Access Ordering and Effective Memory Bandwidth

Steven A. Moyer

Technical Report CS-93-18
April 5, 1993

Access Ordering and Effective Memory Bandwidth

Steven A. Moyer

Computer Science Department
School of Engineering and Applied Science
University of Virginia
Charlottesville, Virginia 22903
(*sam2y@virginia.edu*)

High-performance scalar processors are characterized by multiple pipelined functional units that can be initiated simultaneously to exploit instruction level parallelism. For scientific codes, the performance of these processors depends heavily on memory bandwidth. To achieve peak processor rate, data must be supplied to the arithmetic units at the peak aggregate rate of consumption.

Access ordering, a loop optimization that reorders *non-caching* accesses to better utilize memory system resources, is a compiler technology developed in this thesis that addresses the memory bandwidth problem for scalar processors executing scientific codes. For a given computation, memory architecture, and memory device type, an access ordering algorithm determines a well-defined interleaving of vector references that maximizes effective bandwidth. Consequently, analytic models of performance can also be derived.

Access ordering is fundamentally different from, though complementary to, both caching and access scheduling techniques that attempt to overlap computation with memory latency. Simulation results demonstrate that for a given computation, access ordering can significantly increase effective memory bandwidth over that achieved by the natural reference sequence.

Table of Contents

List of Figures

List of Tables

List of Theorems

List of Symbols

Memory system parameters:

w	word size
p	page size
$T_{p/r}$	page-hit read cycle time
$T_{p/w}$	page-hit write cycle time
$T_{p/m}$	page-miss overhead
$T_{u/r}$	uniform-access read cycle time
$T_{u/w}$	uniform-access write cycle time

Stream parameters:

v	stream start address (vector accessed)
s	stride of access
d	data size
m	mode of access
σ	number of data items referenced per functional iteration (logical streams)
ε	number of words accessed per loop iteration (physical streams)

MAP notation:

a_i	access to the next element of stream t_i
a_i^k	k^{th} access from t_i in a given access sequence
S	set of all streams in a given computation
$\tilde{S}$	access sequence that embodies streams S
N	number of streams in S
V	number of different vectors referenced by streams in S
b	depth of loop unrolling

General properties of physical stream t_i:

γ_i	number of data items per word
θ_i	intermix factor

Properties of physical stream t_i for a sequentially interleaved architecture:

μ_i number of modules referenced

Z_i set of modules referenced

ξ_i module stride

ψ_i maximum number of accesses serviced at any module for a given loop iteration

Properties of physical stream t_i for a multicopy architecture:

$\hat{\mu}_i$ number of modules referenced

$\hat{\xi}_i$ module stride

Modeling functions:

$\phi(s, d)$ average number of accesses per page referenced

$\eta(s, d, c, V)$ average per iteration page miss count

$h\rho(s, d, c)$ average per iteration page miss count for intermixed write stream

$\omega(s, d, c)$ average per iteration page miss count for wrap-around adjacent read stream

$imix(s, d, c, h, V)$ effect of intermixing on page miss count of write stream

$wadj(s, d, c, V)$ effect of wrap-around adjacency on page miss count of read stream

Performance measures:

T_{avg} average time per data item accessed

BW processor-memory bandwidth

1 Introduction

Scientific computing, the application of computers to the solution of science and engineering problems, has traditionally been one of computing's most demanding fields. Until recently, special high-speed vector computers provided the only means for solving most scientific problems at acceptable computation rates. However, advances in VLSI technology have allowed manufacturers to produce scalar microprocessors with sufficient peak performance to make them viable alternatives to traditional vector processors, singly or as components of parallel machines.

High-performance scalar processors are characterized by multiple pipelined functional units that can be initiated simultaneously to exploit instruction level parallelism.[1] For scientific codes, the performance of these processors depends heavily on memory bandwidth. To achieve peak processor rate, data must be supplied to the arithmetic units at the peak aggregate rate of consumption.

Extensive tests of systems constructed from one such processor, Intel's i860, show that as a result of insufficient bandwidth, the average performance of hand optimized scientific kernels is only 1/5 peak processor rate; for compiler generated code average performance is an order of magnitude below peak performance [Lee90, Moye91]. The majority of improvement in hand-coded routines over compiler generated code results from tailoring accesses to memory system performance characteristics.

In general purpose scalar computing, the addition of cache memory is often a sufficient solution to the memory latency and bandwidth problems given the spatial and temporal locality of reference exhibited by most codes. For scientific computations, vectors are normally too large to cache. Iteration space tiling [CaKe89, Wolf89] can partition problems

1. Common superscalar and VLIW architectures incorporate concurrent pipelined functional units.

into cache-size blocks, however tiling often creates cache conflicts [LaRW91] and the technique is difficult to automate. Furthermore, only a subset of the vectors accessed will generally be reused and hence benefit from caching. Finally, caching may actually reduce the effective bandwidth achieved by a computation by fetching extraneous data for non-unit strides. Thus, as noted by Lam *et al* [LaRW91], 'while data caches have been demonstrated to be effective for general-purpose applications..., their effectiveness for numerical code has not been established'.

Access ordering is a compiler technology developed in this thesis that addresses the memory bandwidth problem for scalar processors executing scientific codes. Access ordering is a loop optimization that reorders *non-caching* accesses to better utilize memory system resources. For a given computation, memory architecture, and memory device type, an access ordering algorithm determines a well-defined interleaving of vector references that maximizes effective bandwidth. Consequently, analytic models of performance can also be derived.

Access ordering is fundamentally different from, though complementary to, both caching and access scheduling techniques that attempt to overlap computation with memory latency. Simulation results demonstrate that for a given computation, access ordering can significantly increase effective memory bandwidth over that achieved by the natural reference sequence.

In a study of the Intel Touchstone Delta distributed memory parallel computer, Stevens [Stev92] notes that for many scientific codes

> per node performance is still the number one problem in obtaining good overall applications performance. The majority of [codes surveyed] are not communications bound. Thus improving basic compiler technology is absolutely necessary.

Access ordering represents a new compiler technology complementary to existing technologies aimed at improving performance for scientific codes. Together, these compilation techniques can be applied towards meeting the demands of high-performance scalar computing.

The following sections introduce access ordering and define the scope of this work. Section 1.1 defines the general system model. An intuitive notion of access ordering is provided via a simple example in section 1.2. Sections 1.3 and 1.4 define the computation domain and the scope of memory architectures, respectively, for which access ordering algorithms are derived. Finally, a discussion of performance modeling based on access ordering is presented in section 1.5.

1.1 General System Model

Access ordering algorithms developed in this thesis presume a general system model in which a single scalar processor drives a dedicated memory system, as depicted in Figure 1. The memory system is dedicated in that only one processor is serviced, implying that memory state is dependent on a single reference sequence. This general system model is representative of uniprocessor systems and single-processor nodes of distributed memory parallel machines.

The processor is presumed to implement a non-caching load instruction, ala Intel's i860 [Inte89], allowing the sequence of requests observed by the memory system to be controlled via software. For access ordering, *all memory references are assumed to be non-caching*. Combining caching and non-caching accesses is discussed with other implementation issues in chapter 7.

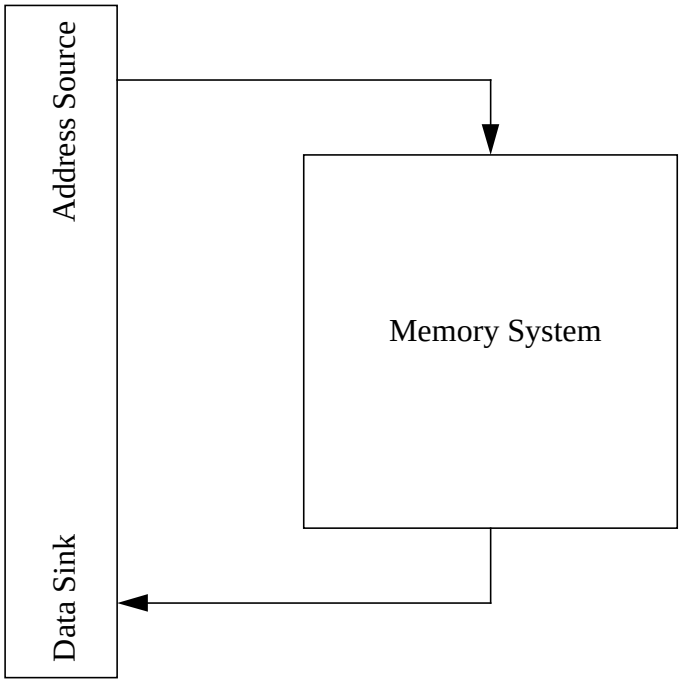

Figure 1 General System Model

1.2 Access Ordering Observation

Access ordering formalizes the notion of reordering accesses to exploit memory system resources. To illustrate this concept, a simple example is presented below.

Consider a single module memory system constructed from *page-mode* DRAMs. Page-mode DRAMs operate as if implemented with a single on-chip cache line, referred to as a *page*. An access that does not fall within the address range of the current DRAM page forces a new page to be accessed, requiring significantly more time to service than an access that ‘hits’ the cached page. Thus, the effective bandwidth is sensitive to the sequence of requests. Nearly all DRAMs currently manufactured implement a form of page-mode operation [Quin91].

