Compiler Optimization of Data Storage

Thesis by

Rajiv Gupta

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1991

(Submitted July 13, 1990)

To my parents,

who gave me much more than mere life

Acknowledgements

There are many who have profoundly influenced my life and, for better or for worse, have had a hand in the molding of this person I call, "I". As such they deserve some credit for all my accomplishments. In the context of the body of work documented in this thesis, Jim, Mani, and Al deserve special mention.

To the world, Jim, Mani, and Al are Prof. James T. Kajiya, Prof. K. Mani Chandy, and Prof. Alan H. Barr, respectively. To me, they are mentors, and friends. They were a constant source of ideas, encouragement, and support. Whenever I sought guidance about my immediate research, future plans, or life in general, I was confident that they would take the time and interest to give me sound advice.

As a thesis advisor, Jim is second to none. He inspires creativity and unconventional thought. He encouraged an independence that was largely responsible for making my Caltech experience truly pleasurable and fulfilling.

The list of friends and Caltech colleagues who proof-read my manuscripts and brainstormed with me on research ideas would not be complete without the names of Ronen Barzel, Pieter Hazewindus, John Wawrzynek, Mike Newton, and Tim Kay.

Abstract

The system efficiency and throughput of most architectures are critically dependent on the ability of the memory subsystem to satisfy data operand accesses. This ability is in turn dependent on the distribution or layout of the data relative to the access of the data by the executing code. Page faults, cache misses, truncated vectors, global communication, for example, are expensive but common symptoms of data and access misalignment.

Compiler optimization, traditionally synonymous with code optimization, has addressed the issue of efficient data access by manipulating the code to better access the data under a fixed, default distribution. This approach is restrictive, and often suboptimal. Data optimization, or data-layout optimization, is presented as an integral part of compiler optimization.

For scalar data, a good compile-time approximation of the "reference string," or sequence of data accesses, is advanced for the purpose of distributing the data. However, the optimal distribution of the scalar data for such, or any, reference string is proved NP-complete. A methodology and a polynomial algorithm for an approximate solution are developed. Experiments with representative, but scaled, scientific programs and execution environments display a reduction in cache misses up to two orders in magnitude.

For array data, compile-time predictions of the patterns in which the data is accessed by programs in scalar and array languages are examined. For arbitrary computations in an array language, the determination of the optimal layout of the data is proved to be NP-complete. Polynomial techniques for the approximate solutions to the optimal layout of arrays in both languages, scalar and array, are outlined.

The general applicability of the techniques, in terms of environments other than hierarchical memories, and in terms of interdependence with code manipulations, is discussed. New code optimizations inspired by the data distribution techniques are motivated. The prudence of compiler- over user-optimized data distribution is argued.

Contents

List of Figures

Chapter 1

Introduction to the thesis

Compiler optimization has been traditionally synonymous with manipulations of code: dead-code removal, code motion, common sub-expression elimination, operator strength reduction, etc. [3]. Data is considered only in its effects on the optimization of code, *e.g.,* flow, anti–, and output dependencies between data items constrain the parallelization and vectorization of code [1, 5, 6, 25, 31, 33]. Data optimization, specifically the layout or the distribution of the data, is usually reserved for clever programmers hoping to eke out higher efficiencies of their systems. Although there is little argument that data optimization is largely responsible for the superiority of hand-optimized code over compiler-optimized code, and although the benefits of data optimization are potentially enormous, code optimization boasts of a disproportionately higher research effort and level of automation than data optimization. The disparity may result from two subconscious biases rooted in tradition: First, the physical data space is imagined as a homogeneous monolith, and second, the virtual data space is considered too unstructured for the deduction and analysis that precede optimization.

The former reasoning implies that the cost of any two accesses in the data space is constant, and thus any optimization can at best result in fewer data accesses; which again falls within the purview of code optimization. While the physical data space of some of the earlier computers may have been homogeneous and monolithic, the physical data space of most modern systems rewards data locality. Tacit in the latter bias is the belief that data accesses are essentially random and that any meaningful data optimization requires complete disambiguation of all data accesses. Thus, the bias may follow, even cases that permit data optimization offer too parsimonious a return on invested effort. This document attempts to expose this anachronism precisely.

Conforming data storage to the use of the data by the program improves the data locality of programs executing in environments with hierarchical stores. In environments featuring interleaved memories, pipelined processors, vector processors, or multiprocessors, the compile-time knowledge of data access patterns can be analogously used to judiciously distribute the data so that it is efficiently available in the right form, in the right place, and at the right time: the place, form, and time being determined by the access of the data by the executing code.

This leads us to anticipate payoffs that include fewer page faults in hierarchical stores, and improved efficacy of architectural constructs in which the data organization drives the parallelism, *viz.* vector processors, interleaved memories, multiprocessors. The net result: faster process turnaround, better system throughput, and more efficient resource utilization. Most current computers feature more than one of the abovementioned architectural components. The cumulative advantages, and conversely the high cost of inefficiency in these computers, make the work presented in this thesis particularly relevant to computers today.

The optimal distribution of data is concerned not with the absolute physical address, but with the location of data relative to each other. The correlation between the data is embodied in the access of the data by the executing code. Note that the data accesses that generate communication traffic and harbor potential inefficiencies correspond to the executing program, and not the source program. This distinction is critical in determining an appropriate executor of the data optimization, and in motivating the automatic optimization of data that is an integral part of compiler optimization.

Chapter 2 addresses the distribution of a datum relative to other data, when each datum is treated as a singular entity. The datum may, however, be composed of many elements, as an array is an abstraction for many individual elements. Although the distribution of the entire array relative to other data is targeted by the techniques for scalar data, the layout of the elements within the array is the subject of Chapter 3. This intra-operand distribution, as opposed to the inter-operand distribution of Chapter 2, is relevant particularly when the array is larger than, and each element of the array is smaller than, the smallest quantum of information transfer , *viz.* a cache line, bus width, etc.

The complexity of optimally solving many of the distribution problems is explored in Chapter 4. This exercise helps in channeling effort to fruitful areas, or at least in alerting the protagonist that his optimal solution may have finally disproved the P $\neq$ NP conjecture, and in identifying promising approximate solutions. Chapter 5 recapitulates the contributions of this thesis and motivates the direction of related future work. An APL example to illustrate the working of the remap algorithm, an array distribution algorithm suited to data-array languages, can be found in Appendix A. Appendix B is a tabulation of the constraints exerted by APL operators on the storage of array operands.

The main contributions of this thesis are as follows:

- It identifies the inefficacy, and sometimes counterproductivity, of some of the traditional compiler optimizations for improving system efficiencies and program turnaround times. Through argument and example, it motivates a reassessment of code transformations for synergizing code and data.

- It proposes data optimization as a rich source of program improvement and opens this new, potent, and practical platform for further research in compiler optimization. It develops a methodology for the distribution of data based on the general principle of locality. This distribution can then be universally applied to different architectures. It presents approaches and algorithms for implementing some of the optimizations.

- It attempts to expose biases that obstruct fruitful research in compiler optimization. Two of these are a suspected bias against data optimization in general, in terms of its complexity and rewards, and a bias towards relegating the distribution of data to the programmer.

- It proves that some of the data distribution problems are NP-complete. The importance of this work is second only to the central proposal of optimizing the storage of data, because the fundamental complexity of the problems will outlive the particular techniques developed.

Note that the proposed data optimization does not invalidate all traditional compiler optimizations. The optimization of data may be applied after, or in conjunction with, code optimization. In the latter case, the code transformations, both traditional and proposed, are subject to objectives revised with data distribution constraints.

Chapter 2

Compiler Optimization of Scalar Data Storage

2.1 Introduction

Memory hierarchies are an attempt to improve program execution performance and resource utilization by mirroring and better serving the locality of the executing program. The locality of a program can be conceptually partitioned into its code locality and its data locality.

These localities are related – repeated execution of the same set of instructions will access the same set of data (modulo array subscripts.) However, the localities may vary widely in their size. For example, a large set of instructions may access a small subset of the data space. The converse is true for loops in programs that span arrays. For scalar data, the data locality very closely resembles the code locality. Since this chapter deals exclusively with scalar data, data locality will loosely imply the localities of both scalar data and code.

The program's locality changes over the course of the program's execution and the change may not be continuous, as in the case of jumps in the code. The tracking of this locality by the memory hierarchy is discrete, in that the contents change in units of some quantum called a line. The contents of the lower levels of the memory hierarchy often get out of phase with the changing data locality of the executing program. One of the most common indicators of such a skew is a data access that cannot be satisfied by the memory level. We then incur a line miss during which a line of data is loaded into the appropriate memory level; consequently, a line of data is ejected from the memory level to make room for the incoming line. A line miss is thus an attempt to realign the contents of the appropriate memory level with the current data locality.

At any instant of time, a memory level is simply a collection of an appropriate number of such lines. Thus the contents of the memory level, and therefore its ability to satisfy the data accesses of the executing program, depend on the contents of the constituent lines. As an example where the lower memory level represents a cache and the line is synonymous with a cache-line, for a 256 byte cache consisting of 64 cache-lines of 4 bytes each, if each cache-line contains only one byte of data relevant to the current locality of the executing program, only 64 bytes or one-fourth of the cache is gainfully employed in satisfying the data accesses. Note that the remaining 3 bytes of every cache-line could contain data from the same program but still be irrelevant to the particular locality under consideration.

With the efficiency and utilization of the memory hierarchy dependent on the distribution of data into lines, the data should be distributed into lines to permit the appropriate memory level collections of these lines to closely map the localities of the executing program. This objective necessitates a good handle on the data localities of the program. The approach adopted in this chapter uses compile-time approximations to the access trace as a means of predicting the localities.

The common technique, and to our knowledge the only technique, for the distribution of scalar data, distributes the data into lines in the order in which the data is defined in the program. This default distribution will be referred to as "definition-order" in the rest of the thesis. The definition-order distribution scheme has the advantage that the user can optimize the distribution of scalar data simply by reordering his data definitions. This chapter exposes the two fallacies in such a purported advantage: firstly, the distribution of scalar data is not considered an important issue and hence will not be attempted by the user, and secondly, as we later illustrate, the optimization of data by the user is usually counterproductive.

Whatever be the default distribution scheme, definition-order or otherwise, as long as it is not moderated by the eventual accesses of the program, it is subject to many debilitating worst cases, and the distribution is very unlikely to be good.

2.1.1 Motivation

The distribution of data into lines, which reflects the access of the data by the executing code results in fewer line misses in the case of demand fetch, and reduced memory traffic in the case of prefetch or speculative fetch, when compared to the distribution of data that is independent of the data access. For an illustration of the payoffs, consider the piece of nonsense code of Figure 2.1.

Assume that the cache-line size is 2 units, where an operand of type `int` occupies unit space, and the cache is two-way set-associative. Also assume that the definition-order scheme is used for distributing the scalar operands into cache-lines. Note that all of the assumptions above are very typical of extant systems. Further assume that for one of a number of reasons, the variables a, d, x, w have not been allocated to registers. As one possibility, the available registers have all been assigned to other variables, temporaries, and constants. Note that many scientific programs and register sets do exhibit such contingencies.

With the order of data definitions as in the code above, the scalar operands a, b, c, d, x, y, z, w are very likely to be distributed into lines ab, cd, xy, zw. Moreover, if the lines map onto the same cache set, only two of the lines may reside in cache simultaneously. The total size of the cache is irrelevant because these operands cannot appear in any other set of the cache.

If the code above is to be executed, *i.e.,* it is the output of traditional code optimizations, then each iteration through the innermost loop will result in the four cache misses: ab, cd, xy, zw. The distribution and mapping of the scalar operands cannot exploit the rest of the cache and we incur a

```
data definitions
int a,b,c,d;
int A1[100][10], ...
⋮
int x,y,z,w;
program source
⋮
for counter1=1 to n
   for counter2=1 to n
   ⋮
      for counterm=1 to n
         a = f(d)
         x = g(w)
         ⋮
```

Figure 2.1: Subprogram for motivating a distribution of scalar data based on the access of the data by the executing code

cache miss for each access to any one of the scalar operands. For the m loops each with n iterations, the code fragment will incur a total of $4n^m$ cache misses.

From the program code we can observe that operands a, d, x, w are used contiguously. In an informed distribution the scalar operands are distributed into lines ad, bc, xw, yz. This distribution will incur a total of 2 misses for all of the n^m iterations. For this rather contrived example, the payoff for a judicious distribution of the scalar operands into lines is a reduction in cache misses by a factor of $2n^m$.

The m nested loops are meant to underline the effect of loops on cache misses and hence on optimal operand distribution. Since loops encode repeated execution of the enclosed piece of code and are effectively expanded out during the execution of the program, they exaggerate the difference between alternative data distributions.

2.1.2 Related Work

In the literature, most optimizations for improving the efficiency of memory hierarchies preordain a definition-order distribution of the scalar operands into lines which is independent of the accesses of the data by the code. There is little work towards the automatic, compiler-optimized distribution or memory layout of data, either scalar, or array data.

Recent endeavors [2, 6, 17, 34] at improving the efficiency of the memory hierarchy direct compiler optimizations to transform the code so that the accesses to the ravel distributed data incur fewer misses. The improvement, primarily for mapping code and array data localities, and in the use of vector registers, is realized through the loop optimizations, *viz.* loop blocking, fusion, distribution, etc. Although fruitful, these transformations are limited by the compile-time resolution of dependence analysis, and by the interdependencies in the code.

While similar in objective, this body of work differs from ours in that they manipulate code for a given, immutable data distribution, while we optimize the layout of data to map the accesses by the optimized code. Moreover, they are exclusively targeting accesses to array data, as opposed to the scalar data of our case. Thus the research efforts complement each other.

From the perspective of the optimizations of data for memory hierarchies, past research can be classified as mostly "statistical". This takes the form of either fine-tuning fetching strategies or replacement policies in the operating system approaches [37], or optimizing the size, partitioning, and other characteristics of individual levels in the hierarchy in the system design approaches [37]. One body of system design work that targets, albeit statistically, the distribution of array data deals with skewing schemes and interleaved memories [11, 22, 26, 38]. All these statistical approaches utilize actual program traces or some analytical model of the reference patterns, and favor an empirical modeling and calibration style. Note that the statistical approaches attempt to fit the system around fixed reference patterns and default data distributions.

Converse to the statistical, there is the "tailored" optimization of data in which the compiler optimizes the data of individual programs. This approach is more recent and the work so far primarily attempts to optimize the distribution of array data [23]. In a different vein, and primarily motivated by maintaining coherency in multiprocessor caches, there is an effort at tagging the data with attributes such as cacheable–noncacheable, shared–private, dirty–clean, etc. [10].

We are aware of no papers that target the optimal distribution of scalar data. Out of the possible conjectures for this indifference, the two that appear most probable are both based on an expectation of low returns from any attempt to optimize the storage of scalar data. The conjectures are:

- The common wisdom and algorithms for caches is a carry-over from the primary–secondary memory interface with large page sizes for which the optimal distribution of scalar data is indeed inconsequential.

- Even in the case of techniques targeted to caches, the cache model is conceived to be fully associative. Similar to the consideration above, if the cache is fully associative, the number of scalar operands is rarely ever large enough to warrant an optimization effort in distributing the data.

This last conjecture merits further discussion because for reasons of cost and speed, the majority of current memory hierarchies employ the restricted set-associative or direct-mapped caches. In the fully associative model, even if the cache-lines are not efficiently utilized, most inner loops of most scientific codes do not access a sufficient number of scalar operands to make the contribution of the scalar operand accesses a significant fraction of the total cache misses.

As the example in the previous subsection demonstrated, the payoffs in caches with restricted associativity is indeed potentially substantial. Conceptually, for the operands that are mapped onto the same set, the available portion of the cache is restricted to the set in which they reside. Since the associativity of caches of most commercial systems is small [19, 21, 32, 37], it is possible, and likely, that the scalar operands accessed within innermost loops, while few in number, may generate disproportionately many cache misses. For such set-associative caches, the distribution criteria can just as easily, and perhaps more profitably, determine which operands not to map onto the same cache set.

Note that our techniques do not preempt other compiler, system, or algorithm optimizations for improving program attributes and efficiencies. Our optimizations can be applied for a given system, for a given encoding of the algorithm, and for an executable output of other compiler optimizations; as such, they complement the other optimizations. The code does affect the optimal distribution of the data by defining the patterns in which it accesses the data. For the purposes of this document we will assume that the code is in a final, executable form.

2.1.3 Outline of the Chapter

Section 2.2 defines the terms and the assumptions that will be used in the rest of the chapter.

There are many levels of approximation in distributing scalar data into lines. Firstly, it is not possible to obtain the exact reference string at compile-time. Section 2.3 develops a compile-time approximation to the reference string. Independent of how the reference string is obtained, it is NP-complete to obtain an optimal partition of operands into lines, which minimizes the number of line misses for satisfying the accesses of the reference string. Section 2.4 describes a heuristic and an algorithm for the distribution of scalar operands to map the accesses of a given reference string. Thus the approximate reference string yields an even more approximate distribution.

Given the multiple levels of approximation in obtaining a distribution driven by the use of the data, one might be tempted to expect mediocre payoffs. Section 2.5 documents the very encouraging reduction in line misses from applying the techniques developed in this chapter.

Besides instrumenting the scalar data distribution into lines for use by the system fetch and replacement policies, by virtue of its knowledge of the distribution, the compiler can circumvent some of the pathological worst cases of the system policies. An example of such a contrivance for the LRU replacement policy, a moderation of some of the assumptions of Section 2.2, and a discussion of the resulting general applicability of our work to the different levels of the memory hierarchy and to different system architectures are the subject of Section 2.6. This section also motivates compiler-optimized over user-optimized scalar data distribution, and introduces a new type of code optimization aimed at reducing line misses.

Section 2.8 notes some avenues for further research that might be catalyzed by out work, and Section 2.9 concludes the chapter.

2.2 Terminology

2.2.1 Definition of Terms

The following terms are used in the rest of the thesis:

- **Reference String** R: This is the sequence in which the operands are accessed by the executing program. Example: R: $a \cdot b \cdot c \cdot d$ implies that the program will first access operand a, then operand b, and so on.

- **Operand Space** S: This is the unsorted list of the operands of the program.

- **Lower Level Operand Capacity** M: This is the size, in number of operands, of the generic lower level of any two neighboring levels in a memory hierarchy. For example, the lower level could represent the cache or primary store.
- **Line Capacity** p: This is the size, in number of operands, of the generic quantum in which information is transferred between any two neighboring levels in a memory hierarchy. For example the line could represent a cache-line at the cache–primary store interface, or a page at the primary–secondary store interface.
- **Lower Level Line Capacity** t: This is the size, in number of lines, of the lower level. M, t, and p obey the integer relationship, $M = tp$.

2.2.2 Catalog of Assumptions

The following assumptions are tacit in the rest of the chapter:

- **Unit Operands:** All operands are assumed to be of unit size.
- **Uniqueness:** There is only one copy of an operand in the entire memory hierarchy. In particular, no two lines have any operand in common.
- **Compact Distribution:** The unit operands completely fill all but possibly one line. A slack distribution may sometimes reduce the number of line misses.

2.3 A Compile-time Approximate Reference String

The reference string represents the actual accesses of the executing program. In this section we motivate the reference expression, a reasonably accurate compile-time abstraction of the reference string. Approximations at the nondeterminisms in the reference expression give an approximate reference string, which then directs the distribution of the scalar operands into lines.

2.3.1 The Reference Expression

Most programs contain code whose execution is controlled by conditions on input data and other run-time information. Checks for boundary values and exceptions are a common example. The compiler is thus not privy to the exact sequence of operand accesses of the computation. It is instructive to identify the references that generate the non-determinism in the reference string.

The source program can be broken into blocks, where a block is a maximal sequence of contiguous instructions none of which, except the last instruction of the sequence, is a branch, and none of which, except the first instruction of the sequence, is the target of a branch. A program now consists of several blocks delineated by branches. Since they are devoid of conditionals, the accesses of program blocks are determinate at compile-time. Thus the compile-time non-determinism of the reference string is derived solely from the non-determinism in the execution sequencing of the blocks of the program.

Tacit in the definition of the reference string is the use of the *dot* operator to represent contiguity of accesses. In the rest of this thesis, the actual $\cdot$ of the *dot* operator will be omitted. Thus $abcde$ will replace $a \cdot b \cdot c \cdot d \cdot e$, which represents the successive accesses to operands a, b, c, d, e, in that order. From the observations of the previous paragraph, the accesses of a program block can be represented by a similar sequence of operands in the reference expression.

With sequential access, and hence program blocks, represented by the *dot* operator, the next step in a compile-time approximation to the reference string attempts to represent conditionals. In keeping with the influence of regular expressions, we introduce the *either*, or $+$ operator, to represent a conditional or choice, and the *parentheses,* () to delineate. *dot* takes precedence over *either*. Thus $d(ba + ca)$ represents an `if-then-else` conditional, where the `then` branch accesses operands b and a in that order, while the `else` branch accesses operands c and a in that order. An example of source code that would be represented by $d(ba + ca)$ is: `if` d `then` $a = b$ `else` $a = c$.

To represent loops, we adopt the representation of repetition in regular expressions, best explained by the use of an example. $(abc)^3$ represents a repetition of abc three times, *i.e.* $abcabcabc$. Similarly, $(abc)^*$ represents the infinite repetition of the string abc. In many cases, the iteration counts for loops are not known to the compiler, as in the case of `while` loops. In such contingencies the reference expression uses a default for the repetition count, θ. An example of a subprogram and the corresponding reference expression are illustrated in Figure 2.2.

Example: (code)

```
a = b + c
b = p + q
if (b<a) then {
   for (i=b; i<c; i++) {
      m = n/i
      if (n>0.2) then p = r
      else p = q
      n = n - i
   }
   b = b + a
} else b = d
```

Corresponding reference expression: $bcapqbab(b(icnimn(rp+qp)nin)^{c-b}bab+db)$

Figure 2.2: Subprogram and the corresponding reference expression

The reference expression operators allow us to target source code that can be converted to a go-to*less* form, or more accurately programs with reducible flow graphs. Virtually all programs fall into this category as verified in the studies of [4, 24].

Thus the operators: *dot*, *either*, *parentheses*, and *repetition*, are the foundation of the reference expression. Observe that the reference expression is the program accesses stripped of any computation. With sufficient information for choosing between the alternatives of the *either* operator, and for setting the values of θ for the repetition counts, the reference expression can be expanded to contain only the *dot* operator. This then will be identical to the reference string of the executing program. But in the absence of such run-time information, we make approximations at the *either* and *repetition* operators in the reference expression.

2.3.2 Approximations for Conditionals and Loops

In most programs, the bulk of the computation is performed within loops; thus most of the accesses are generated within loops. A first approximation to the reference string concentrates on the accesses within loops. While this may reduce the cost of computing a distribution, the conceptual problem of approximating conditionals or loops is no easier.

The code at the target of most conditionals is executed exclusive-or, *i.e.* for multiple targets as in the `then` and `else` of an `if-then-else`, the program will execute the code of either one or the other, but not both. However for conditionals inside loops, different iterations might, and very often do, follow different targets of the conditional. Hence the mutual exclusivity within an iteration is lacking between iterations. How does one approximate the references of conditionals within loops?

It is helpful to reaffirm that the approximations to the reference string are solely to provide clues for the distribution of operands, which minimizes line misses. Thus the validity of an approximation is measured (inversely) by the number of line misses that will be incurred by the corresponding distribution in satisfying the accesses of the program. We refer to the distribution that is optimal for a sequence of references as the line constraints, or simply the constraints, of that sequence.

In particular, the reverse of a reference string produces constraints identical to those of the original reference string. In the context of the conditional represented by $(abcdef + bdfgeh)$ in the reference expression, the approximation substring concatenating the references of the two targets, $abcdefbdfgeh$, encodes the constraints of both the targets as well as some false constraints arising at the boundary of the concatenation. The boundary constraints are false by the mutual exclusivity argument, which tells us that the conditional $(abcdef + bdfgeh)$ can never produce the reference substring $abcdefbdfgeh$ in the same iteration.

