are inequality constraints, a constrained optimization technique is required.

Intuitively, the compiler should use up all of the available M_1 space, so the optimum value of $\vec{\beta}$ must lie on the memory constraint. That is, (6.3) holds as an equality (and the remaining constraints are all just lower bounds). This reduces the solution space by one dimension. There are two approaches to using this information. In the first approach, (6.3) is solved for some β_k. The formula for β_k can then be substituted everywhere else and the constraint can be dropped (but kept around to reconstruct the optimal β_k). This method is excellent for solving problems by hand, but can be difficult to automate.

In the second approach, the equation is kept as a constraint and gradient projection is used to find an optimal solution by walking along the ($n-1$-dimensional) surface specified by the equation. In essence, the chain rule is applied to find the gradient of the cost function along the surface, and step are taken along the surface in the direction of steepest descent. As an initial guess, the vector $\vec{\beta} = (b, b, \ldots, b)$ can be used.

Penalty function methods could also be used, but must be applied with care. A penalty function method turns the constrained optimization problem into an unconstrained optimization problem by introducing new variables to satisfy the constraints, but adding severe penalties to the cost whenever the constraints aren't satisfied. There are two basic kinds of penalty methods, classified according to whether the optimal solution is approached from within the feasible region, or from outside the feasible region. Methods which approach the solution from inside the feasible region are called interior methods. Methods which approach the solution from outside the feasible region are called exterior methods. For the particular case of the cost model developed here, exterior methods are dangerous in that there is the possibility of encountering a singularity if any β_i is zero. These singularities do not exist inside the feasible region. It is possible that by choosing the correct starting point, the path to a solution can be kept away from these singularity regions; this is left to future work.

To ensure optimality, the loop bounds must be known at compile time. If some of the loop bounds are not known at compile time, they can be approximated with some loss of quality of the final result, so long as the assumption that every loop executes a large number of iterations (large with respect to $\vec{\beta}$) holds. The loop bounds change the constant in each term of the cost model; they do not change whether a particular $\vec{\beta}$ is in the denominator or not. The $\vec{\beta}$ values which appear in the denominator of some term must be made large lest that term contribute too greatly to the cost. In effect, the solution space is bimodal; elements of $\vec{\beta}$ which do not appear in the denominator of the cost formula are set to 1, while elements of $\vec{\beta}$ which do appear in the denominator must be made fairly large.

6.3 A complete example

A complete example will help the reader to understand how all of the theory fits together. Livermore loop kernel six is the most complex example in the benchmark set examined, which includes the Livermore loops, the Perfect Club, and the FORTRAN SPECmark codes. It neatly illustrates many of the features of the theory developed in this thesis.

The source code for Livermore loop kernel number six is shown in Figure 6.2. There is a single loop-carried dependence vector, $(+, *)$. This dependence prevents tiling, since no transformation can make this dependence positive in the `k`-loop.

```
for i = 1 to n do
  for k = 0 to i-1 do
    w[i] = w[i] + b[i,k]*w[i-k-1];
```

Figure 6.2: Livermore loop kernel six

If the programmer can take advantage of the associativity of addition, he or she can re-write the loop as shown in Figure 6.3; the only difference is that the `k`-loop has been reversed. The new version has constant dependences, so it can be made tilable.

```
for i = 1 to n do
  for k = -i+1 to 0 do
    w[i] = w[i] + b[i,-k]*w[i+k-1]
```

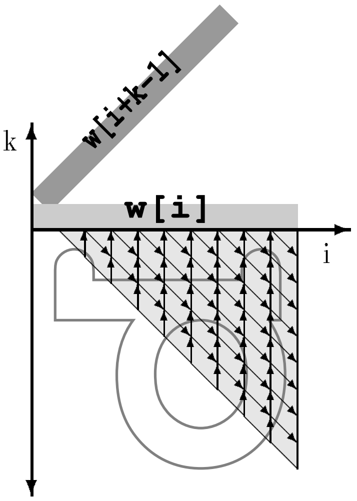


Figure 6.3: Livermore loop kernel six with `k` loop reversed

Three streams are referenced in a two-dimensional loop nest. Two of the streams (`w[i]` and `w[i+k-1]`) are one-dimensional; the third stream (`b[i,-k]`) is two-dimensional. Since only the `w[i]` stream is written, the dependences which restrict the execution order are induced by this stream. There is an output dependence in the `k` direction, and a flow dependence in the `i-k` direction. These dependences are drawn in the iteration space of the figure.

The set of possible candidate vectors is $I \cup D^{+} \cup V \cup V^{\perp} \cup E = \{\mathtt{i}, \mathtt{k}\} \cup \{\mathtt{k}, \mathtt{i} - \mathtt{k}\} \cup \{\mathtt{i}, \mathtt{k}, \mathtt{i} + \mathtt{k}\} \cup \{\mathtt{i}, \mathtt{k}, \mathtt{i} - \mathtt{k}\} \cup \{\mathtt{i} + \mathtt{k}, \mathtt{i}\}$. After filtering against the dependences, only two candidates remain: `i` and $\texttt{i}+\texttt{k}$. There is only a single possible basis, $B = \{\texttt{i}, \texttt{i}+\texttt{k}\}$. The compiler first determines a schedule and computes ρ_v for each stream; it then computes μ_v the space requirement for each stream. Finally, these are combined to compute the cost model.

6.3.1 Finding ρ_v

Since the basis contains only two vectors, there are only two possible schedules to be considered. No locality is possible for the `b[i,-k]`-stream, a two-dimensional stream in a two-dimensional iteration space. In order for a stream to be left local, all its subscript vectors must be perpendicular to the direction of loop execution. This is the case whenever a column of the subscript matrix is zero. Table 6.1 summarizes the effects of choosing various schedules. Since the `w[i]` stream is both read and written, it counts as more total memory accesses, so the best schedule has `i` outermost.

B	B^{-1}	$R_{\texttt{w[i]}}$	$S_{\texttt{w[i]}}$	$R_{\texttt{w[i+k-1]}}$	$S_{\texttt{w[i+k-1]}}$	Local streams
$\begin{bmatrix}1&0\\1&1\end{bmatrix}$	$\begin{bmatrix}1&0\\-1&1\end{bmatrix}$	$[1,0]$	$[1,0]$	$[1,1]$	$[0,1]$	`w[i]`
$\begin{bmatrix}1&1\\1&0\end{bmatrix}$	$\begin{bmatrix}0&1\\1&-1\end{bmatrix}$	$[1,0]$	$[0,1]$	$[1,1]$	$[1,0]$	`w[i+k-1]`

Table 6.1: Summary of scheduling possibilities

The next step is to transform the loop bounds so that the number of refreshes for each stream can be computed. Rewriting the original loop bounds as matrices yields

$$\texttt{for}\ \vec{\imath} = \begin{bmatrix}0&0\\-1&0\end{bmatrix}\vec{\imath}+\vec{1}\ \texttt{to}\ \begin{bmatrix}0&0\\0&0\end{bmatrix}\vec{\imath}+\begin{bmatrix}n\\0\end{bmatrix}\ \texttt{do}$$

This in turns yields the system of inequalities

$$\begin{bmatrix}-1&0\\-1&-1\\1&0\\0&1\end{bmatrix}\vec{\imath}\leq\begin{bmatrix}-1\\-1\\n\\0\end{bmatrix}$$

The next step is to decompose B into a lower-triangular matrix and a unimodular matrix. Since B is already unimodular, the required decomposition is simply $H = I, U = B$.

The transformed inequalities are found by multiplying the coefficients by B, yielding

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ -1 & 1 \end{bmatrix} \vec{\jmath} \leq \begin{bmatrix} -1 \\ -1 \\ n \\ 0 \end{bmatrix}$$

where $\vec{\jmath}$ is the new loop index vector, $\vec{\jmath} = [\mathtt{u}, \mathtt{v}]^{\mathrm{T}}$. Solving for v yields $1 \leq \mathtt{v} \leq u$. Eliminating the inequalities with non-zero v coefficients yields

$$\begin{bmatrix} -1 & 0 \\ -1 & 0 \\ 1 & 0 \end{bmatrix} \vec{\jmath} \leq \begin{bmatrix} -1 \\ -1 \\ n \end{bmatrix}$$

So it is easily seen that the outermost loop bounds are $1 \leq \mathtt{u} \leq n$.

The compiler can now compute the number of refetches for each stream. The `w[i]` stream is local to the innermost loop, but not the outer loop. The number of refetches is therefore

$$\rho_{\mathtt{w[i]}} = \frac{1}{\beta_1} \sum_{u=1}^{n} 1 = \frac{n}{\beta_1}$$

Both the `b[i,-k]` and `w[i+k-1]` matrices are nonlocal. The number of refetches for each is therefore

$$\begin{aligned} \rho_{\mathtt{b[i,-k]}} = \rho_{\mathtt{w[i+k-1]}} &= \frac{1}{\beta_1} \sum_{u=1}^{n} \frac{1}{\beta_2} \sum_{v=1}^{u} 1 \\ &= \frac{1}{\beta_1 \beta_2} \sum_{u=1}^{n} u \\ &= \frac{n^2 + n}{2\beta_1 \beta_2} \end{aligned}$$

6.3.2 Finding μ_v

The first step is to compute the subscript matrices for each stream. The subscript matrices are then used to find space requirements for each stream. The subscript matrices are found using the formula $S = RB^{-1}$.

The subscript matrices are summarized in Table 6.2. As an example of how the data in

Stream	Reference matrix	Subscript matrix	$S\Psi$	Buffer size
`w[i]`	$\begin{bmatrix}1 & 0\end{bmatrix}$	$\begin{bmatrix}1 & 0\end{bmatrix}$	$[\beta_1]$	β_1
`b[i,-k]`	$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 0\\ -1 & 1\end{bmatrix}$	$\begin{bmatrix}\beta_1 & 0\\ -\beta_1 & \beta_2\end{bmatrix}$	$\beta_1{\times}\beta_2$
`w[i+k-1]`	$\begin{bmatrix}1 & 1\end{bmatrix}$	$\begin{bmatrix}0 & 1\end{bmatrix}$	β_2	β_2

Table 6.2: Summary of streams

the table is computed, examine the `w[i]` stream. The reference matrix is read directly from the source code. The subscript matrix is computed by multiplying the reference matrix by the inverse basis matrix. To compute the size requirement, the compiler first checks if skewed rectangular buffering can be applied. Skewed rectangular buffering can be applied in this case because the first column of the subscript matrix is zero. The compiler must then compute the Δ matrix. The columns of Δ subtend the parallelepiped in the data space. The tile is a parallelepiped subtended by the columns of

$$\Psi = \begin{bmatrix}\beta_1 & 0\\ 0 & \beta_2\end{bmatrix}$$

The image of the tile in the data space is a one-dimensional array $S\Psi = \begin{bmatrix}\beta_1 & 0\end{bmatrix}$. Applying Theorem 1, the compiler constructs the matrix Δ which subtends the columns of the parallelepiped in the data space. In this case, the parallelepiped is one-dimensional, and the theorem yields $\Delta = [\beta_1]$. The space requirement is $|\det \Delta|$, and a trivial application of the allocation procedure of Chapter 4 yields a one-dimensional buffer of length β_1. The rest of Table 6.2 is computed in a similar manner. The total memory allocation is

$$\mu_{\texttt{w[i]}} + \mu_{\texttt{w[i+k-1]}} + \mu_{\texttt{b[i,-k]}} = \beta_1 + \beta_2 + \beta_1\beta_2$$

6.3.3 The cost model

Having computed ρ_v and μ_v for each stream, the compiler is ready to construct the cost model. For simplicity, it is assumed that M_2 does not reward block accesses, so each M_2 access costs the same. The cost formula then simplifies to simply the number of M_2 accesses.

The total number of M_2 accesses is

$$\begin{aligned}
X &= 2\mu_{\texttt{w[i]}}\rho_{\texttt{w[i]}} + \mu_{\texttt{w[i+k-1]}}\rho_{\texttt{w[i+k-1]}} + \mu_{\texttt{b[i,-k]}}\rho_{\texttt{b[i,-k]}} \\
&= 2\beta_1\frac{n}{\beta_1} + \beta_2\frac{n^2+n}{2\beta_1\beta_2}\beta_1\beta_2\frac{n^2+n}{2\beta_1\beta_2} \\
&= 2n + \frac{n^2+n}{2\beta_1} + \frac{n^2+n}{2}
\end{aligned}$$

The compiler attempts to minimize X subject to the memory constraint

$$\beta_1 + \beta_2 + \beta_1\beta_2 \le \text{size}(M_1)$$

The minimum occurs when β_1 is as large as possible. Maximizing β_1 means minimizing β_2; since β_2 appears in the memory constraint but not in the cost formula, the compiler sets $\beta_2 = 1$. The memory constraint becomes $2\beta_1 = \text{size}(M_1) - 1$, so $\beta_1 = \frac{\text{size}(M_1)-1}{2}$. The total number of M_2 memory operations is then

$$\begin{aligned}
X &= \mu_{\texttt{X}}\rho_{\texttt{X}} + 2\mu_{\texttt{W}}\rho_{\texttt{W}} + \mu_{\texttt{A}}\rho_{\texttt{A}} \\
&= 2n + \frac{n^2+n}{\text{size}(M_1)-1} + \frac{n^2+n}{2}
\end{aligned}$$

If there were any other possible bases that could be formed from the candidate set, they would be evaluated in the same way. First the compiler computes a formula for the amount of data fetched. Then the compiler chooses a schedule. The schedule is used to compute the number of refresh operations each stream must undergo. The compiler then constructs a cost model. The basis with the lowest cost is chosen for the final code transformation.

6.3.4 Code generation

The transformations to produce the final code are best shown as a series of steps. The source code is first transformed into the new basis space. The transformed loop bounds are copied in, and subscript expressions are replaced by the $S = RB^{-1}$ subscript matrices:

```
for u = 1 to n
  for v = 1 to u
    w[u] = w[u] + b[u,v-u]*w[v-1]
```

Next, strip mining is applied to get a tiled loop nest:

```
$\beta_1$ = size($M_1$) $- 1$
$\beta_2$ = $1$
for u = 1 to n by $\beta_1$
  for v = 1 to u by $\beta_2$
    for uu = u to min(n, u+$\beta_1$-1)
      for vv = v to min(uu, v+$\beta_2$-1)
        w[uu] = w[uu] + b[uu,vv-uu]*w[vv-1]
```

Next, buffering code is added, following the methods of Chapter 4:

```
$\beta_1$ = size($M_1$) $- 1$
$\beta_2$ = $1$
for u = 1 to n by $\beta_1$
  for v = 1 to u by $\beta_2$
    begin
    for k = 0 to $\beta_1$-1
      wi_buf[k] = w[u+k]
    for $k_1$ = 0 to $\beta_1$-1
      for $k_2$ = 0 to $\beta_2$-1
        b_buf[$k_1, k_2$] = b[u+$k_1$,v-u+$k_2$-$k_1$]
    for $k$ = 0 to $\beta_2$-1
      wikplus1_buf[$k$] = w[v+$k$-1]
    for uu = u to min(n, u+$\beta_1$-1)
      for vv = v to min(n, v+$\beta_2$-1)
        wi_buf[uu-u] = wi_buf[uu-u] +
          b_buf[uu-u,vv-v]*wikplus1_buf[vv-v]
    for k = 0 to $\beta_1$-1
      w[u+k] = wi_buf[k]
    end
```

Finally, a few simplifications are made:

```
for u = 1 to n by size($M_1$)-1
  for v = 1 to u
    begin
    for k = 0 to size($M_1$)-2
      wi_buf[k] = w[u+k]
    for $k_1$ = 0 to size($M_1$)-2
      b_buf[$k_1$,0] = b[u+$k_1$,v-u-$k_1$]
    wikplus1_buf[0] = w[v-1]
    for uu = u to min(n, u+$\beta_1$-1)
        wi_buf[uu-u] = wi_buf[uu-u] +
          b_buf[uu-u,0]*wikplus1_buf[0]
    for k = 0 to size($M_1$)-2
      w[u+k] = wi_buf[k]
    end
```

6.4 Conclusion

This chapter completes the theoretical development of techniques for managing data motion using a compiler. A discussion of Fourier-Motzkin elimination is used to determine the transformed loop bounds so that the number of refresh operations can be determined. Back propagation of constraints is added to the elimination process to simplify the loop bounds.