Note that a DRAM page should not be confused with a virtual memory page; this is an unfortunate overloading of terms. A DRAM page is a physical feature of the memory device. Throughout this text the term ‘page’ always refers to a DRAM page.

Figure 2(a) illustrates the ‘natural’ reference sequence for a straight-forward translation of the inner-product algorithm

$$\forall i \qquad s \leftarrow s + a_i b_i$$

Access to the vectors $\bar{a}$ and $\bar{b}$ alternate, incurring a page miss with each access in the likely situation that a_i and b_i reside in a different DRAM page; memory references likely to page miss are highlighted in Figure 2. By unrolling the loop and grouping accesses to the same vector, as demonstrated in Figure 2(b), page miss cost is amortized over a number of accesses, increasing processor-memory bandwidth significantly.

```
loop:
   load a[i]
   load b[i]
   <arithmetic insts>
   <branch when done>
   jump loop
```

(a)

```
loop:
   load a[i]
   load a[i+1]
   load b[i]
   load b[i+1]
   <arithmetic insts>
   <branch when done>
   jump loop
```

(b)

Figure 2 Inner-Product Code

Figure 3 depicts average time per access versus depth of loop unrolling for the inner-product computation, as measured on the Intel IPSC/860 node architecture. For the curve labeled ‘Natural’ the loop body of Figure 2(a) is essentially replicated the appropriate number of times, as is standard practice. For the curve labeled ‘Ordered’, accesses have been arranged as per Figure 2(b); in doing so a performance gain of nearly 150% is realized at a depth of 4.

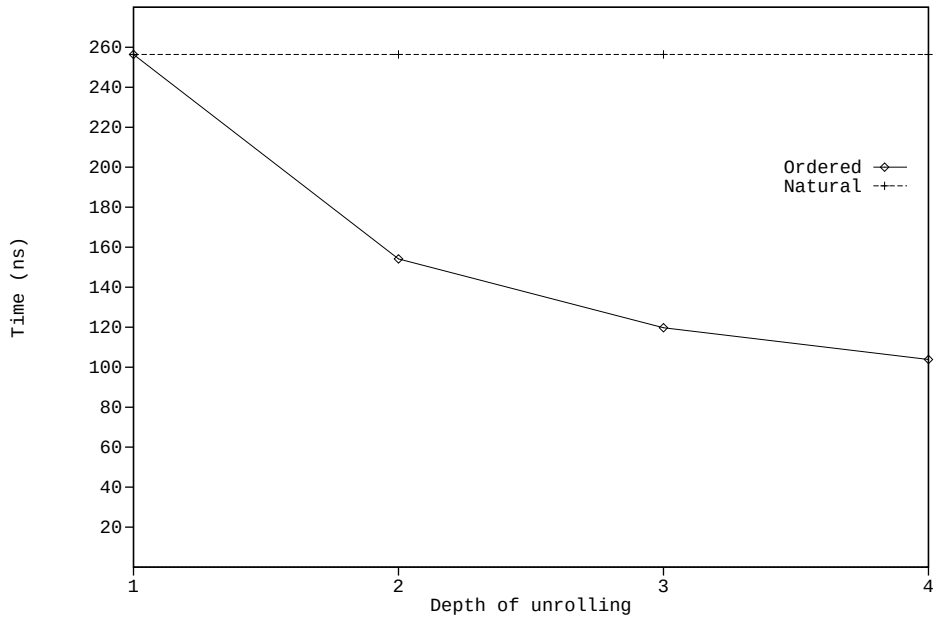

Figure 3 Inner-Product Performance

As noted above, access ordering employs loop unrolling to increase the number of accesses within a given loop that can be reordered. However, loop unrolling creates register pressure and has traditionally been limited by register resources. Techniques that utilize cache memory to mimic vector registers, thereby relieving processor register pressure and effectively increasing register set size, are discussed in chapter 7.

1.3 Computation Domain

The problem domain to which access ordering is applicable is the class of *stream-oriented* computations. A stream-oriented computation interleaves references to some number of streams, where a stream is defined as a linear sequence of accesses to a given vector of fixed sized elements, beginning at a known address, and proceeding at a constant stride. Stream access results in a predictable reference pattern than can be exploited. Processor instructions and scalar constants are assumed to be cached or held in registers, as appropriate.

For example, a scalar processor performing the well known *axpy* operation:

$$\forall i \qquad y_i \leftarrow a x_i + y_i$$

is assumed to generate three distinguishable access streams, one load stream to each of the vectors $\bar{y}$ and $\bar{x}$, and one store stream back to the vector $\bar{y}$.

Many important scientific computational kernels may be classified as stream-oriented computations, including

- BLAS1-3 routines,
- LINPACK routines,
- most of the codes in the Livermore Loops,
- numerous iterative methods for the solution of PDEs,
- diagonally-sparse matrix-vector and matrix-matrix multiply operations,
- the Simplex algorithm for solving Linear Programming problems,
- DSP algorithms such as constant geometry FFT and the linear filters FIR and IIR, and
- numerous recurrence and reduction operations.

Furthermore, many string manipulation and search algorithms may also be classified as stream-oriented computations. Thus, while the scope of reference patterns modeled is limited, the problem space is sufficiently interesting to warrant investigation.

Note that the class of computations considered for accesses ordering, i.e. stream-oriented computations, is precisely the class for which cache memory is an insufficient solution to the memory bandwidth problem. The converse is also true; cache memory provides sufficient bandwidth for computations that do not exhibit the extended patterns of access exploited by access ordering. Thus, the two techniques are complementary.

1.4 Memory System Domain

For stream-oriented computations, access ordering reorders references within an unrolled loop to exploit features of the underlying memory system. Thus, a different access ordering algorithm must be derived for each target memory architecture and device type.

1.4.1 Memory Architectures

Three memory architectures are analyzed in the chapters that follow:

- single module,
- sequentially interleaved, and
- multicopy.

Chapter 4 derives access ordering algorithms for a single module system. Optimal effective memory bandwidth is achieved for a given computation under conditions to be defined. Single module results are used as the basis for analyzing the parallel memory systems defined below.

Chapter 5 derives access ordering algorithms for a sequentially interleaved system. Sequentially interleaved storage is the ‘standard’ parallel memory storage scheme. Given a system of m memory modules, word a maps to module $a \bmod m$. Algorithms are developed assuming both known and unknown stream alignments.

Finally, chapter 6 derives access ordering algorithms for a proposed multicopy memory system, consisting of m identical copies of memory that can be accessed in parallel. In this system, read accesses are tagged for a particular module and write accesses are broadcast to all modules to maintain consistency. Cost and performance of multicopy versus sequentially interleaved systems are discussed.

1.4.2 Memory Device Types

For each of the memory architectures described above, an access ordering algorithm is derived for each of two devices types: uniform-access and page-mode.

Uniform-access components are insensitive to the reference sequence, so that the time to service a given access is not dependent on previous requests; SRAMs are the common example of this device type. The performance of uniform-access components is parameterized by

- $T_{u/r}$, the read cycle time, and
- $T_{u/w}$, the write cycle time.

Page-mode components operate as if implemented with a single on-chip cache line, as discussed in section 1.2; static-column and fast page-mode DRAMs are the common examples of this device type. The performance of page-mode components is parameterized by

- p, the page size,
- $T_{p/r}$, the page-hit read cycle time,
- $T_{p/w}$, the page-hit write cycle time, and
- $T_{p/m}$, the additional page access overhead incurred by a page miss; thus, the page-miss read and write cycle times are $T_{p/r} + T_{p/m}$ and $T_{p/w} + T_{p/m}$, respectively.

For all memory systems analyzed, w is the word size. For systems constructed from page-mode components, page size is a multiple of word size; i.e. $w \mid p$.

Note that for all system parameters, sizes are in bytes and times are in nanoseconds.

1.5 Performance Modeling

For a given computation, memory architecture, and memory device type, access ordering results in code that generates a well-defined sequence of stream references. Consequently,

for each access ordering algorithm, an analytic model of effective memory bandwidth can be derived.

Models of memory system performance have traditionally been based on the assumption that individual modules are insensitive to the sequence of access requests. For modern page-mode DRAM components, this assumption is not correct. Furthermore, memory performance models generally assume a stochastic sequence of references. For stream-oriented computations, this is not the case.

Developing an access ordering algorithm for a given memory architecture and device type provides a unique opportunity to derive a precise analytic model of memory system performance for a large and important class of computations. Models of effective memory bandwidth are derived for the cross-product of architectures and components discussed in section 1.4. Note that to model maximum bandwidth, it is assumed that the processor is sufficiently fast such that performance is limited by the memory system.

2 Survey of Related Work

Access ordering spans a number of interrelated topics from compiler optimizations to performance modeling. To assess the contribution of this work, it is necessary to identify each of these relevant areas and cite previous research.

Section 2.1 discusses stream detection as required for access ordering. Access scheduling techniques are presented in section 2.2. Previous analytic results that capture memory system behavior are discussed in section 2.3. Section 2.4 considers alternative parallel memory storage schemes for increasing the effective bandwidth of vector accesses.