For the purposes of the distribution of scalar operands into lines, the following are possible approximations converting conditionals in the reference expression to the corresponding reference substring:

- An approach akin to trace-scheduling [14] might append the references of blocks to create an approximate reference string. The reference strings of blocks with higher execution probabilities can be appropriately replicated to reinforce their constraints. This will introduce false dependencies between the head and the tail of each replicated block, and at the boundary between two independent traces. On the other hand, with larger straight-line blocks, we reduce the false dependencies at the boundaries of the condition targets.

- Constrain the distribution of the operands by the access sequences of the two targets in isolation. This will avoid the false dependencies at their concatenation, but will also miss the true dependencies at their boundary with the rest of the program. Note that the constraints common to the two distributions will be reinforced in this approach.

- Interleave the operands of the two targets into lines. Example: for the conditional $abcd+efgh$, and $p = 2$, in this approach the line ae is just as desirable as ab and ef. This approach in some sense hedges between the extremes of completely satisfying the constraints of one target while ignoring the other.

- Based on their execution probabilities, ignore the accesses of one or the other target. This will effectively distribute the operands to map the reference string that is the most likely trace through the program.

- Concatenate the references of the targets but replicate the references of each target individually so as to dilute the false dependencies at the boundary. For example, the reference expression conditional $(abcd+efgh)$ gets translated to $(abcd)^n(efgh)^n$ in the approximate reference string. As in the first option above, constraints common to both the targets are reinforced and false dependencies are introduced between the heads and the tails of each replicated target.

- The simplest and most efficient approximation simply concatenates the target references together with no replication. The false dependencies at the boundary are no larger than the false dependencies in any other approach, while the approximate reference string is smaller.

Since the length of the reference string R, $|R|$, figures prominently in the cost of the distribution algorithm of the next section, we chose the last approximation for converting the reference expression to a reference string.

The approximation for the repetition operator involves choosing a value for θ, the repetition count. As earlier, the order of the distribution algorithm dictates a preference for small values. One possible option is to start with $\theta = 1$, compute the operand distribution for the corresponding approximate reference string, and then adaptively increase θ until the distribution stabilizes to an asymptotic value. This approach is expensive. On the other hand, the approximation assuming the constant value of $\theta = 1$ gives a short reference string that may not be accurate because the constraints of the loops are not appropriately reinforced.

The approximations to the conditional and repetition operators characterize the tradeoffs in the solution. An approximation to the reference string with more accurate constraints might entail a longer reference string and hence a more expensive distribution algorithm. However, as we shall see in Section 2.5, the cheapest approximations offer respectable payoffs, especially when compared to the alternative of distributing operands in their order of definition.

2.4 Complexity and Heuristics

It is NP-complete to optimally distribute scalar operands into lines so as to minimize the number of line misses incurred in satisfying the accesses of a reference string. The proof and a general discussion on the complexity of the problem can be found in Chapter 4. This section outlines a polynomial space and time heuristic and algorithm for the distribution problem. Section 2.5 documents the reduction in line misses when operands are distributed as dictated by the algorithm.

2.4.1 Distribution Algorithm

The cache contents at any stage during the execution of the program is nothing more than a cluster of M unsorted variables, called a cache cluster. A line miss occurs when an accessed operand is not available in the cache. The distribution of the operands into lines determines possible cache clusters; conversely, the clusters constrain the distribution in that only certain distributions can realize a given cluster. For example, for $M = 4, p = 2$ realization of cluster $abcd$ constrains the variables a, b, c, d to be distributed as either ab, cd, or ac, bd, or ad, bc. Note that the cluster's constraints may not define a unique distribution of operands into lines.

A distribution is desirable if it permits "good" clusters, where the merit of a cache cluster is quantified by its coverage or the number of accesses that it can satisfy. The algorithm is based on this heuristic notion of the desirability of a distribution. Starting with an equal probability for all possible distributions, the algorithm successively refines the distribution space until we have a unique distribution of the scalar operands into lines.

From the given reference string the algorithm first constructs the size–M operand clusters, each annotated by its coverage. Please note the following in the clustering step:

- The total coverage of a cluster is the sum of the coverages of all its occurrences in the reference string.

  Example: For $M = 4$ and $R = abcdecdbabde$, the cluster $abcd$ occurs two times with coverages of 4 and 6, $\underline{abcd}ecdbabde$ and $abcde\underline{cdbabd}e$, respectively. The total coverage of cluster $abcd$ is 10.

- The clusters overlap in the reference string.

Example For $M = 4$, the reference string $R = abcdec$ has the clusters, $abcd$ and $bcde$.

With the observation that a cluster is effectively a collection of constraints on the distribution, we now have a list of constraints on possible distributions. Since the merit of a cluster is manifest in its coverage, the algorithm successively chooses the cluster with the maximum coverage and refines the distribution to reflect the cluster's constraints. Conflicting constraints are ignored in a greedy approach.

Example: Given $M = 4, p = 2$, and $S = 6$ operands, a, b, c, d, e, f. Suppose the clustering step proposes the clusters $abcd, abce$, and $acef$ in decreasing order of coverage. Cluster $abcd$ constrains the distribution of a, b, c, d to be one of ab, cd, or ac, bd, or ad, bc. The remaining clusters now have to conform to, and choose one of, these distributions. Since each of the constraints of cluster $abce$ – ab, ce, or ac, be, or ae, bc – conflicts with each of the distributions selected by cluster $abcd$, the constraints of $abce$ are ignored. The constraints of cluster $acef$, however, can be resolved with those of $abcd$, and the operands will be distributed as ac, bd, ef.

The clusters whose constraints conflict and hence are ignored in the first pass, are given an opportunity, again in order of decreasing coverages, to influence the distribution in subsequent passes. The example below illustrates the utility of such passes.

Example: Given $M = 6, p = 2$. Suppose the clusters in decreasing order of coverage are $abcdef$, $adfghi$, $cdghij, \ldots$. Now the constraints of the second cluster conflict with those of the first cluster. The first and third clusters can be resolved into constraints $cd, abef, ghij$. Note that $abef$ is a compact representation for a choice between one of ab, ef or ae, bf or af, be. During the second pass, the second cluster can now refine these constraints to $af, be, cd, ghij$.

And finally, we present an account of the constraint accumulation or resolution process. The earlier observation that we start with equal probabilities for all possible distributions was only for purposes of illustration. The cost of such an implementation is $P!/(p!)^{P/p}P/p!$, *i.e.* exponential in P and p. Instead the constraints of a cache cluster are maintained as a line aggregate like $abef$ in the above example. Further refinement of the distribution breaks line aggregates into subaggregates until a subaggregate is the size of a line. This, then, is the final distribution of the p variables that constitute that subaggregate. An illustration of the process follows:

Example: Assume that $M = 6, p = 2$, and the three clusters with the widest coverage are $abcdef, abcdgh, cdegij$. Since we start with no constraints on the distribution, accumulation of the constraints of cache cluster $abcdef$ leaves us with the line aggregate $abcdef$. The resolution of the constraints of $abcdgh$ with those of $abcdef$ results in the line aggregates $abcd, ef$, and gh. Note that the aggregate $abcdef$ is now broken into the subaggregates $abcd$ and ef. The constraints of $cdegij$ now conflict with the constraint ef. The resolution with the constraint $abcd$, however, partitions the aggregate $abcd$ into the subaggregates ab and cd. All the subaggregates are now of size p, resulting in a distribution of the operands into lines ab, cd, ef, gh, ij.

The algorithm of Figure 2.3 is derived from the arguments above.

```
DistributionAlgorithm(int M, int p, char *R)

   GetClusters();            /* extract from R, overlapping clusters of size M */
   SortClusters();           /* sort the clusters in decreasing order of coverage */
   while (AggregatesAreBeingFractured)
      AddConstraints();      /* accumulate/resolve the constraints of the clusters */
```

Figure 2.3: Outline of the polynomial algorithm for distributing scalar operands into lines

For implementing the distribution of the scalar operands as established by `DistributionAlgorithm()`, the compiler simply reorders the sequence of data definitions before continuing with a definition-order scheme.

2.4.2 The Execution Order of the Algorithm

The relevant parameters are: $M, p, |R|, S, l, r$, where $l = \binom{S}{M}$, the total number of distinct cache clusters possible, and $r = \min(|R|, l)$. The cost of `DistributionAlgorithm()` can be seen to be the sum of the costs of clustering, of sorting the clusters, and of resolving the constraints in the two passes. The clustering and the cluster-sorting steps can be performed by sorting the M variables of each cluster, lexicographically sorting all the clusters on the basis of their contents and merging the coverages of the various occurrences of the same cluster, and then sorting the unique clusters in decreasing order of their total coverages. The costs of the steps above are $O(|R|M\log M)$, $O(Mr\log r)$, and $O(r\log r)$, respectively.

The cost of resolving the constraints of a cluster with the $O(p)$ line aggregates is $O(pM)$. With $O(r)$ clusters in each pass, and $O(p)$ passes, resolution of the constraints costs $O(rp^2M)$. Hence the total cost of `DistributionAlgorithm()` is $O(|R|M\log M + Mr\log r + r\log r + rp^2M)$. The cost of the algorithm can be expected to be dominated by the cost of the clustering step, *i.e.* $O(Mr\log r)$.

The NP-completeness proof for the distribution problem strongly suggests that an optimal algorithm for the operand distribution problem will have exponential costs. The cost of the algorithm above is polynomial. It is based on only a heuristic, and although the initial results reported in Section 2.5 are very encouraging, a cheaper and/or better distribution algorithm could very well be devised.

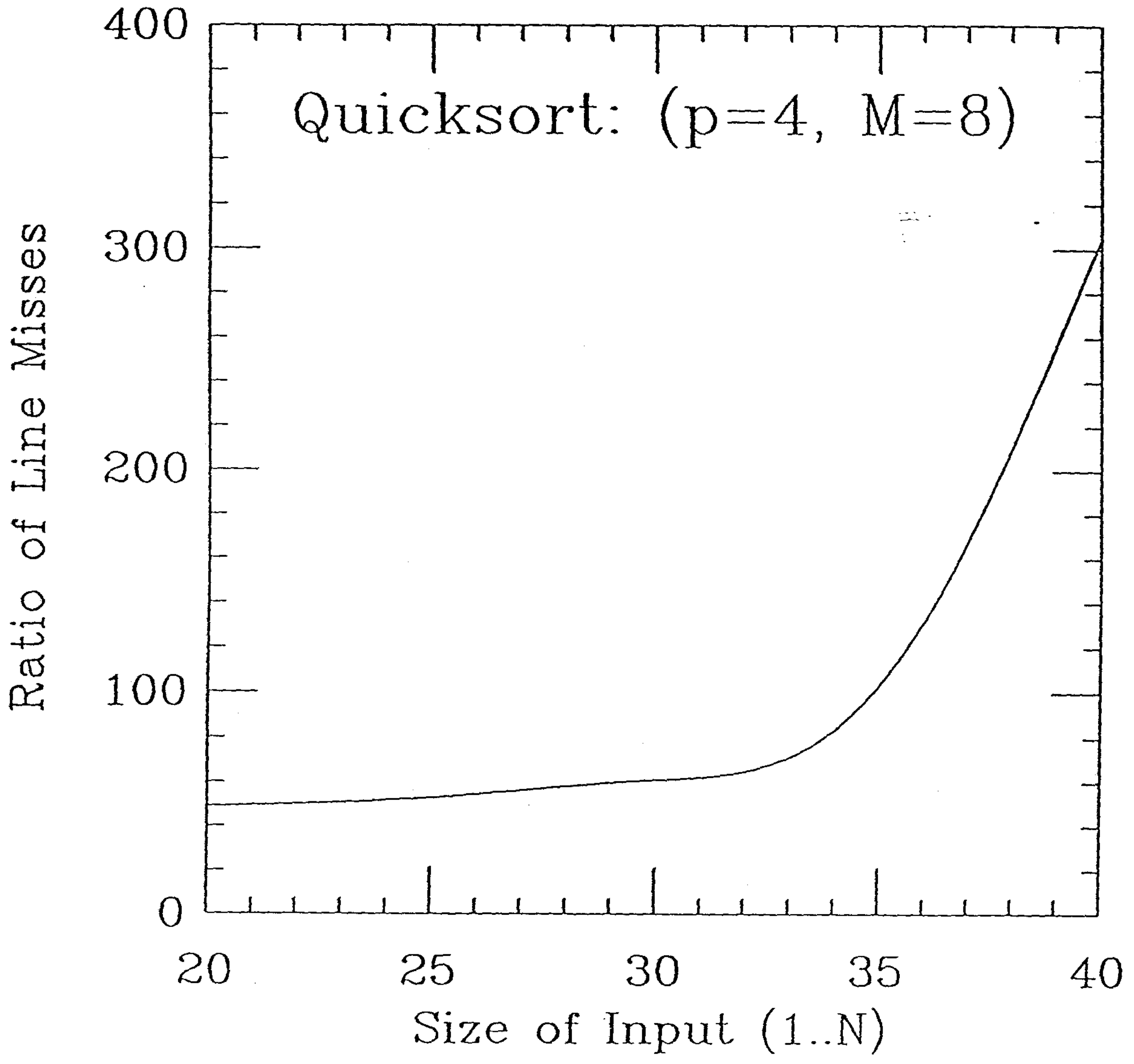
2.5 Initial Results

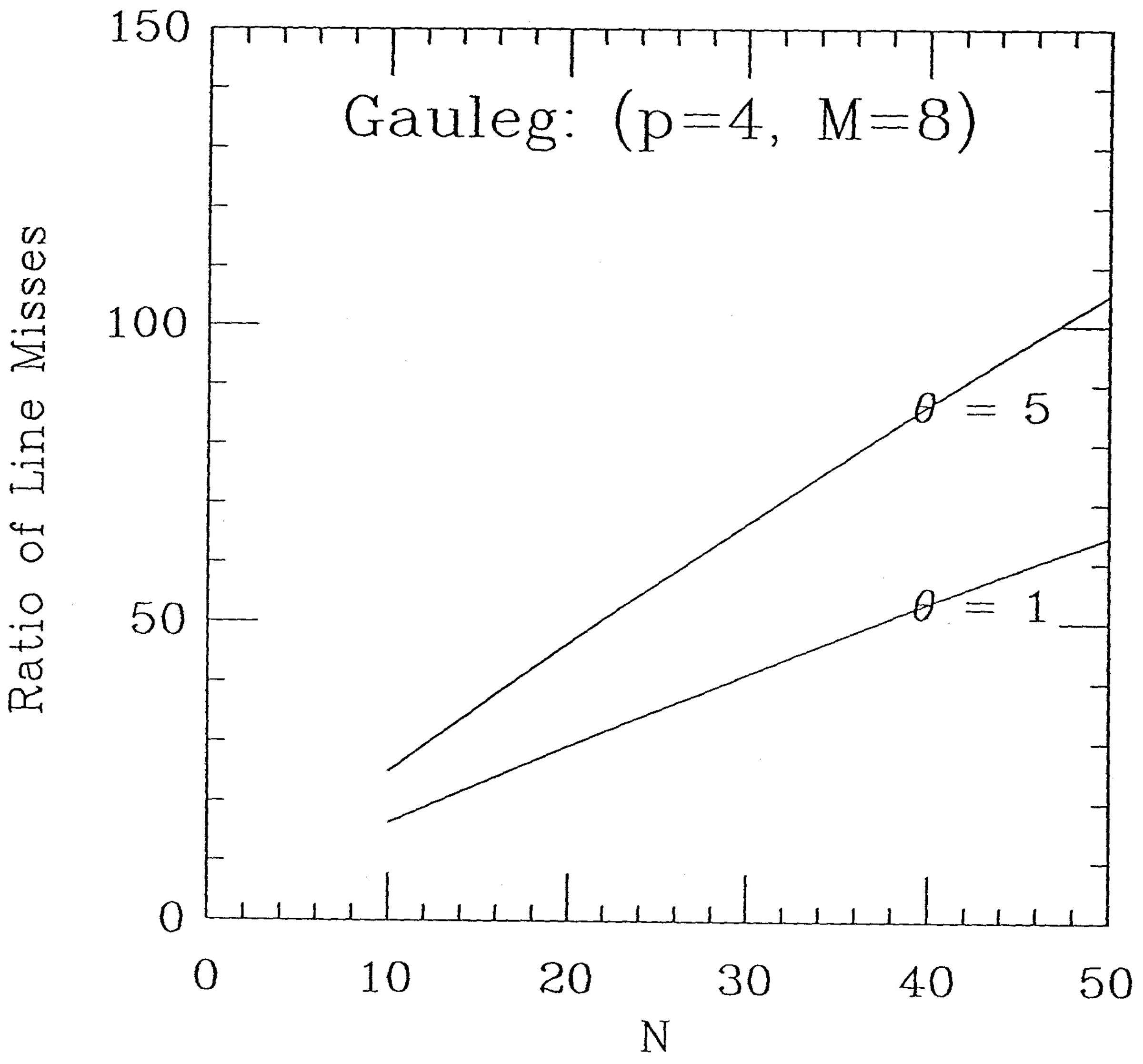
Representative scientific programs from *Numerical Recipes* [35] were chosen for testing the payoffs of a distribution of scalar data as computed by `DistributionAlgorithm()` versus that encoded in the order in which the operands were defined in the source. The programs include `quicksort`, `svd`, (singular value decomposition), `gauleg` (Gauss-Legendre integration). Since we are targeting the minimization of line misses, the scalars in the examples do not reside in registers. This exaggerates the burden on the communication and memory hierarchy, and permits experiments on small, albeit representative programs.

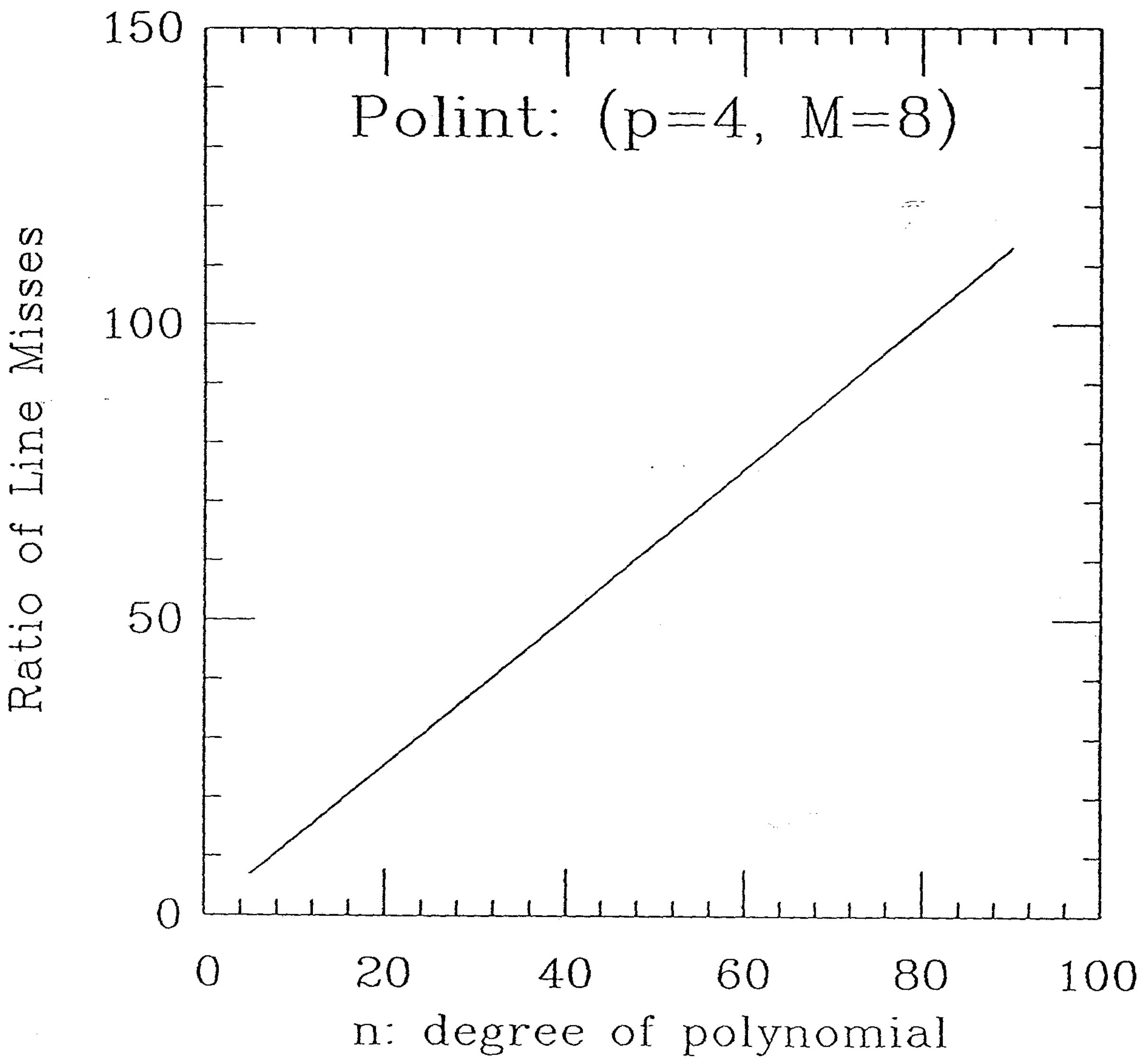
Once the scalars were distributed into lines as per the clustering and definition-order strategies respectively, the lines to be fetched and replaced were determined by the LRU replacement policy. The graphs represent the ratio of page faults incurred by the two distributions in satisfying the requests of the particular program, versus some program constant, *viz.* an input variable controlling the number of loop iterations. Values of θ, the repetition count in the approximate reference string, and of M and p, the system parameters were also varied. In all the experiments, the fully associative cache model was assumed. Although the LRU replacement policy was used for the tests, a random replacement policy, more relevant to caches, should offer similar improvements.

As the following graphs indicate, the reduction in line misses for all the programs is in the range 10 times $\approx$ 160 times, with an average around 40 times fewer misses. Also, even small values of θ offer very promising results.

Note that the reported results pertain to simulations of programs with relatively few scalar data, to memory environments with relatively small values of M and S, and with fully associative caches, and to a processing environment devoid of registers. The observations that the small sizes of M and S, and the full associativity of the simulated cache directly map onto one set of a typical cache, and that cache traffic typically corresponds to an overwhelmed register file lead us to expect that typical workloads and environments will exhibit similar payoffs of an access-driven distribution of scalar data when compared with the default definition-order distribution of the data.


Quicksort: (p=4, M=8)
Ratio of Line Misses
Size of Input (1..N)
0
100
200
300
400
20
25
30
35
40


Gauleg: (p=4, M=8)
Ratio of Line Misses
N
θ = 5
θ = 1
0
50
100
150
0
10
20
30
40
50


Polint: (p=4, M=8)
Ratio of Line Misses
n: degree of polynomial
150
100
50
0
0
20
40
60
80
100

2.6 General Applicability

The benefit of a judicious distribution of scalars into lines is directly related to the size of S, the set of scalar variables, relative to the values M, t, p. For a fixed M, t, p, the larger the size of S, the larger the potential payoffs. As a trivial example, if the number of scalar variables in the code is less than the number that can be accommodated on a line, then all distributions of those variables will cluster them on a line, and are indistinguishable. Note that the values of M, t, p are not restricted by `DistributionAlgorithm()`.

2.6.1 Relaxation of Assumptions

The assumptions of Subsection 2.2.2 serve only to focus attention on the central issues in this chapter. As the succeeding paragraph illustrates, a slack distribution problem can be reduced to a compact distribution problem, and therefore does not impact the validity of the ideas and approach developed. The uniqueness assumption is inconsequential to data optimization that follows code optimization; however, it is the conceptual basis for the input renaming optimization of Section 2.7. The assumptions of unit size, while pernicious from a bin packing perspective, only adds an additional level of detail to the distribution algorithm.