Once the full cost model has been constructed, it is solved either analytically or numerically. The cost model is in general a sum of several terms, where each term takes the form of a multinomial in the elements of $\vec{\beta}$ divided by the the product of the elements of $\vec{\beta}$. The first and second partial derivatives therefore must exist, and are continuous over the region of interest ($\beta_i > 0$ for all i). The existence of these derivatives in the region of feasible $\vec{\beta}$'s allows the use of fast-converging modified Newton methods for gradient descent in finding numerical solutions to the optimization problem.

Finally, a complete example was given illustrating how the techniques used fit together to transform a loop nest. The compiler first finds a set of candidate basis vectors, from which it forms a list of candidate bases. For each basis, the compiler finds a schedule, computes the number of times each buffer must be refreshed, and how large each buffer is, both in terms of the tile size vector $\vec{\beta}$. The compiler then solves for the value of $\vec{\beta}$ which minimizes the total cost given the memory constraint and the constraints that each element of the tile size vector must be at least 1. The value of $\vec{\beta}$ which minimizes the cost formula is used to compute the total cost. The basis with the lowest total cost is chosen for the final code transformation.

In the next chapter, the techniques of this thesis are applied to common scientific program loops, and to other loops designed to contrast the approach taken in this work to similar approaches taken by other researchers.

Chapter 7

Evaluation

Previous chapters have described a new set of techniques for managing data motion using tiling. In this chapter, several examples of tiling illustrate the techniques and demonstrate the advantage of these techniques over previous methods. In the next section, three examples are drawn from common scientific and signal processing programs. The second section is devoted to examples contrasting this work with prior art.

The techniques described in this thesis were implemented in a prototype compiler based on the Fx compiler[50]. Due to time constraints, a numerical solver was not implemented (in the examples which follow, the cost models were solved using analytical rather than numerical techniques). The prototype compiler analyzes the program (using the Omega test[40, 41] for dependence analysis), generates candidate vectors, and evaluates candidate bases. It performs scheduling and data allocation analysis, and emits a cost model for each basis, to be evaluated by hand. Once a numerical solver is implemented, the remaining code transformations are straightforward and easy to implement.

7.1 Common scientific kernels

The following three sections show examples of the techniques described in earlier chapters applied to three loop nests which are common in scientific programming: matrix-matrix multiplication, QR-decomposition, and LU-decomposition.

7.1.1 Matrix multiply

Matrix-matrix multiply is a good example to start with, because the code is simple enough to illustrate the basic principles without introducing any real complexity. Of course, because the code is so simple, any locality-improving transformation should result in near-optimal performance. This example does not motivate the techniques used (later examples will) but rather serves to illustrate the basic principles involved. The input code for matrix multiply is shown once again in Figure 7.1.

```
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            c[i,j] = c[i,j] + a[i,k] * b[k,j];
```

Figure 7.1: Matrix-matrix multiply

The dependence vector set of matrix multiply is $(0,0,1)$, a flow-dependence in the k-loop on `c`. The reference vector set is $\{i,j,k\}$. All of these are legal vectors; the union of these vectors is just I. There is only one tiling basis choice in this case, I.

The code produced by tiling is shown in Figure 7.2. Each loop has been strip-mined, and then the controlling loops were interchanged outwards. The outer n loops select a tile, and the inner n loops select an iteration within that tile.

Once the loop has been tiled, buffer space in M_1 is assigned to each stream (in this case, the buffer variables `a_buf`, `b_buf`, and `c_buf` are assigned to the `a`, `b`, and `c` streams, respectively). Loops to copy blocks of data from M_2 to M_1 and back are then inserted, and references to the original arrays in the loop body are replaced by references to the buffers. The result is the code in Figure 7.3.

```
for i = 1 to n by $\beta_i$ do
  for j = 1 to n by $\beta_j$ do
    for k = 1 to n by $\beta_k$ do
      for ii = i to min (n, ii+$\beta_i$-1) do
        for jj = j to min (n, jj+$\beta_j$-1) do
          for kk = k to min (n, kk+$\beta_k$-1) do
            c[ii,jj] = c[ii,jj] + a[ii,kk] * b[kk,jj];
```

Figure 7.2: Tiled matrix-matrix multiply

```
for i = 1 to n by $\beta_i$ do
  for j = 1 to n by $\beta_j$ do
  begin
    comment{fetch a block of the c matrix}
    for ii = i to min (n, ii+$\beta_i$-1) do
      for jj = j to min (n, jj+$\beta_j$-1) do
        c_buf[ii-i,jj-j] = c[ii,jj];
    for k = 1 to n by $\beta_k$ do
    begin
      comment{fetch a block of the a matrix}
      for ii = i to min (n, ii+$\beta_i$-1) do
        for kk = k to min (n, kk+$\beta_k$-1) do
          a_buf[ii-i,kk-k] = a[ii,kk];

      comment{fetch a block of the b matrix}
      for kk = k to min (n, kk+$\beta_k$-1) do
        for jj = j to min (n, jj+$\beta_j$-1) do
          b_buf[kk-k,jj-j] = b[kk,jj];

      comment{now we can do the computation}
      for ii = i to min (n, ii+$\beta_i$-1) do
        for jj = j to min (n, jj+$\beta_j$-1) do
          for kk = k to min (n, kk+$\beta_k$-1) do
            c_buf[ii-i,jj-j] =
              c_buf[ii-i,jj-j] + a_buf[ii-i,kk-k] * b_buf[kk-k,jj-j];
      comment{a and b are dropped since they are read-only}
    end comment{end of k-loop}

    comment{store back the c matrix block}
    for ii = i to min (n, ii+$\beta_i$-1) do
      for jj = j to min (n, jj+$\beta_j$-1) do
        c[ii] = c_buf[ii-i];
  end comment{end of i-loop and j-loop}
```

Figure 7.3: Tiled matrix-matrix multiply with buffering code

The number of memory operations per refresh for each stream are given by:

$$\begin{aligned}
\mu_{\texttt{c[i,j]}} &= \beta_i\beta_j \\
\mu_{\texttt{a[i,k]}} &= \beta_i\beta_k \\
\mu_{\texttt{b[k,j]}} &= \beta_k\beta_j
\end{aligned}$$

The number of refreshes for each stream are as follows:

$$\begin{aligned}
\rho_{\texttt{c[i,j]}} &= \frac{1}{\beta_i}\sum_{i=1}^{\texttt{n}}\frac{1}{\beta_j}\sum_{j=1}^{\texttt{n}} 1 = \frac{\texttt{n}^2}{\beta_i\beta_j} \\
\rho_{\texttt{a[i,k]}} &= \frac{1}{\beta_i}\sum_{i=1}^{\texttt{n}}\frac{1}{\beta_j}\sum_{j=1}^{\texttt{n}}\frac{1}{\beta_k}\sum_{k=1}^{\texttt{n}} 1 = \frac{\texttt{n}^3}{\beta_i\beta_j\beta_k} \\
\rho_{\texttt{b[k,j]}} &= \frac{1}{\beta_i}\sum_{i=1}^{\texttt{n}}\frac{1}{\beta_j}\sum_{j=1}^{\texttt{n}}\frac{1}{\beta_k}\sum_{k=1}^{\texttt{n}} 1 = \frac{\texttt{n}^3}{\beta_i\beta_j\beta_k}
\end{aligned}$$

The full cost model is therefore given by

$$\frac{2\texttt{n}^2\beta_i\beta_j}{\beta_i\beta_j} + \frac{\texttt{n}^3\beta_i\beta_k}{\beta_i\beta_j\beta_k} + \frac{\texttt{n}^3\beta_k\beta_j}{\beta_i\beta_j\beta_k}$$

(the 2 in the $\texttt{n}^2$ term comes from the fact that `c[i,j]` is both read and written). This simplifies to

$$2\texttt{n}^2 + \frac{\texttt{n}^3}{\beta_j} + \frac{\texttt{n}^3}{\beta_i}$$

This is the function that the compiler must minimize, subject to the constraints

$$\begin{aligned}
\beta_i &\geq 1 \\
\beta_j &\geq 1 \\
\beta_k &\geq 1 \\
\beta_i\beta_j + \beta_i\beta_k + \beta_k\beta_j &\leq \text{size}(M_1)
\end{aligned}$$

A bit of calculus shows that the minimum cost occurs when

$$\vec{\beta} = (\sqrt{\text{size}(M_1) + 1} - 1, \sqrt{\text{size}(M_1) + 1} - 1, 1)$$

The total cost in terms of M_2 operations is then

$$R = 2n^2 + \frac{2n^3}{\sqrt{\text{size}(M_1)+1} - 1}$$

This is the optimal cost. For purposes of comparison, previous researchers choose $\beta_i = \beta_j = \beta_k$, in this case $\vec{\beta} = (\sqrt{\frac{\text{size}(M_1)}{3}}, \sqrt{\frac{\text{size}(M_1)}{3}}, \sqrt{\frac{\text{size}(M_1)}{3}})$. These cubic tiles correspond to multiplying square submatrices. Using this choice results in a total cost of

$$S = 2n^2 + \frac{2\sqrt{3}n^3}{\sqrt{\text{size}(M_1)}}$$

The effectiveness of the new techniques at reducing M_2 bandwidth requirements can be evaluated by comparing R and S, which are both functions of n and size(M_1). The general effectiveness can be evaluated by comparing the total execution time, X, under both methods. X is a function of n, size(M_1), and also the M_2 cycle time, c (in this chapter, the M_2 accesses are modeled as single accesses rather than block transfers for simplicity). Execution time has two components: time for computation and time for I/O. The time for computation is written X_C; for matrix-multiply, there are n^3 iterations, so $X_C = n^3$. Let X_R be the execution time using the new techniques, and X_S be the execution time using the older, cubic tile method. Execution times are therefore given by

$$X_R = X_C + cR = n^3 + c\left(2n^2 + \frac{2n^3}{\sqrt{\text{size}(M_1)+1} - 1}\right)$$

$$X_S = X_C + cS = n^3 + c\left(2n^2 + \frac{2\sqrt{3}n^3}{\sqrt{\text{size}(M_1)}}\right)$$

Figure 7.4 shows the total number of secondary memory operations for M_1 sizes from 4 words to 32Kwords, for a problem size of 120 (i.e., multiplying 120×120 matrices). For this size problem, there are a total of 43,200 words used for array storage. R is shown using a solid line, and S using a dashed line. Both methods require $O(n^3/\sqrt{\text{size}(M_1)})$ accesses. Rectangular tiles require slightly fewer accesses. When M_1 is very small, both methods must constantly fetch and store data, so the extra efficiency of rectangular tiles offers little advantage. Similarly, as M_1 grows larger, the problem begins to fit entirely in M_1, so very few M_2 accesses are required, and the extra efficiency is again little help.

For more direct comparison, Figure 7.5 shows the number of memory operations using