2.1 Stream Detection

Access ordering algorithms derived in this thesis presuppose the existence of compiler techniques to detect stream-oriented computations. Benitez and Davidson [BeDa91] describe a technique for explicitly detecting streaming opportunities, including those in recurrence relations. Furthermore, since stream-oriented computations reference vector operands, well known vectorization techniques are applicable, such as those described by Wolfe [Wolf89].

2.2 Access Scheduling Techniques

Access ordering is a compilation technique for maximizing effective memory bandwidth. Previous work has focused on reducing load/store interlock delay by overlapping computation with memory latency. Such techniques are referred to here collectively as *access scheduling*. Essentially, access scheduling techniques attempt to separate the execution of a load/store instruction from the execution of the instruction which consumes/produces its operand, reducing the time the processor spends delayed on memory requests.

Bernstein and Rodeh [BeRo91] present an algorithm for scheduling intra-loop instructions on superscalar architectures that accommodates load delay. Lam [Lam88] presents a technique referred to as *software pipelining* that structures code such that a given loop iteration loads the data for a later iteration, stores results from a previous iteration, and performs computation for the current iteration. Weiss and Smith [WeSm90] present a comprehensive study in which they classify and evaluate software pipelining techniques implemented in conjunction with loop unrolling. Klaiber and Levy [KlLe91] and Callahan *et al* [CaKP91] propose the use of fetch instructions to preload data into cache; compiler techniques are developed for inserting fetch instructions into the normal instruction stream.

Access ordering and access scheduling are fundamentally different. Access scheduling techniques allow load/store architectures to better tolerate memory latency. Access ordering reorders memory references to increase effective bandwidth. Note that access ordering and access scheduling are complementary. Access ordering can first be applied to a computational kernel to obtain an ordering of load/store instructions that maximizes effective bandwidth. Access scheduling can then be applied to reduce interlock delay while maintaining the specified load/store instruction order.

Access ordering as a compilation technique is an original concept; few references to similar ideas are found in the literature. Ramamoorthy and Wah [RaWa81] present an optimal algorithm for initiating queued requests on a sequentially interleaved memory to maximize module concurrency; however, this is an inherently dynamic technique that assumes independent random requests and knowledge of the modules to which requests are to be mapped. Gupta and Soffa [GuSo88] demonstrate compilation techniques for distributing scalars across parallel memory modules to allow for concurrent access by VLIW architectures; however, these techniques are not applicable to vector data.

2.3 Modeling Memory System Behavior

Deriving an ordering algorithm and corresponding performance predictor requires analytic results that capture the interaction of a reference sequence with memory architecture and components features. Section 2.3.1 discusses reference sequence modeling. Sections 2.3.2 and 2.3.3 survey analytic results pertaining to memory system behavior for scalar and vector processors, respectively.

2.3.1 Access Pattern Models

To model the behavior of a memory system, it is necessary to characterize the reference pattern by which it is driven. While a number of stochastic models of program locality have been derived [AvCK87, ShTu72, SpDe72], they are mainly applicable to the performance analysis of cache memories and virtual memory systems. For the purpose of deriving analytic results, most memory architectures are modeled under the assumption of a uniform random pattern of access, as discussed below.

This study formalizes the notion of a reference sequence consisting of interleaved vector accesses generated by a scalar processor.

2.3.2 Memory Architecture Analysis for Scalar Processors

A number of analytic models have been derived that capture the behavior of a memory system driven by a scalar processor. For single module systems, the characteristics of the memory component type completely determine performance for a given reference sequence. For parallel memory systems, concurrency must be modeled as well.

A single module of uniform-access memory components is insensitive to the sequence of requests, so that performance analysis is trivial. Given a single module constructed from page-mode devices, as described in section 1.4.2, this is not the case. Surprisingly, few analytic results have previously been derived to capture the behavior of any memory

architecture constructed from page-mode components. Peelen and Van de Goor [PeVa87] analyze the performance of page-mode DRAMs in a single module architecture by viewing page-mode as a cache with a single long cache line; a performance predictor is derived in the standard manner as a function of the miss ratio. Goodman and Chiang [GoCh84] consider an architecture in which parallel modules are accessed via a "mid-order" interleaving scheme such that sequential addresses proceed across a page, changing modules at page boundaries. Using trace driven simulation, Goodman and Chiang evaluate the performance of such a system for a number of common UNIX utilities; however, no analytic results are derived. To date, no other studies involving page-mode memory components have been located.

Concurrency in sequentially interleaved memory architectures has been the subject of numerous studies; analytic results are general based on stochastic access sequences. Hellerman [Hell66] assumes a uniform random distribution of accesses and no buffering of conflicting requests in deriving the well known formula that for m memory modules, an average of approximately $m^{0.56}$ are operating concurrently at any given time. Burnett and Coffman [BuCo70] derive an analytic model in which data and instruction requests are separated and serviced alternately; instruction accesses are assumed sequential with a fixed probability of branching, data requests access the next sequential module with probability α and any other module with probability $\beta = (1-\alpha)/N$. Later work [CoBS71] extends this model to include buffering of access conflicts. Ravi [Ravi72] develops an analytic model for the performance of a multiprocessor system accessing a sequentially interleaved memory, assuming a uniform random distribution of accesses. Chang, Kuck, and Lawrie [ChKL77] summarize the work of previous authors and introduce the notion of data dependence between accesses of a multiprocessor system; analytic models are derived for each dependence class, again assuming a uniform random distribution of requests.

Because previous performance studies of memory architectures for scalar processors have been based on stochastic access sequences, their results have limited relevance to the stream-oriented computations considered in this thesis. Furthermore, analysis of memory architectures constructed from page-mode components has all but been ignored.

2.3.3 Memory Architecture Analysis for Vector Processors

The scope of reference patterns considered for access ordering is limited to stream-oriented computations, i.e. algorithms that operated on vectors or vector-like data, as discussed in section 1.3. Thus analytic results pertaining to the performance of vector computer memory systems are potentially relevant.

Modern vector computer systems incorporate multiple independent ports and a large number of sequentially interleaved memory modules. While calculating memory bandwidth is trivial for access to a single vector, it becomes intractable for access to multiple vectors. The difficulty lies in characterizing the conflicts which occur as the access streams contend for memory modules.

Oed and Lange [OeLa85] derive analytic results, based on number of modules and strides of access, for determining when two access streams can proceed conflict-free. In the case of conflict-free access, calculation of bandwidth is trivial and represents an upper bound on performance; for most other cases bandwidth calculation is intractable. Cheung and Smith [ChSm86] perform a simulation study that characterizes reference stream interaction for up to three independent memory ports. Though no analytic results are presented, they too classify steady state conflict situations and provide simulation performance data. Bailey [Bail87] takes a different approach in which performance of a vector computer memory system is modeled analytically under the assumption of uniform random accesses; while these results are not representative of the performance obtained in access-

ing vectors, relationships between the number of processors, number of modules, and module access time carry over to realistic vector accesses, as demonstrated via simulation.

Since the ports of a vector computer memory system operate concurrently and independently, they behave in a fundamentally different manner than a scalar processor that generates a single reference sequence. In particular, one memory port can block on a module conflict while another continues to access a vector, a situation that does not occur with a scalar processor. Therefore, most analytic results derived for vector processors are not applicable. As few analytic results exist that capture vector computer memory system behavior, little is lost in this realization.

2.4 Storage Schemes for Parallel Memories

Access ordering attempts to maximize effective memory bandwidth for a stream-oriented computation by structuring references to exploit memory system characteristics. For parallel memory systems, the storage scheme limits the degree of concurrency achievable by a given computation. This research considers only two such storage schemes: sequentially interleaved and multicopy. However, a number of other schemes have been proposed and are discussed below.

Budnik and Kuck [BuKu71] observed that for a sequentially interleaved storage scheme, only vectors with strides relatively prime to the number of modules *m* can be accessed without conflict, i.e. at maximum system bandwidth. This result led Lawrie and Vora [LaVo82] to propose a memory system based on a prime number of modules. Such a system was developed for the Burroughs Scientific Processor [KuSt82]; however, prime memory systems have proved impractical due to the computational complexity of the storage scheme.

Budnik and Kuck [BuKu71] propose the use of *skewed storage* in which each successive set of *m* storage locations is assigned to *m* memory modules with a skew relative to the

previous set. Harper and Jump [HaJu87] present a comprehensive study of a skewed storage scheme that is shown to reduce conflict over a wide range of strides. Harper and Jump further demonstrate that with skewed storage, vector accesses can reference a sequence of modules with a periodicity that exceeds *m*, allowing queues placed at each module to buffer conflicting requests and increase bandwidth.

As an alternative to implementing a single storage scheme, Harper [HaLi89, Harp89] proposes a *dynamic storage scheme* in which each vector is stored so as to provide optimal bandwidth for a given stride of access; later work demonstrates that dynamic storage schemes can be devised that allow optimal access to a vector for a set of strides [Harp91].