Slack distribution can be simulated by adding *virtual* or *filler* operands to the original operand space, *i.e.* enhancing the size of S with operands that do not appear in the reference string. A distribution that distributes an *original* operand with one or more filler operands is synonymous to distributing the original operand in a partially filled line. Thus the compact distribution of the resulting problem with enhanced S solves the slack distribution of the original problem.

The uniqueness assumption is not germane to data optimization that follows code optimization because the reference string, whether actual or approximate, is based on the accesses of the code. It is for the code, optimized or otherwise, to access the appropriate copy of a non-unique variable. For example, if a and a' are two copies of a variable, the reference string $a \ldots a' \ldots$ accesses them as two unique operands and is different from the reference string $a \ldots a \ldots$.

The assumption of unit size is easily addressed by the clustering approach of the distribution algorithm. When forming either the clusters or the line aggregates, the additional constraint on the sum of the sizes of constituent operands will have to be observed.

2.6.2 Applicability to Other Architectural Models

The reference expression and the approximate reference string are aimed at a good compile-time handle on the localities of the executing program. The clustering approach to the distribution of the scalar operands conforms to this compile-time access locality and the parameters of the memory hierarchy of the executing environment. The compile-time access locality can be similarly used to distribute the scalar operands to conform to a variety of other memory, processor, and system architectures.

We cannot over-emphasize that the memory hierarchy model is used only to illustrate our motivations and techniques, and our work is just as applicable, with or without modification, to most computing environments that reward locality. The compile-time approximations then permit the potential of exploiting such locality rewards. Just as our techniques reduce line misses in hierarchical memories, so will they pare remote accesses and global memory traffic in multiprocessor systems, improve the utilization efficiency of the potential memory bandwidth of interleaved memory systems, etc.

Akin to the lines of hierarchical memories, efficient data communication between the processors of a multiprocessor system is usually quantized. This amortizes the latency of establishing the communication channels, exchanging protocols, etc. Thus consecutively used data operands that are stored contiguously can be streamed through. Similarly, in interleaved memories, consecutively used operands are distributed across the memory modules so that successive accesses are to different modules.

From the perspective that the miss to any datum on a line initiates the entire line to be fetched and hence qualifies as a type of prefetch, judicious distribution of data aims at improving the efficiency of such prefetch. In the same token, the distribution of scalar data is equally applicable to the various fetch schemes, *viz.* demand-driven, prefetch, predictive, preemptive, etc., as it is to the various replacement policies, *viz.* LRU, random, FIFO, etc. This assertion rests on the observation that the fetch and replacement policies determine only the movement of lines between contiguous levels of the memory hierarchy, and have little bearing on the distribution of the scalar operands into these lines.

Also, the distribution techniques are equally applicable to C as to Fortran or Pascal because we are targeting only scalar variables. The indirection of C introduces problems mostly for array variables.

The values of M, t, and p reflect the parameters of the lower memory level in the memory hierarchy. Note that `DistributionAlgorithm()` assumes full associativity of the cache. Caches with restricted associativity can be modeled with small values of M, t and p. The first step in the hierarchical algorithm for such caches distributes the scalars into cache-lines based on the accesses in the approximate operand reference string. This operand distribution is then used to transform the approximate operand reference string to the corresponding approximate line reference string. The same distribution algorithm then distributes these lines into sets based on the line accesses of this approximate line reference string.

Similarly, interleaved memories can be modeled with $t = 1$, and $M = p =$ `InterleaveFactor`. The distribution of data to assist in communication between multiprocessors will analogously use $t = 1$, and $M = p =$ `WidthOfTheCommunicationPath/Bus`.

The application of the techniques developed in this chapter can reduce the cache requirements or the working set for the scalar operands, thereby mitigating the cost of cache flushes in multiprogramming environments. This benefit is independent of the sizes of M, t, p.

The examples and terminology so far have tacitly focused on the hierarchical memory interface between cache and primary memory, and truly scalar operands such as `int`*s* and `float`*s*. The payoffs at other levels in the memory hierarchy are just as promising. Conceptually, independent of the interpretation of line, lower level, and scalar, the cost of memory accesses of data elements depends on the grouping of those elements within the memory hierarchy. At any level, the relevant scalar data elements are such that the program accesses more than can fit within the interpretation of line corresponding to that level. At the primary–secondary memory interface, a "line" represents a page, and "scalar" variables are better represented by arrays or other aggregate data structures. Truly scalar operands are not germane at this interface since most system environments feature large page sizes, and most scientific code do not address more scalar operands than can fit in a page. This is not true of our aggregate "scalars".

The clustering of such scalars into a page is equally pertinent to reducing page faults, improving the utilization efficiency of memory and communication resources, and all the other ramifications of access locality outlined in this chapter. The methodology developed can then be applied to determine a judicious distribution of such scalars into pages. It can be congruently applied to distribute the data and code of entire subroutines into pages. This latter distribution resembles function overlays.

The judicious distribution of data is shown to improve the utilization efficiency of such diverse system features as caches, interleaved memories, communication channels, etc. From a different perspective, judicious data distribution by the compiler motivates a simplification in hardware. For example, a wise distribution of the scalar operands used within loops may avoid the need for full, or high levels of, associativity in caches.

Independent of the compiler distribution of data, or definition-order distribution of data, there are other benefits to compiler-awareness of the distribution. The operating system concerns itself only with physical measures, *viz.* pages. The compiler can translate changes in the operand reference pattern into corresponding changes in the page reference pattern, which the operating system can then use for initiating page migration and replication [36]. Even for operating systems that do not accept compiler directives, the compiler can sometimes circumvent the debilitating effects of statistical system policies.

A pertinent example concerns the LRU replacement policy. Consider $S = 10, p = 2, M = 8$, and the operands are distributed into the lines ab, cd, ef, gh, ij. For the reference string $(acegi)^n$, the sequence of line misses follows $(ab, cd, ef, gh, ij)^n$. Thus a total of $5n$ line misses are incurred. Such a repetitious reference string is likely to be produced by a block of the form:

```
for i=1 to n {
    e = p(a,c)
    g = q(i)
}
```

The *dot* and *repetition* operators of the reference expression can alert the compiler to such a sequence of accesses. Apprised of the distribution of the scalars into lines, and of the adherence to an LRU policy, the compiler can circumvent this worst case of the LRU by introducing pseudo-accesses with the sole intent of resetting the line usage counters of the LRU. The reference string will now more closely resemble $(acegacei)^n$. The sequence of misses follows $ab, cd, ef, (gh, ij)^n$. With the cost of a line miss substantially larger than the cost of a satisfied access, the cost of the pseudo-accesses can be ignored to reflect a saving of $3n - 3$ line misses.

2.7 Optimization of Scalar Data Storage as a Compiler versus a User Responsibility

This chapter has motivated the distribution of data, which reflects the accesses of the executing code. It advocated the machinations of the reference expression to obtain a good compile-time approximation to the reference string. The input source code is subject to numerous code optimizations, and the reference expression derived from this optimized code may be very different from that derived from the programmer's encoding of the algorithm. Note that it is the optimized code that is executed and that generates the accesses, and hence is more relevant to the distribution of scalar operands. It is for this precise reason that we assumed that the code is in a final, executable form.

If the user wants to arrogate responsibility for the distribution of scalar data, he may conveniently do so in a definition-order distribution scheme by appropriately sequencing the definitions of the scalar operands. However, the user's cognizance of the scalar distribution constraints extends only to the source code, and he is usually not privy to the code optimizations by the compiler.

Almost all code optimizations – code motion, common subexpression elimination, induction variable elimination, dead code removal, [3] to name a few – affect the data accesses of the program. At a different level, even intermediate code usually exhibits access sequences very different from the source code. Since the temporaries of the intermediate code quadruples mutate the source code and the reference string, simulations of only source-to-source transformations will be used in illustrating code optimization effects on data distribution.

The following examples contrast the distribution constraints of the source code with those of optimized code resulting from the constant folding and induction variable elimination transformations.

Example: (source code)

```
for i=1 to n {
   b = 4*i
   A[f(b)] = g(B[f(b)])
⋮
```

The source code encourages operands b and i to be stored contiguously. However, the application of constant folding and induction variable elimination transforms the code to:

(optimized code)

```
b = 0;
for i=1 to n {
   b = b+4
   c = f(b)
   A[c] = B[c]
⋮
```

The transformed code encourages operands b and c to be stored contiguously. Thus the distribution constraints derived from the source code are very different from those derived from the optimized code.

User-optimized distribution of scalar operands is usually counterproductive. If the compiler regards the user's assertions as inviolable, it may constrain the code optimization so that the accesses of the executable code map this distribution. This is clearly undesirable. Alternatively, the code optimization may ignore the data distribution, and the data accesses of the optimized code will likely bear no resemblance to the user-optimized data distribution. From the perspective of the execution efficiency of the code, the most advantageous alternative corresponds to the compiler maneuvering the data distribution to map the accesses of the optimized code.

Furthermore, from the clustering rationale of `DistributionAlgorithm()` we note that the optimal data distribution is driven by the parameters of the memory hierarchy, *viz.* M, t, p. Programs with user-optimized data will not be portable across systems. Also, they suffer from low readability, maintainability, etc.

With the compiler responsible for both code and data optimization, the user, free from this lower level optimization detail, can apply himself to the optimization of higher levels, such as algorithm or language choice, that are beyond the scope of the compiler. Besides the ease of programmability, this specialization can lead to more efficient programs – the compiler can now freely restructure and optimize the code, and then optimize the distribution of the data to map the accesses of this final, executable code.

The assumption that the reference expression and data distribution constraints are extracted from the code-optimized output of compilers supports the applicability of the data optimization techniques to most compilers. The existing code optimization can proceed in utter disregard of any data distribution criteria, and the data can be judiciously distributed thereafter. Section 2.5 illustrates the respectable payoffs of this approach with independent code and data optimizations.

However, it is conceivable that some of the data distribution constraints may be adopted in the optimization of code to produce code and data organization that are synergetically optimized. The payoffs of such an interdependent approach are potentially greater. One possible application of scalar distribution constraints is to influence the code generation or linearization of the DAG [3]. In the following example we are again ignoring the effects of intermediate code temporaries and of register allocation:

Example:

```
for i=1 to n
   a = f(b)
:
for i=1 to n {
   p = a + q
   s = t + v
   b = m - n
   l = r - w
:
```

The first loop encourages operands *a* and *b* to be stored contiguously. Devoid of distribution constraints, the instructions of the second loop have no preferred order of execution. The corresponding reference expression, $(\underline{a}qptvsmn\underline{b}rwl)^n$, will distribute the operands into the $p = 2$ lines aq, pt, vs, mn, br, wl. However, the bias towards the line ab derived from the first loop, can motivate the rearrangement of the second loop to conform to this propensity. The rearranged second loop, shown below, now generates the reference expression: $(mn\underline{ba}qptvsrwl)^n$.

```
for i=1 to n
b = m - n
p = a + q
s = t + v
l = r - w
```

2.7.1 New Code Optimizations Motivated by Data Distribution Criteria

Given the high cost of a line miss, the distribution of scalar data can also be seen to motivate new code optimizations with the objective of reducing line misses. One such code transformation, "input renaming" or "constraint splitting," is exactly the converse of the traditional copy propagation optimization [3]. It creates copies of a variable which is subject to conflicting distribution constraints, and splits these constraints onto the copies thus avoiding the conflicts and the concomitant cache misses.

The "dynamic restructuring" or "constraint coalescing" transformation implements a form of dynamic distribution of the scalar variables into lines. Instead of physically redistributing the variables into lines that simplify access, it attempts to redirect accesses by redefining operands so as to exploit the static distribution.

Constraint Splitting

A new copy of a variable is introduced for the purposes of partitioning the clustering constraints on the variable. Consider the following piece of code:

```
for i=1 to n {
    b = $f$(a)
    d = $g$(e)
⋮
}
⋮
for i=1 to n {
    c = $h$(a)
    d = $k$(e)
⋮
```

For purposes of illustration assume that the above code is to be executed in an environment where $M = 4, p = 2$. Operand a is constrained to share a line with b as much as with c. Either alternative will result in $n + 3$ line misses for the two subloops. However, if we could accommodate two copies of operand a, then each copy could satisfy one of the two sets of constraints independently. The following transformed code incurs 5 misses if the operands are distributed into the lines $ab, a'c, de$.

```
a′ = a
for i=1 to n {
   b = $f$(a)
   d = $g$(e)
⋮
}
⋮
for i=1 to n {
   c = $h$(a′)
   d = $k$(e)
⋮
```

In most programs, multiple independent loops very often reuse the iteration counters. Such variable overuse introduces pseudo-dependencies that are alien to the algorithm.

The parallel between input renaming and a common technique for register allocation is conspicuous: During code generation assign every destination or left-hand-side of a quadruple to a unique register thus effectively starting with an infinite register set, and later, on the basis of lifetimes and usage counts, fold these register assignments onto the actual register set available. In the extreme case, the input renaming transformation prompts every use of an operand to access a unique copy, and to later fold the copies into an acceptable set.

One way to deal with unique copies of a variable follows the clustering techniques. First, create the clusters as in `DistributionAlgorithm()`, then combine clusters such as $abcd$, and $a_1bc_2d_4$, where $\forall i, a, \ldots, a_i$ are copies of variable a under the same lifetimes. The way to get around the coherency problem might involve introducing appropriate reassignments, *e.g.* if the two copies of b in $b = \ldots$ and $\ldots = f(b')$ incur fewer misses than the reuse of the same copy in $b = \ldots$ and $\ldots = f(b)$, then the added assignment $b' = b$ after the first expression resolves the coherency issue.

This added assignment might incur an extra miss, but that will be compensated by the savings of misses for the second use of b.

Note that the input renaming transformation may affect other equally important attributes of the code. It will usually, for example, increase the size of the code, and of the scalar data space. If the size of the scalar data space is restricted, the compiler can implement a form of dynamic distribution of the scalar operands into lines. Instead of physically restructuring the lines, the compiler can conjure the same effect by redirecting the accesses to variables in the dynamic restructuring transformation.

Dynamic Restructuring

Consider the reuse of variable a in the code fragment and $M = 4, p = 2$ environment as above. Assume that operand b is of the same type, . *e.g.* `int`, as operand c. Consider the following transformed code:

```
for i=1 to n {
   b = f(a)
   d = g(e)
⋮
}
t = b
⋮
for i=1 to n {
   b = h(a)
   d = k(e)
⋮
```

If all subsequent accesses to operand b are redirected to access operand t, and then all subsequent accesses to operand c are redirected to operand b (as was the transformation of `c = h(a)` to `b =`

h(a)), the computation by the program code remains unchanged. The transformed code incurs only 5 misses if the operands are distributed into lines ab, de. Note that the total number of scalar operands is the same.

The implementation of dynamic restructuring requires a close inspection of def-use chains and other flow analyses by the compiler. Hence, code and data should be optimized in synchrony, and they should reflect constraints of both the processing, and the memory environments.

2.8 Future Work

The ideas documented in this chapter stimulate many avenues of further research and continuing effort. A few appear below:

- Although `DistributionAlgorithm()` operates on a linear reference string, the clustering approach succinctly and effectively encodes locality information, and similar techniques can be employed for a number of other optimizations, old and new.

  The linearization of DAGs to exploit distribution constraints is a typical example. The clustering algorithm would proceed along the lines: Assume each DAG to be a cluster, and partition the clusters until the subaggregates are of size 3 (representing quadruples, 2 for triples, etc.). Note that unlike the line partitioning case, these clusters may be partially ordered, and we have to confirm that each cluster partition is valid, *e.g.* for the DAG generated by the expressions `a = p(b,c); d = q(e,f)`, the partition into aef, bcd is not valid.

  This requires the development of a clustering algorithm on graphs.

- While the input renaming transformation is shown to reduce line misses and an algorithm compatible with `DistributionAlgorithm()` has been outlined, the transformation needs further evaluation. Similarly, although the dynamic restructuring transformation is effective in reducing line misses, a polynomial algorithm for its implementation and further evaluation are warranted.

- By virtue of its knowledge of the distribution of scalar data, the compiler can participate in other optimizations. For some it can provide directives to the operating system [36] or to the loader for implementing distributions of functions into pages, etc. This requires further research in identifying the optimizations, and a better understanding of the tradeoffs involved.
- One source of further work might involve an evaluation of alternative techniques for the approximations to the reference expression, and for the distribution of the operands into lines.
- Most importantly, the techniques need to be incorporated into a working compiler.

2.9 Conclusions

The compile-time nondeterminism of accesses to scalar data is free of the subscript ambiguity plaguing array data, and is limited to the nondeterminism at conditionals and loops. The reference expression encompasses this nondeterminism and motivates various approximations for judiciously distributing the scalar data to map the accesses of the code. The approximations can be successively refined, *i.e.* the finer the refinement, the higher the cost. The accuracy of even the inexpensive approximations is evident in the payoffs of between 10 and 100 times fewer cache misses.

The hierarchical memory interface between cache and primary store was used to illustrate the ideas and techniques. The methodology is equally applicable, albeit with a different interpretation of the terms, to other levels in the hierarchical memory, and to all other environments that reward locality. These include multiprocessor systems, interleaved memories, etc.

The distribution of the data is complementary to other system, code, and algorithm optimizations. It is dependent, however, on the patterns of access in the optimized code and although the experimental results reported assume that data optimization is attempted after all code optimizations, the payoffs of the concurrent optimization of code and data are potentially greater.

From an abstract perspective, the distribution of scalar data as motivated in this document encodes global constraints whereas many of the code optimizations, peephole optimization being the quintessential example, exercise only local information. The discontinuity prompts the application of data distribution criteria to the optimization of code; this premise is the basis of a revaluation of traditional code optimizations, and of a synthesis of new code optimizations. All told, the optimization of code and data to improve the turnaround efficiency of programs is best attempted by the compiler and not the user.

In summary, the process of compiler optimization of scalar data proceeds thus:

1. Derive the reference expression from the source code.

2. Implement approximations to transform this reference expression to an approximate reference string.

3. Apply `DistributionAlgorithm()` to this approximate reference string to obtain a judicious distribution of the operands into lines.

4. Contrive with, or around, the default operand distribution scheme to distribute the operands as decreed by the previous step. As an example, if the default scheme distributes the operands in the order in which they are defined, then rearrange the data definition appropriately.

5. Wherever possible, exploit this compile-time distribution information in other optimizations, *e.g.* introduce pseudo-accesses so as to circumvent pathological worst cases of the replacement algorithm.

Chapter 3

Compiler Optimization of Array Data Storage

3.1 Introduction

This chapter targets the storage organization of arrays or the physical layout of arrays in memory. The distribution of entire arrays relative to other data has been discussed in the previous chapter. If the elements within an array are perceived as individual scalar elements, then they can be distributed in a likewise manner. However, the conglomeration of the elements into one array entity concisely abstracts the indexing and use of the array. For example, the loop, `for i=1 to n {A[i]=0}`, is an abstraction for `A[1]=0; A[2]=0; ... A[n]=0`. If the individual array elements are distributed as scalars and do not adhere to any schema, then we have effectively expanded all loops that sequence through the data array. Such code explosion is usually infeasible. Note that the distribution schema may be different from the source schema.

The methodology and algorithm for optimizing the storage of arrays in declarative languages, more accurately data-array languages, *viz.* Fortran 8x or APL, is different from that for procedural languages, more accurately scalar languages, *viz.* Fortran 77 or C. It should be pointed out that the terms declarative and data-array, and the terms procedural and scalar are used interchangeably in this document. The "remap" algorithm implements the former optimization of array storage, while the suite of "stock" techniques, also employed by the remap algorithm, implement the latter. The algorithms direct the storage of the data arrays to judiciously map the various, sometimes conflicting, patterns in which the array is accessed by the code.

Figure 3.1 illustrates the different ways of storing a matrix. Most compilers today assume and expect that arrays are stored in a fixed order, "ravel." In the case of Fortran 77, ravel order for a matrix may be column-major, while for Pascal this may be row-major. For the purposes of this chapter, ravel order will imply row-major. Note that the axis that moves (unravels) slowest is referred to as "major"; conversely the axis that moves fastest is referred to as "minor." The significance lies not in the difference between the interpretations of ravel, but in the precanned rigidity of a storage organization that is independent of the eventual access of the data.

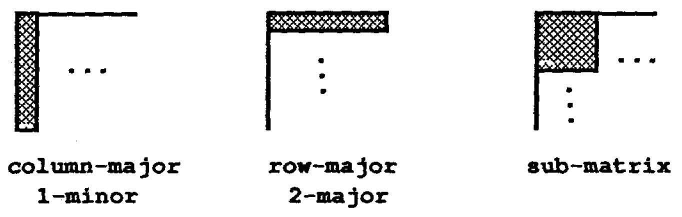


Figure 3.1: Storage Organizations for a Matrix

3.1.1 Motivation

The multiplication in a paging environment of two n-dimensional square matrices, A and B, demonstrates the payoffs obtained by the storage of data, which is prompted by the use of the data, versus the storage of data that is always ravel and independent of data access. Figure 3.2 indicates the typical nested loop that performs the matrix multiplication. The pseudo-code makes it abundantly clear that the optimal storage order for matrix A is row-major and that for matrix B is column-major.

```
for i=1 to n
   for j=1 to n
      for k=1 to n
      Result[i][j] = Result[i][j] + A[i][k] * B[k][j]
```

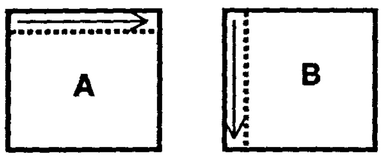


Figure 3.2: Typical Nested Loop for Multiplying Two $n \times n$ Matrices

For a page size of p, and a working space of 3 pages, Table 3.1 gives the expected number of page faults for three different values of n and for the following data storage organizations: ravel, optimal, and submatrix. The page fault computation assumes no optimization or rearrangement of the code; the sequencing in the pseudo-code is followed strictly.

From the table we can see that the improvement in page faults in going from ravel to optimal storage is approximately n, the dimensionality of the square matrices. The improvement in going from submatrix to optimal is approximately $2\sqrt{p}$, where p is the page size.

n	A row major B column major [1]	A row major B row major [2]	A submatrix B submatrix [3]	Improvement Ratio $\frac{[2]}{[1]}$	Improvement Ratio $\frac{[3]}{[1]}$
64	264	16392	16512	62	62
128	2080	262176	131584	126	63
256	16512	4194432	1050624	254	63
N	$(N+2)\frac{N^2}{p}$	$(N^2+2)\frac{N^2}{p}$	$(2N+1)\frac{N^2}{\sqrt{p}}$	$O(N)$	$O(\sqrt{p})$

Table 3.1: Number of page faults generated by the multiplication of two $n \times n$ matrices (p=1024, number of pages available=3)

3.1.2 Related Work

In the literature, most optimizations for improving the efficiency of memory hierarchies prefix the storage organization of operands and either restructure the code in the compiler-based approaches, reorganize and recode the algorithm in the algorithmic approaches [30], or tune the page prefetch and replacement policies in the operating system approaches [8, 9, 15, 39] . The few papers that discuss alternative pagination of arrays [12, 16], for example, emphasize the costs and dynamics of transforming between alternative organizations for specific algorithms.

Even the skewing schemes for distributing arrays over interleaved memories [11, 22, 26, 38] restrict the dimensionality of the operand array and the patterns in which the array may be accessed without bank conflicts. With few exceptions, there appears no general methodology for automatically determining efficient, code-driven storage organizations of operands and intermediate results.

