INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
RICE UNIVERSITY

Improving Data Locality for Caches

by

Karim Esseghir

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Chairman, Associate Professor
Computer Science

Linda Torczon, Faculty Fellow
Computer Science

John K. Bennett, Assistant Professor
Electrical and Computer Engineering

Houston, Texas

September, 1993
Abstract

Improving Data Locality for Caches

by

Karim Esseghir

Since the introduction of cache memories in computer architecture, techniques to improve the data locality of programs with respect to cache have been essential to good performance. This thesis is a contribution in that direction. It is divided in three main parts:

- A survey of the existing techniques that address data locality and the different approaches proposed in the literature.

- The introduction of a new program transformation that is in fact a combination of two known transformations, namely iteration space tiling and copy optimization, to achieve better data locality than can be attained with either of the two techniques by themselves.

- A suggestion of a general approach to data locality improvement as a whole, and an algorithm to compute block sizes for tiled programs that avoid self interferences in cache.
Acknowledgments

This work would not have been possible without the support, and clairvoyant advice of Keith Cooper, my thesis advisor. I am grateful to him for that. All due thanks naturally go to Linda Torczon and John Bennett who graciously accepted to serve on my committee. I am especially thankful to Reinhard von Hanxleden, my colleague, roommate, and most importantly friend who painstakingly read my thesis when it was at a critical stage and provided well-taken points and criticism. I also wish to thank Preston Briggs for offering his insightful suggestions, particularly at the early stage of the thesis. Nathaniel McIntosh was also helpful and encouraging with his comments, especially at the late stages of the work. Thanks to Ivy Jorgensen for proofreading the thesis.

My colleagues, especially Ervan Darnell—who was also my sailing partner—and Ulrich Kremer made my stay Rice enjoyable; I am grateful to all of them.

Finally, thanks to my dear family and my dear friend Saniya, who were always encouraging and supportive from a (long) distance, and, at times, more eager than I was to see this thesis reach completion.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii

1 Introduction 1
1.1 The Problem ............................................. 1
1.2 Contributions of this Thesis .............................. 2
1.3 Overview of the Following Chapters ................... 2

2 Caches and Notions of Locality 4
2.1 Introduction ............................................... 4
2.2 Cache Basics ............................................. 5
  2.2.1 Cache Misses ....................................... 8
2.3 Reuse ................................................... 9
  2.3.1 No Reuse ........................................... 9
  2.3.2 Spatial Reuse ...................................... 9
  2.3.3 Temporal Reuse ................................... 10
  2.3.4 Real Programs ..................................... 10
2.4 Data Dependences ....................................... 10
2.5 Iteration Spaces ........................................ 12
2.6 Transformations ......................................... 12
  2.6.1 Loop Fusion ....................................... 13
  2.6.2 Loop Distribution ................................ 13
  2.6.3 Loop Permutation ................................ 14
  2.6.4 Loop Skewing .................................... 15
  2.6.5 Iteration Space Tiling ............................. 16
  2.6.6 Copying ........................................... 18

3 Literature Survey 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Unimodular transformations</td>
<td>19</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Lam and Wolf</td>
<td>20</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Sarkar and Thekkath</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Iteration Space Tiling</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Wolfe</td>
<td>21</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Porterfield</td>
<td>22</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Lam and Wolf</td>
<td>23</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Carr</td>
<td>23</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Sarkar and Thekkath</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Loop Permutation for Stride</td>
<td>24</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Carr</td>
<td>24</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Sarkar et al.</td>
<td>24</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Kennedy and McKinley</td>
<td>25</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Porterfield</td>
<td>25</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Wolf</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Loop Fusion</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Loop Distribution</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Copying</td>
<td>28</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Lam and Wolf</td>
<td>28</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Gallivan, Gannon, Jalby</td>
<td>28</td>
</tr>
<tr>
<td>3.6.3</td>
<td>PGI</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Copy Optimizations</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Copy Optimization</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Vector Add</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2</td>
<td>5-Spot Stencil</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Combining Tiling and Copying — Motivation</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Tile and Copy — An Algorithm</td>
<td>39</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Estimating the Profitability of Tile and Copy</td>
<td>40</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Tiling</td>
<td>40</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Copying</td>
<td>40</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Putting it all together</td>
<td>48</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Cost</td>
<td>49</td>
</tr>
</tbody>
</table>
5 Observations

5.1 Review ........................................ 61

5.2 Data Locality for Set-Associative Caches .......................... 61
  5.2.1 A General Approach for Data Locality Improvement .... 61
  5.2.2 Determining Block Sizes For Set-Associative Caches .... 63

6 Conclusion ...................................... 69
# Illustrations

2.1 High-level memory hierarchy ........................................ 5
2.2 Stencil computation ............................................... 9
2.3 Program exhibiting spatial reuse ................................. 10
2.4 Data dependences .................................................. 11
2.5 Loop carried vs. loop independent dependences ............... 11
2.6 Loop fusion ......................................................... 13
2.7 Program that cannot be fused safely ............................. 14
2.8 Loop distribution .................................................. 14
2.9 Loop permutation .................................................. 15
2.10 Skew of loop j with respect to loop i ......................... 15
2.11 Strip mine followed by interchange ............................ 16
2.12 Matrix multiply and the data that are reused in one iteration of the outer loop .............................................. 17
2.13 Blocked matrix multiplication and the data that it uses in one iteration of the i loop ............................................. 17

3.1 Fusion for spatial and temporal reuse ............................. 26
3.2 Distribution followed by interchange ............................. 27

4.1 Vector add ......................................................... 31
4.2 Vector add after copying ........................................ 32
4.3 5-spot stencil unoptimized for cache interferences ........... 34
4.4 5-spot stencil after copy optimization .......................... 35
4.5 Tiled matrix multiplication ........................................ 37
4.6 Matrix multiplication after tiling and copying A ................ 38
4.7 Matrix multiply after tiling only, and after tiling and copying. 39
4.8 Loop nest example ................................................ 41
4.9 A simple program where the local array can have fewer dimensions than the original array. 43
4.10 Example of a program where the size of the local array can be pruned. 45
4.11 Algorithm for computing the bounds of a convex polyhedron. 47
4.12 Example 47
4.13 The code with the local copies. 49
4.14 Example 50
4.15 Loop nest format for the heuristic. 51
4.16 Tiling the program for the heuristic. 52
4.17 Copying in. 53
4.18 The computation code. 53
4.19 5-spot stencil 54
4.20 5-spot stencil after square tiling 55
4.21 5-spot stencil tiled and copied 56
4.22 Computation code of the 5-spot stencil tiled and copied with the subscript expressions and the loop bounds rearranged. 57
4.23 SOR after tiling only, and after tiling and copying, run on the i860-XP. 57
4.24 Running time of the stencils before and after tile-and-copy executed on the RS/6000 (Row-wise computation). 58
4.25 The same stencil with the data accessed column-wise. 59
4.26 Unoptimized 9-spot stencil, and its performance on the RS/6000 60
4.27 9-spot stencil after tiling, and tile-and-copy, on the RS/6000 60

5.1 Data locality algorithm for one variable in one loop nest. 62
5.2 Tiled loop with one array variable referenced. 63
5.3 Determining a rectangular block that avoids self interferences 64
5.4 Determining a block that avoids interferences in a 2-way set-associative cache. 65
5.5 Rectangular block for multiple variables 67
Chapter 1

Introduction

1.1 The Problem

The introduction of cache memories in computer architecture brought about a new field in code optimization, that is improvement of data locality. Two main observations led to the introduction of cache memories. The first is that during its lifetime, a program will typically access the same data many times, and that data residing in neighboring memory locations are likely to be accessed within a short lapse of time\(^1\). The second is that DRAM (Dynamic Random Access Memory) technology is not keeping up with the pace of microprocessor technology. Moreover, the high cost of SRAM’s (Static RAM’s) prohibits most designers from building architectures that use on SRAM for their main memory. However, SRAM’s still can be used in small quantities at an intermediate level, between the main memory and the CPU’s registers.

Unfortunately, the first observation is not true of all programs. In many cases, however, these programs might be rearranged for better data locality. As written, however, the locality is not adequately exposed.

In this thesis, we are concerned with code optimizations that improve data locality. We focus exclusively on scalar machines, even though in many cases the solutions can be extended to parallel machines.

Some of the obstacles when trying to improve data locality are the following:

- Caches are typically controlled by hardware exclusively. Consequently, the compiler cannot assign a cache location to a particular datum.

- Caches are relatively small. Data may have to be displaced from cache before it gets reused in order to free cache space for other imminent data.

\(^1\)Similar observations can be associated with the instructions of a program.
• The address maps used in some cache models (direct-mapped and set-associative) can assign the same location to two different data items even when the cache is not filled to capacity.

• Data dependences can prohibit the application of a profitable transformation.

• Some techniques are too expensive to be integrated in a compiler.

• In a timesharing environment, different programs compete for cache usage. In a single-task environment, the program may compete with the operating system. Also, data and instructions compete for cache space when the cache is unified. Almost invariably, compilers assume that the cache is a resource dedicated to the program at hand, and that data and instruction caches are separate: the alternative would be too hard to consider.

1.2 Contributions of this Thesis

1. We show that copy optimization works for uniprocessor machines despite the overhead of the copying.

2. We propose tile and copy, a combination of transformations that seeks to avoid both capacity and interference misses in cache. We then suggest a heuristic for guiding the application of the transformation and apply the heuristic manually on some examples to validate it.

3. We propose an efficient algorithm for computing rectangular block sizes for single and multiple references that avoid self interferences, also taking into account the associativity of the cache.

1.3 Overview of the Following Chapters

In Chapter 2, we describe the most common existing cache models and how they work from an abstract perspective. We also present the various aspects of locality that a compiler could exploit to improve cache performance. We then present a number of transformation techniques that can be used for data locality purposes.

Chapter 3 is a literature survey of the recent work that addresses our topic of interest. We try to highlight the strengths, limitations and costs of every algorithm and heuristic surveyed.
In chapter 4, we show, through examples, the profitability of copy optimization as well as its shortcomings. We then propose a combination of iteration space tiling and copy optimization to overcome one of its limitations when the reused data set is too large. First, we present an algorithm with a large range of applicability but relatively high cost. Then, we propose a heuristic for this technique which is less generally applicable but cheaper to apply.

In Chapter 5, we propose new algorithms to compute block sizes for set-associative caches that avoid self interferences. We also address the problem of optimizing with "real" cache implementations in mind.

In the final chapter, we present a brief summary.
Chapter 2

Caches and Notions of Locality

2.1 Introduction

Recent developments in both semiconductor technology and computer architecture have led to an increase in both processor speed and memory access speed. However, the improvements have not been balanced: DRAM access speed has been increasing at the rate of 7% yearly, while CPU performance improved 18% to 35% per year prior to 1985, and has improved 50% to 100% yearly thereafter [15]. Although other technologies exist which can provide memories with speeds approaching those of today’s fast CPUs, their cost is unfortunately too great to justify such use.

Typical computationally intensive programs reuse the same data several times and execute the same instructions between different loop iterations. These observations led to the introduction of caches—small memories built with this fast and costly technology. A cache holds a subset of the larger main memory, and the system tries to keep the data that will be reused soonest in cache. The idea is simple: if the processor is able to find data (or instructions) in cache when it requests them, the performance of the program increases. However, cache behavior depends greatly on the machine, making it difficult for the programmer to optimize for cache while still retaining a high-level, machine-independent program. Another reason is that they would rather focus on the algorithmic aspects of the problem they have at hand. Consequently, in the last decade, compiler writers have been investigating ways to shift this burden to the compiler. Our contribution in this thesis follows along those lines.

---

2 The memory performances refer to DRAM row access time
3 Typically, main memories are built of DRAM’s (Dynamic RAM’s) while registers and caches are SRAM’s (Static RAM’s). The difference between them lies in that the former require a periodic refreshing of their contents.
2.2 Cache Basics

A cache memory is a small fast memory. A cache stores an image of a subset of some larger, slower memory. In general, it is placed between the main memory and the CPU (Figure 2.1). Alternatively, caches can be used as a large buffer, for instance between the main memory and some I/O device. In the remainder of this thesis, we examine aspects associated with caches which are part of the memory hierarchy of a system.

![Figure 2.1 High-level memory hierarchy.](image)

The same address is used to reference a value in main memory or in cache. Thus, its behavior is transparent, except for the running time of the program. System designers use caches to decrease average memory latency. If a consumer of data uses the same location several times and those data can be kept in a cache, average access time decreases. The average access time for $k$ accesses to a datum $i$ is

$$T_a = R \times T_c + (1 - R) \times T_m,$$

where $T_c$ and $T_m$ are the access times of cache and main memory, and $R$ is the fraction of accesses to $i$ that can be satisfied from the cache. $R$ is called the hit
ratio. Without a cache, $T_a$ is just $k \times T_m$. The best cache behavior would bring 1 into the cache on the first reference and keep it there. This results in $R = \frac{k-1}{k}$. or $T_a = \frac{k-1}{k} \times T_c + \frac{1}{k} \times T_m$. If $T_c < T_m$, then accessing values from the cache decreases $T_a$.

Unfortunately, real cache issues are more complex than this simple example. Since a cache only holds a subset of the larger memory, it requires a map from addresses in the larger memory into cache addresses. The map translates a full-length memory address into a set number and an offset. These two numbers allow the hardware to determine if the data currently resides in the cache. Because the cache is smaller than the surrounding memory, the mapping is an onto function, not a one-to-one function. Two data items may map to the same location, in which case, they interfere in the cache. This simple fact has profound consequences for run-time performance.

Architects have proposed and built many different cache designs. The differences can be characterized in a few parameters. The address map is a function of these parameters.