Rau [Rau91] analyzes a scheme that assigns storage locations to modules in a pseudo-random fashion, rendering memory performance nearly stride insensitive; such a memory system has been incorporated into Cydrome's Cydra 5 Departmental Supercomputer [RaSY89]. As with skewed and dynamic schemes, pseudo-random storage schemes benefit from memory module queues.

The above studies focus on increasing parallelism for accesses to a single vector beyond that achieved by sequentially interleaved storage. However, for a given storage scheme, it is not clear in all cases how references are best made to multiple vectors. Furthermore, for parallel memory systems constructed from page-mode components, it is not sufficient to simply maintain a high degree of concurrency; for maximum performance, reference patterns should also minimize page thrashing. However, it is important to acknowledge the existence of other, more effective storage schemes that may also benefit from access ordering techniques.

3 Model Access Pattern

For deriving access ordering algorithms and performance models, it is useful to define a notation for expressing sequences of requests generated by stream-oriented computations. Section 3.1 defines the Model Access Pattern notation used to describe streams and denote specific reference sequences. Sections 3.2 and 3.3 present a set of general definitions and assumptions applicable to all computations and discuss optimizing accesses with respect to wide words. A restriction placed on stream interaction and the resulting dependencies are discussed in sections 3.4 and 3.5, respectively.

3.1 Basic MAP Notation

Two characteristics define the Model Access Pattern (MAP) for a stream-oriented computation: a set of *access streams* to individual vectors, and an interleaving of stream references into a merged *access sequence*.

An *access stream* is defined by the tuple $t_i = (v_i, s_i, d_i, m_i) : \sigma_i$ where

v_i = vector to be accessed = stream starting address

s_i = stride of access

d_i = data item size

m_i = access mode, read(*r*) or write(*w*)

and σ_i specifies the number of data items referenced from the stream per *functional iteration* of the computation.

A *functional iteration* is defined as a single repetition of an iterative computation. A *loop iteration* is defined as a single repetition of the code implementing a computation. Note that a single loop iteration implements a fixed number of functional iterations, with that number defined to be the *depth of unrolling*.

An *access sequence* describes the interleaving of stream accesses within a loop and is defined recursively as follows:

Let a_i denote an access to the 'next' element of the stream t_i, then

1. $\langle a_i \rangle$ is an access sequence.
2. $\langle A_1, \ldots, A_n \rangle$ is an access sequence where $A_1, \ldots, A_n$ are access sequences; $A_1, \ldots, A_n$ are performed left to right with all accesses in A_j initiated prior to the initiation of accesses in A_{j+1}.
3. $\langle A : c \rangle$ is an access sequence where A is an access sequence and c is a positive integer; A is repeated c consecutive times.

For visual clarity, $\langle \langle a_i \rangle : c \rangle \equiv \langle a_i : c \rangle$ and extraneous brackets are omitted when the meaning is unambiguous. If the access mode is known, then an access to the 'next' element of stream t_i is denoted as r_i or w_i for $m_i = r$ or $m_i = w$, respectively. In discussing a particular access sequence, a_i^k refers to the k^{th} access from stream t_i.

To illustrate, the MAP notation is applied to the axpy operation

$$\forall i \qquad y_i \leftarrow a x_i + y_i$$

The set of logical access streams dictated by the computation is $S = \{t_x, t_{y_r}, t_{y_w}\}$, where $t_x = (x, s_x, d_x, r) : 1$, $t_{y_r} = (y, s_y, d_y, r) : 1$, and $t_{y_w} = (y, s_y, d_y, w) : 1$. The 'natural' access sequence implementing the axpy computation is: $\langle r_x, r_{y_r}, w_{y_w} \rangle$, specifying one read from streams t_x and t_{y_r}, followed by one write from stream t_{y_w}, per iteration of the loop. Note that the access sequence $\langle r_x, r_{y_r}, w_{y_w} \rangle$ implements one functional iteration per loop iteration.

3.2 Definitions and Assumptions

The following definitions complement the MAP notation:

- $S = \{ t_i \mid t_i$ defines an access stream for a given computation$\}$, i.e. S is the set of all access streams for a given computation,
- $N = |S|$, i.e. the total number of access streams in S is N, and
- V = number of unique v_i for all $t_i \in S$, i.e. the number of vectors accessed by streams S is V, where $V \leq N$.

For a set of streams S, it is assumed that for all $t_i \in S$

- $d_i \mid w$, i.e. word size is a multiple of the data size,
- access stream t_i begins at a memory address divisible by d_i, i.e. data is aligned, and
- stride of access s_i is positive.

The first two assumptions represent typical processor constraints and simplify subsequent analysis. The later assumption is customarily satisfied in the class of computations considered; furthermore, the *stream interaction restriction*, to be defined, allows this assumption without loss of generality.

3.3 Wide Word Optimization

For completeness, it is desirable to accommodate wide word access in ordering algorithms and performance models; e.g. 32-bit values referenced from 64-bit words. To optimally utilize wide words, and simplify modeling, a mapping is defined from a set of *logical streams* referencing individual data items, as above, to a corresponding set of *physical streams* referencing memory locations.

Physical streams are defined so as to guarantee that each stream references a given word at most once, resulting in optimal wide-word access. It is assumed that a given logical stream

$t_i^{(L)} = (v_i^{(L)}, s_i^{(L)}, d_i^{(L)}, m_i^{(L)}) : \sigma_i^{(L)}$ can be mapped to a physical stream $t_i = (v_i, s_i, d_i, m_i) : \varepsilon_i$ such that

- t_i begins at an address divisible by the word size *w*, i.e. is word aligned, and
- t_i retrieves exactly the same number of data items for each word accessed.

The second property can be true only if logical stream stride $s_i^{(L)}$ and data item size $d_i^{(L)}$ are restricted such that $w \le s_i^{(L)} d_i^{(L)}$ or $s_i^{(L)} d_i^{(L)} \mid w$. Then the number of data items retrieved per word accessed by physical stream t_i is an integer defined by

$$\gamma_i = \begin{cases} 1 & \text{when} \quad w \le s_i^{(L)} d_i^{(L)} \\ \dfrac{w}{s_i^{(L)} d_i^{(L)}} & \text{when} \quad w > s_i^{(L)} d_i^{(L)} \end{cases}$$

Let κ_i be the minimum number of functional iterations of the computation required to reference all data items contained in a set of accessed words from physical stream t_i. Then κ_i is the least common multiple of the number of data items referenced per functional iteration and the number of data items per word divided by the number of data items per functional iteration, i.e.

$$\kappa_i = \frac{lcm(\sigma_i^{(L)}, \gamma_i)}{\sigma_i^{(L)}}$$

For example, given stream $t_i^{(L)}$ with stride $s_i^{(L)} = 2$, data item size $d_i^{(L)} = 1$, and $\sigma_i^{(L)} = 3$ data items required per functional iteration, and given a word size $w = 4$, then $\kappa_i = 2$ as depicted in Figure 4.

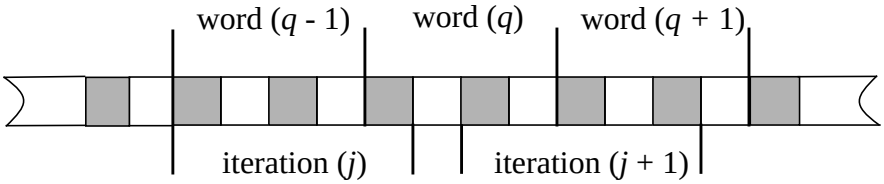

Figure 4 Functional Iteration Diagram

For a set of logical streams $S^{(L)}$, let $c_S = lcm(\kappa_i)$ for all $t_i^{(L)} \in S^{(L)}$. Then c_S is the minimum number of functional iterations required to reference all data items contained in the set of words accessed by the corresponding physical streams.

Given logical streams $S^{(L)}$, a corresponding set of physical streams S can now be defined by mapping each $t_i^{(L)} \in S^{(L)}$ to a distinct $t_i \in S$ such that

$$t_i = \begin{cases} (v_i^{(L)}, s_i^{(L)}, d_i^{(L)}, m_i^{(L)}) : \varepsilon_i & \text{when} \quad \gamma_i = 1 \\ (v_i^{(L)}, 1, w, m_i^{(L)}) : \varepsilon_i & \text{when} \quad \gamma_i > 1 \end{cases}$$

where ε_i is the product of the number of functional iterations c_S and the number of data items referenced per functional iteration divided by the number of data items per word; i.e.

$$\varepsilon_i = \frac{c_S \sigma_i^{(L)}}{\gamma_i}$$

Note that $\gamma_i \mid c_S \sigma_i^{(L)}$ since

$$c_S \sigma_i^{(L)} = h \kappa_i \sigma_i^{(L)} = h \left(\frac{lcm(\sigma_i^{(L)}, \gamma_i)}{\sigma_i^{(L)}} \right) \sigma_i^{(L)} = h\,(lcm(\sigma_i^{(L)}, \gamma_i))$$

where $h = c_S / \kappa_i \in Z^+$.