Mary E. Mace in [27, 28] details an algorithm for determining the optimal storage organization of array operands from a given program graph. Her dynamic programming technique assumes a collapsible program graph furnished with tables that encode the cost of operations for all combinations of storage patterns of operands and result. We demonstrate later in this chapter that her program graph and techniques are suitable mostly for declarative languages. The issues relevant to procedural languages in which all computation is explicitly sequenced, are different. Even for declarative languages, the cost tables are exponential in the dimensions of the subject arrays. Moreover, the restriction to collapsible program graphs severely limits the applicability of her approach. As an example, the program graph of the following computation is not collapsible.

```
c = f(a,b)
d = g(a,b)
result = h(c,d)
```

From that perspective, our techniques attempt to judiciously, even if not always optimally, store array data to map the accesses of unrestricted code of both declarative, and procedural programs.

The Fortran 77 domain has benefited from a vigorous optimization effort aimed at improving the utilization efficiency of memory resources. The loop manipulations of [1, 5, 17, 25, 31, 33] are very effective at vectorization and parallelization or concurrentization of code, and at maximizing the reuse of data in fetched pages. The transformations, however, are handicapped by the rigid, prefixed storage organization of Fortran 77. For example, the loop interchange manipulation can sometimes mask, or ameliorate, the adverse effects of the abiding column-major storage. A loop interchange so necessitated will in many cases compromise the vectorizability of the code, or the complexity of the other manipulations. The storage restriction of Fortran 77 appears to have fettered research in manipulating the storage organizations of array data to be germane to the code.

The work described in [23] is similar in its goals and spirit with that of this chapter. However, they exclusively target SIMD architectures and a data-array source language, in particular the Connection Machine and Fortran 8x, respectively. In contrast, this chapter presents a methodology for the general problem, and a comparison, in the context of compiler optimization, of data storage optimization in scalar languages like Fortran 77 vis-a-vis that in data-array languages like Fortran 8x or APL. This exposes the relevance of data optimization techniques to programs coded in either type of language.

Note that our goal does not preempt other compiler or algorithm optimizations for improving program attributes. This is apparent when we observe that the objects of the respective optimizations are different – in our case it is the physical distribution of data, and in the other cases it is code. Our work complements rather than outmodes other compiler optimizations. For example, some of the loop distribution manipulations often maximize data reuse in ways not targeted by our work.

The code does affect the optimal distribution of data by defining the patterns in which it accesses the data. Therefore data storage and code optimizations are inextricably linked and it is conceivable, and as we shall see also advisable, that some of the data organization considerations may be adopted in the optimization of code to produce code and data organization that are synergetically optimized. For the purposes of this document we will assume that the code is in a final, optimized form.

3.1.3 Outline of the Chapter

Section 3.2 discusses the efficacy of data optimization in declarative versus that in procedural languages. It motivates the program graph and outlines the remap algorithm for accumulating all storage constraints, direct and indirect, on every array operand. Given all the constraints that impact the storage of a data array, the resolution of these constraints for the array is identical to both types of languages, and is targeted by the stock suite. Section 3.3 discusses the general applicability of array data optimization to diverse system architectures. It outlines a technique for optimizing the storage of arrays in Fortran 77 in spite of its column-major constraints. Section 3.4 relates our work to other compiler optimizations and argues that the compiler-optimized approach is more prudent than the user-optimized approach to storage organization. It introduces code optimizations driven by data storage constraints. Section 3.6 concludes this chapter.

3.2 Computing Storage Order Constraints

3.2.1 To what level can data distribution map data access?

The previous section motivates the storage of data in the order in which it will be accessed. Furthermore, this thesis advocates that the compiler be responsible for determining data access and appropriately distributing the program operands. This naturally restricts the resolution to which the eventual data access can be determined, *e.g.* elements of array operands accessed within conditional statements or within dynamically bound loops cannot be *completely* disambiguated at compile-time. But in most cases it is possible to determine at compile-time the *relative order of the axes*, or the pattern in which the data will be accessed, *e.g.* the compiler may be able to deduce that axis i of operand A moves faster than axis j, which in turn moves faster than axis k, and so on. This relative speed of the indices indicates that operand A stored in i-minor, j-next-to-minor, k ... storage order maximizes the probability of addressing contiguous memory locations in consecutive accesses.

Note that such storage organization of operand A by itself does not guarantee optimality over all system architectures. Even in commonplace virtual memory systems the "optimality" of data organization is dependent on the sizes and numbers of operands and pages. In interleaved memories, the interleave factor is an architectural characteristic that affects the optimality of data organization. In vector machines it is the vector length, while in multiprocessors, relevant architectural constructs determine task distribution and include the number and type of processors, the processor connectivity, and communication bandwidth.

However, the ability of a compiler to determine and manipulate operand storage is critical to the efficient execution of programs, as was exemplified by the matrix multiplication of Section 3.1.1. This is a good juncture to mention that in the absence of any system details, the rest of the chapter will use the following definition of optimality for the organization of operand storage: Consecutive accesses by the computation should address contiguous locations in the memory space. This definition coincides with the traditional implications of "locality."

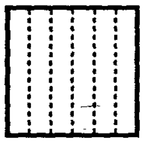
In scalar languages like Fortran 77, which require the programmer to explicitly sequence all computation in terms of scalar operations, the looping and indexing constructs readily provide the compiler with the relative ordering of operand axes. Data-array languages like APL or Fortran 8x, on the other hand, provide the programmer with declarative operators that specify the form of the final result without spelling out the computation sequencing in detail. This flexibility in computation sequencing translates to a corresponding flexibility in the optimal storage organization of the operands.

A few of the data-array operators, however, partially specify the sequencing of the constituent scalar operations over the operands. Some others do not encode any computation; instead, they reorganize their operands. Reduction is typical of the former, and transpose is typical of the latter class of data-array operators. The individual effects of these two operators on optimal operand storage will be studied in the next section as a prelude to their use in illustrations of the working of the remap algorithm.

3.2.2 Reduction and Transpose

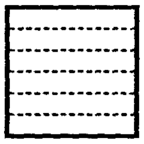
In the general form of the reduction operator, reduce(f, A, i), the scalar function f is applied to reduce the array operand A along the i^{th} axis. If the compiled code is executed in a multiprocessing environment, A will be optimally stored in i-minor order. For $A[m,n]$, the execution of reduce($+, A, 1$) will mirror,

```
doall j=1 to n
    B[j] = 0;
    for i=1 to m
        B[j] = B[j] + A[i][j];
```



and $B[1:n]$ will be returned as the result. If the execution environment is vector-processing, A is optimally stored with i as the major axis. The execution will now mirror,

```
B[1:n] = 0;
for i=1 to m
    B[1:n] = B[1:n] + A[i][1:n];
```



In the general form of the transpose operator, transpose(v, A), where v is a vector containing a permutation of integers $1 \ldots$ rank(A), and rank(A) returns the number of dimensions of A, the vector v directs the transposition of the axes of A. Although transpose does not introduce any new storage constraints, it does affect the determination of the optimal storage order by remapping constraints. As example: if rank–3 operand A is stored in 1-minor, 2-major order, the optimal storage order for transpose((3 1 2), A) is 3-minor, 1-major; while the result of transpose((1 3 2), A) is optimally stored in 1-minor, 3-major order. From Figure 3.3 it becomes apparent that as we transpose the axis at the i^{th} position to the j^{th} position, the relative storage order, in the major–minor spectrum, associated with the i^{th} position is also transferred to the j^{th} position. In other words, the transpose operator transposes the storage attributes in tandem with the shape.

A tabulation of the constraints that APL operators impose on the optimal storage of their

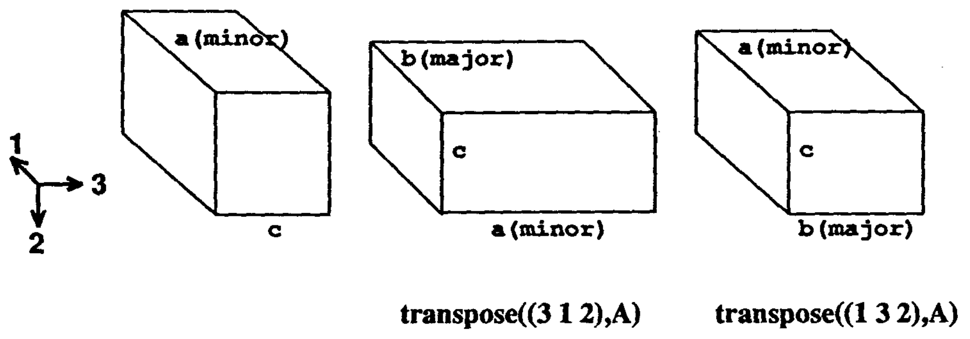


Figure 3.3: Storage Constraints of the Transpose Operator

operands and results can be found in appendix B. APL has been chosen to illustrate our techniques because it is the prototypical data-array language, and because Fortran 8x [7] has adopted, and continues to adopt, many of the APL array operators.

3.2.3 Are Data-Array Languages Especially Amenable to Data Optimization?

The data optimization proposed in this chapter relates the storage organization of an operand to the patterns in which the operand is accessed. It is tacit that we target array operands. Independent of the programming language, every operator or programming construct in the code that accesses an array operand constrains the storage of the operand. In that sense the constraint manipulations developed in this chapter span programming languages. Only the ease of extracting at compile time the sequence of array accesses varies with the language and its use. As an example, some languages like C encourage indirection through pointers in accessing array operands. Compile-time data optimization of programs written in such languages is naturally more complex. Fortran 77 programmers tailor algorithms to the restricted column-major storage of array operands, and the dexterous programmer sometimes exploits such peculiarities, *e.g.* assuming a linear layout of arrays in contiguous memory, making data storage optimization by the compiler counterproductive. The applicability of storage optimization is subject only to the deduction of access sequences. In terms of the efficacy of the optimization of data storage, however, if data optimization follows the optimization of code by transformations isolated from data distribution criteria, declarative, data-array languages do nurse benefits over procedural, scalar languages.

The looping, nesting, and indexing constructs of scalar languages manifest operand array accesses just as transparently as do the semantics of array operators in data-array languages. However, many data-array operators are effectively recurrence-free `for` loops iterating over entire array extents, and data-array languages are characterized by a preponderance of array operands. This minimizes the use of the indirection in accessing arrays, which obscures the compile-time extraction of the patterns in which scalar code accesses the arrays.

The critical advantage of data-array languages, pertinent to storage optimization, rests in the flexibility in computation sequencing that is permitted by array operators. This translates to a corresponding flexibility in optimizing the storage of arrays. In contrast, scalar languages overconstrain the sequencing through arrays. Consider the following initialization fragment in a scalar language other than Fortran 77. We address Fortran 77's column-major restriction in a later section.

```
for i=1 to n
   for j=1 to n
      A[i][j] = 0;
```

In the absence of dependency constraints, the compiler has no motivation to interchange the two loops in this code. Even if all other accesses to array A are column-major, since the storage optimization algorithms take the optimized code as immutable, the relative nesting of loops and indices in this code constrains the storage of A to be row-major; and that is indeed the pattern in which this initialization fragment will access array A.

Consider the same initialization in a data-array language, $A = 0$. Global accesses of array A may lead the remap algorithm to conclude that A is best stored in column-major order. Since the initialization expression does not constrain the access and hence the storage of A, it can be stored in column-major order to satisfy the global constraints. The eventual execution of the initialization code will then adhere to the column-major storage. Note that the loops in the equivalent scalar encoding of the initialization have been effectively interchanged. In exploiting the sequencing flexibility to optimize the storage of the arrays, the remap algorithm has tacitly optimized the sequencing without impacting the declarative code optimization.

From this perspective, data optimization in declarative languages is more effective because even though the data is optimized after the code is optimized, the global storage constraints of arrays can affect the sequencing of the computation. The equivalent loop manipulations can be effected for scalar languages only if the data storage constraints are allowed to motivate the code optimization. In that case, the efficacy of storage optimization in array languages can be matched by that in scalar languages. However, even though data optimization that follows code optimization in scalar languages offers lower payoffs than the data optimization that follows code optimization in array languages, the improvement in the efficiency of the communication and execution systems is significant, especially when compared to the alternative of a fixed ravel storage of all arrays.

3.2.4 The Program Graph

The input to the remap algorithm and the stock techniques is the output resulting from the application of all traditional optimizations on the source code. The output of the storage algorithms is an encoding of the optimal storage organization for each data array in the source code. This storage organization is globally optimal, *i.e.* it is a resolution of all the constraints that directly or indirectly affect the optimal storage organization of the array.

The reduction and transpose operators illustrate the constraints that the array operator may introduce on the optimal storage of its operands and its result. The result or a co-operand may similarly constrain the optimal storage of an operand, and vice versa. As an example, consider the arrays A, B, and C. The array sum in the data-array expression, $A = B + C$, constrains all three arrays to have the identical storage organization. These constraints are transitive; if some other expression in the program relates the storage of C to the storage of D, then the storage constraints on D will constrain the storage of A and B through its constraints on the storage of C.

This motivates the program graph, a graphical representation of the relationships between operands, both intermediate and atomic, and results, both partial and program. In the program graph:

1. The nodes represent the operators in the program computation, and the arcs represent the array operands. Intermediate operands are also the partial results of the computation. The arrays input to, and output from, the program may be constrained to conform to a specific storage organization; hence the `IN` and `OUT` operators.

2. The arcs into a node represent the result of the operator at the node, with the arcs out of the node representing the operands to the operator. Monadic operators have a single arc exiting the node.

An illustration of a data-array program segment and the corresponding program graph may be found in Figure 3.4. Through the transitivity of storage constraints, all operators and operands in the program graph influence the optimal storage of all arrays in the program.

The program graph and constraint transitivity relevant to the computations encoded in data-array languages have no parallel in scalar languages. In scalar languages the constraints on storage are explicit in the procedural operators that explicitly sequence all computation. These operators constrain the storage of each operand and result in isolation of each other. For example, the constraints are embodied in the sequence in which the nested loops access an array operand. For example in the following code:

```
A = B + C

C = E + D

F = A - D
```

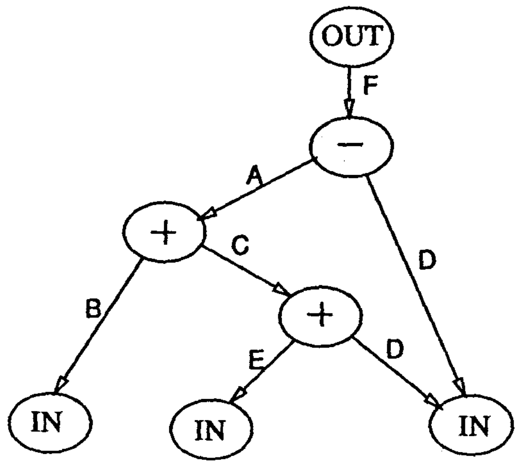


Figure 3.4: Code segment and the corresponding program graph

```
for i=1 to m
    for j=1 to n
        A[i][j] = B[j][i]
```

the storage of array A is constrained to be row-major and that of B to be column-major independent of each other. The constraints flow directly from the looping, nesting, and indexing operators to the operand and result arrays. The corresponding code in a declarative language, $A =$ `transpose`(B), on the other hand, invites a program graph that either constrains A to be stored in row-major order and B in column-major order, or A in column-major and B in row-major, or both in submatrix. Thus the constraints on the operand and result are relative to each other. This assertion rests on the observation that the program graph constraints and the sequencing freedom of declarative languages motivate the interchange of the loops encoding a computation, and the interchange affects all the arrays accessed within that fragment of code.

Note that if the optimization of code in scalar languages is motivated by data constraints, then the program graph is equally pertinent to scalar languages, and as argued in the previous subsection the distinction between data optimization in array and scalar languages is blurred. However, we are assuming that all code optimization precedes data optimization. Thus for procedural languages the stock techniques optimize the storage of an array to the constraints of every access to that array, in isolation of the constraints and optimization of other data arrays. For declarative languages the remap algorithm accumulates all direct and indirect, operand and operator, constraints on an array. The resolution of these multiple, potentially conflicting, storage constraints at each array operand of the declarative program graph can now utilize the same stock techniques. The interdependencies of the remap algorithm intuitively point to its complexity. Chapter 4 proves that it is indeed NP-complete.

3.2.5 Overview of the Remap Algorithm

In determining the optimal storage organization for the operands in an expression, the remap algorithm maintains and manipulates a vector, the "q–vector." The q–vector accumulates constraints on storage order; remember that these constraints specify the preferred relative ordering of the operand axes. This accumulation of constraints mirrors the composition of linear transformations. The particulars of the encoding of the storage constraints in the q–vector are quite simple and are explained in the succeeding subsection. For the present, assume that operand storage constraints are adequately described by the contents of a vector.

The remap algorithm associates a q–vector with every operand array, atomic or intermediate, in the program computation. The starting q–vectors encode the initial constraints on the storage of the corresponding array, *e.g.* the input operand to a reduction operator will have the axis of reduction as the minor-axis in a multi-processing environment, or an atomic array operand may be input to the program in a specific order. If none of the arrays have any constraints, any storage organization, including the default, is equally optimal and the algorithm terminates; however, this is rare.

Starting from the most constrained array, the remap algorithm visits all the remaining arrays in a breadth-first sequence. When two arrays are linked through an operator, the translated q–vector at the destination of the visit is an assimilation of the constraints on the array at the source of the visit, and the constraints, if any, of the operator in the path. If an array has innate constraints, and the incoming q–vector matches these initial constraints, then there is no conflict in composing the effective q–vector at that operand array. However, if the q–vectors do not conform, then the q–vector of that array is the result of resolving the conflicting constraints. This array, with the resolved q–vector as the starting constraint, is now a candidate for initiating a breadth-first visit of the program graph.

At the termination of the remap algorithm, the q–vector at each array is the resolution of constraints, from all other operators and operands, on that array. The optimal storage organization of that array is one that conforms to this q–vector. Note that in the absence of conflict, the remap algorithm has a cost proportional to the total number of arrays in the program graph. Even in cases of conflict, the resolution techniques of Section 3.2.9 expedite the remap algorithm to rapidly converge to a good solution.

As a preview, consider the heuristic that moderates the search of the optimal solution on the basis of the relative costs of the operators. At one extreme, we could ignore the innate constraints of all arrays but those of the arrays associated with the most critical operator. The translated q–vector at each node is adopted without conflict. Another resolution technique, for which the cost of the remap algorithm is proportional to the total number of arrays in the program (note that this is the minimum possible cost), simultaneously satisfies all conflicting storage constraints on an array by the use of subarray storage. For example, if a matrix has 1-minor and 2-minor as the storage constraints, the conflict might be resolved by storing the matrix as submatrices. Even when sub-optimal, this compromise is efficient, simple, and an improvement on a rigid, default storage.

3.2.6 The q–vector

As we have seen, every partial result has an associated q–vector that encodes the optimal storage constraints that must be satisfied by the partial result. At any point, the length of the q–vector is equal to the rank of the corresponding partial result, and the contents are such that the index of the largest element of the q–vector represents the preferred minor axis. In the same token, the index of the smallest element value represents the preferred major axis, and so on for the intermediate values.

As an example, if a rank–2 partial result, A (maybe an atomic operand), has storage constraints described by the q–vector: 1 2, A should optimally be stored in row-major order; while the q–vector: 2 1 implies that A should optimally be stored in column-major order. Thus the ravel organization of a Pascal rank–i array is specified by q–vector: $1 \ldots i$. For Fortran 77 this might be q–vector: $i \ldots 1$. Perfect subarray storage is specified by q–vector: $1\ 1 \ldots 1$, i times. Figure 3.5 illustrates optimal storage corresponding to the six possible q–vectors, not including subarrays, of a rank–3 operand.

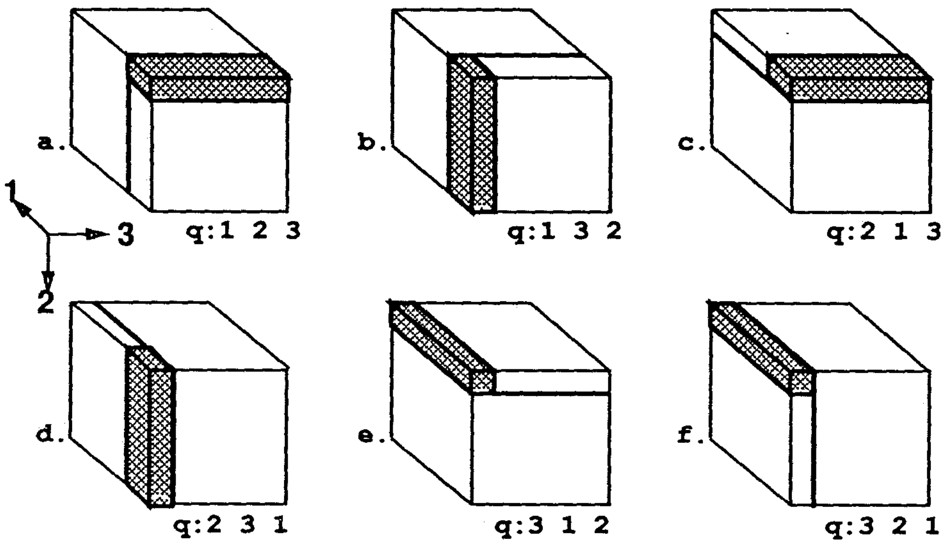


Figure 3.5: Optimal Storage for the Six Possible q–vectors of a rank–3 Operand

3.2.7 The Remap Algorithm

In incorporating the constraints of an operator, the algorithm effectively remaps the storage constraints encoded in the input q–vector into an output q–vector – hence "the *remap* algorithm." Please refer to Figure 3.6 during the algorithm description that follows:

1. **Create the program graph.**

   Note that the q-vectors are associated with the arcs.

2. **Initialize the q–vector of the arcs with the innate storage constraints.**

   These constraints are produced by the operators, *e.g.* the reduction operator constrains its input array operand.

3. **Choose the starting q–vector (arc.)**

   The most constrained array or the array associated with the most critical operator is usually the best choice.

4. **Push the q–vector out in a breadth-first sequence.**

   The q–vector of the arc coincident on a node is produced by accumulating the storage constraints of the operator at the current node into the starting q–vector. Note that the operator constraint may have already been incorporated into the innate q–vector at the coincident arc, as in the case of the reduction operator, or not, as in the case of the transpose operator.

5. **Resolve any conflicting constraints.**

We have considered only static organization of array storage. The dynamic case, where the storage of an array changes over the course of the execution of the code, is pursued as a potential resolution in Section 3.2.9.

3.2.8 Working of the Remap Algorithm: Example 1

In this example we compute: reduce(+, reduce(+, $A+B$, 2), 1), where A and B are rank–3 arrays. The execution environment is multi-processing. Figure 3.6 depicts the program graph.

The result of reduce(+, reduce(+, $A+B$, 2), 1) is a vector, and is therefore labeled with q–vector: 1. From the semantics of the reduction operator in Section 3.2.2, the operand of reduce(+, __, i) is optimally stored in i-minor order. These are the innate constraints.

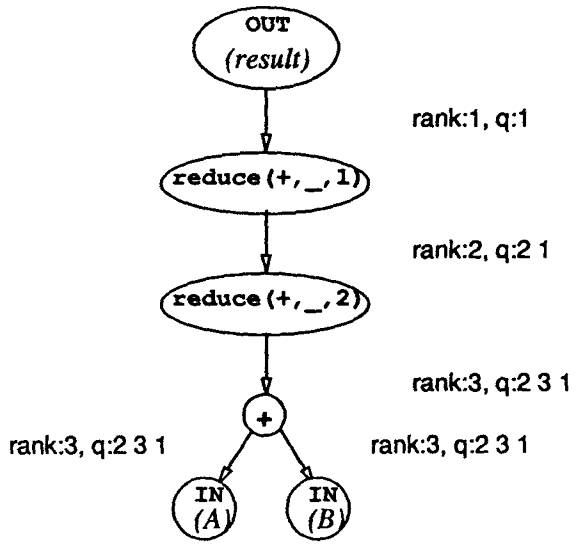


Figure 3.6: Example 1: `reduce(+, reduce(+,`$A+B$`,2), 1)`

We now push the `Result` q–vector: 1, through the expression graph to obtain the storage constraints for all the partial results, and atomic operands A and B. The operand to `reduce(+, reduce(+,`$A+B$`,2), 1)`, which is also the partial result of `reduce(+,`$A+B$`,2)`, accumulates the incoming q–vector: 1, and the constraint of the reduction operator into the translated q–vector: 2 1. Note that a new element was added to the incoming q–vector: 1 to conform to `length(q-vector)` = `rank(corresponding partial result)`. Also note that this q–vector conforms to the innate 1-minor constraint of `reduce(+,__,1)`.