**Cache size:** Cache size is typically specified in bytes. Obviously, larger caches hold more data and are therefore more desirable.

**Line size:** A cache is organized as a series of lines. Transfers into the cache from memory (reads) involve whole lines. Transfers from the cache to memory (writes) may involve whole lines; or modified bytes only. Memory-to-cache bandwidth can be estimated as the product of line size divided by the memory’s interleaving factor, and the memory’s access speed.

**Associativity:** Cache lines are grouped into sets containing one or more lines. The number of lines per set is the associativity of the cache. A cache with associativity of one is termed a direct mapped cache. A fully associative cache has an associativity of $\frac{\text{cache size}}{\text{line size}}$. Caches whose associativity is between these values are called set associative caches. In a cache with associativity greater than one, a lookup in the cache examines all the lines in the set in parallel.

Set associativity and cache size are closely related on code that has not been transformed to improve locality. Hennessy and Patterson’s 7th rule of thumb –The 2:1 Cache Rule– says: The miss rate of a direct-mapped cache of size $X$ is about the same as a 2-way set-associative cache of size $X/2$ [15].
Together, these three numbers characterize the address map of the cache. Given a memory address $addr$, the set number and offset are given by simple formulas.

$$set\ number = \left\lfloor \frac{addr}{line\ size} \right\rfloor \mod \frac{cache\ size}{line\ size \times associativity}$$

$$offset = addr \mod line\ size$$

As long as line size and $\frac{cache\ size}{associativity}$ are powers of two, these computations are simple bit-extractions.

While these parameters specify the address map, several other properties of a cache help determine its effectiveness.

**Address type:** The cache can be organized around either physical or virtual addresses. Physical caches avoid conflicts between identical virtual addresses in different processes; virtual caches must add some sort of process identifier. A physical cache requires address translation before lookup; a virtual cache does not. Virtual caches retain cross-page locality; physical caches do not since adjacent pages in virtual address order may be disjoint in physical address order.

**Replacement algorithm:** When a referenced item is not in the cache, its line is brought into the cache. This may displace another line mapped into the same set; the mechanism for selecting which line to replace is called the replacement algorithm. Much work has been done on replacement algorithms [15]. Two examples are Random Replacement and the Least Recently Used (LRU) algorithm. Random replacement picks the line at random. LRU discards the line within the set that has been used least recently. In a direct-mapped cache, the replacement algorithm is trivial: only one line can be chosen.

**Write policy:** A write-through cache updates the memory location of a modified value each time it is modified. A write-back cache updates memory only when a line is replaced. (At that point, the line is written to memory if it contains a modified value.) These approaches result in different patterns of traffic between the cache and its surrounding memory. A write-back cache may take slightly longer to fill a line when it replaces a modified line.

While this description is simplistic, it captures most of the aspects of cache organization that relate to compiler-based transformations.

There exist slight variations to the cache schemes we described. For example, in presence of a conflict for a cache location, the datum that has to be stored back to
main memory may be written temporarily in a fast buffer, called write buffer so that the new item is placed in cache without having to wait for the effective memory write of the old datum.

Also, direct-mapped caches may be accompanied with an auxiliary fully associative cache, called victim cache [17]. This smaller cache provides additional cache space for data that is displaced from the primary cache due to misses. This scheme has the advantage of avoiding repeated cache misses due to successive references to a single cache location in a direct-mapped cache. However, the effectiveness of the victim cache is reduced for large primary caches, because of its relatively small size.

Agarwal et al [2] devised another technique for reducing the miss rate of direct-mapped caches. They called such caches column-associative. The technique, derived from hash-rehash caches [1], essentially consists of rehashing a datum address that maps to an occupied cache location, so that it gets mapped dynamically to a different set. Therefore, conflicts are resolved not within a set but within the entire cache. However, only two hash functions are used to maintain the placement algorithm cost low.

2.2.1 Cache Misses

There are three kinds of cache misses.

Compulsory misses. Also called intrinsic misses, cold start misses or first reference misses. These occur when a line is referenced for the first time.

Capacity misses. When the program needs more data to remain in cache than the cache can contain, some cache lines may be displaced from the cache to accommodate other immediately needed data. The displaced items would have to be retrieved again when they are referenced later on during the program execution.

Interferences. A code fragment can reference multiple values that all map into a single set in the cache. If the number of referenced items exceeds the cache's associativity, then we say that the items interfere in the cache. As an example of interference, consider a simple five-point stencil, like the program in Figure 2.2. It has references in three different columns of the array a. To cover the reuse, the cache must retain two complete columns of a from a(2,j) to a(n-1,j+1). If the column size is a multiple of \( \frac{\text{cache size}}{\text{line size} \times \text{associativity}} \), all three columns will map to the same set. With associativity \( \geq 3 \), the loop should generate one cache miss every \( \frac{\text{cache size}}{\text{linesize}} \) iterations. With less associativity, it will miss on every iteration, perhaps as often as five times in a single iteration.
do j = 2, n-1  
  do i = 2, n-1  
    a(i,j) = 0.2 * (a(i-1,j) + a(i,j) + a(i+1,j)  
    + a(i,j-1) + a(i,j+1))  
  enddo  
enddo

Figure 2.2 Stencil computation.

2.3 Reuse

To understand how to transform a program to improve its behavior on a machine with one or more levels of cache, we need a firm understanding of the kinds of locality produced by different program constructs. We say that a variable has locality if it exhibits reuse and is in cache at the time it is reused.

2.3.1 No Reuse

A code fragment may not contain any reuse that can be captured by the cache. For example, it may read values from random memory locations. Since it has no locality, we cannot expect to improve its performance with a cache. In fact, its performance will likely be governed by memory access speed.

2.3.2 Spatial Reuse

When cache lines are longer than a single datum, a code fragment that does not reuse a single value may still benefit from cache. For example, a simple vector addition as in Figure 2.3 benefits because each cache miss brings in data for several iterations. In this loop, if there is no interference between a and b or c, references to a will generate $\frac{n}{\text{line size}}$ cache misses. Because references within the cache are faster than misses, longer line sizes benefit this computation. We call this reuse within a line spatial reuse.
\begin{verbatim}
do i = 1, n
c(i) = a(i) + b(i)
enddo
\end{verbatim}

Figure 2.3 Program exhibiting spatial reuse

### 2.3.3 Temporal Reuse

Typically, values are used more than once in a program. For example, each value is defined before it is referenced. If the uses are sufficiently close in time, the second reference may find the value still in the cache. Many programs exhibit a substantial amount of reuse—that is, each region uses a small set of values and they are used more than once. For example, in matrix-matrix multiply, a single column of one matrix is used with each row of the other matrix.

### 2.3.4 Real Programs

In general, real programs display each kind of behavior. They combine different behaviors inside a single loop. This makes it difficult to infer any detailed information from summary execution statistics like hit ratios and timings. We may know that a program runs faster after a certain transformation, but we have little basis for deciding if further improvement is possible.

### 2.4 Data Dependences

To compute reuse information, we need to know when two statements reference the same memory location, or, for spatial reuse, when they reference the same cache line. Data dependence analysis can derive this information [6, 13, 22]. Also, data dependence constitutes a strong criterion to prove the correctness of a transformed program.

Two references in statements $S_1$ and $S_2$ are said to be data dependent if they refer to the same main memory location. In general, we are concerned with the two closest references in time to the same location; if necessary, we extrapolate by transitivity of the dependence relation. There are four types of dependences (Figure 2.4). Assume that $S_2$ is executed after $S_1$ in the original program. A true dependence exists when the reference in $S_1$ defines the value that is later used in $S_2$. An anti dependence
is the opposite case: the use precedes the definition. Here we must ensure that it is
the old value of the variable that is used in $S1$ before it is redefined in $S2$. When
both references are writes, we are in presence of an output dependence. In this case,
we have to make sure that the correct value is kept in the variable, so that later uses
of it refer to the same value. An input dependence exists when both references are
reads. This dependence does not constitute a constraint on the execution order of
the statements, but it provides useful information for improving data locality.

$$(S1) \quad A = \ldots \quad \ldots = A \quad A = \ldots \quad \ldots = A$$

$$(S2) \quad \ldots = A \quad A = \ldots \quad A = \ldots \quad \ldots = A$$

true \quad \text{anti} \quad \text{output} \quad \text{input}

**Figure 2.4** Data dependences.

We further classify dependences as loop independent or loop carried, based on
whether the dependences occur within a single iteration of their enclosing loops or
across different iterations thereof. In the latter case, the loop carrier is the one
outermost that gives rise to it. In Figure 2.5, the anti dependence between the read
$a(i+3)$ in $S1$ and the definition $a(i)$ in statement $S2$ is loop carried, while the true
dependence between the two references $b(i)$ in $S1$ and $S2$ is loop independent.

```
  do i=1, 10
    b(i) = a(i+3) \quad (S1)
    a(i) = b(i) \quad (S2)
  enddo
```

**Figure 2.5** Loop carried vs. loop independent dependences.

A loop carried dependence is characterized by its distance, which is the relative num-
ber of iterations of the loop carrier that separate the accesses to the same memory
location. For example, in Figure 2.5 the loop carried dependence associated with
variable $a$ has a distance of 3. A distance of zero is assigned to loop independent de-
pendences. Alternatively, a distance that is positive, negative, or zero can be labelled
as direction "<", ">", or "=" respectively.

Dependence information is often represented in the form of a dependence graph,
where a node is a statement and an edge from a statement $(S1)$ to a statement $(S2)$
represents a dependence from \((S1)\) to \((S2)\). An edge is also annotated with the level of the loop carrier (or zero if it is loop independent) and the type of the dependence.

As far as cache performance is concerned, we are mostly interested in dependence analysis associated with array variables, relying on the register allocator to manage keeping the scalar variables in registers between their reuses. Dependences associated with scalar variables are necessary to test the safety of applying certain program transformations only.

2.5 Iteration Spaces

An \(n\)-deep loop nest is often modeled as a polytope of the space \(Z^n\) bounded by the loop bounds [12]. An iteration is characterized by a vector \((p_1, p_2, \ldots, p_n)\), where \(p_i\) is the iteration count associated with the loop at level \(i\). Given this definition, a dependence distance vector is also a vector of \(Z^n\) where the components of the vector represent the dependence distance at the corresponding level. Similarly, a dependence direction vector is an \(n\)-tuple such that each component is one of the symbols "\(<"", "\>"", "\=\)" representing the direction of the dependence at the corresponding loop level.

2.6 Transformations

A cache can improve performance in three different ways:

- **spatial reuse**: Architects use longer line sizes to trade off bandwidth against latency. This rewards accessing the data consecutively. Spatial reuse is the result of accessing data residing in the same cache line.

- **temporal reuse**: Temporal reuse is true repeated access to a single value. If the data can be kept in cache, average access time drops as subsequent references hit the cache. Ideally, the compiler should transform the code to ensure that the value remains in the cache until its last "local" reuse.

- **latency hiding**: An alternative way to lower average access time requires direct support in the instruction set. If the architecture provides advisory prefetch and flush instructions, the compiler can use them to hide latency by moving data into the cache in parallel with useful computation. The program avoids the cache miss, saving \(T_m\) cycles, whenever the prefetch has data in cache without
discarding some other necessary data item. This is one way to avoid compulsory misses.

To improve performance, the compiler must capitalize on one of these three properties. In this section, we present the program transformations that are relevant to this thesis.

2.6.1 Loop Fusion

\[
\begin{align*}
do & \ i = 1, \ N \\
\text{body}1(i) & \\
\text{endo} & \\
do & \ j = 1, \ N \\
\text{body}2(j) & \\
\text{endo} &
\end{align*}
\]

\[
\begin{align*}
do & \ i = 1, N \\
\text{body}1(i) & \\
\text{endo} & \\
do & \ j = 1, N \\
\text{body}2(j) & \\
\text{endo} &
\end{align*}
\]

Figure 2.6 Loop fusion.

Loop fusion merges two loops; see Figure 2.6 for an example. This transformation is safe if and only if no loop independent dependence from the first loop to the second becomes loop carried [26]. For an example of a program that cannot be fused legally, see Figure 2.7. In this example, the reference in (S2) reads a value at a(i) that was last altered in (S1) for i ranging from 2 to N – 1. If we fuse the two loops, the values of a read at (S2) would be the old values before alteration at (S1). This may change the program’s behavior.

A fusion can create locality by converting inter-loop reuse into intra-loop reuse. It creates opportunity for more reuse. However, fusion can also increase the memory space required between iterations, leading to interferences or cache overflows. The compiler must resolve this tradeoff.

2.6.2 Loop Distribution

Loop distribution is the inverse of loop fusion [21]. It splits the body of one loop and distributes around it the loop control sub-statement, as in Figure 2.8. Loop distribution is safe when there is no loop carried dependence from the second body to the first body before distribution. In terms of locality effect, loop distribution can
\begin{verbatim}
    do i = 1, N
        a(i) =...
    enddo
    do i = 1, N
        ... = a(i-1)
    enddo
\end{verbatim}

Figure 2.7 Program that cannot be fused safely

be profitable if it separates references to two items that interfere. In particular, if
the interfering references end up in different loops, it can drop interference misses. A
trade off analogous to that of loop fusion has to be taken.

Besides its usage as a transformation to improve locality, loop distribution also
has some pragmatic effects. In many program transformation frameworks, it is used
as a means to simplify the structure of a loop. For instance, many transformation
techniques require that the loop nest be perfectly nested, i.e. nests without conditional
statements or consecutive loops in them. In presence of a loop nest which does not
satisfy this requirement, they rely on loop distribution to simplify it [8, 28].

\begin{verbatim}
    do i = 1, N
        body1(i)
        body2(i)
    enddo
    do i = 1, N
        body1(i)
        enddo
        do i = 1, N
            body2(i)
        enddo
\end{verbatim}

Figure 2.8 Loop distribution.