Whereas $\sigma_i^{(L)}$ specifies the number of data items referenced from logical stream $t_i^{(L)}$ per functional iteration of the computation, ε_i specifies the number of words accessed from the corresponding physical stream t_i per loop iteration, with each loop iteration representing c_S functional iterations. Note that this definition of ε_i for physical stream t_i is not inconsistent with the generic stream definition presented in section 3.1, since a loop iteration can be considered a single 'functional iteration' of the total reference sequence.

A physical stream t_i is said to *embody* a logical stream $t_i^{(L)}$ if t_i accesses all data items referenced by $t_i^{(L)}$.

Theorem 3.1: Given logical streams $S^{(L)}$ and corresponding physical streams *S*, as defined above, $t_i \in S$ embodies $t_i^{(L)} \in S^{(L)}$.

Proof: If $\gamma_i = 1$ it is easily seen that each set of $\varepsilon_i = c_S\sigma_i^{(L)}$ physical accesses from stream t_i per loop iteration references *exactly* the set of $c_S\sigma_i^{(L)}$ data items as logical stream $t_i^{(L)}$ for the corresponding c_S functional iterations.

If $\gamma_i > 1$ then by definition each physical access from stream t_i references exactly γ_i logical data items. Thus each set of ε_i physical accesses from t_i per loop iteration references *exactly* the set of $\varepsilon_i\gamma_i = c_S\sigma_i^{(L)}$ data items as logical stream $t_i^{(L)}$ for the corresponding c_S functional iterations. □

Corollary 3.2: Given logical streams $S^{(L)}$ and corresponding physical streams *S*, $t_i \in S$ is optimal with respect to wide word access.

Proof: Physical stream $t_i \in S$ references a given word at most once, by definition. □

Thus for a set of logical streams $S^{(L)}$, a set of physical streams *S* can be defined such that

- $t_i \in S$ embodies $t_i^{(L)} \in S^{(L)}$ and
- $t_i \in S$ is optimal with respect to wide word access

for any loop depth $b = kc_S$, $k \in Z^+$, implementing kc_S functional iterations per loop iteration. Note that in the most common case of one data item per word per stream, *b* can be any positive integer.

The remainder of this text assumes *S* always to be a set of physical streams that embody a set of logical streams $S^{(L)}$ via the mapping defined above. In deriving *S*, the loop depth *b* is fixed; hence ε_i is fixed for all $t_i \in S$.

Given a loop computation that references a set of data items embodied by *S*, chapters 4 through 6 examine the effect of the specific order of loop accesses on page miss count and concurrency, and hence bandwidth, for a set of memory architectures and device types. Based on these results, algorithms are derived that map the set of accesses defined by streams *S* to a specific sequence of memory references $\tilde{S}$. Note that for a stream $t_i \in S$, accesses from t_i may be placed in any order that preserves dependencies; e.g.

$$\tilde{S} = \langle \ldots, a_i{:}q_1^i, \ldots, a_i{:}q_2^i, \ldots, a_i{:}q_n^i, \ldots \rangle$$

The access sequence $\tilde{S}$ *embodies S* if for all $t_i \in S$

$$\sum_{k=1}^{n} q_k^i = \varepsilon_i$$

That is, the sequence $\tilde{S}$ embodies *S* if $\tilde{S}$ contains exactly the number of accesses per stream required to implement *b* functional iterations of the computation, ordered to preserve dependence, where *b* is the loop depth.

3.4 Stream Interaction Restriction

Recall that for a memory module constructed from page-mode components, the time to complete a given access depends on whether or not the page referenced is the same as that of the immediately preceding access. If two consecutive accesses are from different streams, the impact of the first on the one that follows is difficult to capture analytically as they may or may not reference the same page. To simplify analysis, the following restriction is placed on the streams of a given computation:

> *Stream Interaction Restriction*: For any two streams $t_i, t_j \in S$, $v_i \neq v_j$ implies that the streams have non-intersecting address spaces; in particular, streams reference no pages in common. When $v_i = v_j$ stream parameters are identical except in mode, where by definition $m_i \neq m_j$.

The stream interaction restriction results in stream accesses that interact with memory architecture features in a well defined manner. To illustrate, when two streams have different start addresses, i.e. $v_i \neq v_j$, the stream interaction restriction states that the streams reference no pages in common. Thus it is known that an access from stream t_i preceded by an access from stream t_j will cause a page miss. When two streams have the same start address, i.e. $v_i = v_j$, the stream interaction restriction states that the stream parameters are identical except in access mode, accommodating read-modify-write operations. Thus, within a given loop iteration, the k^{th} accesses from each of t_i and t_j reference the same word and hence the same page.

Strict adherence to the stream interaction restriction limits the applicability of access ordering algorithms to a subset of the class of vectorizable computations. However, the remaining problem domain is still large and encompasses all computations previously listed in section 1.3. Furthermore, under the stream interaction restriction, optimality results are obtained for single module access. Relaxation of this restriction to encompass a superset of the vectorizable loops is discussed in chapter 7.

Though the stream interaction restriction is specifically aimed at simplifying analysis for systems constructed from page-mode components, for consistency it is applied for all memory architectures considered.

3.5 MAP Dependence Relations

Access ordering alters the sequence of instructions that access memory. In performing this reordering, dependence relations must be maintained. As discussed below, the stream interaction restriction limits the types of dependencies that can exist between accesses from different streams. Rules are derived for maintaining dependencies during access ordering.

Briefly, *output* and *input dependence* results when two write or two read accesses, respectively, reference the same data item. *Antidependence* occurs when a read from a data item must precede a write to that datum. Finally, *data dependence* occurs when a write to a data item must precede a read from the same. A dependence relation between two accesses from the same instance of a loop iteration is said to be *loop-independent*, while a dependence between accesses from different instances is said to be *loop-carried*. A detailed treatment of dependence analysis can be found in [Wolf89].

3.5.1 Output and Input Dependence

Output and input dependence can not exist as a result of the stream interaction restriction; two streams of the same mode have a non-intersecting address space. Therefore, dependence relations of this type need not be considered.

3.5.2 Antidependence

The stream interaction restriction states that two streams referencing the same vector do so with stream parameters that differ only in access mode. Thus, antidependence is limited to loop-independent antidependence between corresponding components of a read stream t_i

and write stream t_j implementing a read-modify-write. So, if $v_i = v_j$, then w_j^k is antidependent on r_i^k; notationally $r_i^k \ \bar{\delta} \ w_j^k$.

Simply specifying t_i and t_j such that $v_i = v_j$ is assumed to imply antidependence; the only alternative, a loop-independent data dependence, is redundant and the read stream unnecessary. Compilation is assumed to remove extraneous access streams.

A computation with a loop-carried antidependence that does not form a self-dependence cycle can often be transformed to an equivalent computation with a loop-independent antidependence. For example, restructuring the loop of Figure 5(a) to the loop of Figure 5(b) replaces the loop-carried antidependence with a loop-independent antidependence that conforms to the stream interaction restriction. Loop-carried self antidependence can be eliminated via renaming of the assignment variable.

```
for i = 1 to n
{
   y[i]  = <statement>;
   v[i]  = fn(y[i+1]);
}
```

(a)

```
y[1] = <statement>;
for i = 2 to n
{
   v[i-1]= fn(y[i]);
   y[i]  = <statement>;
}
v[n] = fn(y[n+1]);
```

(b)

Figure 5 Loop-Carried to Loop-Independent Antidependence Transformation

Though the loop of Figure 5(b) does not implement a read-modify-write of the vector $\bar{y}$ in the strict sense that each computed value is a function of the old value, the reference sequence is equivalent.

3.5.3 Data Dependence

Data dependence does not exist between access streams in the usual sense that a memory location is written and later read during the execution of a loop. Loop-independent data dependence implies an extraneous read stream, as discussed above. Loop-carried data dependence can not exist as a result of the stream interaction restriction.

Note that loop-carried data dependence can often be removed, resulting in a computation that conforms to the stream interaction restriction. For example, restructuring the loop of Figure 6(a) to the loop of Figure 6(b) eliminates the read stream and hence the data dependence. Similarly, self data dependence cycles representing recurrence operations can also be removed, as described in [BeDa91].

```
for i = 1 to n                      v[1] = fn(y[0]);
{                                   for i = 1 to (n-1)
   y[i]  = <statement>;                v[i+1] = fn(y[i] = <statement>);
   v[i]  = fn(y[i-1]);              y[n] = <statement>;
}
      (a)                                 (b)
```

Figure 6 Loop-Carried Data Dependence Elimination

Though data dependence does not exist in the usual context, it is present in the data flow sense; that is, as right-hand-side values required in performing a computation. A write operation represents the assignment of a computation result and as such usually requires that some set of read operations precede it. In this sense, a write operation w_j^k is data dependent on a read operation r_i^q if r_i^q defines a value used in the computation of the result assigned by w_j^k; notationally, $r_i^q \ \delta \ w_j^k$.

3.5.4 Dependence Rules

Summarizing the above, dependence between accesses belonging to different streams is limited to two types under the stream interaction restriction: loop-independent antidependence between a read and write stream that access the same vector, and data dependence in the data flow sense. This observation leads to the following two rules necessary for maintaining data dependence in access ordering algorithms.