Similarly, the operand to `reduce(+,`$A+B$`,2)`, which is also the partial result of $A+B$, acquires the q–vector: 2 3 1. Note that the index into the q–vector with the largest element value, 3, is the preferred minor axis, 2; and the order of the inherited values: 2 1, remains unchanged. The array sum operator, + of $A+B$, is equivalent to the unity transformation in that it does not introduce any additional constraint on operand storage; therefore the q–vector: 2 3 1 is passed unaffected to the leafs, A and B. The operands A and B should optimally be stored to conform to the constraints described by q–vector: 2 3 1.

A more involved illustration in APL of the working of the remap algorithm can be found in Appendix A. The APL expression there computes the inner product of two arrays. It does not use the inner product operator of APL (whose storage constraints we know from Appendix B) but instead uses a composition of other APL operators. Such an equivalent expression may arise because of, or in spite of, a clever programmer. Program analysis or peephole optimization for extracting the functional equivalence to the inner product and hence determining the corresponding q–vector manipulation is obviously difficult. The remap algorithm, on the other hand, operates on the semantic specification of the individual operators of the equivalent program and results in the same operand storage constraints as those resulting from the use of the inner product operator. Note that the inner product is merely an example, and any two "equivalent" programs should uniquely constrain the operands. This argument in some sense establishes a "completeness" of the q–vector transformations for the APL operators, and has significant ramifications on the equivalence of programs.

3.2.9 The Stock Suite

The techniques stock all the constraints on the storage of a data array into the optimal distribution of that array – hence "the *stock* suite." For procedural languages these constraints are direct constraints from sequencing operators to operands, while for declarative languages these constraints include the indirect constraints from all operators and operands in the program graph. The resolution techniques in the stock suite can be applied either individually or collectively. A few are enumerated below:

- incorporate the cost or the weight of the various constraints in the resolution (**cost based**),
- resolve the constraints into a compromise storage organization (**compromise**),
- motivate multiple copies of the subject array, each conforming to a conflicting constraint (**multiple copies**),

- induce the storage of the array to change dynamically during the course of the program (**dynamic change**), and
- for interleaved memory banks, skew the storage of the array into a scheme that maps the diverse patterns in which the array is accessed (**skew directives**).

The motivation of the constraint resolution techniques is to reduce global memory traffic and inefficiencies, *e.g.* to minimize the total numner of page faults and line misses summed over all the accesses of the computation. For the procedural case this corresponds to minimizing the misses for the accesses to each data array individually.

3.2.10 Cost Based

The cost-based technique basically permits the relative cost of the conflicting accesses to determine the dominating constraint. It expects each constraint to carry a cost measure reflecting the relevance of that constraint to the objective of minimizing the memory misses. To understand the significance of the cost measure, assume that $m_1 = n_1 = 2^{10} =$ `PageSize`, and $m_2 = n_2 = 2^3$ in the code fragment in Figure 3.7. Also, assume that the primary store has an available capacity of 4 pages. The illustrations in the figure identify the relative array accesses that generate the constraints.

Assuming that the code fragment as in Figure 3.7 generates the accesses to the array A, a row-major storage of A incurs 2^{10} faults for the accesses in the first set of nested loops, and 2^6 faults for the second set of nested loops. A column-major storage of A, on the other hand, incurs 2^{20} faults for the first set of nested loops, and 2^3 faults for the second set of nested loops. Thus although the storage of array A is subject to two conflicting constraints, the constraints derived from the accesses of the first set of nested loops are more critical than those from the second set. Thus extents of the loops that enclose the accesses to the array operands are a major characteristic in the cost measure of storage constraints.

Note that if the loop extents are not statically available to the compiler, then all constraints

```
for i1=1 to m1
   for j1=1 to n1
   ... A[i1][j1] ...
⋮
for i2=1 to m2
   for j2=1 to n2
   ... A[j2][i2] ...
⋮
```

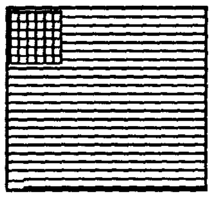


Figure 3.7: Cost criteria of storage constraints

can be assumed to carry unit cost. In that case, the number of times that the array is accessed in a particular order, *i.e.* the number of identical occurrences of a constraining q-vector, forms the weight or cost of that storage order. Whatever the origin of the cost factor, the cost-based resolution technique completely satisfies some constraints to the detriment of others. The following alternative technique resolves the constraints into a compromise storage organization that may not completely satisfy any of the conflicting constraints individually.

3.2.11 Compromise

In the program fragment of Figure 3.7, each of the two accesses to array A constrains its storage to map the respective access sequence. The constraints conflict – the first access constrains array A to conform to row-major storage, while the second access constrains A to column-major storage. In terms of the q-vectors, the first constraint corresponds to q-vector 1 2, while the second to q-vector 2 1.

In the general case, an n-dimensional array will be constrained by many q-vectors, each corresponding to an access of the array. One compromise technique for optimally resolving the constraints corresponds to an assignment of a unique major–minor axis to each dimension of the array. In other words, we preclude the possibility of storing some of the axes as subarrays. Each q-vector encourages a particular storage of the array, and assigns a unique storage axis to each dimension of the array. For example, the q-vector 1 2 3 4 constrains the 4-dimensional array to be stored with the first dimension as the major axis, and the fourth dimension as the minor axis. For any dimension the corresponding q-vector element specifies for that dimension the preferred storage axis in the major–minor spectrum. In other words, a q-vector relates the array dimension to the storage axis.

For the resolution of q-vector constraints, the inverse of the q-vector, $\overline{\text{qvector}}$, is more appropriate. $\overline{\text{qvector}}$ relates the storage axis to the array dimension. Thus for a storage axis in the major–minor spectrum, $\overline{\text{qvector}}$ specifies the preferred array dimension. The last entry of $\overline{\text{qvector}}$ specifies the array dimension that should be stored as the minor axis, and the first entry corresponds to the major axis. For example, q-vector 4 2 1 3 constrains the first dimension to be stored as the minor axis. The corresponding $\overline{\text{qvector}}$ is 3 2 4 1.

The technique for resolving multiple q-vector constraints into an optimally compromised q-vector rests on the following observation: Consider $\overline{\text{qvector}}$ 3 2 4 1. The preferred minor axis is the first dimension. If the first dimension cannot be stored as the minor axis, then the fourth dimension should be stored as the minor axis. The first dimension will then be championed as the next-to-minor axis. This minimizes the offset between the memory locations of successive accesses to the array elements. In other words, the elements of the $\overline{\text{qvector}}$ from left to right specify in increasing order the dimension preferred as the minor axis.

Thus the strategy of resolving the constraints of $\overline{\text{qvectors}}$, is to continually press constraints, in order from minor to major, *i.e.* from right to left, until they are satisfied. If a dimension preferred as the j^{th} axis cannot be stored as such, then the same dimension is the preferred next major axis. Thus the technique stores that dimension as the minor axis that appears most often as the n^{th} element in the $\overline{\text{qvectors}}$ that constrain the n-dimensional array. Ties are broken by summing the number of appearances of the tied dimensions in the $(n-1)^{\text{th}}$ and n^{th} position of the $\overline{\text{qvectors}}$, and so on. The unsatisfied constraints of the n^{th} $\overline{\text{qvector}}$ position alongwith the constraints of the

$(n-1)^{\text{th}}$ position determine the dimension to be stored as the next-to-minor axis, and so on, until every storage axis is assigned a unique dimension.

For example, if a 4-dimensional array is subject to the constraints of $\overline{\text{qvectors}}$ 1 3 2 4 and 1 2 3 4, the resolution of these two constraints stores the fourth axis as the minor axis, either the second or the third as the middle two axes, and the first axis as the major axis. If the array was additionally constrained by $\overline{\text{qvector}}$ 4 2 1 3, the fourth dimension continues to be the minor axis. However, the third dimension is now the preferred next-to-minor axis because the unsatisfied constraint of the last $\overline{\text{qvector}}$ – third dimension as the minor axis – now breaks the tie between the third and second dimensions as the next-to-minor axis.

Another compromise technique resolves conflicting constraints by storing the array as appropriate subarrays. For example, if the two constraints on a 3-dimensional array are represented by the q-vectors 1 2 3 and 1 3 2, the resulting compromise is represented by the q-vector 1 2 2. Figure 3.8 illustrates this compromise technique.

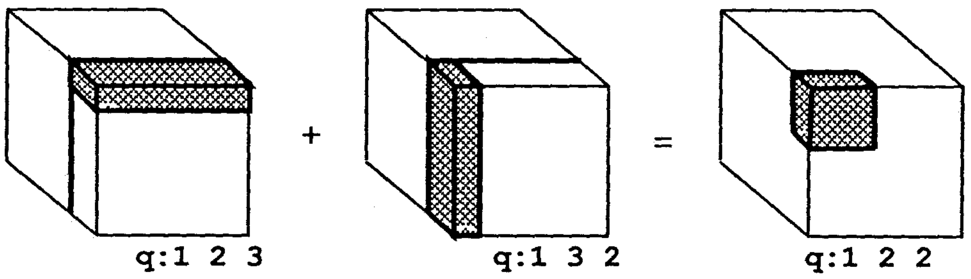


Figure 3.8: The compromise resolution of conflicting storage constraints

3.2.12 Multiple Copies

The resolution technique motivating multiple copies of the array, each conforming to a unique conflicting constraint, is very similar to the input renaming transformation of the distribution of scalar operands into lines. We assign a new operand to each set of identically sequenced accesses. Assignments to the various copies of the array address coherency concerns similar to the scalar case. The payoffs for multiple copies of an array operand are potentially greater than that for scalar operands because a page fault is more expensive than a line miss. Moreover, unlike the effect of scalar copies on the reference string, the assignments to array copies do not impact the patterns of access, array operands are usually not overused as are scalar operands, and the additional complication of the distribution of the scalar copies into lines is not germane to the copies of array operands whose distributions are independent of other operands.

3.2.13 Dynamic Change

The option of dynamically transposing the storage of an array operand to better map a change in the access pattern entails that the data optimization insert code for the transposition. Unlike the $\overline{\text{qvector}}$ compromise technique, the order of the constraining q-vectors is now critical in determining a suitable opportunity to initiate the transposition. As a motivation to dynamically redistributing the array operand, note that it is asymptotically cheaper to translate two row-major matrices to submatrix storage, perform a block inner product, and then translate the result to the required storage, than to attempt the inner product on the default matrices.

Since the $\overline{\text{qvector}}$ technique is computationally inexpensive, one technique for determining an appropriate transposition point might sort the $\overline{\text{qvectors}}$ in the order in which the corresponding accesses occur in the code, resolve neighboring $\overline{\text{qvector}}$ clusters, and watch for large variations in the resolved $\overline{\text{qvectors}}$ of successive clusters. Note that this technique motivates code motion to group code that accesses an array in similar patterns of access.

3.2.14 Skew Directives

Different skewing schemes for distributing arrays over memory banks provide efficient accesses to different patterns through the array. For interleaved memories, the conflicting accesses of the array operand may motivate the choice of an appropriate skewing scheme.

And finally, some or all of the above techniques for resolving access constraints into the judicious storage of the subject array can be applied concurrently. An example of a hybrid between the cost-based and the subarray compromise techniques stores the array into subarrays with an aspect ratio determined by the costs. For example, suppose that two row-major accesses, and one column-major access to a matrix need to be resolved. The matrix may be stored as submatrices with the row extent twice the column extent. Please see **Figure 3.9.**

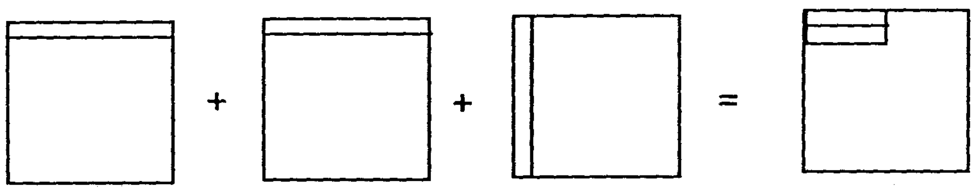


Figure 3.9: The concurrent application of cost-based and subarray compromise techniques

Only the $\overline{\text{qvector}}$ technique is polynomially determinate and has been developed as an implementable algorithm. The other techniques need further evaluation.

3.3 General Applicability

Congruent to the general applicability of the distribution of scalar operands into lines, the layout of array operands can conform to a variety of memory, processor, and system architectures other than the hierarchical memory interface between primary and secondary memory. For example, when applied to the cache-primary memory interface, a reduction in "page faults" translates to a reduction in "line misses." Besides the suitability to the multiprocessors, interleaved memories, etc. of before, the storage of arrays, which maps patterns of access, is particularly beneficial to vector processing environments.

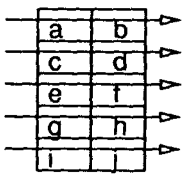
Fortran 77 was exempted from the discussions involving the storage of array data in procedural languages because it imposes column-major storage on all its arrays. The following subsection demonstrates that the storage of arrays in Fortran 77 can be optimized in spite of the storage restriction of the language.

3.3.1 Storage Organization of Arrays in Fortran 77

All meaningful accesses of arrays are enclosed within loops in the source code – in scalar languages like Fortran 77 these loops may be explicit, while in array languages like Fortran 8x they may be mostly implicit. The loops, implied or otherwise, are schemas that concisely abstract the patterns of access in arrays; each array access indexed with the induction variable of an enclosing loop is a schema for accesses to multiple array elements. The juxtaposition of loops and the indices within these schematic array accesses represent the patterns of access which the stock techniques then resolve to get the globally optimal storage organization for that array. While it is possible to program without loops, the accesses to array elements are then accesses to scalar variables and the code effectively has no arrays.

Loop manipulations therefore strongly affect the optimal storage organization of data arrays. Conversely, a prefixed storage for arrays has repercussions on the efficiency of loop manipulations. Consider the following Fortran 77 example:

```
DIMENSION A[5][2]
for i=1 to 5
   for j=1 to 2
      A[i][j] = A[i-1][j] + 1
```



The code fragment can be directly vectorized to,

```
for i=1 to 5
   A[i][1:2] = A[i-1][1:2] + 1;
```

However the vectorized code accesses array A in row-major order. This conflicts with the column-major storage of A. Although the code is vectorized, a vector processor that requires its operands to be in contiguous memory locations will not execute the code as efficiently as it would if only array A had been stored in row-major order. In addition, the non-unit stride between successive vector elements might distribute the vectors in the array over more pages than are available in the memory, or lines than are available in the cache.

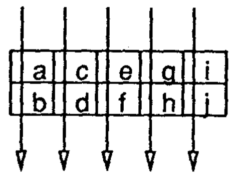
On the other hand, if the compiler interchanges the loops to conform to the localities of the column-major storage of A, the resultant code,

```
for j=1 to 2
   for i=1 to 5
      A[i][j] = A[i-1][j] + 1;
```

cannot be vectorized. A row-major storage for A would satisfy the constraints from the vector processor, as well as from the paged hierarchical memory of the system.

Consider the following transformed code:

```
DIMENSION A[2][5]
for i=1 to 5
   for j=1 to 2
      A[j][i] = A[j][i-1] + 1
```



This code fragment can be directly vectorized to,

```
for i=1 to 5
   A[1:2][i] = A[1:2][i-1] + 1;
```

and this vector code satisfies the vector processor as well as the memory system by accessing the array in column-major order. From the accompanying figures depicting the accesses of A, we note that the transformed code preserves the dependences and hence the semantics of the original code, but it effectively transposes A.

The above example illustrates the transformation that permits the storage optimization of arrays in Fortran 77, while abiding by the column-major constraints on the storage. Whereas the data optimization techniques for arrays can be directly applied to code in languages that do not restrict the storage organization, optimization of array layout in Fortran 77 needs such transpositions to administer the storage recommendations of the optimization. For two dimensional arrays the transposition can be implemented by:
(if any array is optimally stored in row-major order)

- Transpose the definition of the array, *i.e.* `DIMENSION A[i][j]` to `DIMENSION A[j][i]`
- Transpose each use of the array, *i.e.* `A[i][j]` to `A[j][[i]`.

The generalization for arrays of higher dimensionality is straightforward.

Such transposition is necessary when passing array data between functions encoded in languages with differing default storage organizations, *e.g.* C and Fortran 77. Typically the user explicitly programs such transpositions for the interface arrays. The compiler can now motivate the automatic transposition of internal arrays to improve program efficiency while adhering to a rigid storage organization.

3.4 Optimization of Array Data Storage as a Compiler versus a User Responsibility

Unlike the distribution of scalar operands, the payoffs of the judicious distribution of array operands has been recognized as a fruitful exercise. However, many current proposals call for the programmer to assume responsibility for appropriately distributing his array data. This section attempts to dispel the myth that the programmer is the best judge and final arbiter on optimal data storage. Instead, it promotes the optimization of data as an integral part of compiler optimization.

Given a processing task and a system architecture for executing the process, the payoffs for a judicious choice of algorithm that implements the task cannot be overestimated. However, the interdependence between the encoding of the algorithm in the program and the storage organization of the operands to a large extent dictates the execution efficiency of the algorithm. The matrix multiplication example of Section 3.1.1 bears testimony to that. There are two routes to matching code and data organization: The program may be designed to map the operand organization, or conversely, the operand storage may be organized to map the access structure of the program. Matrix multiplication again can be used as an example. A clever program can be designed to multiply two $n \times n$ matrices where both the matrices are stored in column-major order. Alternatively, the matrices may be stored as subarrays to exploit a block multiplication algorithm. Currently, compilers can manipulate loops to partially map differing operand storage organizations; they can, as we have shown, also support the automatic reorganization of operand storage to closely map differing access sequences. The user, on the other hand, may assume the responsibility of matching algorithm and data organization in either of the two routes.

Although the proficient user may optimally match his algorithm and operand storage, he needs to be alert to the optimizations that the compiler performs on his code. The storage constraints pertain to the code executed by the machine *i.e.* to the object code output by the compiler. The storage constraints in the programmer's encoding of the algorithm may very well be different, and oftentimes contradictory, to those of the object code produced by the compilation and optimization of the source code. Some of the loop restructuring manipulations, most notably loop interchange, harbor such potential pitfalls.

As an example, the author of the following code,

```
for i=1 to m
   for j=1 to n
      A[i][j] = A[i][j-1] + 1;
```

stores operand A in row-major order to map the pattern of access of his code. The vectorizing compiler will however interchange the i and j loops to move the recurrence to the outermost loop. The machine will execute code that more closely resembles,

```
for j=1 to n
   A[1:m][j] = A[1:m][j-1] + 1;
```

with operand A accessed in column-major order.

The loop transformation optimizations are very effective, especially when compiling Fortran 77 programs for vector-processing and multi-processing environments. Thus the user who wishes to explicitly specify the storage organization of operands needs to inactivate these optimizations for fear of adversely affecting the efficiency and speed of his computation. This is clearly undesirable. Furthermore, such user-optimized programs suffer from low readability and maintainability.

If, on the other hand, the compiler were to optimize the storage of operands, the user would be liberated from this additional level of expertise and diligence. He would then be free to apply his ingenuity to higher levels, such as algorithm choice, which are beyond the scope of compiler optimizations, while the compiler would be responsible for all lower level code and data optimizations. Besides the ease of programmability, readability, portability, etc., this specialization can lead to more efficient programs – the compiler can now freely restructure and optimize the code for the running environment and then optimally distribute the data to match the data accesses of this final executable code, or it can use data distribution constraints to drive the code optimizations, or it can apply any iterative combination of the two.

The influence of data constraints in motivating loop transformations fueled the higher efficacy of data optimization to declarative languages as opposed to procedural languages. Moreover, some of the techniques in the stock suite, notably dynamic restructuring and multiple copies of arrays, manifest the concurrent optimization of code and data.

3.5 Future Work

Many of the ideas in this chapter warrant further evaluation. They can also spawn research in potentially fruitful issues, documented in, or related to, our work. Some of these include:

- A methodology for the dynamic restructuring technique of the stock suite needs to be developed.
- The payoffs promised by the remap algorithm and the various techniques of the stock suite need to be confirmed. Alternative resolution techniques may be added to the stock suite.
- All the constraints on the storage of the arrays are very dependent on the physical parameters. We have attempted to determine profitable distributions of the arrays based solely on the access of the arrays. Introducing system characteristics, *viz.* page size, number of pages, etc., will temper some of the constraints. This needs further study.
- Most importantly, the techniques need to be incorporated into a working compiler.

3.6 Conclusions

The anachronism that leads us to assume that the physical data space is homogeneous and monolithic needs little comment. The other bias obstructing the optimization of data has been the complexity of analyzing an often unstructured data space. This chapter has demonstrated that data optimization need not necessitate the complete disambiguation of all data accesses. The optimization of data storage for array operands can proceed with information only of the relative access order of the operand axes; and for this a methodology and an algorithm have been presented that are not limited to any one programming language. The payoffs of optimizing the data storage are substantial, especially for system architectures that are data-driven or that reward data locality. Furthermore, the payoffs are practically achievable, considering the simplicity of the remap algorithm and some of the stock techniques.

There are compelling reasons for the compiler to optimize data in coordination with the optimization of code. Firstly, optimal data storage mirrors the access of the data by the code, but the code optimization by the compiler may reorganize these accesses. The option of restricting the compiler reorganization of code is self-defeating. Secondly, data storage constraints are useful evaluators of alternative code transformations by the compiler. And thirdly, the user, released from the low-level responsibility of optimizing data can now better spend his energy optimizing at higher levels, which hold the promise of large improvements in execution efficiency but which are beyond the scope of the compiler.

Chapter 4

Complexity Issues in the Optimal Storage of Data

The algorithms of previous chapters for distributing scalar and array data very loosely describe the solutions as "optimal". No justification for this superlative is forthcoming as the techniques are based on insights into the problem, and the solutions are not optimal in general. In this chapter we explore the complexity of developing truly optimal solutions to the distribution problems. We prove that the problems are NP-complete, strongly suggesting that the cost of optimal solutions will be exponential. In a sense, we vindicate the heuristic approach adopted.

Sections 4.1 and 4.2 explore the complexities of the optimal distribution of scalar and array data respectively. In the spirit of complexity arguments, each proof appropriately restricts the problem and then proves that even the restricted version is NP-complete. Section 4.1 further discerns between similarly the NP-complete cases of the static and the dynamic distribution of scalar data into lines. The chapter is liberally interspersed with examples.

4.1 Distributing Scalar Data into Lines

Problem Statement: Given a reference string R, lower level operand capacity M, and line capacity p. Determine the distribution of the scalar operands into lines, which minimizes the number of line misses necessary to satisfy the accesses of the reference string.

There are two types of distributions: static and dynamic. In the static approach, the distribution of the scalar operands into lines remains unchanged for the entire duration of the reference string. Conversely, in the dynamic approach, the distribution of the scalar operands into lines changes during the duration of the reference string. The following example illustrates the distinction.

Example: Assume $M = 4, p = 2, S = 6$.

Reference String: $abcdefca$

Starting Distribution: ab, cd, ef

Static Line Transitions: $abcd \rightarrow cdef \rightarrow abcd$.