2.6.3 Loop Permutation

Permutation is a generalization of interchange. Interchange applies on two loops.
permutation is the result of some sequence of legal interchanges. We illustrate this
transformation with an example (Figure 2.9). Assuming a column-major ordering of
data, the original loop will access elements of array a successively from one column of
the matrix to the next. Thus, if a is larger than the cache, every reference may result in a miss. After interchanging the loops, however, the accesses become consecutive in memory and fewer cache misses will occur assuming multiple word cache lines\(^4\). Loop interchange is legal if and only if the interchange does not place a direction “>” in the direction vector before any “<” [3].

\[
\begin{align*}
\text{do } i &= 1, N \\
\text{do } j &= 1, N \\
a(i,j) &= i-j \\
\text{endo}
\end{align*}
\]

\[
\begin{align*}
\text{do } j &= 1, N \\
\text{do } i &= 1, N \\
a(i,j) &= i-j \\
\text{endo}
\end{align*}
\]

\[\text{endo}\]

\[\text{endo}\]

**Figure 2.9** Loop permutation.

2.6.4 Loop Skewing

Loop skewing eliminates dependences that prevent interchange (and parallelization) by changing the iteration space [24]. Figure 2.10 shows a skewing of loop \(i\) with respect to \(j\); \(i\) has been added to the bounds of the \(j\) loop and then subtracted from the subscript where \(j\) appears. This has the effect of transforming the iteration space from a rectangle to a skewed parallelogram. Loop skewing does not change the execution order of the statements therefore it is always legal.

\[
\begin{align*}
\text{do } i &= 1, N \\
\text{do } j &= 1, M \\
\text{body}(i,j) \\
\text{endo}
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= 1, N \\
\text{do } j &= i+1, i+M \\
\text{body}(i,j-i) \\
\text{endo}
\end{align*}
\]

\[\text{endo}\]

\[\text{endo}\]

**Figure 2.10** Skew of loop \(j\) with respect to loop \(i\).

\(^4\)Also, fewer DRAM page misses will occur if the columns were “very” large.
2.6.5 Iteration Space Tiling

Iteration space tiling, also called blocking, is a combination of skewing, strip mining and interchange [24, 27]. The application of the skewing transformation depends on the existence of dependences preventing the interchange. For example, in Figure 2.11, we show the original program followed by strip mining followed by a permutation of the stripped loops.

\[
\begin{align*}
\text{do } & i=1, N, S \\
\text{do } & j=1, M \\
\text{body}(i, j) \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } & i=1, N, S \\
\text{do } & j=1, M \\
\text{body}(i, j) \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

Figure 2.11 Strip mine followed by interchange.

Strip mining is not a reordering transformation, therefore, it never changes the meaning of a program. It is a preliminary step that precedes another reordering transformation, like interchange in this case. It partitions the computation into strips of a given number of iterations, S in the example. After interchange, the iteration space is tiled: the outer loop —called the tile loop— steps through the tiles of the iteration space, and the inner loop —called the element loop— advances within a tile. Overall, the traversed iteration space is the same in the tiled and non-tiled versions of the loop nest. In terms of data locality, tiling can subdivide the data space needed by the new inner loops of the tile and hence make their reuse more apt to be converted into a hit. In other words, tiling can eliminate capacity misses.

For example, in matrix multiplication \( A=B\times C \) (Figure 2.12) each element of \( B \) is used a number of times equal to the width of \( C \), and each element of \( C \) is used a number of times equal to the height of \( B \). However, if the matrices are large relatively to the cache size, the reused elements are likely to be flushed out of caches. For instance, matrix \( B \) is totally reused between iterations of the \( i \) loop, as well as columns of \( C \). However, the cache may be too small or its associativity too little to take advantage of it.

The blocked version in Figure 2.13 makes a better reuse of the data. Tiling the iteration space can decrease the volume of reused data between iterations of the \( i \)
do i=1,N
do j=1,M
  do k=1,N
    A(k,i)=A(k,i)+B(k,j)*C(j,i)
  enddo
enddo
enddo

Figure 2.12 Matrix multiply and the data that are reused in one iteration of the outer loop.

loop. In the matrix multiply example, a block of matrix B of size $b \times b$ only and a column of C of length $b$ only need to remain in cache. The value of $b$ is to be determined in a way that makes a maximal use of the cache (to reduce the number of iterations of the tiled loop) and cause a minimal number of conflicts between iterations of the loop carrying the reuse (loop $i$ in this case). Note that this is only possible in the case of a LRU replacement policy. A random replacement algorithm can destroy the benefits of this transformation.

do js=1,M,b
do ks=1,N,b
do i=1,N
  do j=js,min(js+b-1,N)
    do k=ks,min(ks+b-1,N)
      A(k,i)=A(k,i)+B(k,j)*C(j,i)
    enddo
  enddo
enddo
enddo
enddo
enddo

Figure 2.13 Blocked matrix multiplication and the data that it uses in one iteration of the $i$ loop.
2.6.6 Copying

Copying non-contiguous data into contiguous blocks of the memory can avoid interferences that are due to a lack of cache associativity. Copying in scalar machines avoids interferences by introducing an explicit loop-specific address map under the program's control. It is profitable only in the presence of data reuse. It can also make a better use of long cache lines. Copying simulates moving data in blocks to local memories on distributed memory architectures to minimize non-local memory accesses [5, 11]. It is similar to the vector \textit{gather} and \textit{scatter} instructions made popular on vector machines in the 1970's and 1980's. Chapter 4 addresses copying in greater detail.
Chapter 3

Literature Survey

The previous chapter described the basic transformations available for improving data locality. It did not, however, address the issue of how to decide when and where to apply them. This chapter looks at recent work on decision procedures for locality improvement.

3.1 Unimodular transformations

A unimodular matrix is an integral square matrix with a determinant equal to 1 or -1. Unimodular transformations consist of several loop transformations that can be obtained by an application of unimodular matrices to the iteration space. They were first introduced by Banerjee [7].

As an example, to interchange two loops composing a nest, apply unimodular matrix $M$ to the vectorized iteration space represented by vector $(i, j)^T$:

$$
M = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix}
$$

A principal advantage comes from their being closed under multiplication and inversion. This allows for a compact representation of a sequence of transformations. The unimodular transformations include loop permutation, loop skewing, and loop reversal.

- Loop permutation consists of finding a loop ordering for which the fewest references result into misses. Reuse of cache lines (spatial reuse) is an important component of the problem.

- In the context of optimizing for locality, loop skewing is used to make a loop nest fully permutable.

- Loop reversal allows the shortening of dependences, possibly making a loop-carried dependence independent.
However, the limitation to square transformation matrices restricts to mapping n-dimensional iteration spaces to n-dimensional spaces. That is, transformations that change the dimensionality of the iteration space, such as strip mining, can not be performed within this framework.

### 3.1.1 Lam and Wolf

According to Wolf, the data-locality optimization problem for a given iteration space with

1. a set of dependence vectors or distance vectors, and

2. uniformly generated reference sets

consists of finding the unimodular and/or tiling transformation, subject to data dependences, that minimizes the number of memory accesses per iteration \([24]\).

Two references \(a(f(\tilde{i}))\) and \(a(g(\tilde{j}))\) are uniformly generated if

\[
\begin{align*}
    f(\tilde{i}) &= h(\tilde{i}) + C_f \\
    g(\tilde{j}) &= h(\tilde{j}) + C_g
\end{align*}
\]

where \(h\) is a linear transformation and \(C_f\) and \(C_g\) are constant vectors. This concept was first proposed by Gannon \textit{et al.} in \([12]\).

Lam and Wolf evaluate the locality available within a loop nest using the metric of number of memory accesses per iteration of the innermost loop \([25]\).

The biggest restriction of this technique is that the dependences must be uniformly generated \([12]\). Within this scope, their work is the most complete work that solves the problem of data locality in a structured way. Their algorithms account for long lines and set associativity.

Wolf also presents a method to deal with non-perfectly nested loops \([24]\). He proposes the application of the \textit{non-basic-to-basic-loop transformation} to render a non-perfectly nested loop nest eligible for unimodular transforms. His approach is more general than loop distribution: if loop distribution can produce a perfectly nested set of loops, then the technique he proposes can perform the same transformation on the nest directly.

He also assesses the legality requirements for his transformation to apply. One drawback of his transformation is that it inserts if guards in the innermost loop. However, he tries to push them back out whenever possible.
This technique was not implemented in the SUIF compiler — the compiler designed at Stanford University that implements Wolfe’s techniques — at the time he published his dissertation. For non-perfectly nested loops, the SUIF compiler is limited to skewing only. Skewing is safe even with non-perfectly nested loops.

3.1.2 Sarkar and Thekkath

Sarkar and Thekkath [23] introduce a framework for iteration-reordering loop transformations, including a set of unimodular transformations. This scheme is for representing and applying transformations and not for deciding where and when to apply them. The framework consists of a number of transformation templates, each of which includes a set of rules for mapping dependence vectors, a set of rules for mapping loop bound expressions, and a set of rules for creating new initialization statements. The framework is closed under these transformations. Each transformation requires a perfect loop nest and produces another perfect loop nest that could also be directly subject to a new transformation. They can handle loops whose bounds are linear in the induction variables, as long as the increments are constant (this set includes triangular and trapezoidal loops).

One major characteristic of this work, is that it does not help with the decision process. For instance, it does not decide which transformation to apply and with which parameters. Other techniques are also supported in this framework (See sections 3.2.5 and 3.3.2).

3.2 Iteration Space Tiling

3.2.1 Wolfe

Wolfe addresses the issues involved in tiling iteration spaces for locality (as well as for parallelism) [27]. More specifically, he shows how to perform tiling for rectangular and triangular iteration spaces. Unfortunately, he does not give an algorithm. He determines the tile size such that the footprint of a loop, i.e. the amount of memory used by the loop is less than the cache size. He recognizes that large cache line sizes are problematic in evaluating the footprint of a variable due to non-unit strides and misalignment. He does not address set-associativity. Finally, he shows that finding an optimal loop ordering is $O(d!)$, where $d$ is the loop nesting depth.
3.2.2 Porterfield

Porterfield relies on uniformly generated dependences to estimate the overflow iteration of a loop [21]. The overflow iteration of a loop is the first iteration causing cache misses. This requires knowledge about how much memory is needed. To that end, he distinguishes between memory accessed on the first iteration of a loop, and the additional memory needed on later iterations of the loop, i.e. how much reuse there can be. This is computed by examining the dependence edges in the dependence graph and the position of their endpoints in the given loop nest. Computing the overflow iteration for every loop takes $O(nD)$ where $n$ is the maximum loop nesting depth and $D$ is the number of dependences in the loop.

Strip mine and interchange (tiling, blocking) is effective if the inner-loop carried dependences are all hits and dependences carried by the outer loop (of the perfectly nested loop pair) are all misses. This is translated in terms of the overflow iteration as

$$Ov(inner) > 0 \quad \text{and} \quad Ov(outer) = 0$$

where $Ov(l)$ represents the overflow iteration of loop $l$.

When applying strip mine and interchange, the strip size is chosen so that the overflow iteration of the new "middle" loop is larger than any dependence distance carried by the inner loop. His algorithm handles symbolic loop bounds by adding a three-line prologue (in FORTRAN) for computing the strip mine width at run time.

*Unroll and jam* is another form of tiling. The difference between blocking and unroll/jam is that the former moves inner dependences outward to transform them into hits while the latter tries to move outer dependences to the inner loop and transforms them into hits. Porterfield gives an inequality analogous to that of strip mine and interchange in order to find the best unrolling factor.

In cases where tiling is illegal because of fusion- or interchange-preventing dependences, Porterfield uses skewing and peeling in combination with strip mine/interchange and unroll/jam.

Porterfield also proposes an algorithm to choose the proper blocking transformation from strip mine/interchange and unroll/jam. He does this by computing the number of misses under each transformation given the optimal blocking factors. The algorithm requires that all distances of the outer-loop carried dependences be known.

His work assumes a fully associative cache with a one-word cache line.
3.2.3 Lam and Wolf

Lam and Wolf [25] give an algorithm for finding a near optimal loop reordering as well as a method for determining a block size. They design a model to compute the miss rate of the reuse on a variable \( v \) and find the maximum block size that minimizes the miss rate for \( v \). Sources of misses are self interferences (interferences between elements of a same array variable) and cross interferences (interferences between different variables). The hard part is to compute the self interferences. However, it is possible to find the largest block size for \( v \) that reduces self interferences to 0 with an \textit{ad hoc} method. As for cross interferences, they use a probabilistic approach based on the fraction of cache that another variable needs in one iteration of the loop being studied. They then compare the resulting block size with the one computed using the model and take the smaller of the two as the value for the effective block size.

They show how to account for unit, non-unit constant, and varying strides as well as long cache lines. In the presence of a set-associative cache they use a simple heuristic to compute the "best block size." They recommend additional optimizations such as \textit{block copying} to accompany \textit{tiling}, but give no algorithm for its application.

3.2.4 Carr

In his thesis [8], Carr introduces \textit{index-set splitting}, a technique that he uses to tile complex iteration spaces automatically. He is then able to block trapezoidal and rhomboidal iteration spaces. Index-set splitting is also used when recurrences do not allow blocking the whole iteration space but only a portion of it. Another technique, which he calls \textit{IF-inpection}, is used to block loop nests with control flow. Finding tile sizes was, however, beyond the scope of the dissertation.

Carr observed cases (such as LU decomposition with partial pivoting) where dependence information is not sufficient to detect the blockability of a nest. For these cases, the compiler might need pattern-recognition to find row permutations and whole-column updates for instance.