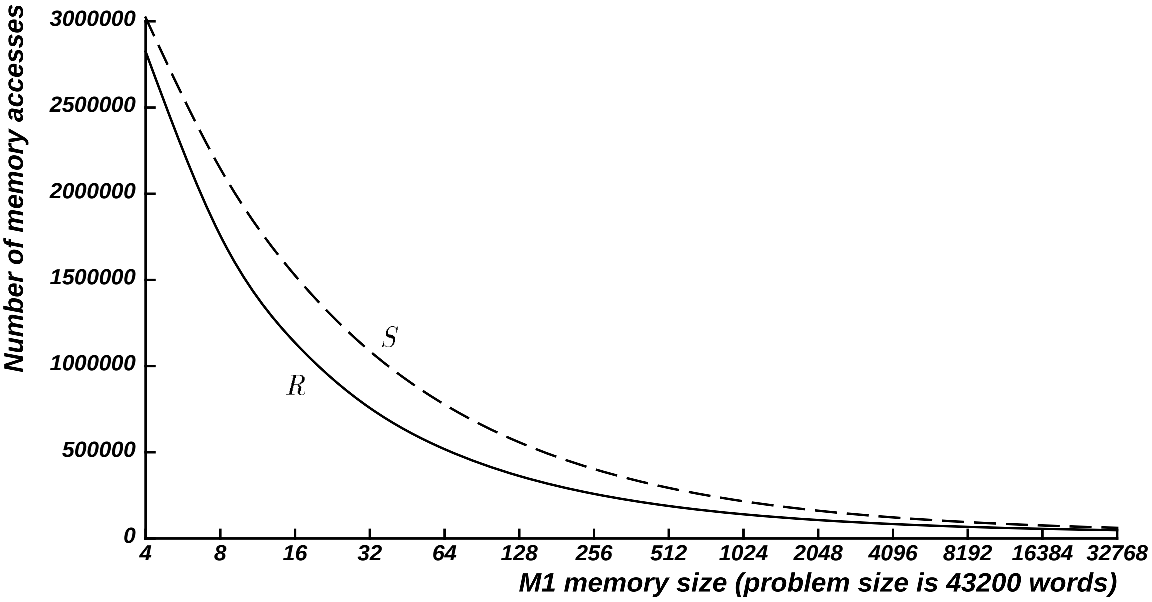


Figure 7.4: M_2 operations of optimal versus square tiles for MM

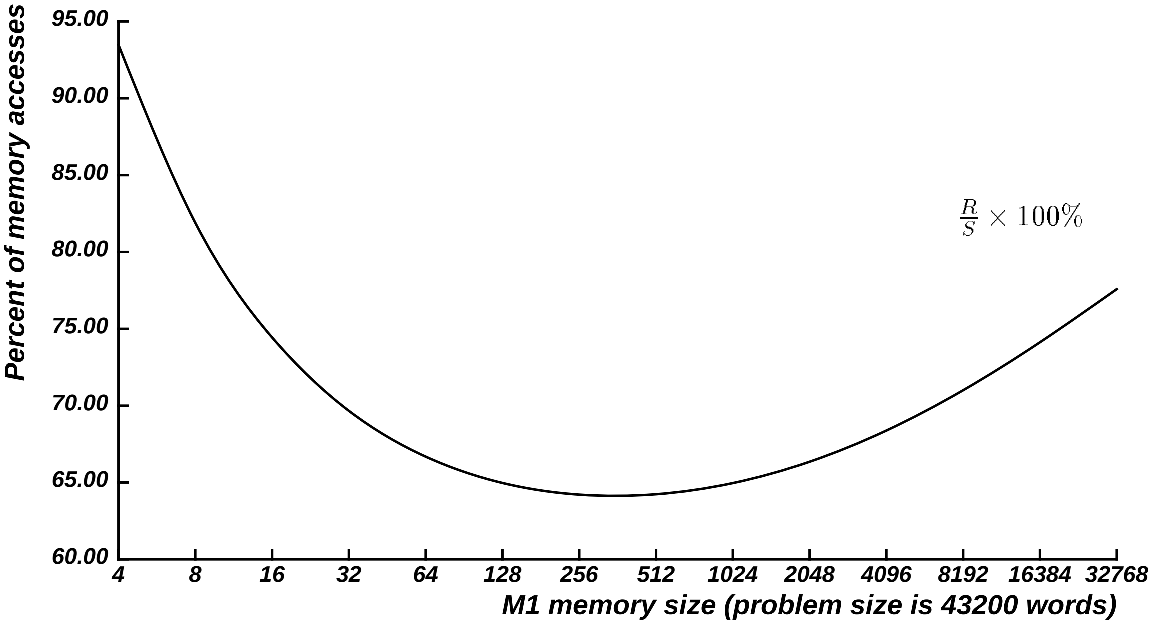


Figure 7.5: Relative I/O costs for MM

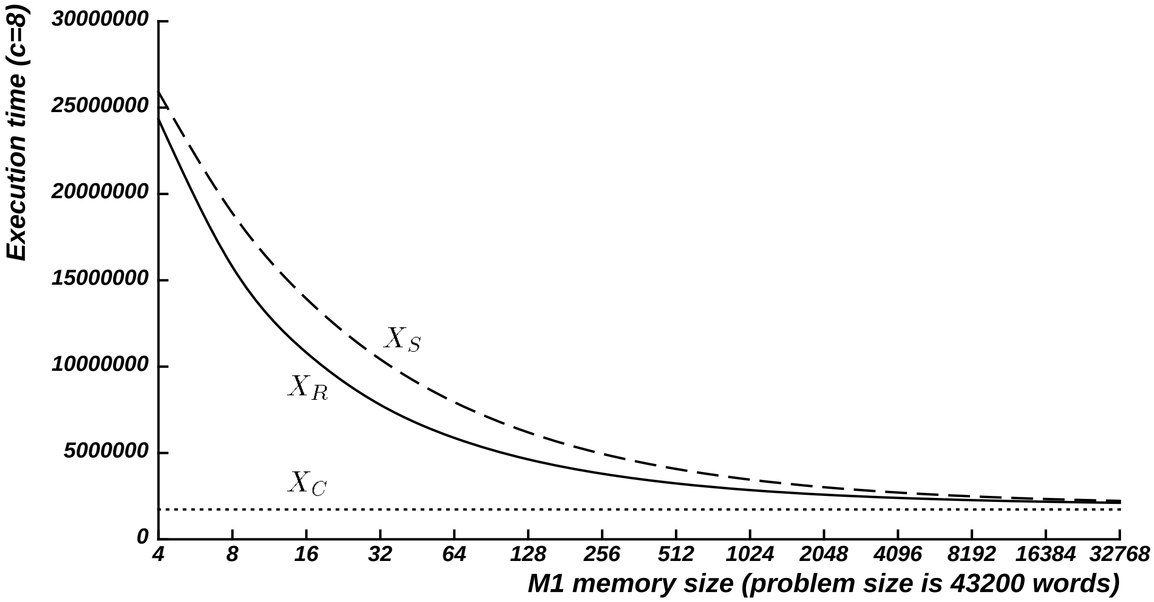


Figure 7.6: Execution times for MM

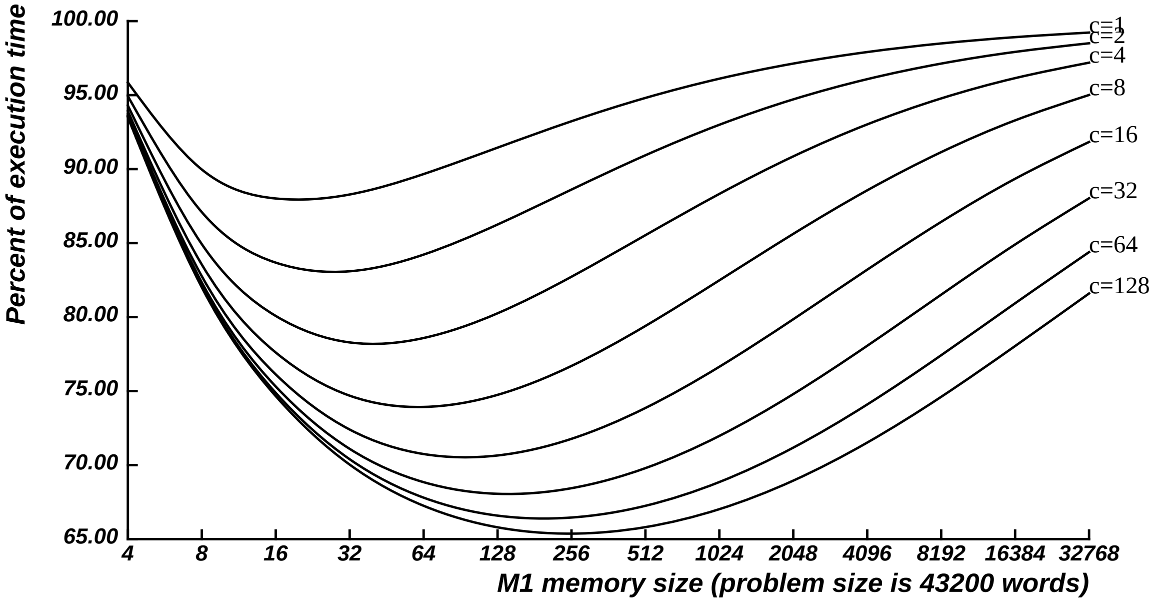


Figure 7.7: Relative execution times for MM ($X_R/X_S \times 100\%$)

the optimal scheme as a percentage of the number of operations using square submatrices as a function of M_1 size (i.e., $100R/S$). In this example, the optimal method saves about 30% of the M_2 accesses compared to the standard method for M_1 sizes between 1/1000th and 1/10th of the total memory accessed (from size$(M_1) = 43$ to size$(M_1) = 4320$). When M_1 is very small, both methods require the processors to refresh constantly, so no savings are realized. Over most reasonable M_1 sizes, a roughly constant improvement is achieved by the more efficient use of M_1 space. As M_1 grows and the problem begins to fit in fast memory, the efficiency of the optimal method becomes less important, because there are fewer M_2 memory operations required. This results in the bowl-shaped curve.

Figure 7.6 shows total execution time curves for 120×120 matrix multiply. The horizontal axis is once again M_1 size in words. The dotted line at the bottom of the figure is the time spent doing computation. The solid line is the time spent waiting for M_2 using optimally-shaped tiles. The dashed line is the time spent waiting for M_2 using cubic titles. The graph is drawn using M_2 cycle time $c = 8$ clocks.

Figure 7.7 shows the relative cost of optimal tiles as a percentage of the cubic-tile cost. Each curve uses a different M_2 cycle time: c=1 is the curve for 1 clock M_2 cycle time, c=2 represents 2 clock cycle time, and so forth. All curves are for a problem size of 120×120.

The extra efficiency leads to only a small improvement in execution time until M_2 cycle times grows large. In a uniprocessor M_2 cycle times are not likely to be very large, but in a parallel machine where an M_2 access may involve interprocessor communication, large M_2 cycle times are not uncommon.

In Figure 7.5, as M_1 sizes grow large, the relative number of M_2 operations decreases, so the extra efficiency becomes less important. A second factor also comes into play when comparing execution times. As M_2 becomes large, the computation time itself begins to dominate, so that optimal tiling becomes less important. This is why the curves in Figure 7.7 are higher on the right side of the graph than one would be lead to expect from Figure 7.5.

This effect can be seen more clearly in Figure 7.8 and Figure 7.9. These figures show relative execution times (X_R/X_S) as a function of size(M_1) for varying problem sizes. Figure 7.8 uses $c = 8$ while Figure 7.9 uses much slower M_2 memories with $c = 128$. For small memories, the extra efficiency leads to a significant improvement in execution time. As the memories become larger, the $\vec{\beta} = (\beta, \beta, \ldots)$ tiling increases the computation-to-I/O

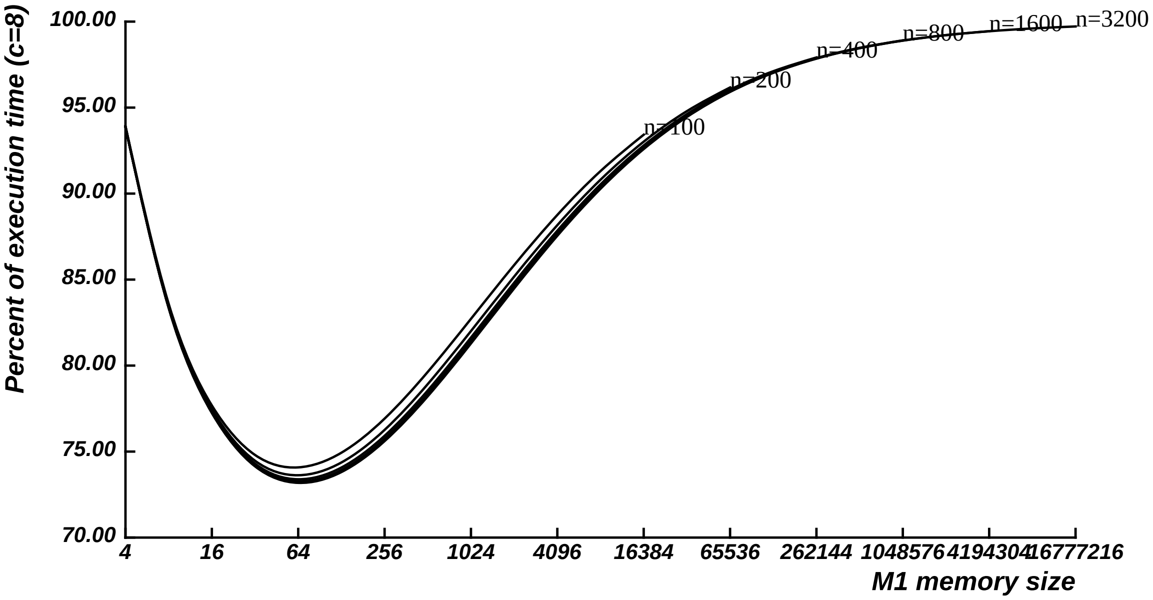


Figure 7.8: Relative improvement in execution time for MM ($c = 8$)

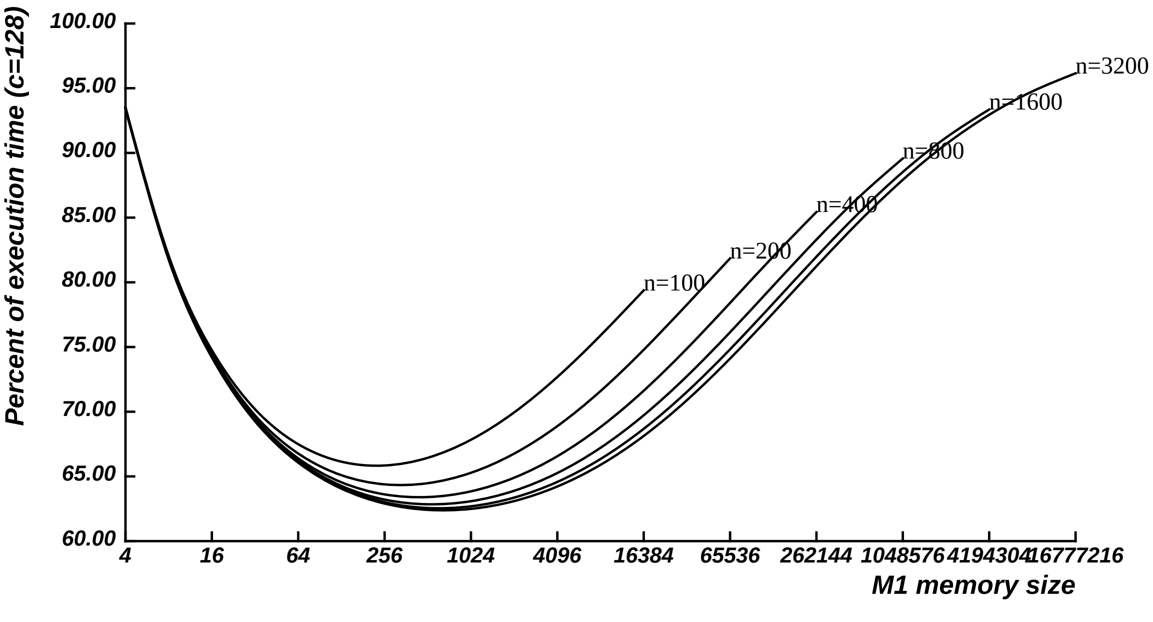


Figure 7.9: Relative improvement in execution time for MM ($c = 128$)

ratio enough that the program becomes computation bounded, so the added efficiency in I/O does not yield improvements in execution time.

The new techniques do have important benefits for relatively small memories. The added efficiency is beneficial when the target M_1 is a large register file (for example, the iWarp processor has 128 registers), or when targeting an on-chip buffer memory, such as those typically found in Transputers.

7.1.2 Where the improvement comes from

It is important to understand where the 30% decrease in M_2 operations comes from. Only a constant factor decrease will ever be possible, because the inherent computation-to-I/O ratio of matrix multiply is $O(\sqrt{\text{size}(M_1)})$. Any tiling must perform at least $O(n^3/\sqrt{\text{size}(M_1)})$ M_2 operations. Among tilings performing $O(n^3/\sqrt{\text{size}(M_1)})$ M_2 operations, there can be at most a constant factor improvement regardless of the choice of n or size(M_1).

The standard $\vec{\beta} = (\beta, \beta, \beta)$ tiling is therefore within a constant factor of the optimal cost. This tiling is illustrated in Figure 7.10. Each tile is cubic (except tiles which are cut off by the borders of the iteration space), and so is fully symmetric. No effort is made to take advantage of the fact that the compiler will execute the tiles in a specific order (along the `k` axis in the tiled code).

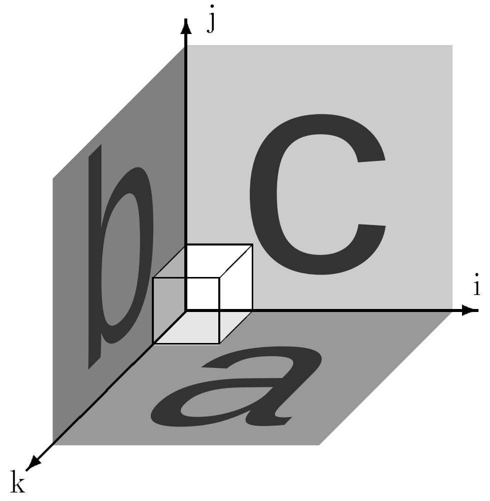


Figure 7.10: The $\vec{\beta} = (\beta, \beta, \beta)$ tiling

Using the techniques presented in this thesis, the compiler chooses $\beta_k = 1$. The tiles are flat squares, as shown in Figure 7.11. This shifts the allocation of M_1 to favor the `c[i,j]` stream. Since the compiler knows that the tiles will be executed in the `k` direction, there is no point in wasting M_1 memory space bringing in a square of data from the `a[i,k]` or `b[k,j]` streams. Each data item will be used a number of times proportional to the width or height of the tile ($\beta_{\mathtt{i}}$ or $\beta_{\mathtt{j}}$), and independent of the $\beta_{\mathtt{k}}$ tile dimension.