Dynamic Line Transitions: $abcd \rightarrow acef$.

From the above example, it may be observed that the for a static distribution, the lower level memory contents are maintained as t lines, each of p operands. A fetched line replaces one of these t lines. For a dynamic distribution, on the other hand, the lower level memory is maintained as M homogeneous operands. A fetched line replaces any p of these M operands. However, even for a dynamic distributions, the p operands that are pushed out as a line from the memory have to be fetched as a line into the memory. Thus the metamorphosis of the dynamic distribution is effected only in the lower memory level.

The spatial locality of the accesses of a program can vary widely with time. Intuitively, a dynamic distribution can better map the changing localities of the reference string, while a static distribution has to optimize one, or the other has to compromise by optimizing neither. The following example underlines the inherent advantages of a dynamic distribution.

Example: Assume $M = 4, p = 2, S = 8$.

Reference String: $(abcd)^n(aefg)^n$

Starting Distribution: ab, cd, ef, gh

Static Line Transitions: $abcd \rightarrow (abef \rightarrow abgh \rightarrow)^n$.

Dynamic Line Transitions: $abcd \rightarrow abgh \rightarrow aefg$.

It is important to note that we need to address the complexity of three problems: how to distribute the scalar operands into lines, which line to load into the lower level memory, and which line to spill out of the lower level memory to make room for the incoming line. Only for the static distribution, the incoming and outgoing lines are optimally determinate in polynomial time [9, 29].

The rest of this chapter, uses the term "page" to represent our generic line, "page fault" to represent a line miss, and "cache" to represent the lower level of the memory hierarchy. The motivation to use "cache" instead of the unwieldy "lower level of the memory hierarchy" prompted the change of terminology.

4.1.1 Complexity of the Static Distribution Problem

The $M = tp$ domain of the distribution problem has five subdivisions, each with its unique characteristic. Each subdomain demands a different approach to reasoning about the complexity of the problem for corresponding values of M, t, and p. These subdomains are treated as the following cases:

case 1: $t > 0, p = 1$

case 2: $t = 1, p = 2$

case 3: $t = 1, p > 2$

case 4: $t > 1, p > 2$

case 5: $t > 1, p = 2$

case 1: $M = tp, t > 0, p = 1$

Operands are trivially distributed one per page.

case 2: $M = tp, t = 1, p = 2$

Consider the execution of the program that begot the reference string. The cache holds the page containing the recently accessed operand. The next operand to be accessed will incur a page fault unless the operand is available in the cache. Since the cache has a page capacity of one, the next access will not incur a page fault only if the operand being accessed resides on the same page as the operand of the preceding access. From this perspective, an optimal solution to the distribution problem for $M = p$ maximizes the number of times that operands cohabiting a page occur consecutively in the reference string. Alternatively, the optimal solution minimizes the number of times that the operands of successive references occur in different pages. This holds true irrespective of the value of p.

The solution to the operand distribution problem for $p = 2$ and the arguments on the complexity of the problem for $p > 2$ require the following definitions:

Definition: A *Reference Multigraph* corresponding to a reference string is the multigraph $G = (V, E)$, whose vertices are the operands in the reference string, and whose edges are such that the reference string is an Euler path [13] through the graph.

A simple technique to obtain G traces the reference string onto the vertices of G and then ignores the sequencing between the edges and the directions on the edges. This ensures that the graph has an Euler path that is identical to the reference string. Note that a reference string corresponds to a unique reference graph, while a reference graph may correspond to as many reference strings as there are Euler paths through the graph. Please refer to Figure 4.1.

Definition: A *Weighted Reference Graph* is obtained from a reference multigraph by coalescing parallel edges in the multigraph into a single edge with weight equal to the number of parallel edges replaced.

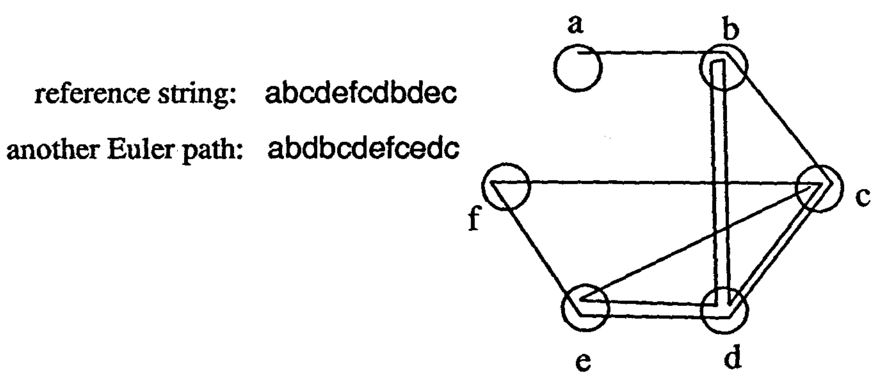

reference string: abcdefcdbdec
another Euler path: abdbcdefcedc
a
b
c
f
e
d

Figure 4.1: A reference string and the corresponding Reference Multigraph

Figure 4.2a depicts the weighted reference graph corresponding to the reference multigraph of Figure 4.1. Since the vertices in the weighted reference graph represent the operands in the distribution problem, paging the operands into pages of size p is analogous to partitioning the vertices of the graph into sets of size p. The uniqueness restriction on the paging demands that these sets be disjoint. The analogue is illustrated in Figure 4.2 where the weighted reference graph is superimposed with a distribution of the operands into pages. This graph theoretic perspective to the distribution problem is encapsulated in the following theorem.

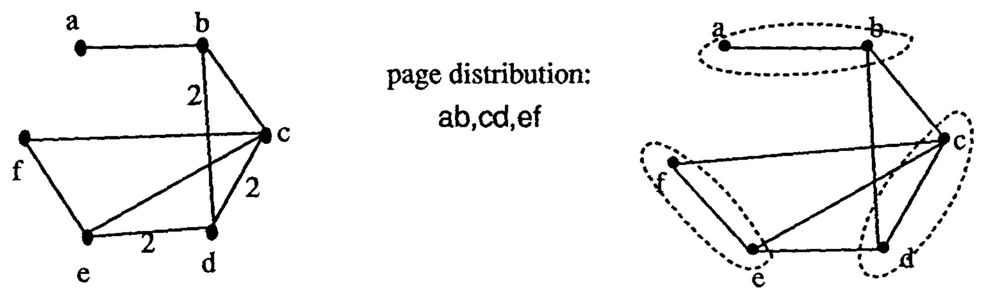


Figure 4.2: Weighted Reference Graph and its partition into disjoint sets matching the pagination of the operands

Theorem 4.1 A solution to the operand distribution problem for $t = 1$ is optimal iff the corresponding partitioning of the weighted reference graph is optimal, *i.e.* the pages partition the weighted reference graph $G = (V, E)$ into disjoint sets $V_1, V_2, \ldots, V_m$ of size p such that if $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets, then $\sum_{e \epsilon E'} l(e)$ is minimum over all such partitions, where $l(e)$ is the weight of edge e.

Also, $\sum_{e \epsilon E'} l(e)$, corresponding to the optimal partition of the weighted reference graph is equal to the number of page faults for any Euler path through the weighted reference graph with the operands distributed into pages matching the sets of the partition.

Proof: From the construction of the weighted reference graph, it is clear that the weight of an edge is the number of times that the operands incident on the edge appear successively in the reference string. Earlier assertions also inform us that for $t = 1$, the number of page faults is the number of times that the operands of successive references occur in different pages.

Since the pages of the distribution problem are represented by the sets in the partition problem, for every edge $e = (a, b)$, if $a, b \epsilon V_i$ *i.e.* a and b appear in the same set and hence on the same page, then all $l(e)$ reference string transitions from a to b or b to a can be satisfied from the cache and do not incur a page fault. Conversely, if $a \epsilon V_i, b \epsilon V_j, i \neq j$ *i.e.* a and b appear on different pages, then each b following an a and each a following a b in the reference string, a total of $l(e)$ such successions, will incur a page fault. Thus the sum, $\sum_{e \epsilon E'} l(e)$, for a partition of the weighted reference graph is equal to the number of page faults for the given reference string and a distribution of the operands into pages mirroring the disjoint sets of the partition. However, the given reference string is but one of many Euler paths through the graph. Therefore, the assertion holds for all Euler paths through the weighted reference graph.

if: Assume that a page distribution $\mathcal{D}$ is optimal with respect to a given reference string but $\mathcal{P}$, the matching partition of the corresponding weighted reference graph, is not. The distribution $\mathcal{D}$ requires $\sum_{e \epsilon E'_{\mathcal{P}}} l(e)$ page faults to satisfy the accesses of the reference string. Since $\mathcal{P}$ is not optimal,

there exists a partition, $\mathcal{P}_1$, such that $\sum_{e \epsilon E'_{\mathcal{P}_1}} l(e) < \sum_{e \epsilon E'_{\mathcal{P}}} l(e)$. Since the page distribution $\mathcal{D}_1$ corresponding to $\mathcal{P}_1$ has $\sum_{e \epsilon E'_{\mathcal{P}_1}} l(e)$ page faults, the original page distribution could not have been optimal.

only if: The proof is congruent to the if part above.

$\square$

For $p = 2$ matching techniques [18] optimally partition the weighted reference graph in polynomial time. By Theorem 4.1 these optimal solutions also optimally distribute the operands into pages of size 2 for all reference strings that are Euler paths through the graph.

case 3: $M = tp, t = 1, p > 2$

The construction of the weighted reference graph and Theorem 4.1 of the previous section are independent of the value of p so long as $p > 1$ and $t = 1$. The problem of partitioning the weighted reference graph into sets of size 2 is polynomial in time and this gave us the polynomial–time solution to the distribution of operands into pages for $p = 2$. The intransigence of the partitioning problem for $p > 2$ is used to prove that the decision problem corresponding to operand distribution for $t = 1, p > 2$ is NP-complete. The Exact Graph Partitioning problem, the statement of which appears below, is very similar to the NP-complete Graph Partitioning problem [18]. The only difference is that the partitions in the Exact Graph Partitioning problem are fixed in size, whereas they are only upper-bound in the Graph Partitioning problem. Not surprisingly, the NP-completeness proofs of both are based on Partitions into Triangles [18].

Problem: Exact Graph Partitioning (**EGP**)

INSTANCE: Graph $G = (V, E), |V| = mq, \forall i, |V_i| = q$, weights $l(e) \epsilon Z^+$ for each $e \epsilon E$, positive integer J.

QUESTION: Is there a partition of V into disjoint sets $V_1, V_2, \ldots, V_m$, such that if $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets $V_i, V_j, i \neq j$, then $\sum_{e \epsilon E'} l(e) \leq J$?

COMMENT: Remains NP-complete for fixed $q \geq 3$.

The decision problem corresponding to the optimization problem of distributing operands between pages to minimize the number of page faults for a given reference string is:

Problem: Static Operand Distribution

INSTANCE: Reference string R, page size p, cache size $M = tp$, number of page faults k, where $p, t, k \epsilon Z^+$.

QUESTION: Is there a static distribution of the operands into disjoint pages of size p, such that the number of page faults generated in satisfying the requests of R is at most k?

Clearly this decision problem is no harder than the optimization problem. If we could find the page distribution that minimizes the number of page faults, then we could use that distribution to find the minimum number of page faults in polynomial time, and we could compare this number with the given bound k. Thus if Static Operand Distribution is proved to be NP-complete, the corresponding optimization problem is at least as hard.

Theorem 4.2 Static Operand Distribution for $t = 1, p > 2$ is NP-complete.

Proof: It is easy to see that Static Operand Distribution for $t = 1, p > 2 \ \epsilon$ NP, since a nondeterministic algorithm need only guess a starting distribution of the operands into pages and check in polynomial time to see whether the number of page faults is less than the bound k.

We transform EGP to Static Operand Distribution for $t = 1, p > 2$. Let $G = (V, E)$, $|V| = mq$, $l(e) \epsilon Z^+$, and $J \epsilon Z^+$, be the arbitrary instance of EGP. We construct a transformation from EGP with $q > 2$ to Static Operand Distribution for $t = 1, p > 2$. This requires a reference string R and a positive integer k such that there is a distribution of the operands into $p > 2$ sized pages, which requires at most k page faults to satisfy the references of R for $t = 1$ if and only if there is a partition of V into m disjoint sets for which $\sum_{e \epsilon E'} \leq J$, where $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets, $V_i, V_j, i \neq j$.

We let $p = q, k = 2J$. To construct the reference string, we first create graph $G' = (V, E^*)$ where $\forall e \epsilon E, e^* \epsilon E^*$ and $l(e^*) = 2l(e)$. This ensures the existence of an Euler path through G^* [13]. R can now be chosen as any Euler path through G^*; G^* is therefore the weighted reference graph for R. This completes the construction.

if: From Theorem 4.1 we know that for $t = 1$, a solution, $\mathcal{D}$, to the distribution of operands into p sized pages for R is optimal if the corresponding solution, $\mathcal{P}$, that partitions the vertices of G^* into p sized sets, is optimal. In particular, the number of page faults generated by $\mathcal{D}$ in satisfying R is equal to $\sum_{e^* \epsilon E^{*\prime}} l(e^*)$ for the $\mathcal{P}$-partition of G^*. For identical partitions of G and G^* into sets $V_1, V_2, \ldots, V_m, \sum_{e^* \epsilon E^{*\prime}} l(e^*) = 2 \sum_{e \epsilon E'} l(e)$, where $E^{*\prime} \subseteq E^*$ is the set of edges that have their two endpoints in two different sets $V_i, V_j, i \neq j$, and $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets $V_i, V_j, i \neq j$. Therefore $\mathcal{D}$ generates k page faults in satisfying R if $\sum_{e^* \epsilon E^{*\prime}} l(e^*) = k$, or $\sum_{e \epsilon E'} l(e) = k/2 = J$ for the $\mathcal{P}$-partitions of G^* and G.

only if: The proof is congruent to the **if** part above.

It is clear that this transformation can be performed in polynomial time.

□

case 4: $M = tp, t > 1, p > 2$

A transformation from Static Operand Distribution for $t = 1, p > 2$, which was proved to belong to NP, will help us prove that Static Operand Distribution for $t > 1, p > 2$ is also NP-complete. Note that EGP can be directly reduced only to the cases of Static Operand Distribution for which $t = 1$ and therefore the NP-completeness proof of Static Operand Distribution for $t > 1, p > 2$ cannot follow directly from the NP-completeness proof of Static Operand Distribution for $t = 1, p > 2$.

Theorem 4.3 Static Operand Distribution for $t > 1, p > 2$ is NP-complete.

Proof: Again, it is easy to see that Static Operand Distribution for $t > 1, p > 2 \in$ NP since a nondeterministic algorithm need only guess a distribution of the operands into pages and check in polynomial time to see whether the number of page faults satisfying R is less than the bound k.

Assume the arbitrary instance of Static Operand Distribution for $t = 1, p > 2$: reference string R, page size p, and bound on the number of page faults, k. We need to construct a reference string R' and a positive integer k' such that there exists a distribution of the operands into pages of size p that requires at most k' page faults to satisfy R' with $t > 1$ if and only if there exists a distribution of the operands into pages of size p that requires at most k page faults to satisfy R with $t = 1$.

Intuitively, the operands of the Static Operand Distribution problem for $t > 1, p > 2$ are the operands of the problem for $t = 1, p > 2$ plus some "external" operands. The reference string R' will be constructed so that these external operands continually occupy $t - 1$ pages in the cache and the "native" operands are distributed into p-sized pages to be optimal for $t = 1$ and an effective reference string identical to R. The optimal distribution of the native operands into pages for R' and $t > 1, p > 2$ will then also be optimal for R and $t = 1, p > 2$.

We introduce $(t-1)p = r$ new variables, $a_1, a_2, \ldots, a_r$, into the instance of Static Operand Distribution for $t > 1, p > 2$. R' is constructed from R by adding after every reference in R, the string $(a_1 a_2 \ldots a_r)^{|R|}$, where a^n implies n repetitions of a, and $|R|$ represents the length of reference string R. The following example serves to illustrate.

Example 4.1 $R = bcb$,

$R' =$ b $\underline{a_1 \ldots a_r}$ $\underline{a_1 \ldots a_r}$ $\underline{a_1 \ldots a_r}$c $\underline{a_1 \ldots a_r}$ $\underline{a_1 \ldots a_r}$ $\underline{a_1 \ldots a_r}$b $\underline{a_1 \ldots a_r}$ $\underline{a_1 \ldots a_r}$ $\underline{a_1 \ldots a_r}$.

Thus R' contains $|R|$ repetitions of the substring $a_1 a_2 \ldots a_r$. Our construction is complete by setting $k' = k + t - 1$.

The optimal number of page faults for R is trivially less than $|R|$. If $a_1, a_2, \ldots, a_r$ are not permanently resident in the cache, then the number of page faults for R' is going to be greater than

$|R|+t-1$, where t page faults load the blank cache at start-up. If they do reside in the cache, then the cache can satisfy all requests for $a_1, a_2, \ldots, a_r$ with only the $t-1$ page faults that initially load them into the cache, and all remaining page faults are generated by the loading and spilling of pages containing the native operands.

Since the external operands are distinct from the native operands, they cannot satisfy any requests to the native operands in R'. Therefore, with the external operands permanently in the cache, the occurrences of $a_1, a_2, \ldots, a_r$ in R' can be ignored. But this reduces R' to R. Also with $a_1, a_2, \ldots, a_r$ permanently occupying $t-1$ pages in the cache, there is only one cache page available for the loading/spilling of pages containing native operands. Therefore, t is effectively equal to 1. Now for $t = 1, p > 1$ the optimal number of pages for R is less than $|R|$. Therefore, the total number of page faults is less than $|R| + t - 1$, and any optimal distribution of the operands for R' and $t > 1, p > 2$ will page $a_1, a_2, \ldots, a_r$ into $t-1$ pages and retain them permanently in the cache. More precisely, the optimal distribution for R' and $t > 1, p > 2$ requires $k + t - 1$ page faults, where k is the number of page faults generated by the optimal distribution for R and $t = 1, p > 2$.

if: Assume that $t = 1, p > 2$ a distribution, $\mathcal{D}$ requires k page faults to satisfy the accesses of reference string R. From the construction above, if we distribute the external operands $a_1, a_2, \ldots, a_r$ into $t-1$ pages, and the native operands into pages matching $\mathcal{D}$, then we can satisfy the requests of R' for $t > 1, p > 2$ in $k + t - 1$ page faults.

only if: Assume that the optimal distribution for R' and $t > 1, p > 2$ requires $k + t - 1$ page faults but there exists no distribution for $t = 1, p > 2$ that can satisfy the requests of R in k page faults. Again from the arguments above we know that the optimal distribution for $t > 1, p > 2$ scatters $a_1, a_2, \ldots, a_r$ in $t-1$ pages and retains them permanently in memory. This requires $t-1$ page faults. But this implies that the native operands are paged to satisfy the requests for R and $t = 1, p > 2$ in k page faults.

The construction of R' from R has cost $O(|R|(t-1)p)$, and the transformation from Static Operand Distribution for $t = 1, p > 2$ to Static Operand Distribution for $t > 1, p > 2$ is polynomial.

□

case 5: $M = tp, t > 1, p = 2$

Static Operand Distribution for $t > 1, p = 2$ has the misfortune that Static Operand Distribution for $t = 1, p = 2$ is polynomial and therefore does not offer any insight into the complexity of this case. Intuitively, all $t > 1$ cases have a degree of "persistence," a carry-over of $t - 1$ pages from past cache contents that very strongly affects future occurrences of page faults. The persistence can potentially span the entire reference string because it is possible that the page loaded to satisfy the first access of the reference string never leaves the cache. This persistence is not easily amenable to quantification or even to formal characterization.

In the $t = 1$ cases, every page fault clears the cache and the problem of optimally distributing the operands into pages is independent of the history of cache transitions. Since Static Operand Distribution for $t = 1, p > 2$ was proven to be NP-complete, the proof for $t > 1, p > 2$ simply nullified the effects of the extra $t - 1$ cache pages with persistent pages containing the r external operands, and hence reduced the $t = 1, p > 2$ case to a $t > 1, p > 2$ instance. To reason about the complexity of the $t > 1, p = 2$ case, we start by exploring the effects of persistence.

Let us restrict the problem to $t = 2$, *i.e.* the cache can hold 2 pages, each with 2 operands for a total of 4 operands. If this case can be proved to be NP-complete, the remaining cases of $t > 2$ can be proved NP-complete by a transformation from the $t = 2$ case analogous to the transformation of $t = 1, p > 2$ to $t > 1, p > 2$.

We can restrict the persistence to n references in the reference string by clearing the cache every n operand accesses. We have seen that the cache can be cleared by introducing repeated references to external operands. Optimal page distribution of a reference string whose persistence is restricted to 2 references is polynomially solved by matching techniques similar to the optimal distribution for $t = 1, p = 2$. Example 4.2 illustrates that for the case of $t = 2, p = 2$, reference strings with persistences that last not more than 4 operand references can be solved by matching techniques too.

Example 4.2 Consider the reference string: $\ldots a_1 a_2 \ldots a_n \ldots$. If we want to restrict the persistence to i accesses of program operands, we introduce accesses to external operands, $x_1, x_2, \ldots, x_M$, after every i accesses to the program operands, *e.g.* $\ldots (x_1 x_2 \ldots x_M)^r a_1 a_2 \ldots a_i (x_1 x_2 \ldots x_M)^r \ldots$. In the following tabulation, we correlate for $t = 2, p = 2$ the influence that restricted persistences exert on the optimal pagination of program operands. $ab : 1$ implies that irrespective of the distribution of the other operands, paging a and b together saves a page fault for the current reference substring. ab :? signifies that the advantages of paging a and b together depend on the distribution of the remaining operands.

Persistence 2: $\ldots ab \ldots$	Persistence 3: $\ldots abc \ldots$	Persistence 4: $\ldots abcd \ldots$	Persistence 5: $\ldots abcde \ldots$
$ab : 1$	$ab : 1,\ ac : 1,\ bc : 1$	$ab : 1,\ ac : 1,\ ad : 1,$	$ab : 1,\ ac : 1,\ ad :?,\ ae :?$
		$bc : 1,\ bd : 1,\ cd : 1$	$bc : 1,\ bd :?,\ be :?$
			$cd : 1,\ ce : 1,\ de : 1$

For $t = 2, p = 2$, nondeterminism is introduced for reference strings when persistence is permitted for 5 references. Consider the substring $\ldots abcde \ldots$ of Example 4.2. Operands b and e have an affinity for the same page unless a and d are paged together. Starting from a relatively fresh cache, *i.e.* none of a, b, c, d, or e inhabit the cache, if operands a, b, c, d, e are restricted to the pages ad, ax_1, be, bx_2, and cx_3, then for the substring *abcde*:

ad, be as pages	4 page faults
ad, bx_2 as pages	4 page faults
ax_1, be as pages	4 page faults
ax_1, bx_2 as pages	5 page faults

Thus if we have either ad or be or both as pages, the number of page faults is less than if we have neither. This important distinction is used to reduce Maximum 2-Satisfiability [18] to prove that the case of $t > 1, p = 2$ is also NP-complete.

Theorem 4.4 Static Operand Distribution for $t > 1, p = 2$ is NP-complete.

Proof: As before, Static Operand Distribution for $t > 1, p = 2 \in$ NP.

Let U, C, and J correspond to the arbitrary instance of Max-2SAT. We need to construct, in polynomial time, an instance of Static Operand Distribution for $t > 1, p = 2$ so that there is a solution to Max-2SAT if and only if there is a solution to the corresponding distribution problem. More explicitly, we need to construct a reference string R, and a positive integer k, such that an optimal distribution of the operands for $t > 1, p = 2$ can satisfy the requests of R in at most k page faults if and only if there is a truth assignment to U that satisfies at least J of the C clauses of Max-2SAT.