3.2.5 Sarkar and Thekkath

The framework of Sarkar and Thekkath [23] (see also section 3.1.2) supports blocking [23]. The precondition for applying this transformation is that the loop bounds be affine functions of the induction variables and that the step function be constant.
Moreover, the authors say that \texttt{min} and \texttt{max} functions in the loop bound expressions can be problematic. Also, the template requires that it be given the block size.

### 3.3 Loop Permutation for Stride

Although loop permutation can fit in the set of unimodular transformations when the dependences are uniformly generated, we study it separately here as it can also be achieved within a different framework.

Loop permutation consists of reordering a loop nest in order to take advantage of the largest number of reuse on the array variables.

#### 3.3.1 Carr

To measure the memory performance of a loop nest, Carr uses a model divided in two parts [8]. One part models the ability of cache to retain values between reuse and one part measures the cost associated with each memory reference in the innermost loop. The former uses dependence analysis, the latter uses the system's characteristics such as cache line size, hit time and miss penalty.

Carr assumes that no interference occurs in the inner loop and that all outer loop references result in misses. He then proceeds by trying every loop as innermost (providing the interchange is legal, which he assesses using McKinley's legality test [20]) and computing the memory costs of its references. The final loop order is such that cheaper loops are placed deeper in the nest. Note that this is achieved in $O(d)$, where $d$ is the depth of the loop nest. If the nest is non-perfectly nested, he distributes it when it is safe and then performs the interchange.

#### 3.3.2 Sarkar \textit{et al.}

One of the templates of the framework [23], \textit{ReversePermute}, allows permutation and reversal of loops. It basically tests for legality and rewrites the loop nest following rules for transforming loop bounds and dependences.

In another work [10], Ferrante, Sarkar, and Thrash choose the best loop ordering by finding an upper bound on the number of distinct lines accessed for each array given all possible orders. Their method is based on Banerjee's inequality [6] for determining the lower and upper bounds of a linear subscript expression. They are able to predict the cases where their evaluation will underestimate misses due to interferences, but
they cannot avoid it beforehand, i.e. at the time of counting the necessary cache lines. They suggest padding the arrays, if the semantics of the language permit it, or copying. They do not, however, give details of how to do this.

3.3.3 Kennedy and McKinley

Kennedy and McKinley adopt an approach based on dependence analysis as well [18]. Their algorithm determines the cost of a loop if it were innermost by estimating the number of cache lines that it would access across the iteration space. A few simplifications are made: the references of the outer loops are assumed to be misses, non-unit stride accesses are assumed to carry no reuse even if the dependence distance separating the reference is less than the cache line size, and finally self interferences are ignored in the innermost loop. Also, the cache is fully associative in their model.

The algorithm MemoryOrder leads to a loop ordering such that the inner loops access fewer cache lines. Then, if the permutation that leads to that order cannot be achieved for legality constraints, a "nearby" permutation is performed. The algorithm that finds this permutation is guaranteed to place a loop innermost if there exists a legal permutation where the loop is innermost. It takes $\max(D, n^2)$ time in the worst case, where $n$ is the loop nest depth and $D$ is the number of dependences.

3.3.4 Porterfield

Porterfield's study of the profitability of loop interchange [21] considered the overflow iteration as the only indicator of profitability since he assumed one-word cache lines (no room for spatial reuse). He can find a best ordering between two loops based on which loop carries reuse and an estimate of the amount of memory needed between two iterations when each loop is placed at the inner level. His algorithm takes $O(D)$ where $D$ is the number of dependences. He does not show how to extend the algorithm for larger loop nests.

3.3.5 Wolf

In order to find the legal orders, Wolf [24] starts by detecting those loops that can be placed outermost because they carry no reuse, or that must be placed outermost because it is illegal to move them. This prunes the set of loops to be searched for a best ordering. The remaining set of loops is made fully permutable using the Skew-Reverse-Permute (SRP) transformation (an algorithm that uses unimodular transformations
to make a loop nest fully permutable). Each loop is considered in the innermost level and the number of memory references per iteration is evaluated. The transformation that leads to the best cost is applied\(^5\). The algorithm is exponential in the number of loops.

### 3.4 Loop Fusion

In the context of data locality, loop fusion is used to bring closer together references that exhibit temporal reuse between two loops so that the references become hits, or at least the reference in the originally second loop becomes a hit. It is also an opportunity to transform temporal reuse into spatial reuse as in Figure 3.1 where fusion is applied twice. Fusion can also be counter-productive if it increases the number of memory references per iteration in such a way that it makes the cache thrash.

```plaintext
   do j = 1, M
      do i = 1, N
         a(i,j) = b(i,j) * c(j)
      enddo
   enddo
   do j = 1, M
      do i = 1, N
         d(i,j) = a(i,j) + b(i+1,j)
      enddo
   enddo

   c After two-level fusion
   do j = 1, M
      do i = 1, N
         a(i,j) = b(i,j) * c(j)
         d(i,j) = a(i,j) + b(i+1,j)
      enddo
   enddo

Figure 3.1 Fusion for spatial and temporal reuse
```

\(^5\)The reordering transformation is actually used in conjunction with tiling in the algorithm for improving data locality
Porterfield [21] estimates the profitability of fusion by counting the misses that become hits after fusion, and the hits that become misses. The latter case occurs when two references reusing the same line are made too far apart by the introduction of additional memory references in between them after fusion (the former case is an occurrence of the inverse phenomenon). The profitability estimates are once again based on the overflow iteration and the distances associated with the dependences. But again, his model for memory performance assumes a fully associative cache with one-word lines.

Loop fusion can also be used in order to expose parallelism. Hall, Kennedy, and McKinley [14] propose a method that extracts loops from within called procedures and later fuses them or interchanges them to improve performance.

\[
\begin{align*}
    \text{do } i &= 1, N \\
    \text{do } j &= 1, M \\
    \quad a(i,j) &= b(i,j) + c(i,j) \\
    \text{enddo} \\
    \text{do } j &= 1, M \\
    \quad b(j,i) &= 1/3 \times (b(j-1,i) + b(j,i) + b(j+1,i)) \\
    \text{enddo} \\
\end{align*}
\]

\[
\begin{align*}
    \text{do } j &= 1, M \\
    \text{do } i &= 1, N \\
    \quad a(i,j) &= b(i,j) + c(i,j) \\
    \text{enddo} \\
    \text{do } i &= 1, N \\
    \text{do } j &= 1, M \\
    \quad b(j,i) &= 1/3 \times (b(j-1,i) + b(j,i) + b(j+1,i)) \\
    \text{enddo} \\
\end{align*}
\]

**Figure 3.2** Distribution followed by interchange

### 3.5 Loop Distribution

In the context of data locality optimization, loop distribution is used in three ways. One is to reduce the working set so that references that exhibit reuse, which might result in misses because of interferences, can become hits [21]. The second is to make loops perfectly nested and apply the techniques that require this property [8, 28]. The third way is to distribute when a loop nest has conflicting interests in one transformation or another [8]. For instance, in Figure 3.2, the first statement favors an interchange of the loops, and the second favors the original loop order. By
distributing the outer loop, we can interchange the first resulting loop nest without interfering with the second. The analysis in the example assumes column-major order.

3.6 Copying

Copy optimization consists of copying non-contiguous reused data into a contiguous area of memory in order to avoid self-interferences due to long stride accesses. Cross-interferences (interferences between different variables) can also be eliminated with the appropriate mapping. Copying is sometimes not safe, for example, when the data to be copied is aliased, or in the presence of dependences between the copied and the non-copied data. The profitability of copying is highly dependent on the ratio of the copying cost to the amount of effective computation and reuse. To diminish this cost, one can reduce the number of misses induced by the copying itself by taking advantage of long cache lines when applicable. Some machines, like the Intel i860, provide instructions to move data to and from registers bypassing the cache (non-allocating loads and stores).

3.6.1 Lam and Wolf

In their model for counting the number of misses per iteration for a loop, Wolf and Lam [25, 24] set the self-interference term to zero when using copying and based on that they decide the best blocking factor. Copying is not implemented in the SUIF compiler but Wolf claims that it is a straightforward task for the compiler.

3.6.2 Gallivan, Gannon, Jalby

Gallivan, Gannon, and Jalby present a preliminary study on how to move array sections from global memory to local memories, mainly for distributed and hierarchical memory parallel machines [11]. They put the problem into a mathematical context in the following way. Let \( C \) be the iteration space defining a loop nest of depth \( k \) (\( C \subseteq Z^k \)). And let \( a(h(i) + v) \) be a memory reference at statement \( S \) of the inner loop, where \( h \) is a linear function \( h: Z^k \to Z^d \) and \( v \) is a vector of \( Z^d \). Let region \( R \) be the set of elements of \( a \) referenced by \( S \): \( R = h(C) + v \). The problem consists of finding the geometric structure of region \( R \), the number of elements that the region contains, a simpler geometric structure that encloses \( R \), and the code that moves the region \( R \) between the levels of the memory hierarchy.
They assume for their implementation scheme that a system function called \textit{block-fetch} will perform the copying from the global memory to the local memory. It takes an address, a linear indexing expression and a sequence of ranges for unbounded variables in the expression. A second system function (\textit{block-replace}) does the inverse. While these functions are not available in most microprocessor memory systems, they could be implemented artificially and assumed to have a similar effect. Furthermore, caches are generally not manageable by the compiler, and the placement and replacement policies are hard to predict accurately at compile time.

The method they use is based on linear algebra. They find the image vector space of function \( h \) and then enclose it in a smaller region, so as to avoid duplicate copying operations on the same elements of the copied array.

### 3.6.3 PGI

The Portland Group's FORTRAN and C compilers for the Intel i860 machines use a form of copying called \textit{strip mining and streaming} [19]. They use this technique in order to avoid successive DRAM page misses. They use the non-allocating \texttt{pfld} instruction to load a portion of the data into registers. These are then written into an array that is assumed to be used frequently enough to remain in cache. The computation is actually done on this array. Strip mining serves the purpose of subdividing the data into cache length portions. The strip length is chosen such that the amount of streamed data accessed in the strip loop approaches the cache size. This is computed in accordance to the fact that they are able to bypass the cache with the \texttt{pfld} instructions. When there is reuse of the data in an outer loop, interchange is further invoked.
Chapter 4

Copy Optimizations

4.1 Introduction

Copy optimization consists of copying data from one array to another in which the used data are placed in contiguous memory locations. This serves the purpose of avoiding interference due to conflict misses. In this chapter, we show examples of how effective this technique can be and examine the difficulties of automating it. Then, we propose a technique called tile and copy as an alternative solution.

4.2 Copy Optimization

In order to evaluate the profitability of copy optimization, we ran several experiments on the IBM RS/6000. The processor's cache is 64K bytes large, 4-way set associative, and line size is 128 bytes. The replacement policy follows the least recently used scheme. The write policy is write-back.

4.2.1 Vector Add

In this experiment, we are given a 5-column matrix. We vary the column length from 1024 to 17000 double-precision floating point elements. This has the effect of generating all possible relative alignments. The computation consists of adding the first 1024 elements of the 5 columns into another vector \( x \). To get an idea of when copying starts to become profitable, we varied the number of additions from 1 to 21 by increments of 4. This way, we get different amounts of computation for the same number of data transfers between the memory levels. The time measured corresponds to the call to the routine showed in Figure 4.1. This figure plots only a small representative portion of the experiment we performed. It shows the results that were obtained when the vector length varies from 14000 to 17000 elements.

Given that double-precision words are 8 bytes long, one vector of 2048 elements can fill one fourth of the cache, in such a way that elements of the same row have the
subroutine vadd (a, n, x, t)
double precision a(n, 5), x(1024)
integer n, t
integer i, j
do j = 1, 5
   do i = 1, 1024
      a(i,j) = 2.0
   enddo
enddo

do k =1, t
   do i = 1, 1024
      x(i) = x(i) + a(i,1) + a(i,2) + a(i,3) + a(i,4) + a(i,5)
   enddo
enddo

Figure 4.1 Vector add

same offset with respect to each bank of the cache — i.e., they belong to the same set. Hence, in addition to the cross interferences that may happen between arrays a and x, self interferences will occur at each reference to array a, since we access more than four elements of the same row on each iteration of the inner loop. Hence, the limited associativity results in pathological behavior — one miss per memory reference to array a. So, even though the computation could take advantage of significant spatial reuse (reducing memory accesses per iteration by a factor of 8), the self interferences make this impossible. That is, the cache has a counter-productive effect on the performance of the program, since a load with cache miss is more expensive than a register load.

To remove the self interferences, one can change the data layout in memory before loading it into cache. One other solution is to break the computation in two, and accumulate the addition of the first three vectors in x in a first loop, and continue with the remaining two vectors in a second loop. This would get rid of the interferences due to a limited associativity. However, this technique is highly dependent on the nature of the operations used and, in general, is prohibited by data dependencies. Iteration space tiling does not help here; it would work only if the cache had more associativity. In fact, its application makes sense only in cases where the volume
of data can overflow the cache or when the interferences take place across different iterations. In this study, we examine the first suggested method and operate by copying the data needed for the computation into a different array. We then perform the computation using the newly created array. The code after transformation is in Figure 4.2.

```fortran
subroutine addcop (a, b, n, x, t)
double precision a(n,5),b(5,1024),
x(1024)
integer n, t
integer i, j
do j = 1, 5
   do i = 1, 1024
      a(i,j) = 2.0
   enddo
endo
do i = 1, 1024
   do j = 1, 5
      b(j,i) = a(i,j)
   enddo
endo
do k = 1, t
   do i = 1, 1024
      x(i)=x(i)+b(1,i)+b(2,i)+b(3,i)+b(4,i)+b(5,i)
   enddo
endo
```

**Figure 4.2** Vector add after copying

Self interferences cause the worst degradation when the column size of the matrix is a multiple of 2048, but even nearby column sizes (more or less 4 elements) can be problematic. It appears that copying automatically, i.e. independently of the vector length, does not cause a degradation of the performance when the vector lengths are not critical. However, when the vectors are 16K long, we observe a particularly high peak in the running time of the vector add routine. This is due to a conflict in the translation lookaside buffer (TLB). The TLB is a cache that is used to map virtual memory addresses to cache addresses. In the RS/6000, the TLB is a 128 entry 2-way
set associative cache. Since the page size is 4K, in order to have conflicts in the TLB, there have to be at least three different references to virtual memory that are 64*4K bytes apart from each other. In this example, the conflict in the TLB occurs for references $a(i)$, $a(i+2)$ and $a(i+4)$\textsuperscript{6}.