For read stream t_i and write stream t_j, an access sequence maintains all dependencies if

1. r_i^k precedes w_j^k when $r_i^k \ \bar{\delta} \ w_j^k$, i.e. a read precedes its corresponding write in a read-modify-write operation, and
2. r_i^q precedes w_j^k when $r_i^q \ \delta \ w_j^k$, i.e. a read operation that defines a value used in the computation of a result precedes the write of that result.

Dependence information is derived from context. As discussed in section 2.1, it is assumed that stream information has been provided for the access ordering algorithm; it is assumed that dependence information is provided as well.

3.5.5 Other Dependencies

The above discussion completely characterizes the dependence that can exist between accesses belonging to different streams under the stream interaction restriction. However, two other types of dependence may exist: loop-carried input dependence within a single read stream, and control dependence.

Loop-carried input dependence can result from the transformation of a more complex sequence of read accesses to a single read stream. Consider the finite difference approximation to the first derivative

$$\forall i \qquad dv_i = \frac{(v_{i+1} - v_{i-1})}{2h}$$

Analysis techniques [BeDa91, CaCK90] can transform the ‘natural’ pattern of access to vector $\bar{v}$ to a simple stream requiring one access per iteration; two values of $\bar{v}$ are pre-loaded prior to entering the loop, and each successive value accessed is carried in a register for two iterations. The loop-carried input dependence created in the transformation has no affect on the ordering of memory access instructions.

Control dependence results from branch statements within a loop. When control dependence is present, access ordering can still be applied by considering each path through the loop body independently. Ordering and code generation is performed for each path, with the code segment to be executed on each iteration determined dynamically. For the remainder of this discussion, loops are assumed free of control dependence.

4 Single Module Architecture

This chapter derives access ordering algorithms and performance predictors for a single module memory system as depicted in Figure 7. Systems constructed from both uniform-access and page-mode components are considered. Optimal effective memory bandwidth is achieved in both cases.

Section 4.1 develops techniques for minimizing page overhead, where applicable. Sections 4.2 and 4.3 derive ordering algorithms and performance models for a single module of uniform-access and page-mode components, respectively. The effectiveness of access ordering and accuracy of performance models are demonstrated via simulation in 4.4. Section 4.5 summarizes results.

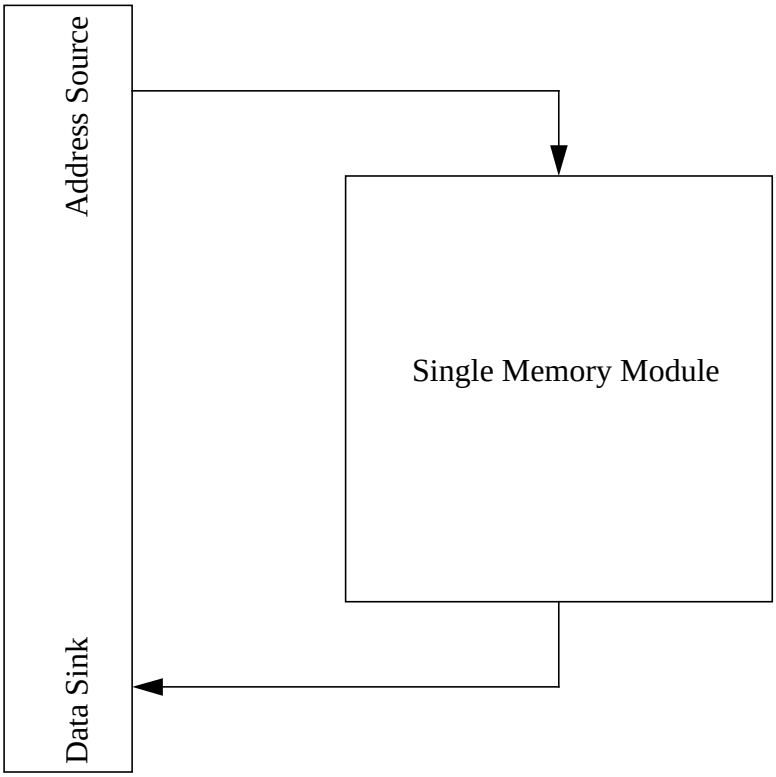

Figure 7 Single Module Architecture

4.1 Minimizing Page Overhead

Consider a single module of page-mode components. For access ordering to generate a reference sequence that achieves optimal effective memory bandwidth for a given computation, page overhead resulting from accesses that page miss must be minimized. Page overhead is measured in terms of total page miss count. Given a stream not involved in a read-modify-write, minimizing page overhead is trivial. For streams implementing this operation, page overhead is minimized via *intermixing* and *wrap-around adjacency.*

Given stream $t_i \in S$ such that t_i does not participate in a read-modify-write, i.e. $v_i \neq v_j$ for all $t_j \in S$, minimum page miss count for t_i is achieved by performing a sequence of accesses a_i without an intervening access from a second stream a_j. This follows from the observation that a_i^{k+1} only results in a page miss if it does not reference the same page as a_i^k; an intervening access a_j is guaranteed to generate a page miss by the stream interaction restriction.

The average page miss count for accesses grouped by stream is derived as follows. For stream t_i with stride s_i and data size d_i, the average number of accesses per page referenced is

$$\phi(s_i, d_i) = \begin{cases} 1 & \text{when} \quad p \leq s_i d_i \\ \dfrac{p}{s_i d_i} & \text{when} \quad p > s_i d_i \end{cases}$$

Then arranging accesses from t_i as $\langle \ldots, a_i{:}c, \ldots \rangle$, the average per iteration page miss count for the set of c memory references is

$$\eta(s_i, d_i, c, V) = \begin{cases} \dfrac{c}{\phi(s_i, d_i)} & \text{when} \quad V = 1 \\ 1 + \dfrac{(c-1)}{\phi(s_i, d_i)} & \text{when} \quad V \geq 2 \end{cases}$$

That is, when the number of vectors referenced is one, i.e. $V = 1$, the average page miss count for c consecutive accesses to t_i is simply the number of accesses divided by the number of accesses per page. For $V \geq 2$, a_i^1 is guaranteed to page miss, so that the average page miss count is one plus the remaining number of accesses, $c - 1$, divided by the number of accesses per page.

Lemma 4.1: In the sequence $\langle \ldots, a_i{:}\, c, \ldots \rangle$, the average page miss count per access, $\eta(s_i, d_i, c, V)/c$, is either constant or inversely proportional to c.

Theorem 4.2: Given streams S and $t_i \in S$ such that t_i does not participate in a read-modify-write, i.e. $v_i \neq v_j$ for all $t_j \in S$, minimum page miss count for stream t_i at the specified depth of unrolling is achieved by the access sequence $\tilde{S} = \langle \ldots, a_i{:}\, \varepsilon_i, \ldots \rangle$.

Proof: Recall from section 3.3 that in mapping logical streams $S^{(L)}$ to physical streams S, the depth of loop unrolling b, and hence the number of accesses ε_i from stream t_i per loop iteration, is fixed. Thus it follows immediately from Lemma 4.1 that the sequence $\tilde{S} = \langle \ldots, a_i{:}\, \varepsilon_i, \ldots \rangle$ must result in minimum page miss count for stream t_i at the loop depth b. □

Corollary 4.3: Given physical streams S and S' that result from mapping logical streams $S^{(L)}$, $\eta(s_i, d_i, \varepsilon_i, V)/\varepsilon_i \leq \eta(s_i, d_i, \varepsilon'_i, V)/\varepsilon'_i$ if $\varepsilon_i > \varepsilon'_i$ $(b > b')$. That is, the average page miss count per access for sequence $\tilde{S} = \langle \ldots, a_i{:}\, \varepsilon_i, \ldots \rangle$ must be less than or equal to that of sequence $\tilde{S}' = \langle \ldots, a_i{:}\, \varepsilon'_i, \ldots \rangle$ if $\varepsilon_i > \varepsilon'_i$ $(b > b')$.

Thus, minimum page miss count is achieved in grouping accesses by stream. Furthermore, the average page miss count per access is either constant or inversely proportional to the depth of loop unrolling.

4.1.1 Intermixing

Given read stream t_i and write stream t_j that implement a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$, it is often possible to reduce the page miss count of the write stream below that achieved by the access sequence $\langle \ldots, r_i{:}\varepsilon_i, \ldots, w_j{:}\varepsilon_j, \ldots \rangle$.

Consider the *general intermix sequence*

$$\langle \ldots, \langle r_i{:}c, w_j{:}c \rangle{:}h, \ldots \rangle$$

that generates the string of references

$$\ldots, r_i^1, r_i^2, \ldots, r_i^c, w_j^1, w_j^2, \ldots, w_j^c, r_i^{c+1}, \ldots$$

Lemma 4.4: The general intermix sequence $\langle \ldots, \langle r_i{:}c, w_j{:}c \rangle{:}h, \ldots \rangle$ is an optimal interleaving of accesses, as demonstrated in Appendix A.1.

Since r_i^c and w_j^c refer to the same location, r_i^{c+1} will only page miss when referencing a page different from that referenced by r_i^c. Thus, the page miss count for the read stream is unchanged. However, the sequence of accesses $w_j^{(k-1)c+1}$ through w_j^{kc}, $1 \le k \le h$, suffers a page miss only when $r_i^{(k-1)c+1}$ and r_i^{kc} reference a different page.