For each variable $u_i \epsilon U$, the Static Operand Distribution instance has the four operands, x_i, y_i, z_i, v_i. For each clause $c_j \epsilon C$, there corresponds operand w_j. Additionally, the distribution instance has the operands, p_1, p_2, p_3, p_4, that are used for clearing the cache and resetting the persistence. Thus the operands of the distribution instance,
$S = \{x_1, \ldots, x_{|U|},\ y_1, \ldots, y_{|U|},\ z_1, \ldots, z_{|U|},\ v_1, \ldots, v_{|U|},\ w_1, \ldots, w_{|C|},\ p_1, p_2, p_3, p_4\}$,
need to be distributed among p sized pages.

The reference string has 3 types of components: clause-defining, persistence-resetting or cache clearing, and variable-defining. There is a clause-defining substring for each clause c. After every clause-defining substring, the persistence between clauses is cleared by imposing a persistence-resetting substring. There is a variable-defining substring for each variable u of Max-2SAT.

Consider variable u_i. Any valid truth assignment will assign this variable to be either *true* or *false*. In other words, either u_i is valid, or $\overline{u_i}$ is valid, but not both. Such mutual exclusivity can be seen in the paging of operands too – operand x_i can be on the same page with either y_i, or z_i, but not both. Thus if variable $u_i \epsilon U$ is valid, its corresponding operand x_i is paged with y_i, and operand z_i is paged with v_i. Conversely if $\overline{u_i}$ is valid, x_i is paged with z_i, and y_i with v_i.

Consider clause $c_j = (u_1 + \overline{u_2})$. c_j is satisfied for either variable $u_1 =$ *true*, or $\overline{u_2} =$ *true*, or both.

In Max-2SAT we want the truth assignment to the variables, which maximizes the number of satisfied clauses. Therefore we need a way of relating the number of page faults generated in satisfying the substring corresponding to c_j, with the satisfaction of c_j. From our earlier arguments on persistences over 5 references, and the parallel between truth assignments to variables and paginations of the corresponding operands, we notice that there does exist just such a relationship. Thus if $x_1x_2w_jy_1z_2$ is the clause-defining substring corresponding to clause c_j, then the number of page faults for either x_1y_1 paged together *i.e.* $u_1 = true$, x_2z_2 paged together *i.e.* $\overline{u_2} = true$, or both is 4, whereas if neither x_1y_1 nor x_2z_2 is paged together, *i.e.* c_j is unsatisfied, then the number of page faults is 5.

For the clause-defining component as described in the previous paragraph, the persistence has to be restricted to the 5 operands corresponding to the clause. In other words, we want the cache to be in a state so that the accesses of the clause-defining substring incur either 4 page faults if the pages are chosen to satisfy the corresponding clause, or 5 page faults otherwise. The persistence-resetting substring, $(p_1p_2p_3p_4)^n$, appears after every clause-defining substring and performs exactly that function. n is chosen to be large enough that every optimal distribution will herd p_1, p_2, p_3, p_4 into 2 pages and incur only 2 page faults for satisfying the requests of the substring. Any other choice will result in at least $2n + 1$ page faults.

The variable-defining substring simply ensures that either operand x_i can be paged with y_i, in which case z_i can be paged only with v_i, or x_i can be paged with z_i, in which case y_i is to be paged with v_i. Any other page distribution of x_i, y_i, z_i, v_i is meaningless to the instance of Max-2SAT. For each variable u_i, the reference string has the substring, $(p_1p_2x_iy_i)^n$ $(p_1p_2z_iv_i)^n$ $(p_1p_2x_iz_i)^n$ $(p_1p_2y_iv_i)^n$. If page p_1p_2 is available in the cache at the start of the variable-defining substring, a distribution choosing either x_iy_i, z_iv_i, or x_iz_i, y_iv_i as pages, incurs $4n+2$ page faults in satisfying the substring. (Say x_iy_i and z_iv_i are chosen as the pages, and p_1p_2 already resides in the cache; since $t = 2$, we need 1 page fault for loading x_iy_i and this satisfies all of $(p_1p_2x_iy_i)^n$, one page for loading z_iv_i and this satisfies all of $(p_1p_2z_iv_i)^n$, and $2n$ pages for satisfying each of $(p_1p_2x_iz_i)^n$ and $(p_1p_2y_iv_i)^n$.) Any other distribution will require at least $8n$ page faults. Note that p_1p_2 appear in the substrings for each of the $|U|$ variables and therefore any optimal distribution will page them together or risk at least $4n(|U|-1)$ additional page faults. Thus p_1p_2 and p_3p_4 as pages satisfy the persistence-resetting as well as the variable-defining constraints on p_1, p_2, p_3 and p_4.

The structure of the reference string corresponding to the variables $u_1, \ldots, u_{|U|}$ and to the clauses $c_1, \ldots, c_{|C|}$ of the arbitrary instance of Max-2SAT is:

$R = cds_1\ prs\ cds_2\ prs \ldots\ prs\ cds_{|C|}\ prs\ vds_1\ vds_2 \ldots\ vds_{|U|}$,

where cds_i translates to the clause-defining substring corresponding to c_i, prs translates to the persistence-resetting substring $(p_1p_2p_3p_4)^n$, and vds_i translates to the substring,

$(p_1p_2x_iy_i)^n\ (p_1p_2z_iv_i)^n\ (p_1p_2x_iz_i)^n\ (p_1p_2y_iv_i)^n$. Please see Example 4.3.

The construction is complete by relating the value of k, the minimum number of page faults to satisfy R for $t = 2, p = 2$, to J, the maximum number of clauses satisfiable by a truth assignment to U. Each satisfied clause requires 4 page faults and each unsatisfied clause requires 5 page faults. Therefore, we have $4J + 5(|C| - J)$ page faults for the clause-defining substrings. Each persistence-resetting substring optimally requires 2 page faults, giving a total of $2|C|$ page faults for the $|C|$ occurrences of the substring in R. Each variable-defining substring requires $4n + 2$ page faults, giving a total of $(4n + 2)|U|$ page faults for the $|U|$ variable-defining substrings. Note that each variable-defining substring requires $4n + 2$ page faults subject to the prior availability of p_1p_2 in the cache. This is indeed the case because p_1, p_2 are accessed in all contiguous substrings starting from the last persistence-resetting substring, through all the variable-defining substrings, to the end of R. Thus the page p_1p_2 will persist in the cache for all the variable-defining substrings.

Thus $k = 4J + 5(|C| - J) + 2|C| + (4n + 2)|U|$. As we might expect, J and k are inversely related – a distribution that minimizes the number of page faults for R and $t = 2, p = 2$, will maximize the number of clauses satisfied by the corresponding truth assignment, and vice versa. To reiterate, the truth assignment is easy to obtain from the optimal page distribution: x_iy_i paged together implies that the optimal solution for Max-2SAT has variable u_i set to *true*; conversely x_iz_i paged together implies that u_i is set to *false*.

From the construction it is easy to see that if a truth assignment to the variables U satisfies at least J clauses, the corresponding page distribution of the operands requires at most $k = 4J + 5(|C| - J) + 2|C| + (4n+2)|U|$ page faults to satisfy the accesses of R. Conversely, we have chosen n and R such that an optimal distribution of the operands corresponds to a valid truth assignment to the variables of U. Furthermore, if a page distribution requires at most k page faults for R and $t = 2, p = 2$, the corresponding truth assignment satisfies at least $J = 7|C| - k + (4n+2)|U|$ clauses.

All that remains to this NP-completeness proof is arguing that the above transformation is polynomial. The instance of Static Operand Distribution has $4|U| + |C| + 4$ operands. The length of R has a contribution of $4n$ from each persistence-resetting substring of which there are $|C|$, 5 from each clause of which there are $|C|$, and $16n$ for each variable of which there are $|U|$. Thus $|R| = (4n+5)|C| + 16n|U|$. Therefore, the above transformation of Max-2SAT to Static Operand Distribution for $t = 2, p = 2$ is polynomial if n is polynomial in $|C|$ and $|U|$.

The choice of n affects the distribution of p_1, p_2, p_3, p_4 into pages. If these operands are not distributed into 2 pages, we suffer more than $2n + 1$ page faults instead of 2 for each occurrence of the persistence-resetting substring. With $|C|$ occurrences, the difference is $2|C|$ versus $(2n+1)|C|$. A distribution not paging p_1 and p_2 together might, at best, satisfy all $|C|$ instead of J clauses, corresponding to a reduction from $4J + 5(|C| - J)$ to $4|C|$ page faults. Since $J \geq 0$, this corresponds to a saving of $|C|$ page faults. Thus a value of n as low as 2 would deter this alternative. The choice of n also affects the distribution of x_i, y_i, z_i, v_i into pages. Again, choosing a distribution that pages neither $x_i y_i$ and $z_i v_i$, nor $x_i z_i$ and $y_i v_i$, incurs $8n$ instead of $4n + 2$ page faults. If such a distribution, too, saves $|C|$ page faults in satisfying more clauses, a choice of $n = |C|$ would deter such an alternative. Thus it is sufficient that $n = |C|$, and the transformation is indeed polynomial.

$\square$

Example 4.3 Consider the arbitrary instance of Max-2SAT: $U = \{u_1, u_2\}, C = \{(u_1 + u_2), (\overline{u_1} + u_2)\}, J$. We will construct, in polynomial time, an instance of Static Operand Distribution for

$t = 2, p = 2$: R, S, k, such that there is a distribution of the operands into pages that can satisfy the requests of R in k page faults if and only if there is a truth assignment to U that satisfies at least J of the C clauses.

$$\begin{aligned}
S &= \{x_1, x_2, y_1, y_2, z_1, z_2, v_1, v_2, w_1, w_2, p_1, p_2, p_3, p_4\} \\
n &= 2 \\
R &= \underline{x_1x_2w_1y_1y_2}\ \underline{p_1p_2p_3p_4p_1p_2p_3p_4}\ \underline{x_1x_2w_2z_1y_2}\ \underline{p_1p_2p_3p_4p_1p_2p_3p_4} \\
&\ \underline{p_1p_2x_1y_1p_1p_2x_1y_1p_1p_2z_1v_1p_1p_2z_1v_1p_1p_2x_1z_1p_1p_2x_1z_1p_1p_2y_1v_1p_1p_2y_1v_1} \\
&\ \underline{p_1p_2x_2y_2p_1p_2x_2y_2p_1p_2z_2v_2p_1p_2z_2v_2p_1p_2x_2z_2p_1p_2x_2z_2p_1p_2y_2v_2p_1p_2y_2v_2} \\
|R| &= 90 = (4n+5)|C| + 16n|U|
\end{aligned}$$

$(u_1, u_2) = (true, true)$ is an optimal truth assignment for which $J = 2$.

The corresponding page distribution,

$(x_1y_1, z_1v_1, x_2y_2, z_2v_2, p_1p_2, p_3p_4, w_1w_2)$, requires 32 page faults to satisfy the requests of R. Thus, $k = 32 = 4J + 5(|C| - J) + 2|C| + (4n+2)|U|$.

4.1.2 Complexity of the Dynamic Distribution Problem

Since the page contents in dynamic distribution may be shuffled in the cache, the dynamic distribution problem reduces to the static distribution problem when the cache can hold only one page. Thus in the $M = tp$ domain of the dynamic distribution problem, the cases corresponding to $t = 1$ have the same arguments and complexity as the corresponding static cases discussed in the previous section. The distribution of operands for $p = 1$ is similarly trivial. Therefore, the only 2 important cases are:

case 1: $t > 1, p = 2$

case 2: $t > 1, p > 2$

In some sense the dynamic distribution problem is more complex than the static distribution problem because we have to determine not only the optimal starting distribution, but also the page to load and the p operands to flush as a page out of the cache. Unlike the case of static distribution where the page to be flushed out of the cache, a potential of t choices, is determined by a simple polynomial lookahead of the reference string (the *greedy* approach), dynamic distribution has no such simple determinism pointing to the p operands that should be paged out, a potential of $\binom{M}{p}$ choices. In the following example, the greedy strategy pages out the p operands that occur after the remaining $M-p$ operands in the cache.

Example 4.4 Our problem has $M = 4, p = 2, |S| = 6$. Assume that we have reached a state where the remaining reference string, current contents of the cache, and current pagination, which describes at any state the distribution of the $|S| - M$ operands into p sized pages, are given by:

Reference String: *abcdefcdbdec*

Current Cache Contents: *abcd*

Current Pagination: *ef*

Cache Transitions *(Greedy)*: $abcd \rightarrow efcd \rightarrow abde \rightarrow cfab$.

Cache Transitions *(Optimal)*: $abcd \rightarrow abef \rightarrow cdbe$.

Note that for any state where $|S| = (M-1)p$, the current pagination is trivial because the p operands not in the cache have to be paged together.

Similar to the nondeterminism in the outgoing page, dynamic distribution has a nondeterminism in the incoming page too. The greedy strategy, which on encountering an access whose operand is not in cache, loads the page containing the operand, and which is proved to be optimal for static distribution, is not always optimal in the dynamic case. This is illustrated in the following example.

Example 4.5 Our problem has $M = 4, p = 2, |S| = 8$. Assume that we have reached a state where the remaining reference string, current contents of the cache, and current pagination, are given by:

Reference String: *abcdefagefag*

Current Cache Contents: *abcd*

Current Pagination: *ef*, *gh*

Cache Transitions *(Greedy)*: $abcd \rightarrow efab \rightarrow ghef \rightarrow abgh$.

Cache Transitions *(Optimal)*: $abcd \rightarrow ghab \rightarrow efag$.

Note that for $|S| = (M-1)p$ there is only one possible incoming page, and hence no uncertainty in the page to be loaded. For $M = 4, p = 2$, there is an uncertainty in the incoming page for $|S| \geq 8$, and as in the example, it is not optimal to always load the page containing the faulting operand.

Let us attempt to understand the source of the complexity in static page distribution as a precursor to gaining an intuition into the complexity of the dynamic distribution problem. In static page distribution each cache state was a cluster of operands that satisfied accesses of the reference string. Alternatively, successive accesses of the reference string had to be satisfied by either the same or successive clusters. The intersection between the contents of successive clusters was defined as the persistence. In static page distribution the operands were paged to permit the construction of "profitable" clusters, where the profitability of a cluster is related to the number of accesses in the reference string that it can satisfy, and also to the persistence that it offers in the construction of proceeding clusters. Once the operands were distributed into pages, the optimal clustering was polynomially determinate.

In dynamic page distribution, the objective is similarly to facilitate the construction of profitable clusters. Determining the optimal starting distribution appears no easier than the static case. However, even with a given starting page distribution, the clustering of operands in the cache constricts, and is constricted by, the choice of page that is loaded into the cache at a page fault, and by the choice of the p operands that are paged out of the cache at a page fault. Therefore, even with a given starting distribution, there is a nondeterminism in the optimal clustering.

It is NP-complete to determine the optimal starting page assignment for the dynamic operand distribution problem for $t > 1, p = 2$ and $t > 1, p > 2$. Moreover, for a given starting page assignment, it is NP-complete to determine the optimal page to load and p operands to spill in the dynamic operand distribution problem for $t > 1, p = 2$ and $t > 1, p > 2$. The proofs parallel the arguments for Static Operand Distribution for $t > 1, p = 2$.

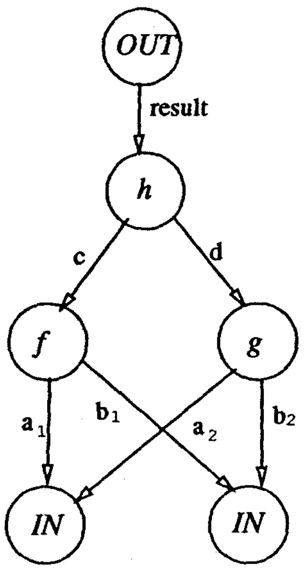
4.2 Distributing Array Data into Pages

4.2.1 Complexity of the Optimal Array Layout Problem for Declarative Languages

Problem Statement: Given a program graph, where the nodes represent the operators in the program computation, and the edges represent the array operands and results of the corresponding operators. The leaf edges incident on the *IN* operators represent the atomic operands of the program, while the others represent the intermediate operands or the partial results of the computation. The execution cost, including the cost of accessing the operands from the memory hierarchy, of the operator at each node depends on the layout of the array operands and results, and each node is attended with a tabulation of this cost for all the possible combinations of storage organizations of the operands and result of the node. Determine the layout for all the operands, atomic and intermediate, that minimizes the total cost of the program computation.

An example program graph with the concomitant cost tables can be found in Figure 4.3.

Note that the benign cost table of each node conceals an additional exponential cost – the number of possible distributions of an n-dimensional operand array. An n-dimensional operand may be stored in any one of $n!$ ways, for the possible combinations of major-minor axes as encoded by the q-vectors $1\ 2 \ldots n$, $2\ 1\ 3 \ldots n$, etc. This does not entertain the possibility of storing some of the axes as subarrays as represented by q-vectors $1\ 1\ 2 \ldots (n-1)$, $2\ 1\ 2\ 3 \ldots (n-1)$, etc. If we include such partial subarray layout, an n-dimensional operand may be stored in any of $\sum_{i=1}^{i=(n-1)} \binom{n}{i} (n-i)!$



function		
operand_1	operand_2	cost
r	r	10
r	c	110
c	r	110
c	c	200

function			
operand_1	operand_2	operand_3	cost
r	r	r	30
r	r	c	120
r	c	r	120
r	c	c	210
c	r	r	120
c	r	c	210
c	c	r	210
c	c	c	300

Figure 4.3: Program graph and computation cost tables

ways.

The following complexity arguments assume polynomially restricted cost tables, *i.e.* the operand arrays may be stored in any of only a polynomial number of layouts.

Problem: Array Operand Distribution for Declarative Languages (**AODDL**)

INSTANCE: Program Graph $G = (V, E)$, polynomially restricted cost table for each node $v \epsilon V$, positive integer J.

QUESTION: Is there an assignment of layout to each edge $e \epsilon E$ such that the sum of the costs of the node computations is at most J?

Clearly, this decision problem is no harder than the optimization problem. If we could find the assignment of layouts to the operands that minimizes the cost of the computation, then we could use that assignment to find the minimum cost in polynomial time, and compare this number with the given bound J. Thus if AODDL is proved to be NP-complete, the corresponding optimization problem is at least as hard.

Theorem 4.5 AODDL is NP-complete.

Proof: It is easy to see that AODDL ϵ NP since a nondeterministic algorithm need only guess an assignment of layout to operands and check in polynomial time whether the sum of the costs of all the computation nodes is less than the bound J.

We transform 3SAT to the restricted AODDL where each operand can be distributed in one of only two possibilities. Let the sets C and U correspond to the arbitrary instance of 3SAT, where $C = \{c_1, c_2, \ldots, c_m\}$ is the set of clauses, and U is the set of variables. We need to construct, in polynomial time, an instance of AODDL so that there is a solution to 3SAT if and only if there is a solution to the corresponding distribution problem. More explicitly, we need to construct a program graph $G = (V, E)$, the cost tables at each node, and a positive integer J, such that an assignment to the edges incurs a cost at most J if and only if there is a truth assignment to U that satisfies the C clauses of 3SAT.

For the corresponding program graph $G = (V, E)$, $V = \{x_1, x_2, \ldots, x_{|U|}, z_1, z_2, \ldots, z_{|C|}\}$, $E = \{(x_i, z_j)|$ variable u_i appears in any form, negated or non-negated, in clause $c_j\}$. Thus the vertices can be partitioned into the *variable* vertices x_i and the *clause* vertices z_j. Consequently, there are no edges (x_p, x_q) or (z_r, z_s), and each vertex z_j has a degree of 3.

Before we attempt to articulate the construction of the cost tables, it will be useful to motivate the endeavor. The layout of the array operand corresponding to each edge is restricted to the set (T, F) to mirror a truth assignment. Now for a clause to be satisfied, at least one of its literals should be *true*, *e.g.* for the clause $(a + \bar{b} + \bar{c})$ to be satisfied, at least one of $a, \bar{b}, \bar{c}$ need to be *true*; equivalently either $a =$ *true*, or $b =$ *false*, or $c =$ *false*, or any combination thereof. Conversely, only one truth assignment to the variables will fail to satisfy the clause. In the example above, only if $a =$ *false*, $b =$ *true*, $c =$ *true*, the clause $(a + \bar{b} + \bar{c})$ is not satisfied.

The set of possible layouts, (T, F), is synonymous with the truth assignments *(true, false).*

The computations costs in the tables for the clause nodes are also binary, $(0, 1)$. For the layout assignment to the three edges incident on the clause node that does not satisfy the clause, the cost of the computation is 1; in all the other cases the computation cost is 0. Thus only if all the clauses are satisfied, the sum of the costs of the clause node computations is 0.

While the cost tables at the clause codes insure clause satisfiability, the cost tables at the variable nodes insure consistency of the assignments to the edges. In isolation, clause nodes might promote an assignment to the edges such that the corresponding 3SAT variables assume different truth assignments for different clauses. This inconsistency is rectified by the cost tables at the variable nodes. The computation costs in the tables for the variable nodes are also binary, $(0, 1)$.

Consider the n appearances of variable u_i in the clauses. Each of these appearances corresponds to an edge incident on the variable node x_i corresponding to the variable u_i, and also incident on the appropriate clause node z_j. For identical layout assignments, either all T or all F, to these n edges incident on x_i, the cost of the computation is 0; in all other cases the computation cost is 1. Thus only if all the variables are assigned consistently, the sum of the costs of the variable nodes in 0.

Thus the sum of the costs of the node computations, both clause nodes and variable nodes, is 0 only for an assignment of layout to the edges in the program graph that is consistent, and that satisfies all the clauses. The construction is complete by assigning a value of 0 to J. Example 4.6 illustrates the construction of the Array Operand Distribution that corresponds to a given arbitrary instance of 3SAT.

From the construction it is patent that any assignment of layouts to the edges in the program graph for which the sum of the costs of the node computations is 0, is synonymous to a truth assignment to the variables of the corresponding instance of 3SAT that satisfies all the clauses of the 3SAT instance. The synonym lies in the identity of the possible layouts (T, F) with the possible truth assignments *(true, false)*. Conversely, any valid truth assignment to the variables of the 3SAT instance will incur 0 cost at the variable nodes because the truth assignment is consistent across the clauses, and hence the edges incident on a variable node will be identically assigned corresponding layouts. Moreover, if this truth assignment satisfies the 3SAT clauses, the clause nodes will incur 0 cost for the corresponding layout assignment, and the sum of the costs of all the node computations in the program graph will be at most $J = 0$.

All that remains to this NP-completeness proof is the argument that the above construction is polynomial. The program graph, $G = (V, E)$, has number of nodes $|V| = |C| + |U|$, and number of edges $|E| = 3|U|$. The cost table at each clause node has basically 2 entries, one that does not satisfy the clause and has a cost of 1, and the others that do satisfy the clause and cost 0. The cost tables at each variable node have 3 entries, one each for identical T and F assignments to all the edges incident on the node, which has a cost of 0, and the others, which cost 1. Hence the transformation is indeed polynomial.

□

Example 4.6 Consider the arbitrary instance of 3SAT:

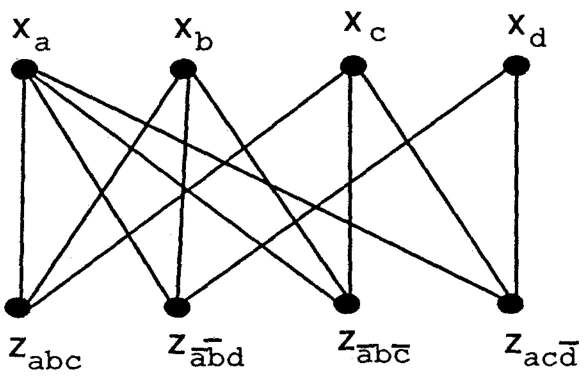
$U = \{a, b, c, d\}, C = \{(a + b + c), (\overline{a} + \overline{b} + d), (\overline{a} + b + \overline{c}), (a + c + \overline{d})\}$.