To avoid interference, a simple map consists of transposing the submatrix subject to the computation into a new matrix $b$. Now the accesses at each iteration of the inner loop are made on successive elements of one column of $b$ without interferences. More precisely, 7 times out of 8 on average, the data will already be in cache when they are needed by the CPU because the line size is 8 double precision words.

The speed-ups obtained from copying depend on the number of times the addition is performed. For instance, for a single addition, at the critical points, \textit{i.e.} where the vector length is a multiple of 1024 (except for a length equal to 16K), the execution times are .0088sec and .0037sec for the copied and the original version, respectively. For most other vector lengths, the execution times are 0.0026sec .0013sec, respectively. On the other hand, for 13 additions, at vector length equal 14K, the execution time is 0.0137sec, for the copied version against 0.0375sec, for the original version. At any non-critical length, the execution times are comparable for both versions of the code.

The major trick that was used in this example is the mapping function. \textit{i.e.} the idea of transposing the matrix. Is it realistic to ask from a compiler to find such maps? This seems quite improbable. Are there other non-optimal and yet satisfactory mappings which a compiler can reasonably find? These are the questions that we try to answer in section 4.4.

\subsection*{4.2.2 5-Spot Stencil}

Our second experiment consisted of running an unoptimized abstracted version of the SOR method (Figure 4.3), and then the "copy-optimized" version of it. We varied the column length —parameter $n$ in the algorithms— from 4000 to 6000 elements. We also varied the time step of the relaxation —parameter $t$— from 1 to 21 by increments of 4. We initialized the matrix such that the compiler optimizations do not interfere with our performance measurements.

The performance of this routine was irregular and hardly predictable. While the relaxation was always made on the same 100-by-100 element block of the matrix —\textit{i.e.}
The amount of computation was constant for all matrix sizes—we observed a large variation in the running time of the algorithm across different matrix sizes. Also, the performance due to the data layout degraded more often and to a larger degree than with the vector add routine (excluding the TLB conflicts observed with the vector add).

The optimization of the program consisted of simply copying the data necessary to do the computation into a smaller array—small enough, in this case, that a major part of the interferences were avoided (Figure 4.4). It is hard to predict in general how large the new array should be. The goal is to bring the reused data closer together in order to diminish cache interferences. After the computation is done, it is necessary to copy the data back to the original array since the matrix b was modified by the program.

It appears that for time steps greater than one, the optimized version runs faster for all matrix sizes. For example, when the time step is 5, at vector length 4092, the running time of the copied code was 3.52 times faster than the original code, while at vector length 4374, the speed-up was 3.06. Again, the larger the time step, the larger
Figure 4.4 5-spot stencil after copy optimization.
the speed-up. For instance, for a time step of 21, at vector length 4092, the speed-up was 4.23.

The improvement is due to decreased self interference in the copied array. In the original code, given the size of the columns, there is a great chance that whole columns conflict with each other. In fact, for a size equal to 4K elements, the column is 32K bytes long, meaning that all elements of the same row map to the same set and since the inner loop iterates along the rows, each row between row number 3 and row number 97 (reused twice on iterations of the middle loop i) will be flushed out of cache before it is reused. While in the 100-by-100 array, the chances for interference between elements of the same row are reduced to zero. Only capacity misses remain: their effect is small in comparison, since they correspond to references that are reused on the outermost loop.

In this example, we did not change the mapping functions, but we altered the size of the arrays. However, typical SOR computations are made on whole matrices, and often the working set necessary for the stencil is much larger than the cache can hold. In those cases, copying alone is of no help. This suggests that if given some computation involving a large data set, it might be worthwhile at times to subdivide it into smaller chunks that fit into cache, and thus use copying for each of those chunks of computation. The following section addresses this in greater detail.

### 4.3 Combining Tiling and Copying — Motivation

*Tile and copy* is a combination of the tiling and copying transformation techniques. Its purpose is to bring reuse closer together in the course of the computation — through tiling — and to reduce the interferences that might occur due to long stride accesses — through copying. Consider matrix multiplication as an example. We show in Figure 4.5, a tiled matrix multiply. In this program, the code was tiled so that a block of $A$ of size $ts \times ts$ is kept in cache and reused across iterations of loop $j$, and a column of $b$ of length $ts$, reused across iterations of loop $i$ is also expected to be kept in cache.

When the matrices are too large, we cannot tile the iteration space effectively without causing self-interferences on $A$. For instance, consider double precision matrices of sizes $512 \times 512$. In a 4-way set associative 16K cache — like the i860-XP, assuming that the matrix origin is aligned with the cache origin, elements of the same row will map to the same set in cache, and will therefore interfere with each other at least
every four elements in the same row depending on the cache replacement policy. This suggests square block sizes of at most 4×4, which leaves the cache largely unexploited.

\[
\begin{align*}
    &\text{do } \text{ks} = 1, n, ts \\
    &\quad \text{do } \text{is} = 1, n, ts \\
    &\quad\quad \text{do } j = 1, n \\
    &\quad\quad\quad \text{do } i = \text{is}, \min(n, \text{is+ts-1}) \\
    &\quad\quad\quad\quad \text{do } k = \text{ks}, \min(n, \text{ks+ts-1}) \\
    &\quad\quad\quad\quad\quad C(i, j) = C(i, j) + A(i, k) \times B(k, j) \\
    &\quad\quad\quad\text{enddo} \\
    &\quad\quad\text{enddo} \\
    &\quad\text{enddo} \\
    &\text{enddo}
\end{align*}
\]

**Figure 4.5** Tiled matrix multiplication.

If we copy the part of matrix A that is used within a tile into a new array AL of size \(ts \times ts\), such that \(ts \times ts \times \text{element-size} < \text{cache-size} - ts \times \text{element-size}\), then we would have eliminated self-interferences, and reduced the cross-interferences between the new array AL and array B.

We show in Figure 4.6 the tiled-and-copied version of the program. First, we insert the code for copying A into AL, then we operate the matrix multiplication using AL instead of A. Note that we do not have to copy AL back into A since AL is not altered in the computation.

We ran these programs on one processor of the Paragon XP/S computer. This machine has 54 Intel i860-XP processors. Each processor has a 16K 4-way set-associative data cache with lines of 32 bytes and a random replacement policy. We varied the matrix sizes from 100×100 to 600×600 with steps of 10. The matrix elements are 8-byte double precision elements. The results of the executions are shown in Figure 4.7. In the tiled version, we chose the block sizes \(ts\) as follows. Let \(ts = cs/(i \times 8) - 1\) where \(cs\) is the cache size, and \(i\) is the column length of the matrix. To account for the space required for matrix B, we count the difference between the cache size and the space used by A if \(ts\) were the block size. If it is smaller than \(ts\), which is the length of one tiled column of matrix B, we reduce the tile size by one. If the tile size is greater than \(i\), that means the array is small enough to be kept in cache, and we do not block the matrices. In the opposite case, where \(ts\) as computed is less than
do ks = 1, n, ts
  do is = 1, n, ts
c  Copyin in
    do k = ks, min(n, ks+ts-1)
      do i = is, min(n, is+ts-1)
        AL(i-is+1, k-ks+1)=A(i,k)
      enddo
    enddo
c  Computing
    do j = 1, n
      do i = is, min(n, is+ts-1)
        do k = ks, min(n, ks+ts-1)
          C(i,j)=C(i,j)+AL(i-is+1, k-ks+1)*B(k,j)
        enddo
      enddo
    enddo
  enddo
enddo

Figure 4.6 Matrix multiplication after tiling and copying A.

8 (the cache line size), we simply set it to 8 since a tile smaller than the line size is counter-productive.

In the tiled-and-copied version, we let ts = \(\sqrt{cs/8} - 2\). If this number is greater than i (A fits entirely in cache), we reset ts to i, otherwise we leave it unchanged. We declare array AL as an array of size ts \times ts. This choice of ts is small enough to avoid self interferences in AL (the random replacement policy of the cache makes it hard to assert with certitude), and hopefully not cause a large number of cross interferences (we have no way to control that formally since the address mapping of the arrays are unknown at compile-time).

A small peak is observed in the plot of the tiled version 4.7. This occurs when the matrix size is neighboring 512 \times 512 because in that neighborhood, self interferences in A are frequent. In fact, for that length, elements on the same row map to the same set in cache, so interferences are very likely to occur, even for tiles of width equal to 4 (the set-associativity), since the replacement policy is random. We chose to set the minimum tile sizes to 8 because for matrices with comparable sizes (excluding sizes
between 508 and 516), the interferences do not occur for a tile size of 8 making it a better choice than 4.

Also, the curves in both plots look the same; this stems from the fact that the cost of multiplication — \(O(n^3)\) — dominates the cost of the algorithm.

Clearly, this technique is limited in practice by the complexity of the case in study. The main complexity comes from finding the mapping associated with the copying. We feel that simple heuristics, while not optimal, can be performed with little overhead and significant improvement of the overall performance of the computation.

### 4.4 Tile and Copy — An Algorithm

Tile and copy should be seen as part of a more general framework that addresses data locality in general. The reason for this is that, in the course of the algorithm, one should determine whether simpler transformation techniques can achieve the same results. For instance, in many cases, it is possible that tiling by itself, or loop permutation is sufficient to achieve the same or a better result. One other reason is that part of the algorithm consists of tiling the iteration space for locality, which is already a set of program transformation techniques in itself.
This section shows how we can fit tile and copy into such a framework. The next section proposes a simpler and less expensive heuristic that seems to achieve good results.

4.4.1 Estimating the Profitability of Tile and Copy

The profitability of tile and copy depends heavily on memory reuse and the stride of accesses to the referenced arrays in a loop nest.

The first step is to determine whether there is reuse. One way to do this is to count the number of iterations and the number of distinct memory locations that are accessed for each array; this number can be approximated by a method described by Gallivan et al.[11]. If the count of iterations is greater than the number of distinct memory locations, then there is reuse.

The second step is to test whether the reuse can be exploited efficiently through tiling the iteration space only (i.e. without copying). For each array, if the strides of accesses across different iterations —or within the same iteration among different references— are large (such as in cross-column accesses), and if a loop permutation does not reduce the stride of accesses, then tiling and copying is profitable. Typically, only large working sets will necessitate copying after tiling.

4.4.2 Tiling

We can follow Wolf’s algorithm for improving data locality [24] which permutes and tiles the iteration space in order to minimize the number of memory references per iteration of the inner loop. The tile sizes are better chosen later, once it is decided which data is to be copied. In cases where all the arrays are to be copied, one can tile as if the cache were fully associative, given that the objective of the copying is to remove interferences that are due to lack of associativity. For those cases, one can follow Porterfield’s algorithm which tiles according to the overflow iteration [21]. However, for this to work properly, all local arrays must be laid out in a contiguous memory zone and the whole memory used in the element loops be smaller than the cache size.

4.4.3 Copying

This part is based on the works of Ancourt [5].
Definition 4.1 (Polyhedron) A convex polyhedron $P \subseteq \mathbb{R}^n$ is a set of points that satisfy a finite number of linear inequations; i.e.,

$$P = \{ \bar{x} \in \mathbb{R}^n | A\bar{x} \leq \bar{b} \},$$

where $A$ is an $n \times m$ matrix and $\bar{b}$ a vector in $\mathbb{R}^m$.

Definition 4.2 (Z-module) A $Z$-module of $\mathbb{Z}^n$ is defined as the set of linear integer combinations of linearly independent vectors of $\mathbb{Z}^n$.

For an $n \times m$ integer matrix $A$, the $Z$-module is

$$L(A) = \{ y \in \mathbb{Z}^n | y = Ax, x \in \mathbb{Z}^m \}$$

Definition 4.3 (Z-Polyhedron) A $Z$-polyhedron is the set of integer points resulting from the intersection of an integer polyhedron (an integral linear system of inequations) and a $Z$-module.

```
do i =1, 20
  do j =i, 20
    ...
  enddo
enddo
```

Figure 4.8 Loop nest example

$Z$-polyhedrons are used to model iteration spaces. For example, the iteration space in Figure 4.8 can be rewritten as an integral linear system as follows:

\[
\begin{align*}
-i & \leq 1 \\
\quad i & \leq 20 \\
-j & \leq i \\
\quad j & \leq 20
\end{align*}
\]

This integral system is defined by the intersection of, on one hand, the $Z$-module, itself defined by the identity matrix and the base vector

$$\bar{x} = \begin{pmatrix} i \\ j \end{pmatrix},$$
and, on the other hand, the polyhedron defined by $A\bar{x} \leq \bar{b}$, with matrix

$$A = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}$$

and vector $\bar{b} = \begin{pmatrix} -1 \\ 20 \\ -i \\ 20 \end{pmatrix}$.