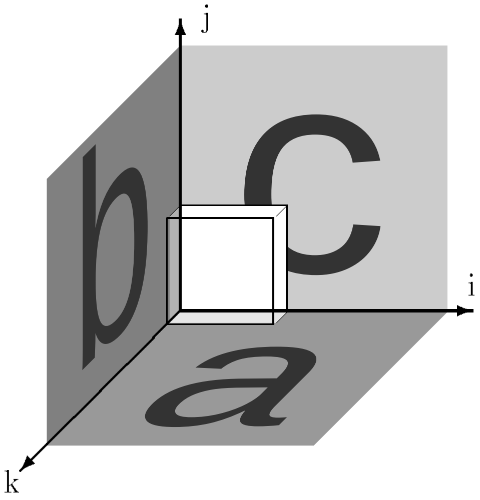


Figure 7.11: The $\vec{\beta} = (\beta, \beta, 1)$ tiling

Maximizing these other dimensions maximizes the reuse of the `a` and `b` streams while simultaneously minimizing the number of times the data for these streams will be refetched. This is shown graphically by noting that while the tile in Figure 7.10 requires the same amount of data per tile as the one in Figure 7.11, the flat tile is wider, so it has more iterations in the `i` and `j` directions. This means it reuses `b[k,j]` and `a[i,k]` more times in a given tile. Furthermore, these streams are refetched a number of time equal to `n` divided by the width of the tile in the respective directions, so widening the tiles reduces the number of refetches as well.

Note that the formulas for the I/O costs R and S are derived parametrically. The only assumption that must hold is that n is much larger than size(M_1), so multiple tiles are executed in each direction. R will always be smaller than S since the difference in the formulas is essentially a factor of $\sqrt{3}$. In fact, we would expect theoretically that R/S would asymptotically approach $1/\sqrt{3}$ as n grows large, or about 58%. Figure 7.12 shows that this is indeed the case.

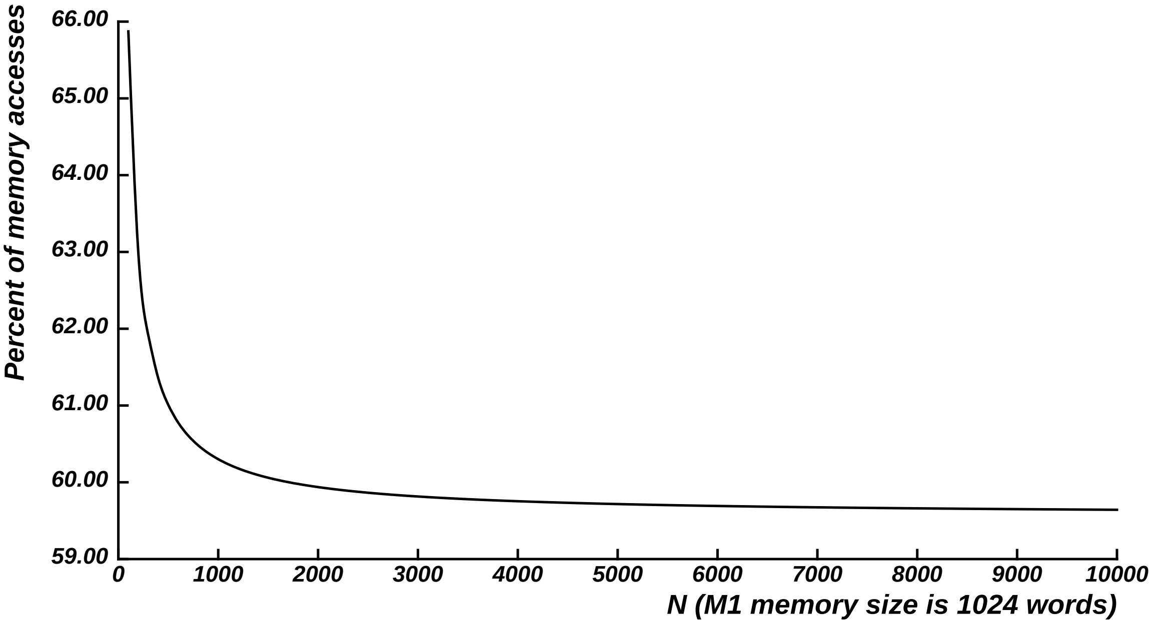


Figure 7.12: R/S as n grows very large

Because tiling increases the computation-to-I/O ratio of a program, one might conclude that very large programs would be computation bound, so that the improvements in I/O requirements outlined here would become less important. This is not usually the case. In matrix-multiply, for example, there are n^3 iterations. Before tiling, there are $O(n^3)$ I/O's; after tiling, there are $O(n^3/\sqrt{\text{size}(M_1)})$ I/O's. As n grows, the number of I/O's grows proportionally with execution time (fixing size(M_1)).

Tiling improves the computation-to-I/O ratio of a program, but only up to the M_1 memory size. Perhaps the best way to conceptualize this is to think of a simple cubic tiling of matrix multiply. If the tile size is so large that the entire iteration space fits in a single tile, only $3n^2$ fetches and n^2 stores are required for n^3 iterations. If the tile size shrinks to a single iteration, each computation requires 2 fetches, plus $1/n$ fetches and $1/n$ stores for the `c` matrix. For a fixed tile size, increasing the problem size makes the relative tile size approach the single-iteration case.

It is possible to write programs which have loops in which all data is local. Increasing the number of iterations in such a loop increases the computation-to-I/O ratio of the program. In such a program, increasing the "problem size" increases the computation-to-I/O ratio of the program, because it increases the number of computations without increasing the amount of data accessed (or it increases the number of computations faster than it increases the amount of data). In such a program, the relative impact of the increased efficiency would drop as the computation-to-I/O ratio of the program increased, because more and more time would be spent in the compute phase and the I/O time would become relatively insignificant. Fortunately, nearly all loops in scientific programs access new data, and increasing "problem size" does not usually change the computation-to-I/O ratio of the program.

7.1.3 QR decomposition

The source code for QR decomposition is shown in Figure 7.13. There are two streams, `a[i,j]`, and `r[k,j]`. The dependence matrix for QR decomposition is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$; that is, there are dependences carried by the `k` and `j` loops.

```
for k = 0 to 119
  for i = k to 119
    for j = k to 119
      if (i == k) then
        r[k,j] = a[i,j];
      else
        if (j == k) then
          c = r[k,j]/SQRT(r[k,j]*r[k,j]+a[i,j]*a[i,j]);
          s = a[i,j]/SQRT(r[k,j]*r[k,j]+a[i,j]*a[i,j]);
          r[k,j] = c*r[k,j] + s*a[i,j];
        else
          rt = r[k,j];
          r[k,j] = s*a[i,j] + c*rt;
          a[i,j] = c*a[i,j] - s*rt;
        endif
      endif
```

Figure 7.13: Source code for QR decomposition

The tiling candidate set is {`k, i, j`}. In the last `else` clause there are read and write accesses to both `a[i,j]` and `r[k,j]`. In the middle clause of the loop body, there is no write of `a[i,j]`, so the compiler should choose to keep `r[k,j]` local, since it represents slightly more memory operations. This is achieved by interchanging the `i` loop innermost.

The number of refreshes for each stream is then given by

$$\begin{aligned}\rho_{\mathtt{r[k,j]}} &= \frac{1}{\beta_k}\sum_{k=1}^{\mathtt{n}}\frac{1}{\beta_j}\sum_{j=k}^{\mathtt{n}}1 = \frac{\mathtt{n}(\mathtt{n}+1)}{2\beta_k\beta_j}\\ \rho_{\mathtt{a[i,j]}} &= \frac{1}{\beta_k}\sum_{k=1}^{\mathtt{n}}\frac{1}{\beta_j}\sum_{j=k}^{\mathtt{n}}\frac{1}{\beta_i}\sum_{i=k}^{\mathtt{n}}1 = \frac{\mathtt{n}(\mathtt{n}+1)(2\mathtt{n}+1)}{6\beta_i\beta_j\beta_k}\end{aligned}$$

The buffer size for each stream is

$$\begin{aligned}\mu_{\mathtt{r[k,j]}} &= \beta_k\beta_j\\ \mu_{\mathtt{a[i,j]}} &= \beta_i\beta_j\end{aligned}$$

Which makes the total cost

$$\frac{\mathtt{n}(\mathtt{n}+1)}{2} + \frac{\mathtt{n}(\mathtt{n}+1)(2\mathtt{n}+1)}{6\beta_k}$$

To minimize this cost subject to the memory constraint

$$\beta_i\beta_j + \beta_k\beta_j \leq \text{size}(M_1)$$

the compiler chooses $\beta_i = \beta_j = 1$, and $\beta_k = \text{size}(M_1) - 1$. The total cost is then

$$R = \frac{\mathtt{n}(\mathtt{n}+1)}{2} + \frac{\mathtt{n}(\mathtt{n}+1)(2\mathtt{n}+1)}{6(\text{size}(M_1)-1)}$$

Using square tiles, the compiler chooses $\vec{\beta} = (\beta, \beta, \beta)$. The total cost is

$$S = \frac{\mathtt{n}(\mathtt{n}+1)}{2} + \frac{\mathtt{n}(\mathtt{n}+1)(2\mathtt{n}+1)}{6\sqrt{\text{size}(M_1)/2}}$$

The number of iterations for QR decomposition is given by

$$X_C = \sum_{k=0}^{n}\sum_{i=k}^{n}\sum_{j=k}^{n}1 = \frac{6+13n+9n^2+2\,n^3}{6}$$

Note that using optimally-shaped tiles, the computation-to-I/O ratio is $O(\text{size}(M_1))$; that is, for each M_2 access, on the order of size(M_1) computations are performed. Using cubic tiles, the computation-to-I/O ratio is only $O(\sqrt{\text{size}(M_1)})$.

QR decomposition more clearly shows the advantage of solving for each tile dimension separately to minimize the total cost, as illustrated in Figure 7.14. This figure shows the number of secondary memory operations for optimally shaped tiles and for square tiles ($\beta_j = \beta_k = \sqrt{\text{size}(M_1)/2}$) in QR decomposition. The number of M_2 operations for square tiles is shown using a dashed line, while the number of M_2 operations using optimally-shaped tiles is shown with a solid line.

Figure 7.15 shows the number of M_2 accesses made by the optimal solution expressed as a percentage of the number of accesses taken by the square-tile solution. The curve is like that for matrix-multiply, in that when M_1 is very small, both methods must fetch data constantly, so less savings can be realized. As M_1 grows large enough to allow locality, the optimal method's efficiency quickly out-paces the square-tile method. As M_1 continues to grow, the problem comes closer to fitting in M_1, so the greater efficiency becomes less important. Nevertheless, for M_1 sizes between 1/1000th and 1/10th of the full problem size, the optimal method requires less than 10% of the accesses required by the square-tile method. The plots are for a problem size of 120×120.

Figure 7.16 shows the total execution time curves for QR decomposition of 120×120 matrices. In QR-decomposition, each iteration is assumed to take 4 clocks. When the cost of an M_2 access is very small, computation time dominates, but as the cost of each access grows, the savings realized by the more efficient buffering becomes apparent. Even with very large M_2 access times, however, as M_1 size becomes large enough to fit most of the problem, the number of I/O's using either method drops to the point that execution time begins to dominate. This is why the relative percentage of execution time taken by the optimal method increases for larger M_1 sizes.

In Figure 7.17, the execution time of QR is shown for an M_2 cycle time of 8 clocks. The dotted line is execution time, the solid line is the time spent waiting on M_2 using optimally shaped tiles, and the dashed line is the time spent waiting on M_2 using cubic tiles. In this figure the difference between the two methods is clear. The optimal-tile method decreases the computation-to-I/O rate to $O(\text{size}(M_1))$, while the cubic-tile method can only perform $O(\sqrt{\text{size}(M_1)})$ operations per I/O. The optimal method allows the program to become computation-bounded much earlier than the cubic-tile method.