For write stream t_j, the average page miss count in performing each set of c write accesses in the intermix sequence $\langle \ldots, \langle r_i{:}c, w_j{:}c \rangle{:}h, \ldots \rangle$ is derived in Appendix A.2 as

$$\rho(s_j, d_j, c) = \begin{cases} \dfrac{2(c-1)s_j d_j}{p} & \text{when} \quad (c-1)s_j d_j + d_j \le p \\ 1 + \dfrac{(c-1)}{\phi(s_j, d_j)} & \text{when} \quad (c-1)s_j d_j + d_j > p \end{cases}$$

Thus, the average page miss count in performing all ch write operations for a given iteration is $h\rho(s_j, d_j, c)$.

Based on the preceding analysis, for a computation that references two or more vectors the intermix sequence $\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$ results in a lower page miss count for write accesses than the sequence $\langle \ldots, r_i : ch, \ldots, w_j : ch, \ldots \rangle$ if $h\rho(s_j, d_j, c) < \eta(s_j, d_j, ch, V)$. Similarly, for a computation that references exactly one vector the intermix sequence $\langle \langle r_i : c, w_j : c \rangle : h \rangle$ results in a lower page miss count for write operations than the sequence $\langle r_i : ch, w_j : ch \rangle$ if $h\rho(s_j, d_j, c) < \rho(s_j, d_j, ch)$. Then for write stream t_j, the effect of intermixing on average per iteration page miss count is computed as

$$imix(s_j, d_j, c, h, V) = \begin{cases} \rho(s_j, d_j, ch) - h\rho(s_j, d_j, c) & \text{when} \quad V = 1 \\ \eta(s_j, d_j, ch, V) - h\rho(s_j, d_j, c) & \text{when} \quad V \geq 2 \end{cases}$$

It can be shown algebraically that if $c = 1$ or $((c-2)h + 1)\, s_j d_j < p$ then $imix(s_j, d_j, c, h, V) > 0$, i.e. intermixing reduces write access page miss count.

Lemma 4.5: In the sequence $\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$, if $imix(s_j, d_j, c, h, V) > 0$ the average page miss count in performing each set of *c* write accesses, $\rho(s_j, d_j, c)$, is directly proportional to *c*. Thus, choosing *c* as small as possible minimizes page miss count for the write operations.

4.1.1.1 Intermix Factor

For the general intermix sequence, the values of the *intermix parameters c* and *h* that minimize page miss count for the write stream are a function of both the stream parameters and data dependence information. Intuitively, the intermix parameter *c* is chosen to be the minimum value that preserves data dependence while optimally utilizing wide word access. If write stream t_j is not data dependent on read stream t_i, implying the computation is not a strict read-modify-write, then $c = 1$. Otherwise, *c* is the minimum number of accesses required to reference all data items for a number of functional iterations such that

all data items in the words accessed are consumed; this minimal value of *c* is referred to as the *intermix factor.*

For write stream t_j the intermix factor is computed as

$$\theta_j = \begin{cases} 1 & \text{when } t_j \text{ is not data dependent on } t_i \\ \dfrac{lcm(\sigma_j^{(L)}, \gamma_j)}{\gamma_j} & \text{otherwise} \end{cases}$$

Note that the value $\sigma_j^{(L)}$ from the corresponding logical stream $t_j^{(L)}$ is required in deriving θ_j, since $\sigma_j^{(L)}$ specifies the number of data items that must be accessed from t_j per functional iteration of the computation. From the derivation of ε_j in section 3.3, it is easily seen that the number of accesses from stream t_j per loop iteration is a multiple of the intermix factor θ_j; i.e. $\theta_j \mid \varepsilon_j$.

Theorem 4.6: Given streams *S* with read stream t_i and write stream t_j that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$, if $imix(s_j, d_j, c, h, V) > 0$ for intermix parameters $c = \theta_j$ and $h = \varepsilon_j / \theta_j$ then page miss count for write stream t_j is minimized by the intermix sequence $\tilde{S} = \langle \ldots, \langle r_i : \theta_j, w_j : \theta_j \rangle : (\varepsilon_j / \theta_j), \ldots \rangle$. Page miss count for read stream t_i is unaffected by intermixing and equivalent to that of the sequence $\langle \ldots, r_i : \varepsilon_i, \ldots \rangle$.

Proof: Follows immediately from Lemma 4.5. □

Corollary 4.7: Given streams *S* with read stream t_i and write stream t_j that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$, the average page miss count per access from write stream t_j in the sequence $\tilde{S} = \langle \ldots, \langle r_i : \theta_j, w_j : \theta_j \rangle : (\varepsilon_j / \theta_j), \ldots \rangle$ is independent of the depth of loop unrolling *b* chosen in mapping logical streams $S^{(L)}$ to physical streams *S*. However, the average page miss count per access from read stream t_i is either constant or inversely proportional to *b*, as per Corollary 4.3.

Thus, page miss count for write streams can be minimized by intermixing. For the general intermix sequence, the average page miss count per access for write operations is independent of the depth of loop unrolling; for read operations the average page miss count per access is either constant or inversely proportional to the loop depth *b*.

4.1.2 Wrap-around Adjacency

Assume a read stream t_i and write stream t_j that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$. In the preceding section, intermixing is employed to reduce page miss count for the write stream. Alternatively, *wrap-around adjacency* can often reduce the page miss count of the read stream.

Streams t_i and t_j are *wrap-around adjacent* if accesses to each occur at the beginning and end of an access sequence, respectively; i.e.

$$\langle r_i{:}\,\varepsilon_i, \ldots, w_j{:}\,\varepsilon_j \rangle$$

Note that in the special case where t_i and t_j are the only streams in a computation, the intermix sequence $\langle \langle r_i{:}\,\theta_j, w_j{:}\,\theta_j \rangle{:}\ (\varepsilon_j / \theta_j) \rangle$ also results in wrap-around adjacency.

Since $r_i^{\varepsilon_i}$ and $w_j^{\varepsilon_j}$ reference the same location, then for a given loop iteration r_i^1 will only page miss when referencing a page different from that referenced by $r_i^{\varepsilon_i}$ on the previous iteration. Thus the read stream proceeds as if no other vector is accessed, so that page miss count is computed by $\eta\,(s_i, d_i, c, V)$ where $V = 1$.

Then for read stream t_i, the average per iteration page miss count for *c* wrap-around adjacent accesses is

$$\omega(s_i, d_i, c) = \frac{c}{\phi\,(s_i, d_i)}$$

The affect of wrap-around adjacency on per iteration page miss count for read stream t_i is computed as

$$wadj(s_i, d_i, c, V) = \eta(s_i, d_i, c, V) - \omega(s_i, d_i, c)$$

Theorem 4.8: Given streams *S* with read stream t_i and write stream t_j that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$, minimum page miss count for read stream t_i is achieved via the wrap-around adjacent sequence $\tilde{S} = \langle r_i{:}\varepsilon_i, \ldots, w_j{:}\varepsilon_j \rangle$. Page miss count for write stream t_j is unaffected.

Proof: In the sequence $\tilde{S} = \langle r_i{:}\varepsilon_i, \ldots, w_j{:}\varepsilon_j \rangle$, read stream t_i proceeds as if no other vector is referenced, guaranteeing minimum page thrashing □

Corollary 4.9: Given streams *S* with read stream t_i and write stream t_j that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$, the average page miss count per access from t_i in the sequence $\tilde{S} = \langle r_i{:}\varepsilon_i, \ldots, w_j{:}\varepsilon_j \rangle$ is independent of the depth of loop unrolling *b* chosen in mapping logical streams $S^{(L)}$ to physical streams *S*. However, the average page miss count per access from write stream t_j is either constant or inversely proportional to *b*, as per Corollary 4.3.

4.1.3 Summary of Techniques

As demonstrated above, for a single module of page-mode components, grouping accesses by stream minimizes page overhead for streams not involved in a read-modify-write; for streams implementing this operation, intermixing and wrap-around adjacency are employed. The result is an access sequence with at most two distinct stream reference patterns, e.g.

$$\langle \ldots, r_k{:}\varepsilon_k, \ldots, \langle r_i{:}\theta_j, w_j{:}\theta_j \rangle{:}\ (\varepsilon_j / \theta_j), \ldots \rangle$$

The ordering algorithm derived in section 4.3 determines the specific reference sequence that minimizes page miss count for a given computation. Recall that wide-word access is optimized via the logical to physical stream mapping defined in section 3.3.

Note that though intermixing can minimize page miss count for write operations, the resulting sequence may not be amenable for execution on pipelined processors; alternating read and write accesses can force scalar-mode (non-pipelined) arithmetic operations. However, intermixing is justified if the additional access time resulting from a sub-optimal reference sequence exceeds the additional cost of performing scalar-mode computation. This issue is discussed in detail in chapter 7.

4.2 Single Module of Uniform-access Components

Deriving an access ordering algorithm for a single module of uniform-access components is trivial and presented here only for completeness. Since uniform-access components are insensitive to the sequence of memory requests, any order that preserves dependencies results in optimal effective memory bandwidth.