In the procedure that evaluates the profitability of the transformation, the array candidates for copying are identified based on the reuse and the stride of accesses. For each such array, we have to perform the following steps:

1. Find a new base for the image domain of the access function to the new array, which we will call local (as if it were local to the cache memory). This part also determines the dimensionality of the local array, as it can be less than the original’s.

2. Find the new local array access functions within the computation code.

3. Generate the local array declaration.

4. Generate the code for the data transfer. This consists of computing the iteration space in the new base that will be used to transfer the data between the global and the local arrays.

**Image Domain Base** To find the image domain base we use Hermite’s theorem for matrix decomposition.

**Theorem 4.1 (Hermite)** If $F$ is an $n \times m$ matrix of rank $r$, then there exists unimodular matrices $P, Q$ s.t.

$$PFQ = \begin{pmatrix} L & 0 \\ S & 0 \end{pmatrix} = H$$

where $H$ is the Hermite reduced form matrix associated to $F$ of rank $r$, $L(r \times r)$ a lower triangular matrix, $S((n - r) \times r)$ a matrix, and $P(n \times n)$ a permutation matrix.

So, if the access function to array $T$ is $F^{\bar{j}} + \bar{f}_0$ then it can be rewritten $P^{-1}H(Q^{-1}\bar{j}) + \bar{f}_0$. 
Lemma 4.1  The image domain of $F$ is the same as the image domain of $H$.

This matrix decomposition allows us to find a base and a new iteration space that covers the same data domain accessed by $F$ but in fewer iterations than the original code does, i.e. avoiding multiple copies of the same elements of the array.

```
do i
  do j
    ... T(i+j+3, i+j-1) ...
  enddo
endo
c Copy in
  do t
    TL(t+3) = T(t+3, t-1)
  enddo
c The new computation code goes here
... 
```

Figure 4.9  A simple program where the local array can have fewer dimensions than the original array.

Array Access Functions  The access function to the local array in the new base $\bar{i} = Q^{-1}j$ is in general $\varphi = P^{-1}H\bar{i} + \bar{f}_0$. However, there exist cases where it is possible to prune the number of dimensions of the local array: in Figure 4.9, only one dimension is necessary for the local array. The original code is to the left of the figure, and the copying code is to the right. The following lemma explains how to find a new access function of dimension $r$ in those cases [5].

Lemma 4.2  Let $\varphi = P^{-1}H\bar{i} + \bar{f}_0$ the access function to elements of array $T$ of dimension $n$, and $r < n$ the dimension of the image domain referenced by $T$. By multiplying $\varphi$ to the left by unimodular matrix $Q''$, we obtain a new access function to the same elements of $T$ of dimension $r$. The matrix $Q'$ is the result of the reduced Hermite decomposition of matrix $(P^{-1}H)'$:

$$(P^{-1}H)' = P'^{-1}H'Q'^{-1},$$

Local Array Declarations  Let $\varphi = F\bar{j} + \bar{f}_0$ be the access function to the original array and $A\bar{j} \leq \bar{\alpha}$ the iteration space. In a previous step, when we computed the new access function to the local array, we determined the number of dimensions of the local
array. In this step, we compute the sizes of the local array in each dimension. These depend on the upper and lower bounds taken by vector $\vec{t}$ since the access function is a linear combination thereof. We consider two cases depending on whether or not the dimension of the image domain is equal to the dimension of the iteration space.

Case 1: $\dim(\varphi) = \dim(\vec{j})$: $F$ is reversible, we can then apply the base change $\vec{j} = F^{-1}\vec{t}$. The domain accessed by $F$ is then defined by the inequality $AF^{-1}\vec{t} \leq \vec{\alpha}$. We compute the bounds associated to index $t_i$ of $\vec{t}$ by projecting all variables but $t_i$ on this new domain. For example, suppose we have the program:

```plaintext
do i = 1, 10
  do j = 1, 5
    ... A(i+j,j) ...
  enddo
endo do
```

The iteration space domain is defined by matrices $A$ and vector $\vec{\alpha}$ as follows:

$$A = \begin{pmatrix}
1 & 0 \\
-1 & 0 \\
0 & 1 \\
0 & -1
\end{pmatrix} \quad \text{and} \quad \vec{\alpha} = \begin{pmatrix}
10 \\
-1 \\
5 \\
-1
\end{pmatrix}$$

the access function is

$$F = \begin{pmatrix}
1 & 1 \\
0 & 1
\end{pmatrix} \quad \text{which implies} \quad F^{-1} = \begin{pmatrix}
1 & -1 \\
0 & 1
\end{pmatrix}$$

Now,

$$AF^{-1}\vec{t} \leq \vec{\alpha} \quad \Rightarrow \quad \begin{pmatrix}
1 & -1 \\
-1 & 1 \\
0 & 1 \\
0 & -1
\end{pmatrix} \begin{pmatrix}
t_1 \\
t_2
\end{pmatrix} \leq \begin{pmatrix}
10 \\
-1 \\
5 \\
-1
\end{pmatrix}$$

Which defines the system,

$$\begin{cases}
t_1 - t_2 \leq 10 \\
-t_1 + t_2 \leq -1 \\
t_2 \leq 5 \\
-t_2 \leq -1
\end{cases}$$
By projecting $t_2$ on the first and second inequalities, we obtain the system

$$
\begin{align*}
  t_1 & \leq 15 \\
  -t_1 & \leq -2 \\
  t_2 & \leq 5 \\
  t_2 & \leq -1
\end{align*}
$$

or

$$
\begin{align*}
  2 & \leq t_1 \leq 15 \\
  1 & \leq t_2 \leq 5
\end{align*}
$$

meaning that the local array will have to be declared as $AL(2:15,1:5)$.

Case 2 $\text{dim}(\varphi) < \text{dim}(\vec{f})$: For an exact solution, we use Feautrier's parametric integer programming method [9]. Alternatively, we could use successive projections as in Case 1 at the expense of over-estimating the size of the array.

In cases, there can be opportunities for reducing the size of the local arrays when the subscript functions can be factorized. In the example in Figure 4.10, the local array can be declared as $TL(1:N+M,1:L)$ instead of $TL(2:2N+2M,3:3L)$.

```fortran
  do i = 1,N
    do j = 1,M
      do k = 1,L
        T(2i+2j,3k)...
      enddo
    enddo
  enddo
```

**Figure 4.10** Example of a program where the size of the local array can be pruned.

**Code Generation for the Data Transfers** In this part of the algorithm, we generate the copying code by determining the bounds associated with each loop of the new iteration space. It is based on simplification and projection of the integral linear systems. Initially, the system defines the iteration space in the new base. We first eliminate the extra dimensions, if they exist, using the Fourier-Motzkin method of variable elimination. Then, by successive projections starting from the first variable, we determine the bounds associated with each non-redundant variable of the new base. This procedure actually generates the smallest convex polyhedron containing the image domain of the access function.
Let \( A\vec{j} \leq \vec{\alpha} \) be the Z-polyhedron defining an iteration space. Let \( \varphi \) be the access function to array \( T \), \( \varphi = P^{-1}H(Q^{-1}\vec{j}) + \vec{f}_0 \). The new base \( \vec{i} \) is defined by the equation \( \vec{i} = Q^{-1}\vec{j} \Rightarrow \vec{j} = Q\vec{i} \). Therefore, the iteration space is expressed in the new base by the inequality \( AQ\vec{i} \leq \vec{\alpha} \).

When the image domain has rank \( r \) less than the dimension \( n \) of the iteration space, i.e., \( \text{rank}(H) < \text{dim}(\vec{j}) \), then in order to reduce the number of redundant copy operations between the global and the local arrays, we project the iteration domain on the first \( r \) variables of the base \( \vec{i} \). Irigoin [16] originated the algorithm shown in Figure 4.11[16].

**Example** Let's examine the different steps of the copying algorithm on the example in Figure 4.12.

The matrix \( A \) and the vector \( \vec{\alpha} \) define the polyhedron (iteration space) according to the system \( A\vec{j} \leq \vec{\alpha} \). Their values are:

\[
A = \begin{pmatrix}
-1 & 0 \\
1 & 0 \\
0 & -1 \\
0 & 1
\end{pmatrix}
\quad \text{and} \quad
\vec{\alpha} = \begin{pmatrix}
-1 \\
10 \\
-1 \\
10
\end{pmatrix}.
\]

The access function to \( T \) is defined by the matrix \( F \) and the constant vector \( \vec{f}_0 \) as follows.

\[
F = \begin{pmatrix}
1 & 1 \\
1 & 1
\end{pmatrix},
\quad \vec{f}_0 = \begin{pmatrix}
0 \\
0
\end{pmatrix}.
\]

The decomposition of \( F \) according to Hermite gives

\[
F = \begin{pmatrix}
1 & 1 \\
1 & 1
\end{pmatrix} = P^{-1}HQ^{-1} = \begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix} \begin{pmatrix}
1 & 0 \\
1 & 0
\end{pmatrix} \begin{pmatrix}
1 & 1 \\
0 & 1
\end{pmatrix}
\]

The iteration domain in the new base is defined by the system \( AQ\vec{i} \leq \vec{\alpha} \). Therefore, given that:

\[
Q = \begin{pmatrix}
1 & -1 \\
0 & 1
\end{pmatrix},
\]
Let $AQ\bar{t} \leq \bar{\alpha}$ be the inequality that defines a polyhedron in the new base.

1. The bounds of the induction variable $t_i$ corresponding to the innermost loop are given by the inequality $AQ\bar{t} \leq \bar{\alpha}$ as function of the enclosing loops induction variables.

2. To compute the $(r - i)^{th}$ loop bounds, we project the system w.r.t. induction variable $t_i$. This allows us to express $t_{i+1}$ as a function of the $r - i - 1$ first indices of the loop nest.

3. Project the system w.r.t. the $r - 1$ first variables to obtain the bounds of the outermost loop indexed by $t_r$.

At each step of the algorithm, eliminate the redundant variables according to the following rules:

- For each induction variable, at least two constraints have to be kept, in order to generate the loop bounds corresponding to the variable.

- The redundant constraints corresponding to the innermost loops are eliminated first, so that the bounds are expressed in the simplest manner possible.

**Figure 4.11** Algorithm for computing the bounds of a convex polyhedron.

```plaintext
do i=1,10
  do j=1,10
    T(i+j,i+j) = ...
  enddo
enddo
```

**Figure 4.12** Example
the system is rewritten:

\[
\begin{pmatrix}
-1 & 1 \\
1 & -1 \\
0 & -1 \\
0 & 1
\end{pmatrix} \vec{t} \leq \begin{pmatrix}
-1 \\
10 \\
-1 \\
10
\end{pmatrix},
\]

which defines the system of constraints:

\[
P = \begin{cases}
-t_1 + t_2 & \leq -1 \\
-t_1 - t_2 & \leq 10 \\
-t_2 & \leq -1 \\
t_2 & \leq 10
\end{cases}
\]

Since \( \text{rank}(F) = 1 \) and \( \text{dim}(T) = 2 \), we can reduce the dimension of the local array TL. By projecting the iteration domain on the last variable of the new base, we obtain a new projected polyhedron defining the new iteration space:

\[
P P = \begin{cases}
t_1 & \leq 20 \\
-t_1 & \leq -2
\end{cases}
\]

This new system does not contain constraints on variable \( t_2 \). The bounds of the new polyhedron are here explicit (there is only one variable left).

Since the access function domain is of rank 1, we obtain the new access function to array \( T \) in the new base following Lemma 3.2.

\[
\vec{\varphi} = \begin{pmatrix} t_1 \\ t_1 \end{pmatrix}.
\]

Finally, the data transfer code to the local array and back to the original array are shown in Figure 4.13.

### 4.4.4 Putting it all together

In the previous section, we described an algorithm for generating copy-optimized code for one reference in a loop nest, independently of our original intention to tile a loop nest before copying. If we take this fact into account, we notice that the method we described to generate the local array declaration is not appropriate because it requires that each tile be associated with one particular declaration of the array to
be copied. This is not acceptable because a loop nest can be tiled into a considerably large number of tiles, or worse, into a number of tiles unknown at compile time.

The solution we propose to overcome this "engineering problem" is to declare one single local array which we will use for all the tiles with different access functions. This heuristic will work in most of the cases because we only consider convex tiles, that is, tiles with bounds expressions that are affine functions of enclosing induction variables. Non-convex tiles have their bounds defined with integer divisions and are used in cases where an exact coverage of the modified area of a copied array is needed [4]. We ignore this problem in our study since we are concerned with scalar machines only.

We illustrate by an example (Figure 4.14) how to make use of the same array for all tiles. The array in this case is declared as AL(1:TS).

4.4.5 Cost

The overall cost is dominated by the tiling part and the copying part of the algorithm. Both can have a time complexity exponential in the loop nest depth. Tiling is of complexity $O(d!)$ if $d$ is the loop nest depth. The Fourier-Motzkin elimination algorithm, used in the copying procedure to find the iteration space that scans the image domain of one array access function, is exponential in $d$. It has to be invoked for each set of references with distinct access functions, or, more precisely, for each distinct set of uniformly generated references.
4.4.6 Safety

Tile and copy is safe if both the tiling transformation and the copying are safe separately. Tiling as described by Wolf [24] is legal when the loop nest is fully permutable. Copying requires that no dependences exist between the copied data and the non copied data, and that the copied data not be aliased within the loop nest.

4.5 Tile and Copy — A Heuristic

In light of the cost of the algorithm presented in the previous section, we propose in this section a heuristic for achieving tile-and-copy with a restricted domain of applicability but at a substantially lower cost.