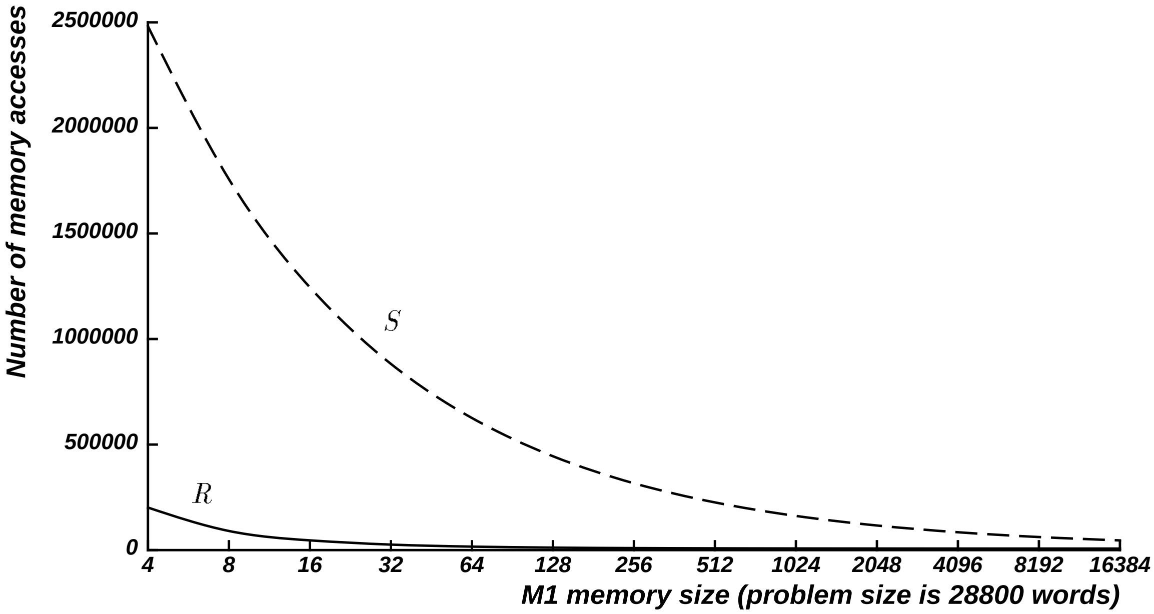


Figure 7.14: M_2 operations of optimal and square tiles for QR

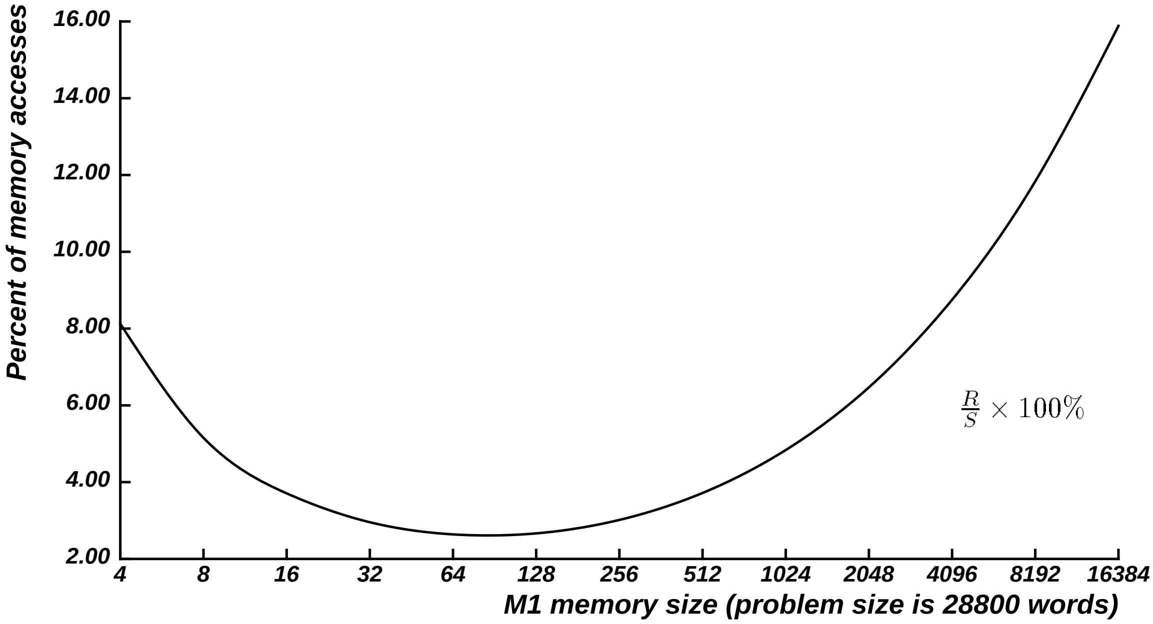


Figure 7.15: M_2 operations of optimal versus square tiles for QR

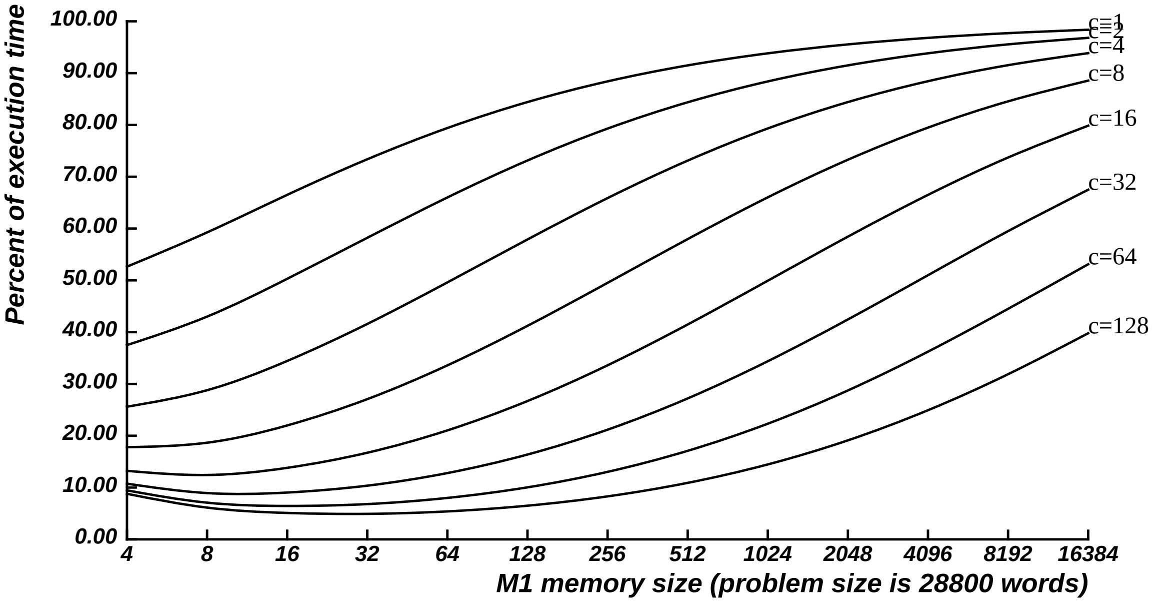


Figure 7.16: Execution time of optimal versus square tiles for QR ($X_R/X_S \times 100\%$)

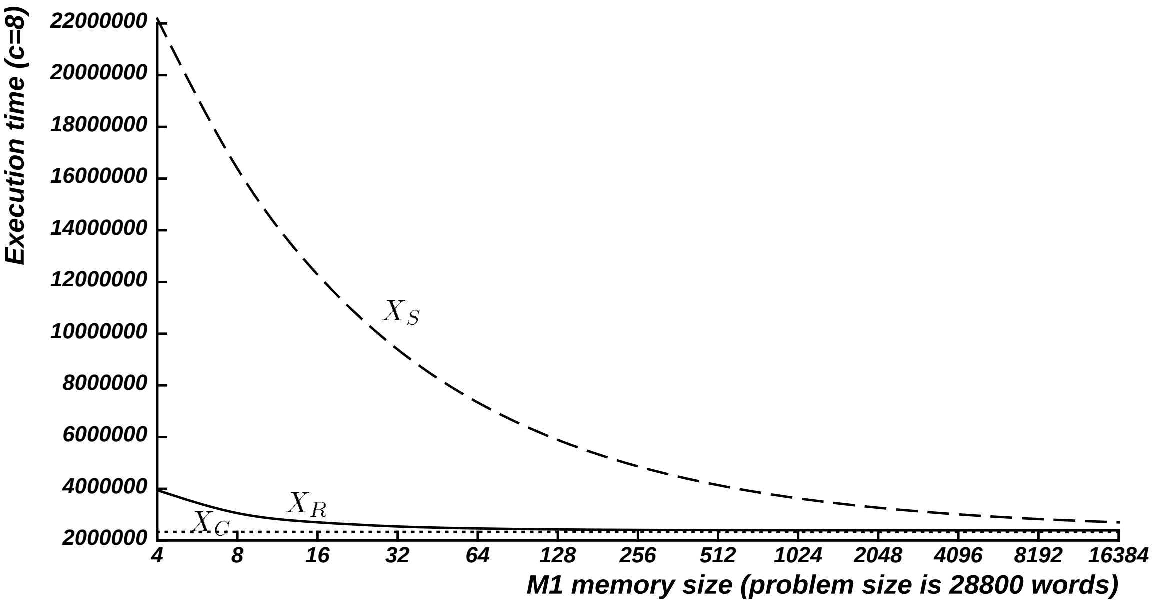


Figure 7.17: Execution time of optimal and square tiles for QR ($c = 8$)

7.1.4 LU decomposition

Note to the reader: LU decomposition is not significantly different from matrix multiply from the point of view of this thesis; it is included for completeness of comparison to other works. A reader uninterested in another example may skip to Section 7.2 on page 121.

LU decomposition is an algorithm for decomposing a matrix into two matrices: an upper-triangular matrix U and a lower-triangular matrix L. It is closely related to Gaussian elimination, since the lower triangular matrix generated can be use for solving a set of equations using back-substitution. The code for LU decomposition is shown in Figure 7.18. In this version, the original matrix is stored in the variable `a`. Upon return, the L matrix is stored in the variable `l`, and the U matrix is stored in the variable `a`. The variable `x` is used only as a temporary variable.

```
for k = 1 to n
  for i = k to n
    for j = k to n
      if (i == k) then
        begin
        if (j ≥ k)
          x[k,j] = a[i,j];
        end
      else
        if (j == k) then
          l[i,k] = a[i,j] / x[k,j];
        else
          a[i,j] = a[i,j] - l[i,k] * x[k,j];
        endif
      endif
```

Figure 7.18: Source code for LU decomposition

There are three streams: `a[i,j]`, `l[i,k]`, and `x[k,j]`. All three are both read and written. There are only three different reference vectors, `i`, `j`, and `k`. The tiling candidate set is therefore just I. The best ordering of the controlling loops has `k` innermost, leaving `a[i,j]` local. The transformed code is shown in Figure 7.19.

The buffer sizes for each stream are given below:

$$\mu_{\texttt{a[i,j]}} = \beta_i \beta_j$$

```
for i = 1 to n by $\beta_i$
  for j = 1 to n by $\beta_j$
    begin
    comment{fetch a block of stream a[i,j]}
    for ii = i to min(i+$\beta_i$-1,n)
      for jj = j to min(j+$\beta_j$-1,n)
        a_buf[ii-i,jj-j] = a[ii,jj];
    for k = 1 to min(i,j) by $\beta_k$
      begin
      comment{fetch a block of stream l[i,k]}
      for ii = i to min(i+$\beta_i$-1,n)
        for kk = k to min(ii,jj,k+$\beta_k$-1,n)
          l_buf[ii-i,kk-k] = l[ii,kk];
      comment{fetch a block of stream x[k,j]}
      for jj = j to min(j+$\beta_j$-1,n)
        for kk = k to min(ii,jj,k+$\beta_k$-1,n)
          x_buf[kk-k,jj-j] = x[kk,jj];
      comment{Main computation loop}
      for ii = i to min(i+$\beta_i$-1,n)
        for jj = j to min(j+$\beta_j$-1,n)
          for kk = k to min(ii,jj,k+$\beta_k$-1,n)
            if (ii == kk) then
              if (jj $\geq$ kk) x_buf[kk-k,jj-j] = a_buf[ii-i,jj-j];
            else
              if (jj == kk) then
                l_buf[ii-i,kk-k] =
                  a_buf[ii-i,jj-j] / x_buf[kk-k,jj-j];
              else
                a_buf[ii-i,jj-j] =
                  a_buf[ii-i,jj-j]-l_buf[ii-i,kk-k]*x_buf[kk-k,jj-j];
              endif
            endif
      comment{write back stream l[i,k]}
      for ii = i to min(i+$\beta_i$-1,n)
        for kk = k to min(ii,jj,k+$\beta_k$-1,n)
          l[ii,kk] = l_buf[ii-i,kk-k];
      comment{write back stream x[k,j]}
      for jj = j to min(j+$\beta_j$-1,n)
        for kk = k to min(ii,jj,k+$\beta_k$-1,n)
          x[kk,jj] = x_buf[kk-k,jj-j];
      end
    comment{write back of a[i,j] stream deleted for space reasons}
    end
```

Figure 7.19: Tiled code for LU decomposition

$$\begin{aligned}\mu_{\texttt{l[i,k]}} &= \beta_i\beta_k\\ \mu_{\texttt{x[k,j]}} &= \beta_k\beta_j\end{aligned}$$

The number of refreshes for each stream are as follows:

$$\begin{aligned}\rho_{\texttt{a[i,j]}} &= \frac{1}{\beta_i}\sum_{i=1}^{\texttt{n}}\frac{1}{\beta_j}\sum_{j=1}^{\texttt{n}} 1 = \frac{\texttt{n}^2}{\beta_i\beta_j}\\ \rho_{\texttt{l[i,k]}} &= \frac{1}{\beta_i}\sum_{i=1}^{\texttt{n}}\frac{1}{\beta_j}\sum_{j=1}^{\texttt{n}}\frac{1}{\beta_k}\sum_{k=1}^{\texttt{min(i,j)}} 1 = \frac{\texttt{n}+3\texttt{n}^2+2\texttt{n}^3}{6\beta_i\beta_j\beta_k}\\ \rho_{\texttt{x[k,j]}} &= \frac{1}{\beta_i}\sum_{i=1}^{\texttt{n}}\frac{1}{\beta_j}\sum_{j=1}^{\texttt{n}}\frac{1}{\beta_k}\sum_{k=1}^{\texttt{min(i,j)}} 1 = \frac{\texttt{n}+3\texttt{n}^2+2\texttt{n}^3}{6\beta_i\beta_j\beta_k}\end{aligned}$$

The total cost is then given by

$$2(n^2 + \frac{\texttt{n}+3\texttt{n}^2+2\texttt{n}^3}{6\beta_j} + \frac{\texttt{n}+3\texttt{n}^2+2\texttt{n}^3}{6\beta_i})$$

The minimal cost is achieved when $\beta_k = 1$, and $\beta_i = \beta_j = \sqrt{\text{size}(M_1)+1} - 1$. This leads to the completed I/O cost function

$$R = 2(n^2 + 2\frac{\texttt{n}+3\texttt{n}^2+2\texttt{n}^3}{6(\sqrt{\text{size}(M_1)+1}-1)})$$

Using the square-tile method, $\vec{\beta} = (\sqrt{\text{size}(M_1)/3}, \sqrt{\text{size}(M_1)/3}, \sqrt{\text{size}(M_1)/3})$. The I/O cost function for this method is then

$$S = 2(n^2 + 2\frac{\texttt{n}+3\texttt{n}^2+2\texttt{n}^3}{6(\sqrt{\text{size}(M_1)/3})})$$

These are very close to the values for matrix multiply. Recall that in matrix-multiply the computation-to-I/O ratio is $order\sqrt{\text{size}(M_1)}$. The computation cost for LU decomposition is

$$X_C = \sum_{k=0}^{n}\sum_{i=k}^{n}\sum_{j=k}^{n} 1 = \frac{6+13n+9n^2+2\,n^3}{6}$$

which is $O(n^3)$, resulting in a computation-to-I/O ratio of $O(\sqrt{\text{size}(M_1)})$. The value of $\vec{\beta}$ for the two programs are permutations of one another, because the cost models are so similar. This is reflected in the graphs for LU-decomposition, which look just like those for matrix-multiply.

Figure 7.20 shows the number of secondary memory operations for optimally shaped tiles and for cubic tiles in LU decomposition. The total number of M_2 operations for cubic tiles is shown with a dashed line, while the number of M_2 operations for optimally-shaped tiles is shown with a solid line. As expected, R and S are very similar.

The next plot, in Figure 7.21 is the number of M_2 accesses made by the optimal solution as a percentage of the number of accesses made by the traditional square-tile solution. Note that for M_1 sizes from 1/1000th to 1/10th of the problem size, the new methods achieve a 30-35% decrease in M_2 memory bandwidth.

Figure 7.22 shows the execution time taken by the optimal solution as a percentage of the execution time taken by the traditional square-tile solution, with the M_2 cycle time (labelled "c") varying from 1 clock to 128 clocks. Each iteration is assumed to take 2 clocks.

Figure 7.23 shows the total execution time for optimally shaped tiles and for cubic tiles in LU decomposition. The plot shows the total execution time spent in computation (dotted line), the time spent waiting for M_2 with cubic tiles (dashed line) and for optimally-shaped tiles (solid line), assuming the M_2 cycle time is 8 clocks.

7.2 Comparison to Wolf's work

In this section, several examples demonstrate the contributions of this thesis, by comparing it to the work of Wolf[57], the most thorough work on tiling for locality to date. Because he concentrated on tiling for machines with caches, Wolf made some assumptions which do not hold for compiler-controlled memories (like RAMs). Furthermore, Wolf's techniques for choosing $\vec{\beta}$ may be appropriate for cache-based systems, but it is less than ideal for software-controlled memories. Wolf also does not address compilation for machines with block-oriented memories; the framework provided in this thesis does handle this problem. Finally, Wolf abstracts the reuse space of a program to the set of loops carrying reuse, which is unnecessary given our techniques for choosing B.