In the corresponding program graph $G = (V, E)$ of Figure 4.4, $V = \{x_a, x_b, x_c, x_d, z_{abc}, z_{\overline{a}\overline{b}d}, z_{\overline{a}b\overline{c}}, z_{ac\overline{d}}\}$. The cost tables for the nodes accompany the nodes in the figure. Each edge is annotated by an assignment of layout to the operand represented by that edge, such that the cost of the program computation is 0.

The interpretation of (T, F) as *(true, false)* gives the corresponding truth assignment to the 3SAT variables, which satisfies all the clauses.

variable node

clause_1	clause_2	clause_3	cost
T	T	T	0
F	F	F	0
-	-	-	1



clause node $z_{\bar{a}b\bar{c}}$

x_a	x_b	x_c	cost
T	F	T	1
-	-	-	0

Figure 4.4: 3SAT instance and corresponding program graph

Chapter 5

Concluding remarks

We have presented a new type of optimization: the distribution or layout of data. The impetus for the judicious distribution of data derives from two sources: from the hardware, in that the data organization drives the parallelism of most current architectures, and from the software, in that the use of most high-performance architectures, precisely the ones that are data-driven, is crowded with scientific programs with small code sizes computing over large data aggregates.

The payoffs of Section 2.5, a reduction in line misses by a factor up to two orders of magnitude, are especially significant given the high cost of line misses and of the communication bottleneck in most systems. The data optimization is based on the principles of locality in general, without regard to the interpretation of locality peculiar to any one computing environment. Once the locality of the data accesses has been determined by the compile-time techniques presented, then the data can be distributed to map the communication paradigm of any architecture, *viz.* shared or distributed memories, hierarchical memories, interleaved memories, vector processors, etc. This embodies one aspect of the general applicability of our work.

Another manifestation of the generality can be observed in the different levels at which the data optimization transformations can be applied. These range from the completely non-invasive where the data is distributed after all traditional code optimizations, to the transgressive where the data distribution constraints motivate some of the code transformations. The examples in the thesis illustrated the counterproductivity of some of the traditional optimizations to increasing system efficiency and performance, and identified the potential of higher payoffs in the latter, more aggressive approach. These motivate a reassessment of traditional code optimizations to target both the execution and communication efficiencies, rather than either in isolation. Furthermore, the "input renaming" and "dynamic restructuring" transformations were developed to exemplify new code optimizations motivated by the new constraints. Note that the experiments, based on the non-invasive approach, represent the conservative rewards of data optimization.

It was argued that the distribution of the data should be attempted only by the compiler. The two ramifications of this assertion are: Firstly, it bucks the trend of relegating data distribution to the programmer. Examples illustrated the drawbacks of user-optimized data distributions. Secondly, it motivates the development of automatic, data distribution techniques integral to compiler optimization. The techniques developed in this thesis are simple and can be easily incorporated into a compiler.

Different techniques were developed for the optimization of different types of data in different types of languages: `DistributionAlgorithm()` for the distribution of scalar data, the remap algorithm for the distribution of array data in declarative languages, and the stock suite for the distribution of array data in procedural languages. The distribution of scalar data necessitates a good compile-time handle on the reference string. The reference regular expression developed clearly identifies the sources of access non-determinisms in the source code. From the alternatives for approximations at these non-determinisms that were presented, even the simplest and computationally least expensive approximation accurately distributed the scalar data in the experiments. The resulting storage optimizations of both scalar and array data can be implemented within the current framework and practices by appropriately permuting the order of scalar definitions, and by transforming the shapes of array definitions, respectively.

The constraints on the distribution of the data were accumulated from all accesses of the data.

The optimization of the data to these global constraints was proved to be NP-complete. The fundamental proposal of optimizing the storage of data, either after, or concurrently with, the optimization of code, and the complexity proofs are the long-term contributions of this thesis. They will influence compiler optimizations of the future.

We see a further explosion in architectures fueled by advances in technology, both hardware and software. Even today, the level of parallelism, ratio of execution to communication cost, etc. vary widely between different systems, or even systems from the same family. It will become impossible to target a different compiler to each system. Just like the development of intermediate languages, and table-driven code generation, it will become necessary for compilers to optimize to parametrized system descriptions. This thesis has advanced that endeavor by the use of parameters M, t, p in optimizing the storage of data.

Appendix A

Working of the Remap Algorithm: Example 2

From the multiplication algorithm presented in Section 3.1.1, we saw that the multiplication of two n-dimensional square matrices, A and B as the left and right operands respectively, is optimal when A is stored in 2–minor and B is stored in 1–minor order. Extrapolating this observation to array operands of higher rank is relatively straightforward. For a rank–3 operand A and a rank–4 operand B, the optimal storage order for A is 3–minor and for B is 1–minor order; the relative order of the remaining axes is determined by the constraints on the result. If the rank–5 result, C, has constraints encoded by the q–vector: 5 2 1 4 3, the constraints on A are given by the q–vector: 2 1 3, and that on B by q–vector: 4 1 3 2.

From the semantics of array multiplication, we know that the first two axes of C correspond to the first two axes of A, and the remaining three axes of C correspond to the last three axes of B; the third axis of A and the first axis of B are reduced by the multiplication. From the constraints on C we now proceed to develop the q–vector for A; the considerations for operand B are congruent. The q–vector for A, 2 1 3, satisfies the 3–minor order requirement of the previous paragraph. Additionally, the first axis of C is minor relative to the second axis, and since the first two axes of C are composed of the first two axes of A, the first axis of A too is relatively minor to its second axis. Thus the page accesses for operand A will cycle through the first axis before the second, and the combined page accesses of the multiplication computation will cycle through operands A and B in the following order: axis–3 of A, axis–1 of B, axis–1 of A, axis–3 of B, axis–4 of B, axis–2 of A, axis–2 of B. This order follows from the q–vector: 5 2 1 4 3 of the result.

In Example 2 we apply the remap algorithm to an equivalent program for computing the inner product of two arrays. The operands A and B, and the result C, are the same as above: C is constrained by the q–vector: 5 2 1 4 3, A has a rank of 3, B has a rank of 4, and the last axis of A and the first axis of B have the same extent. Figure A.1a depicts the APL program, and Figure A.1b labels the program variables with information from the dimensions pass. The rank of neither operand A nor B can be deduced from the dimensions pass through the program. The remap algorithm could assume variables r_a and r_b and proceed; however, for purposes of illustration let us assume that from some other program segment, r_a can be deduced to be 3 and r_b to be 4. The exact shape vector remains unknown. Therefore, Figure A.1b assumes shape vectors consisting of variables $a \ldots f$ and conforming to the ranks r_a and r_b. Figure A.1c incorporates the dimension information of Figure A.1b to define the expression in terms of A and B.

Referring to Figure A.1c, the q–vector in the remap algorithm starts with the constraints on the result of the program, *i.e.* q–vector: 5 2 1 4 3. From the discussions of Example 1 and Section 3.2.2, the operand of $+/[KA]$ (the same as $+/[3]$), *i.e.* $TA \times TB$, inherits the q–vector: 5 2 6 1 4 3. As with the array add operation before, the $\times$ operator of $TA \times TB$ does not introduce any additional constraints, and the storage organizations of both TA and TB are constrained by the q–vector: 5 2 6 1 4 3.

The q–vector constraining TA is now pushed down the expression that computes TA to obtain the storage constraints on operand A. The first operator we encounter is the dyadic transpose with ZA as the control argument and $WA\rho A$ as the operand; let variable X be the result of $WA\rho A$. The

```
                                   ρA = a b c; ρB = c d e f
WA ← (1 ↓ ρB),ρA                       WA = d e f a b c

VA ← ιρWA                              VA = 1 2 3 4 5 6

KA ← ρρA                               KA = 3

ZA ← KA ⌽ VA                           ZA = 4 5 6 1 2 3

TA ← ZA ⍉ WAρA                         ρTA = a b c d e f

WB ← (¯1 ↓ ρA),ρB                      WB = a b c d e f

TB ← WBρB                              ρTB = a b c d e f

       (a)                                   (b)
```

```
C ↔ A+.×B ↔ +/[KA]TA × TB ↔ +/[3](4 5 6 1 2 3 ⍉ d e f a b c ρA)×(a b c d e f ρB)

                                  (c)
```

Figure A.1: (a) The APL program segment for multiplying two arrays. (b) The program variables with values obtained in the dimensions pass. (c) A simplified expression for the result C

dimensions pass gives us (4 5 6 1 2 3) as the value of ZA. In the normal right–to–left execution of the dyadic transpose, the fourth axis of X would have become the first axis of the result. From the q–vector of the result we know that this first axis of the result has a q–vector value of 5; thus the left–to–right accumulation of the dyadic transpose into the q–vector gives 5 in the fourth position of the q–vector constraining X. By extending this reasoning, the accumulation of the dyadic transpose with control argument (4 5 6 1 2 3) transforms the q–vector: 5 2 6 1 4 3 constraining TA into the q–vector: 1 4 3 5 2 6 constraining X.

The q–vector constraining X is now pushed down the expression that computes X, *viz.* $WA\rho A$. Borrowing a terminology from Guibas & Wyatt [20], the reshape in this expression is called a "conforming" reshape because WA is of the form: $(Y, \rho A)$, where Y is a vector or a scalar. Such a reshape preserves the structure of A; it merely creates multiple copies of A along the added dimensions Y. Conforming reshapes pass on the "normalized" suffix of the q–vector to the operand. The corresponding suffix of the q–vector, Q, that will be inherited by A is: $(-\rho\rho A) \uparrow Q$. The range of the values in the q–vector suffix pertains to the parent q–vector Q, *i.e.* between 1 and $\rho\rho Q$. Normalization conserves the relative order of the suffix values but makes them pertinent to operand A, *i.e.* it converts the values to lie between 1 and $\rho\rho A$. If S is the unnormalized suffix, then $+/[2](S \circ . \geq S)$ normalizes S.

In our case the relevant suffix, S, of the q–vector is: 5 2 6, and the q–vector passed on to operand A is: 2 1 3. This constraint on operand A is the same as that obtained by the analysis of the multiplication of arrays A and B with the rank–5 result, C, constrained by the q–vector: 5 2 1 4 3. A similar set of manipulations shows that operand B is constrained by the q–vector: 4 1 3 2.

Appendix B

Operand Storage Constraints of APL Operators

(index–origin: 1)

The tabulation follows the format:

Operator	*General Case*	*Operand Type*

Most Optimal Operand Storage

- **Shape** ρA A : any

  No storage constraints on operand A.

- **Rank** $\rho\rho A$ A : any

  No storage constraints on operand A.

- **Reshape** $A\rho B$ A: vector; B: any

  For conforming reshapes the storage organization of B is not constrained and axes $(-\rho B)\uparrow\rho A$

of the result, which are copied along the added dimensions $((\rho A) - \rho B) \uparrow \rho A$ of the result, have the same storage order as operand B. For all other reshapes the most optimal storage organization of B is ravel and the result is stored in ravel order too.

- **Index** $A[I_1; I_2; \ldots]$ $I_1, I_2, \ldots$: vector

The axis with the largest indexed extent relative to the length of the respective axis should optimally be the minor axis *i.e.* p–minor axis, if $\forall j, \mathtt{length}(I_p)/((\rho A)[p]) \geq \mathtt{length}(I_j)/((\rho A)[j])$. The result has the same storage order as operand A.

- **Ravel** $,A$ A : any

Operand A is best stored in ravel order.

- **Reduction** $f/[I]B$ f : scalar fn.; I : axis of B; B : array

If I is known at compile–time, B is optimally stored in I–minor order; and if I cannot be deduced at compile–time, B is best stored in subarray order. The result has the storage order of B with axis I elided.

- **Scan** $f\backslash[I]B$ f : scalar fn.; I : axis of B; B : array

If I is known at compile–time, B is optimally stored in I–minor order; else B is best stored in subarray order. The result has the same storage order as B.

- **Outer product** $A \circ .fB$ f : scalar dyadic fn.; A : any; B : any

No storage constraints on either operand A, or B. The result, however, is stored in an order produced by merging the storage orders of A and B, respectively.

- **Inner product** $Af.gB$ f, g : scalar dyadic fns.; A : any; $B : 1 \uparrow \rho B = {}^{-}1 \uparrow \rho A$

Inner product has an interesting tradeoff between memory requirements and memory efficiency. However, subarray storage for A and B, yielding a result in subarray storage too, has the best space–time characteristics.

- **Reverse** ⌽ $[I]A$ I : axis of A

No constraints on the storage of A. The result has the same storage order as operand A.

- **Monadic transpose** ⍉ B B : any

Monadic transpose does not add any constraint to the storage of operand B; it merely transposes the constraints alongwith the axes.

- **Matrix inverse** ⌹B $\rho\rho B = 2$

The most optimal storage for operand B depends on the implementation; perhaps it is subarray like the inner product.

- **Catenate** $A,[I]B$ A : any; B : any array; I : integer

Since catenate combines the arrays A and B into one entity, A and B should be stored in the same order that is also the storage order of the result; and this order is preferably I–major. Subarray storage for the operands is a safe bet that satisfies the common storage order requisite and yields a result in subarray order too.

- **Laminate** $A,[I]B$ A, B : any; I : non–integer

Similar to catenate above, laminate expects operands A and B to be stored in the same order. The added axis, $\lceil I$, becomes the major axis in the result; the remaining axes in the result retain the relative storage order of the operands. The laminated operands, A and B, have no preferred storage order. Note that if A and B are stored in subarray order, the subarray order of the result will not incorporate the added axis $\lceil I$, and is therefore a skewed subarray.

- **Index of** $A\iota B$ A : vector; B : any array

B has no storage constraints. The result has the same storage order as operand B.

- **Member of** $A\epsilon B$ A, B : any

No storage constraints on either A or B. The result has the same shape and storage order as A.

- **Compress** $L/[I]A$ A : any; L : binary vector; I : integer

  A is best stored in I–major order. The result has the same storage order as operand A.

- **Expand** $L\backslash[I]A$ A : any; L : binary vector; I : integer

  A is best stored in I–major order. The storage order of the result is the same as that of operand A.

- **Rotate** $A \;⌽\; [I]B$ $(\rho A) = I \downarrow \rho B$; B : any; I : integer

  B is best stored in I–minor order. The remaining axes of B should have the same relative storage order as the axes of A. The result is stored in the same order as B.

- **Decode** $A \perp B$ A : any; $B : 1 \uparrow \rho B = {}^{-}1 \uparrow \rho A$

  Operand A is best stored in $\rho\rho A$–minor order and operand B is best stored in 1–minor order. The storage of the result is determined by the relative storage orders of the remaining axes of A and B. However, if additional memory (= `SizeOfSubarray` $\div$ ${}^{-}1 \uparrow$ `ShapeOfSubarray`) can be used to forward partial products, the inner product dominates and constrains the storage of A and B to be subarray.

- **Encode** $A \top B$ A, B : array

  A is best stored in 1–minor order. The other axes of A and the storage of B are not constrained. The result with shape $\rho A, \rho B$ has the minor axis of A, *i.e.* axis 1, and the minor axis of B, as the two most minor axes; the storage order of the remaining axes of the result is determined by the storage order of the remaining axes of A and B.

- **General dyadic transpose** $A \;⍉\; B$ A : vector; B : any

  If control vector A is known at compile–time and has no repeated elements, the storage order of operand B is not constrained. The operator transposes any previous constraints on the storage of operand B along with the axes to give the new axes and storage order of the result. If, however, A is not statically determinate, operand array B is best stored as a subarray. And if there are repeated elements in A, the corresponding axes should be stored as subarrays in the most optimal storage of operand B. Subarray storage for B, in whatever shape, yields a result with the same subarray storage.

- **Take** $V \uparrow A$ A : any; V : vector

  If V is statically determinate and the different axes have largely disparate extents taken, then the axes with the largest taken extents relative to the lengths of the respective axis are the preferred minor axis for the storage of A. In all other cases A should optimally be stored as subarrays. The result is stored in the same order as operand A.

- **Drop** $V \downarrow A$ A : any; V : vector

  Similar storage constraints as in Take except that the preferred minor axes have large relative extents remaining, or in other words the axes with the smallest relative extents dropped are the preferred minor axes. Subarray storage is similarly a hedged bet.

- **Matrix divide** $A⌹B$ A, B : matrices

  The storage constraints on operands A and B, like the inner product case, are dependent on the implementation algorithm. Subarray storage is probably the best.

- **Indexed assignment** $A[I] \leftarrow B$ A, B : arrays

  The storage constraints are identical to those of the index operator, fourth in this tabulation. Indexed assignment merits an independent itemization for the additional constraint: the storage order of the indexed axes of A should be identical to that of B.

Note that all the observations above are for constrained memory; the constraints may be parametrized by an interleave factor, the number versus the size of pages at the various levels of a hierarchical store, the number and size of operands and results, the shape of the operands relative to the size of a page, etc. The severity of the constraint determines the importance and payoffs of the data distribution. A hierarchical non–interleaved store is assumed in the discussions of constrained memory systems that follow.

At one end of the spectrum lies the completely unconstrained state with sufficient primary memory, *e.g.* cache, for all operands and results. Similarly, a memory with many, small pages is more tolerant of a nonoptimal data distribution. Also, operand sensitivity to optimal data distribution is proportional to the skew in the shape of the operand.

Subarray storage offers a good compromise, a sort of hedging of bets between the optimality of an optimal storage order and the inefficiency of a non–optimal storage order. As we saw in Section 3.2.9, it is also the resolution of a compromise between conflicting constraints. Moreover, operators with operands stored in subarray order usually yield their result in subarray order too. Even though the subarray order of the result is oftentimes imperfect, where the *perfect subarray* is defined as having equal extents along all axes, the skewed subarray storage is usually a better default than ravel.

Note that perfect subarray storage is recommended in cases where nothing is known of the operators at compile–time. When the knowledge is partial or when intelligent guesses can be made, the subarray storage can be deliberately skewed to represent the information available. This notion is encapsulated in the cost measure of Section 3.2.9.

The operand storage constraints of all APL operators depend entirely on the implementation algorithm used. Thus, when the table asserts that an operand has no storage constraints, the implication is that there is no preferred sequencing of operations in the operator semantics. As an example, the outer product does not impose any constraints on the storage of its operands, A and B. However, if the algorithm is encoded such that all the axes of A are cycled before any axis of B, the first $\rho\rho A$ axes of the result of the outer product will be minor with respect to the remaining axis (which are drawn from B). The storage organization of A is similarly constrained by the nesting order of the loops that cycle through A, independent of the interspersing of the loops cycling through B. For example, in the implementation of the outer product if the last axis of A is nested deeper than the remaining axes of A , *i.e.* axis $\rho\rho A$ of A moves faster than the remaining $1 \ldots (\rho\rho A) - 1$

axes of A, then the outer product constrains operand A to be stored with $\rho\rho A$ as the minor axis.

Conversely, even if a logical operation sequencing is tacit in the operator semantics and if on that basis we assert the operand storage constraints in the tabulation above, the algorithm implementing the operator may access the operand in an entirely different order. It is this access order of the operand that then dictates the optimal storage of the operand. In this light the constraints on the operand storage order tabulated above are merely recommendations. The compiler should impose operand storage constraints that conform to the algorithm implementations provided. As an example, the table declares that the operand of a reduction over axis i is optimally stored in i–minor order. However, if the reduction algorithm is implemented to cycle through the operand with axis j moving the fastest, then the optimal storage order of the operand is j–minor and not i–minor. Similarly, the operands to the inner product are constrained to be stored as subarrays only because we expect that the inner product will be implemented by the efficient block–multiplication algorithm.

This brings up the possibility that the algorithm implementation be flexible to accommodate operands with constrained storage. This can be achieved if the algorithm is provided in source form so that the compiler can not only constrain the operand storage to conform to the nesting of the loops accessing the operands, but whenever necessary it can also permute the loops in the algorithm to access constrained operands optimally. Such compiler control of the data locality by collaborative manipulations of the data and the code promises the best results.

Bibliography

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. Automatic program transformations for virtual memory computers. *National Computer Conference*, pages 969–974, 1979.

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

[3] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison–Wesley, Mass., 1979.

[4] F. E. Allen. Control flow analysis. *Sigplan Notices*, (5:7), 1970.

[5] John Randal Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[6] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[7] Inc. American National Standards Institute. Fortran 8x draft. Technical Report 4, SIGPLAN Special Interest Publication on Fortran, 1989.

[8] Jean-Loup Baier and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Trans. on Software Engineering*, SE–2(1):54–62, March 1976.

[9] L. A. Belady. A study of replacement algorithms for a virtual–storage computer. *IBM Systems Journal*, 1966.

[10] John Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. Technical Report Rice COMP TR89-98, Rice University, Nov. 1989.

[11] Paul Budnick and David J. Kuck. The organization and use of parallel memories. *IEEE trans. on Computers*, Dec. 1971.

[12] Michael W. Condry. On arbitrary array pagination strategies: Storage and reorganization. *Proc. of the Johns Hopkins Conference on Information and System Sciences*, pages 385–390, March 1978.

[13] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall, Inc., 1974.

[14] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The ACM Doctoral Dissertation Award Series, The MIT Press, 1985.

[15] Domenico Ferrari. The improvement of program behaviour. *IEEE Computer*, pages 39–47, Nov. 1976.

[16] Patrick C. Fischer and Robert L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Comm. of the ACM*, 22(7):405–415, July 1979.

[17] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel & Distributed Computing*, (5):587–616, 1988.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness.* W. H. Freeman & Company, 1979.

[19] Ed Grochowski and Ken Shoemaker. Issues in the implementation of the i486^{tm} cache and bus. *IEEE*, 1989.

[20] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. *Fifth Annual Symposium on Principles of Programming Languages*, pages 1–8, 1978.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[22] Edward G. Coffman Jr., G. J. Burnett, and R. A. Snowdon. On the performance of interleaved memories with multiple-word bandwidths. *IEEE trans. on computers*, Dec. 1971.

[23] Kathleen Knobe, Joan D. Lukas, and Jr. Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel & Distributed Computing*, (8):102–118, 1990.

[24] D. E. Knuth. An empirical study of FORTRAN programs. *Software Practices & Experience*, (1:2), 1971.

[25] David J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley & Sons, Inc., 1978.

[26] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE trans. on computers*, C–24(12), Dec. 1975.

[27] Mary E. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic Publishers, 1978.

[28] Mary E. Mace and Robert A. Wagner. Globally optimum selection of memory storage patterns. 1985.

[29] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, (2):78–117, 1970.

[30] A.C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Comm. of the ACM*, 12(3):153–165, March 1969.

[31] David Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. of the ACM*, 29(12):1184–1201, Dec. 1986.

[32] Piyush Patel and Diane Douglass. Architectural features of the i860(tm) microprocessor risc core and on-chip caches. *IEEE*, 1989.

[33] Constantine D. Polychronopoulos. Automatic restructuring of fortran programs for parallel execution. Technical Report 665, Univ. of Illinois, Urbana–Champaign, June 1987.

[34] Allan K. Porterfield. Software methods for improvement of cache performance on supercomputer applications. Technical Report COMP TR89-93, Rice University, May 1989.

[35] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C - The Art of Scientific Computing.* Cambridge University Press, 1988.

[36] Jr. Richard P. LaRowe and Carla Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. Technical Report CS-1990-10, Duke University, April 1990.

[37] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3), Sept. 1982.

[38] Gerard Tel and Harry A. G. Wijshoff. Hierarchical parallel memory systems and multiperiodic skewing schemes. *Journal of Parallel & Distributed Computing*, (7):355–367, 1989.

[39] Kishor S. Trivedi. Prepaging and applications to array algorithms. *IEEE Trans. on Computers*, C–25(9):915–921, Sept. 1976.