We start by recalling the common cases that constitute an opportunity for the application of tile and copy. Those are perfect loop nests that access long columns of arrays of dimension 2 or more with a “large” reuse factor. The columns must be long enough to provoke self-interferences between reuses even after tiling.
The heuristic that we propose is applicable to instances of loop nests that have the format illustrated in Figure 4.15, where $A$ is an array of dimension $p$ referenced $n$ times, $a_i$ are non null integers, and $\gamma_j \in Z$, for $i = 1 \ldots n$ and $j = 1 \ldots p$. Let us assume for simplicity that the lower and upper bounds of the loops are simple expressions independent of preceding induction variables in the same nest, that is to say, they do not induce triangular or trapezoidal iteration spaces.

```
do k = l_k, u_k
  do j_1 = l_{j_1}, u_{j_1}
      ...
      do j_p = l_{j_p}, u_{j_p}
          ... $A(a_1 j_1 + \gamma_1^1, \ldots, a_p j_p + \gamma_p^1)$ ...
          ...
          ...
          $A(a_1 j_1 + \gamma_1^n, \ldots, a_p j_p + \gamma_p^n)$ ...
      enddo
  enddo
endo
```

Figure 4.15 Loop nest format for the heuristic.

**Tiling.** The first step is to test the full permutability of the loop nest. This will enable us to tile the iteration space in any order we want. If the loop nest is not fully permutable, we find the best reordering that promotes data locality according to McKinley’s algorithm that runs in $O(max(d, D^2))$ where $d$ is the loop nest depth and $D$ is the number of dependences in the loop nest. After tiling, we obtain the program in Figure 4.16. For simplicity of notation, we assumed that the nest is fully permutable. In general, we should rename all induction variables $j_i$ to $j_\sigma$, where $\sigma$ is the permutation that leads to the new loop ordering.

Note that we reversed the order of the loops so that the loop that iterates over the first dimension of $A$ is innermost, the loop that iterates over the second dimension is second to innermost etc. This way, we make a better use of data locality assuming column-major ordering. Also note that loop $k$ now iterates over the element loops. This ensures a maximum reuse on each sub-copy of $A$. 
do $jj_p = l_{jp}, u_{jp}, s_{jp} \\
...$ 
\[\text{do } jj_1 = l_{j1}, u_{j1}, s_{j1}\]
\<copy-in code> 
\text{do } k = l_k, u_k 
\text{do } j_p = jj_p, min(u_{jp}, jj_p + s_{jp} - 1) 
\quad ... 
\quad do \quad j_1 = jj_1, min(u_{j1}, jj_1 + s_{j1} - 1) 
\quad \quad \quad \quad <\text{computation code}> 
\quad \quad \quad \quad enddo 
\quad \quad ... 
\quad enddo 
\quad enddo
\<copy-back code> 
\quad enddo 
\text{...} 
\quad enddo

**Figure 4.16** Tiling the program for the heuristic.

**Copying in.** AL is the local array that will hold values of A used within one tile of the iteration space.

Given that the subscript expressions for one dimension across different references to A have the same coefficient $a_i$, we are able to reduce the size of array AL by a factor of $a_i$ for each dimension $i$\(^7\). AL is declared as follows.

$$AL(s_{j1} - \min_{i=1..n} \gamma_i^1 + \max_{i=1..n} \gamma_i^1, \ldots, s_{jp} - \min_{i=1..n} \gamma_i^p + \max_{i=1..n} \gamma_i^p)$$

The code for copying parts of A into AL is in Figure 4.17. For each dimension, we generate a loop that iterates over the space that is accessed within a tile relative to that dimension. We subtract $jj_q + \min_{i=1..n} \gamma_q^i$ for each dimension $q$, so that the local array AL is started at offset 1 and filled up from there on.

Note, here too, that the copying is made in an order that favors stride one accesses for column-major order as in FORTRAN.

**Computation Code** References to array A are replaced with references to AL as in Figure 4.18.

\(\text{\foreignlanguage{italian}Footnote}\) Cases where a subscript expression is loop invariant (not shown in this format for ease of illustration) are trivially dealt with by removing that dimension from the local array AL.
\[ \text{do } j_p = jj_p + \min_{i=1..n} \gamma^i_p, \min(u_{j_p}, jj_p + s_{j_p} - 1) + \max_{i=1..n} \gamma^i_p \]
\[ \therefore \]
\[ \text{do } j_1 = jj_1 + \min_{i=1..n} \gamma^i_1, \min(u_{j_1}, jj_1 + s_{j_1} - 1) + \max_{i=1..n} \gamma^i_1 \]
\[ \text{AL}(j_1 + 1 - (jj_1 + \min_{i=1..n} \gamma^i_1), \ldots, j_p + 1 - (jj_p + \min_{i=1..n} \gamma^i_p)) = A(a_1 j_1 \ldots a_p j_p) \]
\[ \therefore \]
\[ \text{enddo} \]
\[ \therefore \]
\[ \text{enddo} \]

**Figure 4.17** Copying in.

**Figure 4.18** The computation code.

All other references remain unchanged.

**Copying back** The code for copying the data back to the original array A is similar to the code for copying it in with the assignment reversed.

Tile sizes must be chosen so that array AL fits entirely in cache\(^8\). For example, we can set \( s_{j_1} = \ldots = s_{j_p} = s \) and therefore

\[
s = (cs - \prod_{j=1}^{p} \max_{i=1..n} \gamma^i_j - \min_{i=1..n} \gamma^i_j)^{1/p},
\]

where \( cs \) is the cache size.

**4.5.1 Cost**

The cost of this heuristic is divided between the cost of the permutation transformation and the application of the tiling and copying transformation as described in the previous section. The complexity of the tile-and-copy part is linear in the loop nest depth. Existent permutation algorithms have complexities that range from \( O(d) \) [8] to \( O(d!) \) [27, 24], with McKinley's algorithm running in \( O(D, d^2) \). \( D \) being the number of dependences.

---

\(^8\)We assume that no other data requiring cache space are involved in the loop nest. If this is not the case, then an estimate of this cache space must be subtracted from the cache size.
4.5.2 Examples

As an application of the heuristic, we take the 5-spot stencil in Figure 4.19. Let us first see why and when tile-and-copy is a beneficial transformation for this application. If the matrix is very large, i.e. larger than the cache, the reuses carried by the outer loop will result in misses. If we tile the iteration space, such that the size of the block fits in cache, we reduce the number of misses. However, elements of the same block might still conflict with each other. For example, given a direct-mapped cache of 8K bytes, and a matrix of 1024×1024 elements of 8 bytes each, it is clear that blocks of width less than 32 require less than 8K bytes since $32 \times 32 \times 8 = 8K$. However, since columns of the array are 8K long, elements on the same row of the matrix that are 2 columns apart will map to the same cache location since they are 8,192 bytes apart. Therefore, a tile will have to be of a size less than or equal to 2 in order to avoid the interferences. This size is not practical since the 5-point computational stencil accesses three columns for each iteration of the outer loop. Moreover, if the cache line is large, tiles would have to be wider than the cache line in order to take advantage of spatial reuse. To eliminate the interferences and improve the spatial locality of the program, we copy the blocks of the array that are accessed within one tile into a second smaller array.

```plaintext
    do k = 1, T
        do i = 2, N-1
            do j = 2, N-1
                A(i,j) = 0.2*(A(i,j)+A(i,j-1)+A(i,j+1)
                                +A(i-1,j)+A(i+1,j))
            enddo
        enddo
    enddo
```

**Figure 4.19** 5-spot stencil

In a first step, we tile the iteration space to obtain the program as in Figure 4.20. Loop k now iterates over the tiles, and the order of loops i and j is reversed in order to maximize spatial locality.

Next, we insert the code for copying A into B, and replace the references to A by references to B following the heuristic. We show the result of this application in Figure 4.21.
do jj=2,N-1,s
  do ii=2,N-1,s
    do k=1, T
      j=jj, min(N-1, jj+s-1)
      do i=ii, min(N-1, ii+s-1)
        A(i,j) = 0.2*A(i,j)+A(i,j-1)+A(i,j+1)
        *A(i-1,j)+A(i+1,j)
      enddo
    enddo
  enddo
enddo

Figure 4.20 5-spot stencil after square tiling

In this case, because only one array is used, we can further optimize the computation code by shifting the loop bounds and the subscript expressions. We subtract ii-2 from the bounds of loop i and the subscripts at the second dimension of each reference to array B. Similarly, we subtract jj-2 from the bounds of loop j and the subscripts at the first dimension of each reference to B. We obtain the code in Figure 4.22.

We ran the 5-spot stencil routine on one processor of the Paragon XP/S, which has a 16K data cache, 4-way set associative, with 32-byte lines, and a random replacement policy. In one instance, we ran the tiled-only version, and in a second, we ran the tiled-and-copied version. We varied the column size of the matrix from 600 to 1200 double precision elements with steps of 10. In the tiled version, we used square tiles of width cs/(i*8) - 1 where cs is the cache size in bytes and i is the column length of the matrix. This block size is the largest that will not induce self-interferences within a tile. When this value is less than 4, we set it to 4, which is the cache line size in double precision elements.

Given 8-byte double precision array elements and a cache of size cs, we set S to be the larger of the matrix width N and \( \sqrt{cs/8} - 1 \), the tiles are then of size S \( \times S \). This allows for a larger block without interference given that the copied data resides in a more compact area of memory. The performances of both programs are in Figure 4.23.
do ii=2, N-1, S
    do jj=2, N-1, S
        copy in
        do i = ii-1, min(N-1, ii+S-1)+1
            do j = jj-1, min(N-1, jj+S-1)+1
                B(j+jj, i-ii+2) = A(j, i)
            enddo
        enddo
        enddo
        compute
        do k=1, T
            do i=ii, min(N-1, ii+S-1)
                do j=jj, min(N-1, jj+S-1)
                    B(j-jj+2, i-ii+2) = 0.2*(B(j-jj+2, i-ii+2) + B(j-jj+2, i-ii+2-1) + B(j-jj+2, i-ii+2+1) + B(j-jj+2-1, i-ii+2) + B(j-jj+2+1, i-ii+2))
                enddo
            enddo
            enddo
        enddo
        copy out
        do i= ii-1, min(N-1, ii+S-1)+1
            do j= jj-1, min(N-1, jj+S-1)+1
                A(j, i) = B(j-jj+2, i-ii+2)
            enddo
        enddo
        enddo
        enddo

Figure 4.21 5-spot stencil tiled and copied
c compute
    do k = 1, t
       do i=2, min(N-is+1, TS+1)
          do j=2, min(N-js+1, TS+1)
             B(j,i) = 0.2*(B(j,i)+B(j,i-1)+B(j,i+1)
                           +B(j-1,i)+B(j+1,i))
          enddo
       enddo
    enddo
enddo

Figure 4.22 Computation code of the 5-spot stencil tiled and copied with the subscript expressions and the loop bounds rearranged.

Figure 4.23 SOR after tiling only, and after tiling and copying, run on the i860-XP.
We ran the same stencil on the IBM RS/6000 which has a 64K, 4-way set-associative, LRU cache with 128 byte lines. We experimented with the stencil un-optimized, and after tile-and-copy; and we varied the column size from 100 to 500 elements by steps of 10. We also varied the way the data is accessed such that in one time, we accessed them column-wise, that is, in a fashion compatible with the way they are layed out in memory by the FORTRAN compiler, and in a second time, we ran it row-wise. We show the results in Figures 4.24 and 4.25.

![Figure 4.24 Running time of the stencils before and after tile-and-copy executed on the RS/6000 (Row-wise computation).](image)

We notice that the performance improves more with the row-wise computation. This leads us to conclude, on one hand, that long cache lines are quite beneficial for computations that provide spatial locality, and on the other hand that long cache lines recover well from interferences when most of the time the data is accessed contiguously. It also suggests that in the presence of short cache lines —about the same size as the array elements being processed— tile and copy might be more profitable independently of whether the computation is made row-wise or column-wise. Finally, it suggests that tile-and-copy may be more profitable when, due to dependences, the computation can only be done row-wise.

To validate the last argument we experimented on the RS/6000 with a 9-spot stencil where dependences prohibit the loop permutation that would make the data access column-wise (Figure 4.26). The true dependence between the write $a(i,j)$
Figure 4.25 The same stencil with the data accessed column-wise.

and the read $a(i+1, j-1)$ carried by loop $i$ is what makes the interchanging of the inner loops illegal.

We used the same strategy to determine tile sizes for the tiled and the tiled-and-copied versions, as we did for the 5-spot stencil, described above in this section. The performance of the transformed program is illustrated in Figure 4.27.
do k=1, t
   do i=2, n-1
      do j=2, n-1
         a(i,j) = 0.2*(a(i,j)+a(i,j-1))
         +a(i,j+1)+a(i-1,j)+a(i+1,j)
         +a(i-1,j-1)+a(i-1,j+1)
         +a(i+1,j-1)+a(i+1,j+1))
      enddo
   enddo
enddo

Figure 4.26 Unoptimized 9-spot stencil,
and its performance on the RS/6000

Figure 4.27 9-spot stencil after tiling, and tile-and-copy, on the RS/6000
Chapter 5

Observations

5.1 Review

In Chapter 3, we reviewed previous work that addressed the topic of improvement of cache performance by software methods. Each of them made a clear contribution in this direction. However, we feel that more needs to be done. In particular, we would like to generalize these techniques in order to make them applicable for a larger and possibly more common range of cache models. As we noted in Chapter 3, most techniques consider either fully associative or direct-mapped caches, and true LRU replacement policy caches. Practically, none of the studies considered random replacement policy caches or other randomization, like HP's hashing addresses. Furthermore, most researchers, with the exception of Wolfe [26], proposed or studied separate techniques and transformations that can improve data locality, but seldom proposed a unified algorithm that addresses data locality as a whole.