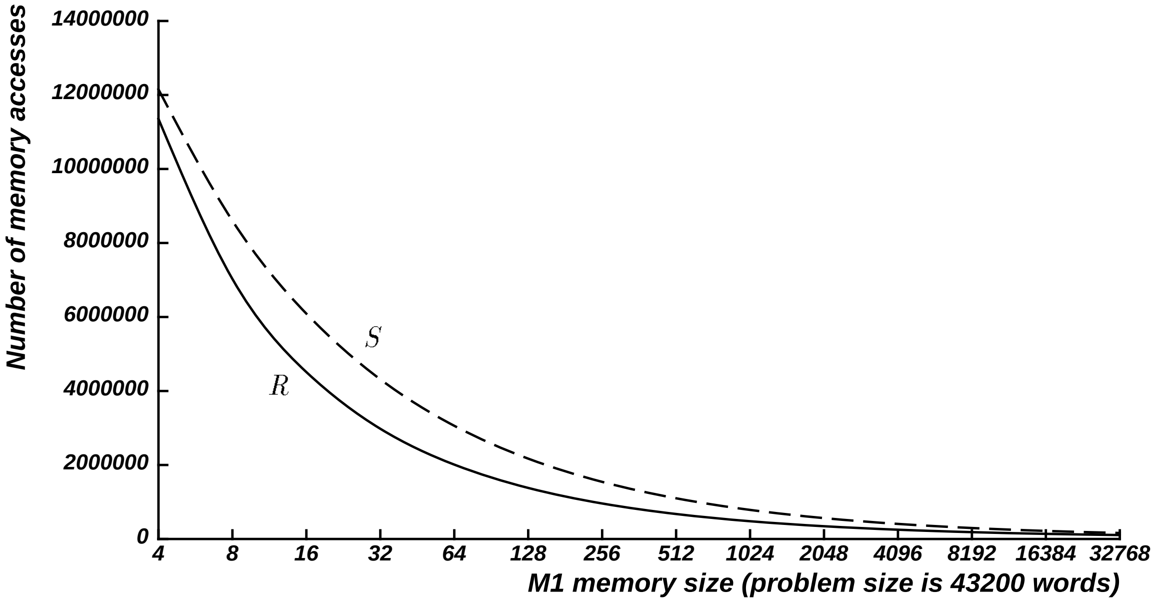


Figure 7.20: M_2 operations of optimal and square tiles for LU decomposition

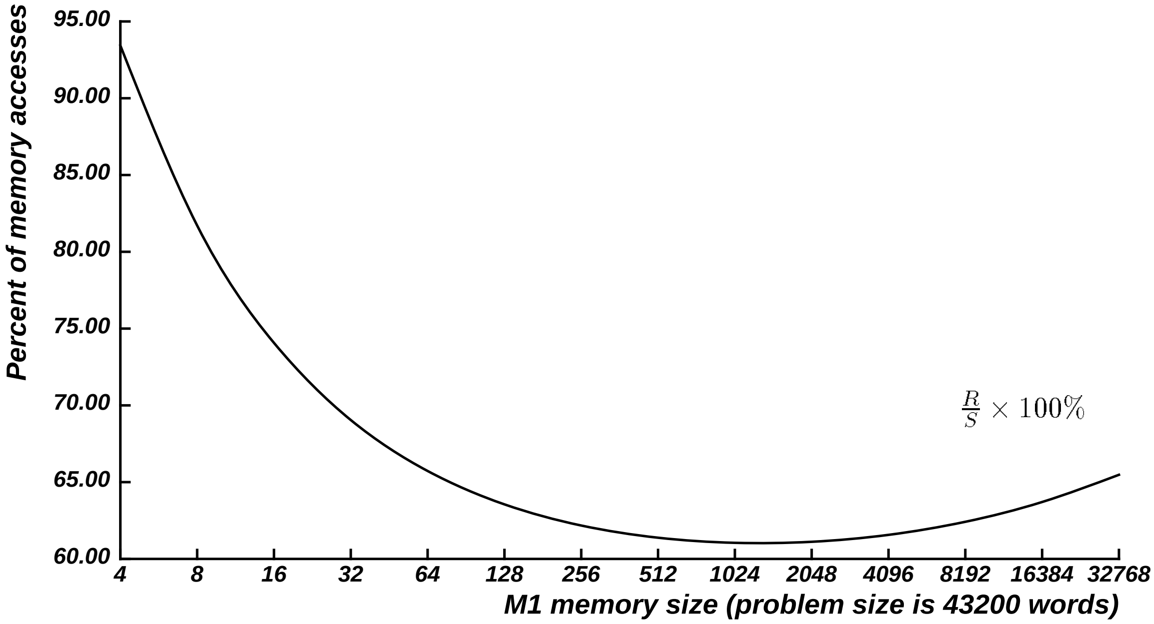


Figure 7.21: M_2 operations of optimal versus square tiles for LU decomposition

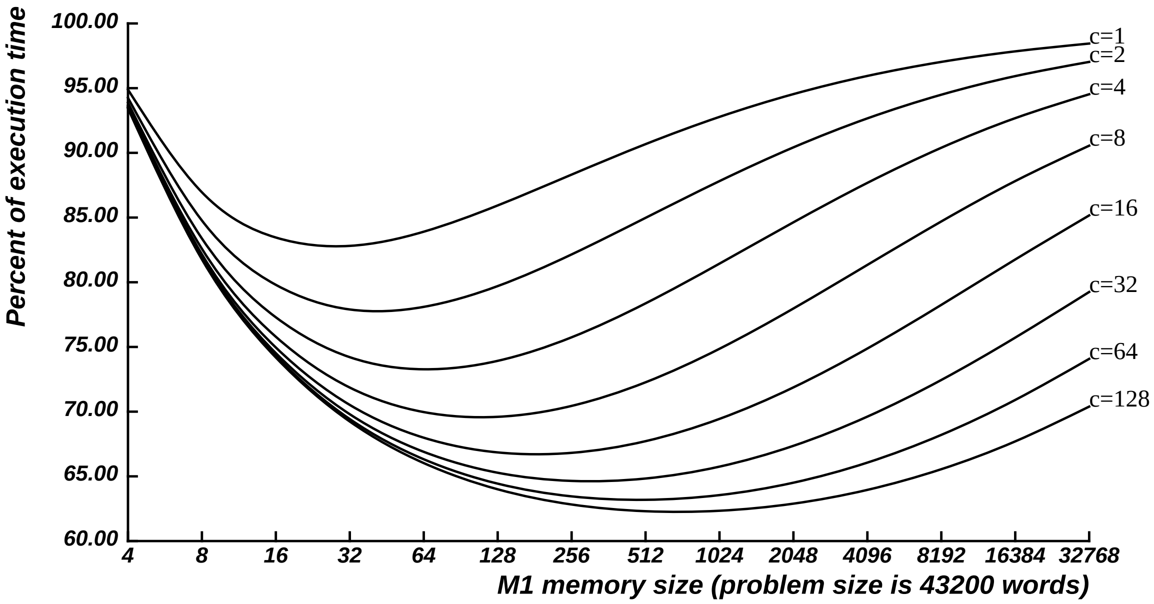


Figure 7.22: Execution time of optimal versus square tiles for LU decomposition

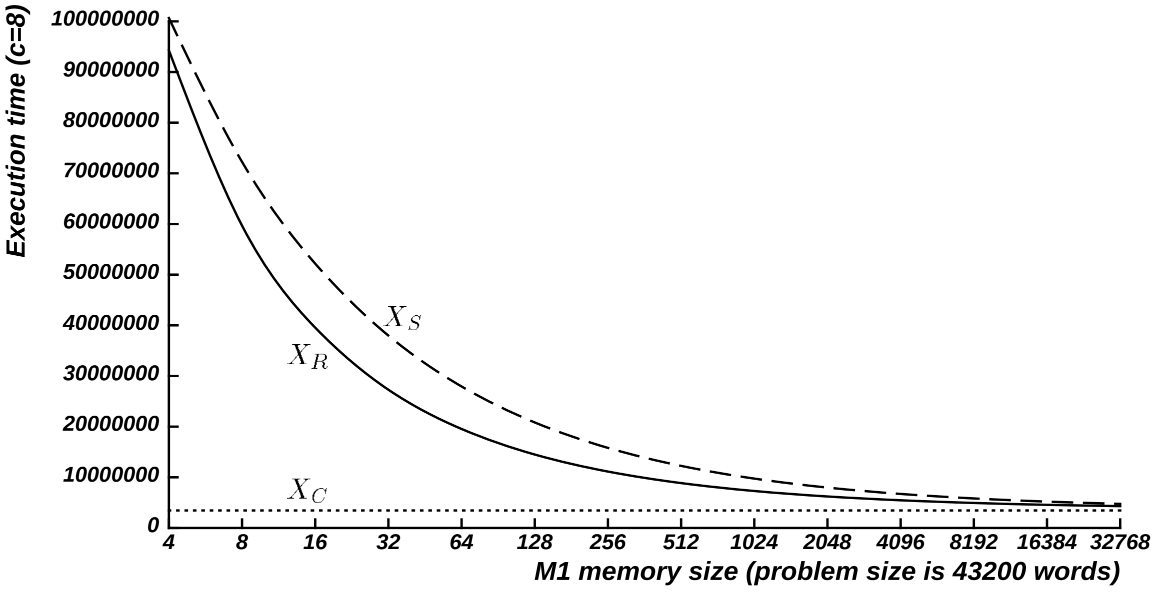


Figure 7.23: Execution time of optimal versus square tiles for LU decomposition

7.2.1 Reuse spaces

Wolf's description of reuse spaces in terms of self-temporal, self-spatial, group-temporal, and group-spatial (R_{ST}, R_{SS}, R_{GT}, R_{GS}), is concise, yet captures all the necessary information. In this work, we do not attempt to capture spatial locality directly. Spatial locality is included indirectly in the cost model by using a block-oriented cost system for memory accesses.

This work also takes a different approach to dealing with dependence limitations. Wolf formulates the expected reuse contributed by tiling each loop, and then selects the tilable subset of loops which maximizes locality. In this work, the tiling basis is chosen to tile all the loops given the dependence set. Extreme vectors of the dependence set are included in the candidate set to ensure that every loop nest can be tiled. In the worst case, the loops are skewed to the point that they are serialized. This approach requires that the compiler be able to predict the number of iterations in each loop, at least in terms of loop-constant program variables.

7.2.2 The problem with localized vector spaces

Wolf uses *localized vector spaces* to model when reuse of data actually results in locality. The localized vector space is the set of iterations in the inner tile of a tiled loop nest, counting from the innermost loop outwards to the first loop the first loop with a large number of iterations (*i.e.,* the first loop whose loop bounds are not compiler-selected).

This model is almost correct for cache-based systems. While it is true that a loop with a large number of iterations *can* access a lot of data, and thus flush previously used data from the cache, it is not necessarily true that it does so. Some loops perform repeated computation with the same data and do not change the data held in a cache.

For compiler-controlled memories, including bypassable caches and RAMs, localized vector spaces do not capture locality correctly. In machines where the compiler controls M_1, there is effectively a separate cache for each stream; accessing large amounts of data for one stream does not flush data held for other streams. There can be significant locality outside of the innermost tile. This locality is called intertile locality and was addressed in detail in Chapter 5.

Taking advantage of this locality requires a computer architecture which allows the compiler to exercise control over the memory hierarchy. The compiler can exercise full control over RAM buffers used in place of caches. Even bypassable caches or caches which allow lines to be "locked in" allow the compiler some amount of control. Another possibility is to include multiple separately addressed data caches, so that the compiler can assign a separate stream to each cache.

Wolf's localized iteration spaces force him to tile loops with large iteration counts which do not access new data, that is, loops which are in the iterations space $\mathcal{I}$ but not in the data space $\mathcal{D}$. If this loop is part of a tilable nest, it can be interchanged to be the innermost loop, and need not be tiled at all. The techniques of this thesis can address this problem in one of two ways: the compiler can recognize such loops and interchange them innermost prior to tiling, or it can add these loops to the set to be tiled. The scheduler will note that these loops allow locality, and will make them the innermost controlling loop; since no data is accessed, no value will be assigned to the β-value for this loop by the tile size optimizer. The compiler can choose the tile size vector to be ∞ in any loop which isn't assigned a value by the tile size optimizer. A simple post-tiling optimizer can remove controlling loops with a tile size of ∞.

7.2.3 Loop jamming: a hack for choosing $\vec{\beta}$

The previous examples have demonstrated that choosing $\vec{\beta} = (\beta, \beta, \ldots, \beta)$ is not generally optimal. In fairness, Wolf handles the examples given so far with a neat trick: he coalesces the outermost tiled loop with the innermost controlling loop (this is the "jam" part of "unroll and jam"). This has the same effect as choosing $\vec{\beta} = (\beta, \beta, \ldots, 1)$. Of course this is not as general as solving for the tile sizes directly. The following is an example where Wolf's method is insufficient (this example is derived from the QR-decomposition code by adding the `w`-stream; the computation performed in the loop body was simplified since it is irrelevant to the locality tiler):

```
for k = 1 to n
    for j = 1 to n
        for i = 1 to n
            a[i,j] = a[i,j] + r[k,j] * w[k];
```

The reuse space is the full iteration space. Wolf will tile the entire space, and then jam the `i`-loop back together to produce code like this:

```
for kk = 1 to n by B
    for jj = 1 to n by B
        for i = 1 to n
            for j = jj to min (n, jj+B-1)
                for k = kk to min (n, kk+B-1)
                    a[i,j] = a[i,j] + r[k,j] * w[k];
```

which results in 1 fetch per element of `w`, one fetch per element of `r`, and $O(N^3/\sqrt{M})$ fetches per element of `a`.

Using the techniques of this thesis, every loop is tiled because the data space spans the iteration space. This results in the code

```
for kk = 1 to n by $\beta_k$
    for jj = 1 to n by $\beta_j$
        for ii = 1 to n by $\beta_i$
            for k = kk to min (n, kk+$\beta_k$-1)
                for j = jj to min (n,jj+$\beta_j$-1)
                    for i = ii to min (n,ii+$\beta_i$-1)
                        a[i,j] = a[i,j] + r[k,j] * w[k];
```

The scheduler selects the loop ordering which minimizes data motion. Table 7.1 represents the scheduler's knowledge. Remember that the scheduler decides on a loop nest ordering before the tile size vector is chosen, so it uses the original loop nest, and not the tiled loop nest, to estimate the number of M_2 operations required by each ordering of the controlling loops.

Loop order	References for each stream			Total
	`a[i,j]`	`r[k,j]`	`w[k]`	
`k,j,i`	$2n^3$	n^2	n	$2n^3 + n^2 + n$
`k,i,j`	$2n^3$	n^3	n	$3n^3 + n$
`j,k,i`	$2n^3$	n^2	n^2	$2n^3 + 2n^2$
`j,i,k`	$2n^2$	n^3	n^3	$2n^3 + 2n^2$
`i,k,j`	$2n^3$	n^3	n^2	$3n^3 + n^2$
`i,j,k`	$2n^2$	n^3	n^3	$2n^3 + 2n^2$

Table 7.1: Data motion costs of different schedules

The fewest M_2 references is achieved by the ordering `k,j,i`, so the scheduler selects that order for the controlling loops. The cost model is then given by

$$M_2 \text{ operations} = \frac{2n^3}{\beta_{\mathtt{k}}} + n^2 + n \tag{7.1}$$

subject to the memory constraint

$$\text{size}(M_1) \geq \beta_{\mathtt{i}}\beta_{\mathtt{j}} + \beta_{\mathtt{j}}\beta_{\mathtt{k}} + \beta_{\mathtt{k}}$$

The minimum of Equation 7.1 occurs when $\beta_{\mathtt{i}} = \beta_{\mathtt{j}} = 1$ and $\beta_{\mathtt{k}} = (\text{size}(M_1) - 1)/2$. After removing the tiled loops with tile size 1, the code looks like this:

```
for kk = 1 to n by βk
    for j = 1 to n
        for i = 1 to n
            for k = kk to min (n, kk+βk-1)
                a[i,j] = a[i,j] + r[k,j] * w[k];
```

which results in 1 fetch per element for `w` and `r`, but requires only $O(n^3/\text{size}(M_1))$ operations for `a`. In essence, we have jammed the `j`-loop as well as the `i`-loop; Wolf cannot do this because of the way he models the localized vector space.

7.2.4 Blocking

Wolf does not address blocking memory accesses when no locality is involved. In this thesis, we model block-oriented memories explicitly. By tiling all loops, and choosing the blocking size β to be 1 in some loops, the compiler is, in effect, choosing to tile exactly the loops which minimize execution time. This effect comes for free; the compiler does not need to consider separately whether it should block a given loop or not.

Consider the following code:

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            a[i] = f(a[i], j, k);
```

(here `f(a[i], j, k)` denotes some function which reads a[i]; for locality purposes, the exact computation is irrelevant). There are n^2 iterations performed between accesses to `a`.

Using the techniques of this thesis, the loop nest would be tiled resulting in the code:

```
for ii = 1 to n by $\beta_{\mathtt{i}}$
    begin
    for i = ii to min (n, ii+$\beta_{\mathtt{i}}$-1)
        abuf[ii-i+1] = a[i];
    for i = ii to min (n, ii+$\beta_{\mathtt{i}}$-1)
        for j = 1 to n
            for k = 1 to n
                abuf[ii-i+1] = f(a[ii-i+1], j, k);
    for i = ii to min (n, ii+$\beta_{\mathtt{i}}$-1)
        a[i] = abuf[ii-i+1];
    end
```

Accesses to `a` are now blocked; by default, the compiler will choose $\beta_{\mathtt{i}} = \text{size}(M_1)$. The techniques used in this thesis could easily be modified to choose tile size vectors so that the block sizes match the block sizes supported by hardware if necessary.

7.2.5 Abstracting the reuse space

Using reference vectors to guide the transformation process allows a more powerful set of transformations to be applied. The example of this section illustrates a case where this added power is needed. This example was deliberately constructed for its illustrative purposes; programs that can use the added power of the techniques suggested in this thesis are rare, because programmers almost always use simple loop indices as subscripts rather than complex linear combinations of loop indices. The techniques suggested in this thesis are inexpensive enough to use in the general case, however; a compiler using them will have the extra power when it is needed.

The techniques suggested by Wolf and Lam view the transformation as a way of creating locality in a loop nest, without specific regard for the exact direction of locality. This leads them to abstract from the directions of locality for a given stream to a set of loops carrying that locality. The code in Figure 7.24 shows an example where this leads to less than optimal performance.

```
for i = 1 to n
    for j = 1 to i
        A[i-j] = A[i-j] + f(i,j);
```

Figure 7.24: An example loop

In this case, there is locality for the `A` stream in direction $i - j$. Wolf would abstract this locality and consider it to be carried by both the `i` and `j` loops. Since this is the only locality available, and the locality-carrying loops are already interchangeable, no transformations are necessary before tiling. Both loops are tiled, resulting in the code of Figure 7.25.

```
for i = 1 to n by $\beta_i$
    for j = 1 to i by $\beta_j$
        for ii = i to min(n,i+$\beta_i$-1)
            for jj = j to min(i,j+$\beta_j$-1)
                A[ii-jj] = A[ii-jj] + f(ii,jj);
```

Figure 7.25: The example loop after tiling

Note however, that while this does result in intratile locality, there is no intertile locality because the reference vector $(1,-1)$ is not perpendicular to either loop direction vector $(1,0)$ or $(0,1)$. The number of refreshes is therefore given by

$$\rho_{\texttt{A[i-j]}} = \sum_{i=1}^{n}\sum_{j=1}^{i}\frac{1}{\beta_i\beta_j} = \frac{n^2}{2\beta_i\beta_j}$$

The buffer space required by a $\beta_i \times \beta_j$ tile is $\beta_i + \beta_j$. The total cost in M_2 transfers is then

$$\frac{2n^2\beta_i + \beta_j}{2\beta_i\beta_j} = O\left(\frac{n^2}{\text{size}(M_1)}\right)$$

(the number of transfers is twice the number of refreshes since each refresh operation is a read and a write).

Figure 7.26 shows the tiling that results from this abstraction of the locality space.

By modeling the locality explicitly for each variable, better performance can be achieved. In this case, the candidate tiling basis set is $\{i, j, i-j\}$. Using basis $\{i, j\}$, the total cost is the same as above. Using the basis $\{i, i-j\}$, however, the iteration space is first skewed, resulting in the code shown in Figure 7.27 Tiling will now leave intertile locality in the `l` loop. In fact, since all data is local to the `l` loop, it can be interchanged outermost, and only the `k` loop need be tiled. The tiled code is shown in Figure 7.28.

In the transformed code, one M_2 read and one M_2 write occur per element of `A`, so the total number of memory operations is `2n`, which is an order of magnitude smaller than n^2.

Figure 7.29 shows the iteration space for the transformed code. Note that only a one-dimensional tiling is required.

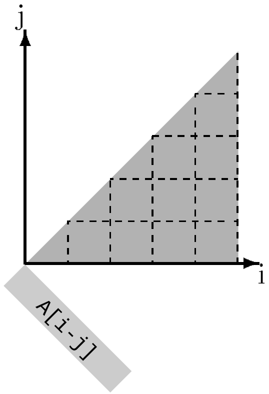


Figure 7.26: Iteration space diagram of tiled code using abstracted reuse space

```
for k = 1 to n
    for l = 1 to n-k+1
        A[k] = f (k+l, l);
```

Figure 7.27: The example loop transformed for locality

```
for l = 1 to n
    for k = 1 to n-l+1 by $\beta_k$
    begin
        for kk = k to min(k+$\beta_k$, n-l+1)
                Abuf[kk-k] = A[kk];
        for kk = k to min(k+$\beta_k$, n-l+1)
                Abuf[kk-k] = f (kk+l,l);
        for kk = k to min(k+$\beta_k$, n-l+1)
                A[kk] = Abuf[kk-k];
    end
```

Figure 7.28: The tiled transformed loop

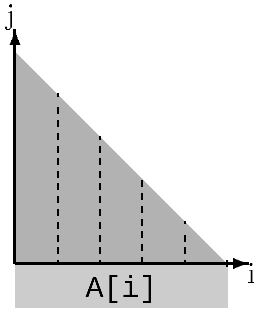


Figure 7.29: Iteration space diagram of tiled code using abstracted reuse space

7.3 Conclusions

In this chapter, several example programs were given. Each one has been tiled for locality. In each case, the techniques suggested in this thesis equal or exceed older techniques in terms of the number of M_2 operations required by the resulting code. The gain is due to increased efficiency in M_1 usage for most programs. This increased efficiency is important for small M_1 memories, but is less important for larger M_1 memories because tiling results in computation-bounded programs which do relatively little I/O. There are some cases where the number of M_2 operations can be reduced by a factor which increases with M_1 size. For nearly all programs in the benchmark set, the tiling basis candidate vector set is just I, so few decisions need to be made on the average.

We have demonstrated that the new techniques address the shortcomings of Wolf and Lam's methods of tiling for locality in machines with software-controlled memory hierarchies. For such machines, a new definition of locality is required, to take into account the fact that different streams cannot interfere with one another as they do in cache-based memory systems. Intertile locality as defined in Chapter 5 allows the compiler to schedule tiles to achieve all the locality possible.

Intertile locality combined with the new technique of solving for the optimal tile sizes allows the compiler forgo making a decision about which loops to tile: the entire loop nest is tiled, and the tile size is set to be 1 or ∞ in loops which need not have been tiled. Loops with a tile size of 1 are placed outside the innermost tile; effectively, they have a controlling loop but no tiled loop. Loops with a tile size of ∞ are placed inside the innermost tile; they have a tiled loop but no controlling loop.

Chapter 8

Conclusions and future work

The illustrations in Chapter 7 showed that the new techniques presented in this thesis reduce the execution time of programs compared to standard tiling methods. The new techniques perform at least as well as previous methods, often better, and are no more expensive in terms of compilation time in the case where each dimension of each array subscript is a function of only a single loop index variable.

The next section reiterates the contributions of this thesis, and the conclusions that can be drawn. The last section describes the limitations of the approach taken in this work, and describes important steps that could be taken to follow up this work.

8.1 Contributions of this work

The tiling techniques investigated in this thesis are an advance in the state of the art in tiling. New techniques are used for modeling the relationship of data to the iteration space. New algorithms are used for tiling, which are no more expensive than prior methods when applied to the simple loops that predominate scientific programs. The new techniques yield faster code in most cases, significantly faster code in a few cases, and never worsen performance in any case.

8.1.1 Mathematical tools

The mathematical foundation of this work makes it easy to integrate parallelism and locality as goals for the tiling software. This thesis has thoroughly investigated tiling for locality on uniprocessors. For multiprocessors, there are two cases. In the first case, all processors operate in parallel to execute a tile, but only a single tile is being executed at a time. This is called intratile parallelism. Intratile parallelism is easily captured by the methods used, because the scheduler does not consider parallelism within a tile. The second parallel case is intertile parallelism, in which tiles are executed in parallel on different processors. In this work, a simple form of intertile parallelism is combined with locality goals; more complex forms of parallelization using wavefronting will fit into the same framework, with an appropriate adjustment in the cost models.

Reference matrices are a powerful tool for modeling array accesses within a program. They directly relate every array element to the iterations using that element, and vice versa. Using a reference matrix for each stream allows the compiler to evaluate directions of locality and directions of parallelism using linear algebra techniques.

Rectangular buffering and skewed rectangular buffering are important techniques for use in block-oriented systems. These addressing techniques precisely map array elements in the global program data space into buffered array elements in M_1. Rectangular buffering describes this mapping when rectangular blocks of data are involved; skewed rectangular buffering generalizes this mapping to allow skewed rectangles of the global data to be buffered in rectangular arrays in M_1. This allows additional flexibility without increasing the address generation cost in the innermost loops of a tile.

The cost models developed in this thesis are another important tool. When tiling an iteration space, the number of iterations does not change. To calculate the relative benefits of one tiling as compared to another, it is sufficient to count the number of times a data item must be moved from M_2 to M_1 and back. Since each item has to be moved at least once, the first time data is moved into M_1 need not be counted either. The concentration is then on the number of times data is *refetched.* Refetches are a direct cost, adding directly to the execution time of a loop nest. Overallocation is an indirect cost. Standard square tiling techniques typically overallocate one or more streams, providing too much space in M_1. More efficient use of M_1 reduces the overall execution time by reducing the number of times data is refetched. In this work, the buffer space required for each stream is calculated to minimizes the number of refetches given a particular basis choice B.

Different memory systems can be easily integrated into the cost model as well. Memories that support block transfers can be modeled easily since all M_1-M_2 transfers are block transfers. Memory systems in which data can be moved directly from slower levels of the memory hierarchy into the processor can be handled by streaming non-local data directly into the CPU. Lastly, the buffering techniques of Chapter 4 can handle simple rectangular blocks or more complex skewed rectangular blocks of data, maximizing the space-efficiency in M_1.

Finally, these cost models can be applied regardless of what technique is used to to find the tiling basis. Once a tiling (and a schedule) is chosen, the number of data items required for each stream in a tile is easy to determine. The number of times that data is fetched *given an ordering of the controlling loops* is also easy to determine. Thus, the technique of solving for tile sizes used in this thesis can be applied to any tiling mechanism.

8.1.2 Algorithmic costs

Compared to the best previous work, presented by Wolf and Lam[54], the tiling techniques investigated in this thesis are expensive in the general theoretical case, but not in practical cases. Wolf and Lam present an algorithm that transforms a loop nest to a tilable loop nest (if possible); their algorithm is $O(n^3d)$ where n is the number of loops and d is the number of dependences. They do not schedule tiles for intertile locality, they do not choose optimal tile sizes, and they do not generate buffering code (their work is for cache-based systems so buffering code is not required).

This work concentrates on quality of the code rather than the compiler effort required to achieve it. Wolf and Lam avoid an exponential search of the transformed space, but at the cost of losing efficiency. Since the search is exponential in loop nest depth, and the nest depth is almost always small, the "exponential" search can actually be carried out very quickly.

In this work, a candidate set of basis vectors is generated, and every linearly independent subset of this candidate set is evaluated. In the general case this approach is exponential in the number of loops and also in the size of the candidate set. Fortunately, for most programs, the candidate set is just I, and there is only one candidate basis to be evaluated. Evaluating a basis requires computing the M_1 memory requirement of each stream, and computing the number of refresh operations given a schedule of the controlling loops. These can be computed in time linear in the number of loops. Once they are computed, the resulting nonlinear optimization problem must be solved either symbolically or numerically. Although the algorithm presented is theoretically exponential in complexity, in practice it is linear in the loop nest depth, except for the code that solves the nonlinear optimization problem. Standard nonlinear solvers can be used for this step.

8.1.3 Code quality

Code quality is improved if possible using the new techniques relative to older techniques for performing the same kinds of transformations. The new techniques for choosing a basis are at least as good as previous methods; in many cases the new methods perform better, because the new techniques a based on a definition of locality developed for software-controlled memories. The new techniques for finding optimal buffer sizes are an improvement over the old methods, which were designed for caches and not buffers.

Wolf and Lam[33, 54, 55, 56, 57] are the most thorough previous work on tiling. Rather than choosing a tiling basis, they find a set of permute, skew, and reverse transformations that result in a tilable loop nest. This is equivalent to finding a unimodular tiling basis. Rather than attempting to find a particular tiling basis that fits naturally to the data, they simply take any basis that gives locality for a subset of the streams. In this work, every tiling basis that can be constructed from a candidate set is examined; the scheduler extracts as much locality as possible for that basis, and then the basis that results in lowest execution time is chosen. Including reference vectors in the candidate set ensures that bases resulting in efficient M_1 usage are chosen if possible. Including the rays of the dependence cone in the candidate set ensures that the compiler can search the breadth of the available space for a legal transformation if necessary.

Given a particular basis choice, Wolf and Lam choose the tiles sizes to be small enough to avoid self-interference in the cache. The buffer-optimizing work in this thesis is a new contribution. It improves the efficiency of M_1 usage for buffers. The matrix-multiply, QR-decomposition, and LU-decomposition examples in Chapter 7 showed the advantage of these techniques. The compiler solves for the buffer sizes that minimize compilation time. Since the buffer size vector chosen will minimize the execution time, performance can only be enhanced by applying this technique. The enhancement in execution time is greatest for relative small M_1 memories, of sizes typically available on-chip (either as large register files or as on-chip buffer memories). Larger M_1 memories allow more straightforward tiling techniques to yield computation-bounded programs, so reducing I/O further does not yield significant performance improvement.

As a result of the extra time spent searching for a tiling basis that matches the reference vectors, candidate-set tiling can produce better code than heuristics based on which loops carry locality in the source program. Reference vectors capture locality information on a per-stream basis; this information allows the candidate set to capture all the information of other methods, but does not artificially reduce the information carried. Wolf and Lam abstract the locality space of a stream to the set of loops that carry the locality; Section 7.2.5 shows an example where this causes their method to perform significantly worse that the techniques developed in this thesis.

The candidate set method therefore will always perform at least as well as other forms of tiling for locality. In some cases, either where square tiles are not optimal, or where the locality in the source code is not accurately modeled by noting which loops carry the locality, the new method performs better.

8.1.4 Limitations of the approach

Several assumptions are critical to the application of this thesis. Most of the assumptions are fairly obvious and were stated in Chapter 1. A few assumptions are more esoteric in nature and were presented in the context of later chapters. A few of these assumptions are reiterated here to ensure that the casual reader has not missed these important points.

First, it is assumed that all loops execute enough iterations that the fragmentation of $\frac{n}{\beta_i}$ can be ignored. For most scientific loops this is not an issue, but one special case is worth considering. Loops with loop bounds which are parameters (subroutine or procedure parameters) may be deliberately written with the intent that the parameter may be usefully set to 1. If the compiler cannot determine the loops bounds at compile time (or at least determine that they are sufficiently large), the cost model may not be optimal. In this case, the compiler could test the size of the parameter, and execute different versions of the code depending on whether the parameter is large or small.

A second assumption is that the constant offset vectors are small, so only a minor tweaking of the $\vec{\beta}$ factors is needed. It is possible that the offset may be the length of an entire row or column; in this case it should be buffered separately, rather than extending the buffer size dedicated to the uniformly generated set of streams.

8.1.5 Conclusions

The compiler can manage the memory hierarchy in both parallel and sequential machines for programs that access large arrays with regular access patterns. In parallel machines, interprocessor communication is part of the memory hierarchy.

The techniques outlined in this work allow the compiler to manage data motion throughout the memory hierarchy without hardware support for this motion. These techniques can be applied to support larger memory spaces on machine like Crays, which do not have virtual memory support. The techniques can be used on machines like iWarp, a systolic array processor with a programmer-controlled memory hierarchy, to increase performance by allowing most array accesses to use data in the faster memory. Some modern shared memory multiprocessors are being built without hardware for cache coherence, like IBM's POWER/4. The techniques for modeling data motion and for selecting tiling bases can be used to support software cache coherence in these machines in a high-performance optimizing compiler.

8.2 Future work

There are several important ways this work could be extended. Integrating software prefetch would allow the compiler to take advantage of more complex memory systems that allow pipelined requests. The effects of distance locality on tiling should be more closely investigated. Machines that allow data to move from the slower levels of the memory hierarchy can be supported with some minor additional work. In this work the compiler assumes that all loops nests are perfect; additional work could be performed to allow the compiler to pick better schedules for non-perfect loop nests. Finally, a major avenue of research that has been opened by this research is developing other ways to integrate scheduling for locality and parallelism.

8.2.1 Software prefetch

This thesis has shown that the compiler can manage the memory hierarchy for linear accesses to array data. The next logical step is to integrate software prefetching to support compiler memory management of all accesses. This problem is an extension of the register allocation problem; the biggest unknown is how to partition M_1 between scalar and array data. In this thesis, scalar data is assumed to be small enough that it will fit in the register file and need never be present in M_1. For the tight loops typical of scientific programs, this is often the case, but this will probably not hold true for C language programs, for example.

8.2.2 Distance locality

The techniques used in this thesis ignore distance locality: for example, the locality that would occur between the references `c[i]` and `c[i-3]`. In this thesis, the two accesses are considered a single stream. To correctly implement the accesses, the length of the offset would be added to M_1 memory requirement expression (for multidimensional arrays, the offset distance is added into the expression in each dimension before multiplying the dimensions together). In the examples encountered, this has not presented a serious problem, but a more complete cost model could take these distance-accesses into account as well.

8.2.3 M_2 streaming

Another important way this work could be extended is to allow direct access from M_2 memory to the CPU without an intermediate stop in M_1. The current cost model does support this type of access correctly. Recall the tiled matrix multiply code of Figure 3.2. Note that the tiled code is optimal for the target machine model, which does not allow data to be moved directly from M_2 into M_1. If data *can* be brought from M_2 into M_1, additional savings are possible. In this case, the elements of `b[k,j]` brought into M_1 are used β_i times, but only one element at a time is used. If data can be streamed directly from M_2 into the processor, only one element of the `b[k,j]` stream should be fetched at a time; it can be stored in a register. The extra space in M_1 would then be divided between the other two streams.

M_2 streaming can be incorporated fairly easily into the framework provided by this thesis. After scheduling and computing buffer requirements, the compiler constructs the cost model, leaving out any streams which do not exhibit significant reuse within a tile. Streams dropped from the cost model are fetched directly from M_2 when needed.

Streams with no reuse within a tile can always be dropped. Streams which exhibit reuse due to constant offsets are dropped if the reuse can be accommodated in the register file by allocating extra space. For example, in Figure 8.1, two streams are each reused once due to constant-offset accesses. The `a` stream's reuse can easily be accommodated if the `j` loop is innermost, by simply allocating two registers, one for `a[i,j]` and one for `a[i,j-1]`. The `b` stream's reuse is in the `i+j` direction; accommodating its reuse requires either allocating a number of registers proportional to the tile size, or transforming the loop so that the innermost loop executes in the `i+j` direction.