In summary, we believe that the next step that the research should take regarding data locality improvement is towards the generalization of the existent techniques to deal with cache models and more general loop nests formats. That is, given a set of facts about the cache and the loop nest, the compiler should be able to improve locality most of the times.

5.2 Data Locality for Set-Associative Caches

5.2.1 A General Approach for Data Locality Improvement

Recall that there are three classes of misses that originate in a cache: compulsory, capacity, and interference (Section 2.2.1). Without hardware support, like prefetching or non-alloacting loads, we cannot avoid the compulsory misses, i.e. every distinct line accessed results in a miss at least once. However, it is possible to avoid the other sources of misses when the data dependences allow it.
We can avoid capacity misses through tiling. Tiling allows us to reduce the working set per iteration of a loop so that the amount of data accessed between reuses carried by the *element* loops —the loops that iterate within a tile— is below cache capacity. However, interference misses may still exist, in which case we have two options for removing them depending on the case. The first option is to reduce the block sizes until no more interferences occur; this may not be possible when, for instance, one single iteration of the inner loop accesses more lines that belong to the same set than there is associativity. The second method to remove the interferences is copy optimization. We outline in Figure 5.1 the basics of an algorithm optimizing data locality for one variable in one loop nest.

<table>
<thead>
<tr>
<th>Input:</th>
<th>Loop nest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cache parameters</td>
</tr>
<tr>
<td>Output:</td>
<td>Transformed loop nest with “optimal” data locality.</td>
</tr>
<tr>
<td>Useful Information:</td>
<td>Loop structure abstraction (<em>Control Flow Graph</em>)</td>
</tr>
<tr>
<td></td>
<td>Dependence analysis (<em>Dependence graph</em>)</td>
</tr>
<tr>
<td></td>
<td>Reuse information</td>
</tr>
<tr>
<td>Algorithm:</td>
<td>if cross-column accesses then</td>
</tr>
<tr>
<td></td>
<td>Permute for spatial locality</td>
</tr>
<tr>
<td></td>
<td>if capacity misses then</td>
</tr>
<tr>
<td></td>
<td>Tile to reduce the working set of the element loops</td>
</tr>
<tr>
<td></td>
<td>to cache capacity or less</td>
</tr>
<tr>
<td></td>
<td>if interferences then</td>
</tr>
<tr>
<td></td>
<td>Let $n$ be the number of distinct lines belonging to</td>
</tr>
<tr>
<td></td>
<td>one same set accessed in one iteration of the innerloop</td>
</tr>
<tr>
<td></td>
<td>if $n &gt; \text{set-associativity}$ then</td>
</tr>
<tr>
<td></td>
<td>Copy data accessed by one iteration of the tile loop into a new array</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Tile with block size $= b$ such that</td>
</tr>
<tr>
<td></td>
<td>the number of distinct lines belonging to</td>
</tr>
<tr>
<td></td>
<td>one same set in $b$ iterations of the element loops is $\leq \text{set-associativity}$</td>
</tr>
</tbody>
</table>

**Figure 5.1** Data locality algorithm for one variable in one loop nest.
The problem of choosing the best block size when tiling the iteration space still remains. In the outlined algorithm of Figure 5.1, we only consider one variable as a source of interferences. We do not compute the block size as a function of the cache parameters.

5.2.2 Determining Block Sizes For Set-Associative Caches

Assume a column-major order layout. Let $A$ be an array of $n$ rows and $m$ columns and the tiled iteration space in Figure 5.2. We want to find $b_1$ and $b_2$ that minimize interferences in the element loops — the two innermost loops — and yet have a block as large as possible. The loop nest accesses submatrix $A_{4,5}$.

First, we consider self interferences only. In other words, we assume only one array variable is accessing the cache. We also assume a true LRU replacement policy.

\begin{verbatim}
  do jj = 1, p, b1
    do ii = 1, q, b2
      ...
      do j = jj, min(p, jj+b1-1)
        do i = ii, min(q, ii+b2-1)
          ... A(i,j) ...
        enddo
      enddo
    enddo
  enddo
enddo
\end{verbatim}

Figure 5.2 Tiled loop with one array variable referenced.

The cache is $a$-way set-associative, and can contain up to $cs$ elements of the size of elements of $A$. The algorithm consists of finding the first element $A(d1,d2)$ that interferes with $A(1,1)$. In an $a$-way set associative cache, this implies finding the $a$-th element that maps to the same set as $A(1,1)$.

The algorithm iteratively finds the array elements that map to the same set as the origin of the matrix. Those that are located outside the area referenced in the loop are not considered. It stops when the number of conflicting elements is equal to the set-associativity of the cache. Unless one column of $A$ is larger than the cache size, the width of the block is equal to the column number of the last found element minus one.
Algorithm: BlockSize(cs,a,n,p,q)
Output: structure with integer fields (height, width)
Input: cache size, set-associativity, matrix column length,
iteration space width, iteration space length

s = cs / a
if (n*p>cs) then
  count = 0
  while (count<n and d2<p) do
    d2 = s div n
    d1 = s mod n
    if (d1<=q) then
      count = count+1
    endif
    s = s + cs/a
  enddo
if (d2=0) then
  b2 = 1
  b1 = cs
else
  b2 = d2
  b1 = q
endif
else
  b2 = p
  b1 = q
endif
return(b1,b2)

Figure 5.3 Determining a rectangular block that avoids self interferences
and the height is equal to \( q \). Note that this algorithm does not guarantee avoidance of self interferences. If more precision is liked, the algorithm should be changed so that all elements that are found to map to the same set as \( A(1,1) \) be counted independently of whether or not they belong to the area of \( A \) that is referenced in the loop nest. However, the algorithm as presented would avoid most of the interferences and produce larger blocks.

![Diagram](image)

**Figure 5.4** Determining a block that avoids interferences in a 2-way set-associative cache.

For example, assuming a 2-way set-associative cache, in Figure 5.4, box 1 refers to \( A(1,1) \). Box 2 refers to the first element of \( A \) that maps to the same set as \( A(1,1) \), but since it is outside the accessed part of \( A \) in the iteration space \( [1..q] \times [1..p] \), it is not counted. However, box 3 is counted and box 4 is the element that also maps to the same set as box 1 and box 3 and would therefore displace one of them before they get reused, so we set the block width to the column that precedes column of box 4, and the block height is here set to \( q \).
Multiple variables  When more than one array variable are accessed in the loop nest, we cannot make assumptions as to whether two distinct arrays will interfere with each other, unless we can determine at compile-time the address mapping of each array, or when the memory space accessed by one variable is larger than the cache. In his model for computing the miss rate on the reuse of a data in a loop nest, Wolf adopted a probabilistic approach to account for cross interferences [24].

We distinguish two cases:

- If the number of arrays accessed in the tiled loop nest is less than or equal to the associativity of the cache, then we determine a block such that the total number of columns accessed per iteration of the tile loops is less than or equal to the cache associativity. For example, if in the loop nest of Figure 5.2 we had two references A(i,j) and B(i,j) instead of A(i,j) only, we would choose $b_1 = \min(q, cs/a)$ and $b_2 = a/2$ where $a$ is the set-associativity of the cache (assuming $a \geq 2$).

- Otherwise, we make our decisions given the argument that relatively small, self interference-avoiding blocks are a good conservative choice, because even when there exist larger blocks that do not cause interferences, the performance loss due to the choice of a smaller block is small. However, we want to be able to reuse an entire line when we bring it to cache before it is flushed out. Therefore, the block height should not be smaller than the cache line size. We propose the following heuristic to determine a block size that minimizes self and cross interferences:

1. Choose a small fixed block height $b_1$, for example equal to the cache line size in elements.

2. For each array, compute the block width that avoids self interferences as in algorithm BlockSize with $q = b_1$.

3. Take the smallest value computed in the previous step as the block width $b_2$.

There exists a trade off between the choices of $b_1$ and $b_2$. If $b_2$ is very small, e.g. equal to the set-associativity, it means that the array length that caused this choice of $b_2$ has very long columns and therefore increasing $b_1$ slightly will
not result in additional interferences for this array. We choose to increment the block size by an amount equal to the line size in number of elements when that happens.

Also, blocks that are too wide should be avoided to reduce the chances of cross interferences. We bound this value from above by

$$\sqrt{\frac{\text{cache size}}{\# \text{ of variables}}}.$$ 

We show a more formal version of this algorithm in Figure 5.5.

```
Var = set of array variables in loop nest
Length(v) = column length of variable v in Var

if # of variables ≤ a then
  b1=min(q,cs/a)
  b2=a/(# of variables)
else
  b1 = line size in number of elements
  b2 = \sqrt{\text{cache size}/\# \text{ of variables}}
  foreach v ∈ Var
    b = BlockSize(cs,a,Length(v),p,b1,linesize)width
    b2=min(b,b2)
  endfor
  If b2≤a then
    b1=b1+linesize
  endif
endif
return(b1,b2)
```

**Figure 5.5** Rectangular block for multiple variables

**Approximated LRU policies** Some processors like Intel's 386 approximate the LRU policy to make the replacement decision faster. For an associativity of $2^a$, only the $2^{a-1}$ most recently used lines are guaranteed to remain in cache. To compute a block size for this type of cache while avoiding the uncertainties of the approximated LRU, we can use the same algorithms we proposed for true LRU caches but assuming an associativity of $2^{a-1} + 1$. 
Random Replacement  It is harder to reason with random replacement caches than LRU caches for the obvious reason that in case of conflict, the line to be replaced cannot be predicted. One hardly acceptable solution is to consider the cache as if it were a direct-mapped cache of size the original cache size divided by the set-associativity. This makes the replaced line easier to know in advance but has the obvious disadvantage of excluding a large portion of the cache.
Chapter 6

Conclusion

A large, yet insufficient amount of work has been dedicated to improving cache performance of programs using software methods. In this thesis, we surveyed a representative set of the existent literature pointing out their strengths and weaknesses. We started by describing caches and how they work, emphasizing the aspects that are important from the standpoint an optimizing compiler. We then described the different software techniques that pursue data locality and described briefly the current works that address these techniques.

We took one particular technique, namely copy optimization and showed two examples where we were able to achieve large speed-ups when the interferences were high. However, these examples use a very large data set and may not be common in practice. When we examined more common programs, we noticed that copy optimization can be profitable if used in conjunction with iteration space tiling. This combination of techniques, tile-and-copy, seeks to avoid capacity misses through tiling and interferences by copying. In an ideal code (with array references only), this would reduce the misses to precisely the compulsory misses. We validated this claim with other common examples, noting the increase of performance obtained from the tiled version to the tiled-and-copied version.

We proposed an algorithm to apply this technique on a relatively wide range of loop nest formats; however, the generality of the algorithm came at the expense of a higher complexity. It was based on work by Ancourt to achieve data transfers from global memory to local memories in a distributed memory parallel machine. In many instances, the algorithm’s complexity is exponential in the number of loops. After this, we proposed a heuristic to apply tile-and-copy on more restricted code samples. This heuristic has the advantage of being faster. We think that it makes more economical sense than the algorithm in a compiler implementation of tile-and-copy.

One of the experiments that we ran suggested that tile-and-copy may be an alternative optimization when loop permutation for spatial locality is not legal.
In the last chapter we summarized the shortcomings of the current techniques that address the cache performance improvement problem. We then proposed the general lines that we think a data locality algorithm should follow.

We also devised an algorithm that computes a rectangular block size for a tiled loop nest with one unit-stride reference. The block computed is the largest guaranteed to avoid self interferences in a set-associative, true LRU cache. We then extended the algorithm to account for multiple references avoiding self interferences and hopefully minimizing the chances for cross interferences. Finally, we suggested ways to deal with approximate LRU and random replacement policy caches.

**Future Work.** As we observed in Chapter 5, the data locality problem should be addressed as a whole. The next step should be to implement the algorithm that we suggested in the same chapter. For this, we recommend taking the already existing techniques and enhancing them in order to account for realistic cache behaviors, and possibly broader areas of applicability.
Bibliography

   Cache performance of operating systems and multiprogramming.

   Column-associative caches: A technique for reducing the miss rate for direct-
   mapped caches.

   Automatic loop interchange.
   In Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.
   Montreal, Canada, June 1984.

   Scanning polyhedra with do loops.
   In Proceedings of the Third ACM SIGPLAN Symposium on Principles and
   Practice of Parallel Programming, Williamsburg, VA, April 1991.

   Génération automatique de codes de transfert pour multiprocesseurs à mémoires
   locales.

   Dependence Analysis for Supercomputing.

   Unimodular transformations of double loops.
   In Advances in Languages and Compilers for Parallel Computing, Irvine, CA.

   Memory-Hierarchy Management.

   Parametric integer programming.

On estimating and enhancing cache effectiveness.

On the problem of optimizing data transfers for complex memory systems.

Strategies for cache and local memory management by global program transformation.

Practical dependence testing.

Interprocedural transformations for parallel code generation.


Code generation for the hyperplane method and for loop interchange.

Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers.

Optimizing for parallelism and data locality.

A vectorizing, software pipelining compiler for LIW and superscalar architecture.

*Automatic and Interactive Parallelization.*  

*Software Methods for Improvement of Cache Performance.*  

[22] W. Pugh.  
The Omega test: A fast and practical integer programming algorithm for dependence analysis.  

A general framework for iteration-reordering loop transformations (technical summary).  

*Improving Locality and Parallelism in Nested Loops.*  

A data locality optimizing algorithm.  

*Optimizing Supercompilers for Supercomputers.*  
PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[27] M. J. Wolfe.  
Iteration space tiling for memory hierarchies, December 1987.  
Extended version of a paper which appeared in *Proceedings of the Third SIAM Conference on Parallel Processing.*

*Optimizing Supercompilers for Supercomputers.*  