```
for i = 1 to n
  for j = 1 to n
    ...a[i,j]+a[i,j-1]...b[i,j]+b[i-1,j-1]...
```

Figure 8.1: Examples of constant-offset reuse

8.2.4 Non-perfect loop nests

The techniques as outlined in previous chapters do not take into account branching in the body of the loop. In the LU decomposition program of Figure 7.18, for example, the last assignment statement (which writes `a[i,j]`) will execute much more frequently than the other two. The compiler should take this into account when selecting the loop ordering for intertile locality. Additional work would be required to integrate this into the prototype compiler.

8.2.5 Integrating tiling for parallelism and locality

This thesis develops a framework for considering the trade-off between parallelism and locality. Specifically, locality was thoroughly investigated for uniprocessors, and it was shown that compiling for intratile parallelism requires little more from the locality-optimizer than compiling for a uniprocessor. Intertile parallelism has only begun to be addressed, however. In this thesis, it was argued that many loops in scientific programs have at least one fully parallel loop; the compiler can therefore find enough parallelism to keep all the processors busy, and locality can be addressed using the other loops in the loop nest.

An important class of loops has no single inherently parallel direction, but can be executed in parallel along a wavefront. This is equivalent to skewing the loop nest, and then parallelizing the resultant loop. If the loop bounds are large enough, there will be enough tiles to keep all the processors busy in the steady state. The compiler does need to consider the start-up and tear-down costs of the wavefront (which is effectively a pipeline). These costs are dependent on the tile size vector $\vec{\beta}$.

The hardest open question is how to schedule the tiles, since the optimal schedule may depend on the start-up costs, and the start-up costs depend on $\vec{\beta}$, which has not been chosen at scheduling time. Optimality can be ensured by examining *every* possible schedule. This may be another example of a theoretically exponential search which in practice can be carried out very quickly.

8.2.6 Compiling for split-memory machines

The techniques of this thesis enable compiler-writers to take new approaches to compiling programs for parallel machines. One example of this is that a programmable systolic array can be viewed in a new way. Traditionally, systolic arrays are treated as an array of individual processors, as shown in Figure 8.2. Each "cell" or processor consists of a processor (CPU), a local memory (LM), and a network interface (systolic pathway segment).

A small systolic array (or a segment of a larger array) can be treated as a single VLIW "superprocessor" with many processing elements (Figure 8.3). The extended processor has a segmented memory system; data stored in memory segment one can only be accessed through memory port 1. However, since systolic arrays can communicate data between processors at (typically) one word per processor per clock, data can be shifted quickly to the correct functional unit for processing.

The primary benefit of this approach is that the size of local memory considered to be "owned" by a processor is much larger, since the local memories of several processors are treated as a single processor. This is especially important in machines where there are large secondary memories attached to only a few processors. Groups of "superprocessors" can be formed around the secondary memories, and scheduled (using VLIW techniques) as a single processing unit.

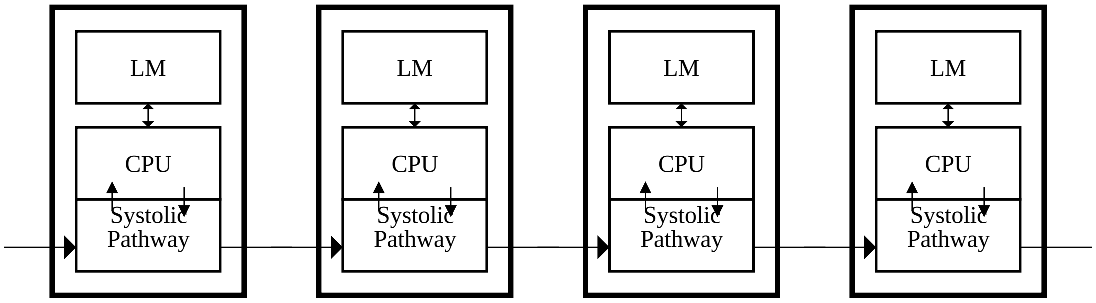


Figure 8.2: The traditional view of a systolic array

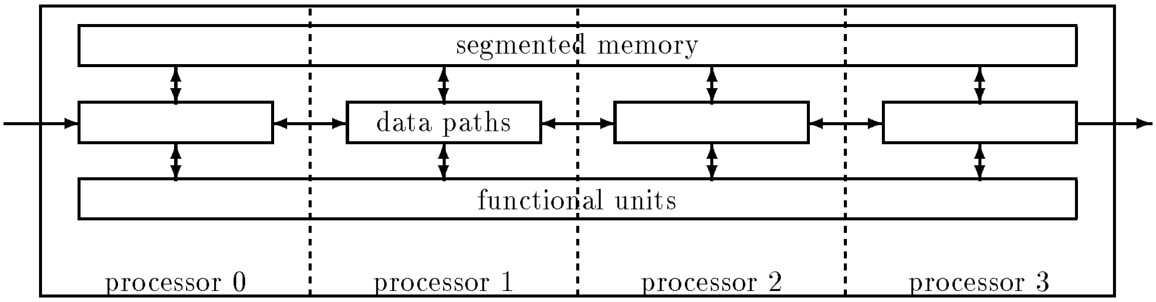


Figure 8.3: Systolic cells combine to form a "superprocessor"

The principle challenge in this approach will be handling exceptions. Since MIMD systolic arrays have separate program counters for each cell, an exception in one processor is not directly communicated to other processors working on the same VLIW "superinstruction" (the concatenation of instructions issued over several processors). Nevertheless, the high communication bandwidth of the systolic communication pathway may be sufficient to allow the necessary synchronization.

Bibliography

[1] Walid Abu-Sufah, David Kuck, and Duncan Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* The Association for Computing Machinery, June 1992. Also available as ACM SIGPLAN Notices, Volume 27, Number 9, September 1992.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, 1986.

[4] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 39–50. The Association for Computing Machinery, April 1991.

[5] Tom M. Apostol. *Calculus.* John Wiley and Sons, Inc, second edition, 1969. Volume II, Multi-Variable Calculus and Linear Alegbra, with Applications to Differential Equations and Probability.

[6] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. An interactive environment for data partitioning and distribution. In *The Fifth Distributed Memory Computing Conference* [25], pages 1160–1170.

[7] Utpal Banerjee. *Dependence Analysis for Supercomputing.* The Kluwer International Series in Engineering and Computer Science: Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, 1988.

[8] Wiliam H. Beyer. *CRC Standard Mathematical Tables.* CRC Press, Incorporated, 2000 N.W. 24th Street, Boca Raton, FL 33431, twenty-sixth edition, 1981.

[9] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* [27], pages 40–52.

[10] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceeding of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 400–405. Society for Industrial and Applied Mathematics, December 1989.

[11] Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. In Lee W. Hoevel and Veljko Milutinovic, editors, *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, pages 277–285. IEEE Computer Society Press, 1989.

[12] Ron Cytron. Doacross: Beyond vectorization for multiprcoessors (extended abstract). In *1986 International Conference on Parallel Processing*, pages 836–844. IEEE Computer Society Press, 1986.

[13] George B. Dantzig and Curtis B. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.

[14] R. J. Duffin. On Fourier's analysis of linear inequality systems. *Mathematical Programming Study*, 1:71–95, 1974.

[15] Christine Eisenbeis, William Jalby, Daniel Windheiser, and François Bodin. A strategy for array management in local memory. In *3rd Workshop on Programming Languages and Compilers for Parallel Computing* [37], pages 238–253. From preliminary proceedings, issued as a technical report by University of California, Irvine.

[16] J. A. B. Fortes and D. I. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2:277–301, 1985.

[17] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. In *1988 International Conference on Supercomputing*, pages 238–253. The Association for Computing Machinery, July 1988.

[18] 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.

[19] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *1990 International Conference on Supercomputing*, pages 354–368. The Association for Computing Machinery, June 1990.

[20] Andrew Grace. *Optimization Toolbox For Use with MATLAB.* The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760, August 1992. Internet e-mail: info@mathworks.com.

[21] Rajiv Gupta and Jim Kajiya. Compiler optimization of array data storage. Technical Report Caltech-CS-TR-90-07, California Institute of Technology, April 1990.

[22] Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990. Manual Part Number: 09740–90039.

[23] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. In *Supercomputing '91* [28], pages 86–100.

[24] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings, Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326–333. The Association for Computing Machinery, 1981.

[25] *The Fifth Distributed Memory Computing Conference.* IEEE Computer Society Press, April 1990.

[26] *1990 International Conference on Parallel Processing.* IEEE Computer Society Press, 1990.

[27] *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* IEEE Computer Society Press, April 1991.

[28] *Supercomputing '91.* IEEE Computer Society Press, 1991.

[29] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM SICACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 319–329. The Association for Computing Machinery, January 1988.

[30] *Supercomputing '89: Proceedings of the Fourth International Conference on Supercomputing.* International Supercomputing Institute, November 1989. Santa Clara, CA.

[31] H. T. Kung. Memory requirements for balanced computer arcitectures. *Journal of Complexity*, 1:147–157, 1985.

[32] H. T. Kung. Computational models for parallel computers. Technical Report CMU-CS-88-164, Carnegie Mellon University, August 1988. Prepared for the Royal Society Discussion Meeting on "Solving Scientfic Problems on Multiprocessors".

[33] Monica Lam, Edward Rothberg, and Michael Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* [27], pages 63–74.

[34] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 285–295. Also available as ACM SIGPLAN Notices, Volume 27, Number 9, September 1992.

[35] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 249–259. Yale University, August 1992.

[36] Cror E. Maydan, Saman P. Amarasighe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Conference Record of the Twentieth Annual ACM SICACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 2–15. The Association for Computing Machinery, January 1993.

[37] *3rd Workshop on Programming Languages and Compilers for Parallel Computing.* Pitman / MIT-Press, August 1990. From preliminary proceedings, issued as a technical report by University of California, Irvine.

[38] Keshav Pingali and Anne Rogers. Compiling for locality. In *1990 International Conference on Parallel Processing* [26], pages II:142–146.

[39] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice Univeristy, May 1989.

[40] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91* [28]. To appear in Communications of the ACM.

[41] William Pugh and David Wonacott. Eliminating false data dependences using the Omega test. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 140–151. Also available as ACM SIGPLAN Notices, Volume 27, Number 9, September 1992.

[42] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Supercomputing '89: Proceedings of the Fourth International Conference on Supercomputing* [30], pages 637–646. Santa Clara, CA.

[43] J. Ramanujam and P. Sadayappan. Nested loop tiling for distributed memory machines. In *The Fifth Distributed Memory Computing Conference* [25], pages 1088–1096.

[44] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *1990 International Conference on Parallel Processing* [26], pages II:179–186.

[45] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[46] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[47] Hudson Ribas. *Automatic Generation of Systolic Programs From Nested Loops.* PhD thesis, Carnegie Mellon University, June 1990.

[48] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, Research Institute for Advanced Computer Science, August 1990.

[49] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1986.

[50] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Programming task and data parallelism on a multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–22, May 1993.

[51] Alan Sussman. *Model-Driven Mapping of Computation onto Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, September 1991.

[52] Ping-Sheng Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, May 1989.

[53] David. A. Wismer and R. Chattergy. *Introduction to Nonlinear Optimization: A Problem Solving Approach.* North-Holland Series in System Science and Engineering. North-Holland, 1978.

[54] Michael Wolf and Monica Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44. The Association for Computing Machinery, June 1991.

[55] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop transformations. In *3rd Workshop on Programming Languages and Compilers for Parallel Computing* [37]. From preliminary proceedings, issued as a technical report by University of California, Irvine.

[56] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[57] Michael Edward Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, August 1992.

[58] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* University of Illinois at Urbana-Champaign, 1982.

[59] Michael Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the Fourth International Conference on Supercomputing* [30], pages 655–664. Santa Clara, CA.