For streams *S*, let t_1 through t_{N_r} be read streams and t_{N_r+1} through t_N be write streams. Then the access sequence employed is

$$\tilde{S} = \langle r_1{:}\varepsilon_1, \ldots, r_{N_r}{:}\varepsilon_{N_r}, w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N \rangle$$

Recall that the stream interaction restriction limits dependencies to loop-independent anti-dependence and data dependence in the data-flow sense, as discussed in section 3.5. Thus, placing all reads prior to the first write maintains all dependencies.

4.2.1 Performance Predictor

A performance predictor for a single module of uniform-access components is computed below for the average time per data item accessed T_{avg}, and the effective processor-memory bandwidth BW.

If t_i is a read stream, the time to complete all references to t_i for a given sequence iteration is computed as the number of accesses ε_i multiplied by the uniform-access read cycle time $T_{u/r}$; i.e. $\varepsilon_i T_{u/r}$.

Then T_r, the time to complete all read accesses for a given iteration, is computed as the sum of the times to complete accesses for each individual read stream, so that

$$T_r = \sum_{\substack{t_i \in S \\ m_i = r}} \varepsilon_i T_{u/r}$$

T_w is defined as the time to complete all write access for a given iteration and is computed analogously to T_r, so that

$$T_w = \sum_{\substack{t_i \in S \\ m_i = w}} \varepsilon_i T_{u/w}$$

Then the average time per data item accessed T_{avg} is the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_r + T_w}{\sum_{t_i \in S} (\varepsilon_i \gamma_i)}$$

The effective memory bandwidth BW is the number of bytes of *relevant* data transferred per iteration divided by the time to complete all access; i.e.

$$BW = \frac{10^3 \sum_{t_i \in S} (\varepsilon_i \gamma_i d_i)}{T_r + T_w}$$

All times are in nanoseconds and sizes in bytes, as discussed in section 1.4.2, with bandwidth measured in megabytes per second.

4.3 Single Module of Page-mode Components

For a single module of page-mode components, an access ordering algorithm is derived that achieves optimal effective memory bandwidth by minimizing page overhead for a given computation while maintaining dependencies. Note that the access sequence generated is ‘statistically optimal’ in that it results in on average best case performance, given that stream alignment within a page is not restricted and therefore not known; such is the case for all algorithms developed for systems of page-mode components.

For streams not involved in a read-modify-write, grouping accesses by stream results in minimum page miss count (Theorem 4.2). Given two streams that implement this operation, further reduction in page overhead may be achieved for write and read accesses by intermixing (Theorem 4.6) and wrap-around adjacency (Theorem 4.8), respectively.

Then for streams S with no pair of streams implementing a read-modify-write, ordering is trivial. Let t_1 through t_{N_r} be read streams and t_{N_r+1} through t_N be write streams. An access sequence that minimizes page overhead while preserving dependencies is

$$\tilde{S} = \langle r_1 : \varepsilon_1, \ldots, r_{N_r} : \varepsilon_{N_r}, w_{N_r+1} : \varepsilon_{N_r+1}, \ldots, w_N : \varepsilon_N \rangle$$

If S contains one or more pair of streams implementing a read-modify-write, then an optimal access sequence $\tilde{S}$ is defined by the following algorithm:

> Determine the total ordering of *access sets* $\langle a_i : \varepsilon_i \rangle$, $t_i \in S$, that maximizes the reduction in page overhead achievable via intermixing and wrap-around adjacency and that maintains the partial ordering of access sets defined by the dependence relations. Reduction in page miss count for a particular ordering is calculated by the functions $imix(s, d, c, h, V)$ and $wadj(s, d, c, V)$ derived in sections 4.1.1 and 4.1.2, respectively.

Determining the total ordering of access sets that maximizes the potential reduction in page overhead is exponential in the number of streams in S. However, in practice, the stream count N tends to be small and dependencies significantly reduce the number of total orderings. Furthermore, page overhead is only affected by the relative position of streams implementing read-modify-writes. Read and write access sets not involved in a read-modify-write may be treated as a single read and write access set, respectively. The result is an efficient algorithm.

4.3.1 Example Problem

The following example illustrates the application of the ordering algorithm defined above. Consider the axpy operation

$$\forall i \qquad y_i \leftarrow a x_i + y_i$$

that generates the set of streams $S = \{t_x, t_{y_r}, t_{y_w}\}$ where $t_x = (x, s_x, d_x, r) : \varepsilon_x$, $t_{y_r} = (y, s_y, d_y, r) : \varepsilon_{y_r}$, and $t_{y_w} = (y, s_y, d_y, w) : \varepsilon_{y_w}$. Antidependence exists between corresponding elements of read stream t_{y_r} and write stream t_{y_w}, and data dependence exists between corresponding elements of t_{y_r} and t_{y_w}, and t_x and t_{y_w}.

The access sets are $\langle r_x : \varepsilon_x \rangle$, $\langle r_{y_r} : \varepsilon_{y_r} \rangle$, and $\langle w_{y_w} : \varepsilon_{y_w} \rangle$ for which two total orderings maintain dependencies: ($\langle r_x : \varepsilon_x \rangle$, $\langle r_{y_r} : \varepsilon_{y_r} \rangle$, $\langle w_{y_w} : \varepsilon_{y_w} \rangle$) and ($\langle r_{y_r} : \varepsilon_{y_r} \rangle$, $\langle r_x : \varepsilon_x \rangle$, $\langle w_{y_w} : \varepsilon_{y_w} \rangle$).

Considering each total ordering in turn, ($\langle r_x : \varepsilon_x \rangle$, $\langle r_{y_r} : \varepsilon_{y_r} \rangle$, $\langle w_{y_w} : \varepsilon_{y_w} \rangle$) presents the opportunity for intermixing $\langle r_{y_r} : \varepsilon_{y_r} \rangle$ and $\langle w_{y_w} : \varepsilon_{y_w} \rangle$ and results in the access sequence

$$\langle r_x : \varepsilon_x, \langle r_{y_r}, w_{y_w} \rangle : \varepsilon_{y_w} \rangle$$

The gross reduction in page overhead achieved by the ordering above is calculated as $imix(s_{y_w}, d_{y_w}, 1, \varepsilon_{y_w}, 2)$; intermix parameters are computed as discussed in 4.1.1.1.

The total ordering ($\langle r_{y_r} : \varepsilon_{y_r} \rangle$, $\langle r_x : \varepsilon_x \rangle$, $\langle w_{y_w} : \varepsilon_{y_w} \rangle$) provides wrap-around adjacency and results in the access sequence

$$\langle r_{y_r} : \varepsilon_{y_r}, r_x : \varepsilon_x, w_{y_w} : \varepsilon_{y_w} \rangle$$

The gross reduction in page overhead is calculated as $wadj(s_{y_r}, d_{y_r}, \varepsilon_{y_r}, 2)$.

The access ordering algorithm determines the total ordering of access sets that maximizes the reduction in page overhead. For the accesses sets of the axpy computation considered above, if $imix(s_{y_w}, d_{y_w}, 1, \varepsilon_{y_w}, 2) > wadj(s_{y_r}, d_{y_r}, \varepsilon_{y_r}, 2)$ then the access sequence $\langle r_x : \varepsilon_x, \langle r_{y_r}, w_{y_w} \rangle : \varepsilon_{y_w} \rangle$ results in optimal effective memory bandwidth, otherwise the sequence $\langle r_{y_r} : \varepsilon_{y_r}, r_x : \varepsilon_x, w_{y_w} : \varepsilon_{y_w} \rangle$ is optimal.

4.3.2 Performance Predictor

For a set of streams *S* and an access sequence $\tilde{S}$ defined by the algorithm above, a performance predictor is derived for the average time per data item accessed T_{avg} and the effective processor-memory bandwidth BW.

Access sequence $\tilde{S}$ is composed of some number of component sequences $\tilde{S}_i$, where the subscript is defined to be that of the stream referenced; for an intermix sequence the subscript is defined to be that of the read stream. Each $\tilde{S}_i$ must be in the form of

- a read access set $\langle r_i : \varepsilon_i \rangle$,
- a write access set $\langle w_i : \varepsilon_i \rangle$, or
- an intermix sequence $\langle \langle r_i : \theta_j, w_j : \theta_j \rangle : (\varepsilon_j / \theta_j) \rangle$.

If $\tilde{S}_i = \langle r_i : \varepsilon_i \rangle$ then $T(\tilde{S}_i)$, the time to complete the sequence $\tilde{S}_i$, is the sum of the number of accesses to t_i multiplied by the page-hit read cycle time $T_{p/r}$ and the average page miss count multiplied by the page miss time $T_{p/m}$; i.e.

$$T(\tilde{S}_i) = \varepsilon_i T_{p/r} + \begin{cases} \omega(s_i, d_i, \varepsilon_i) T_{p/m} & \text{when } t_i \text{ is wrap-around adjacent} \\ \eta(s_i, d_i, \varepsilon_i, V) T_{p/m} & \text{otherwise} \end{cases}$$

Similarly, if $\tilde{S}_i = \langle w_i : \varepsilon_i \rangle$ then $T(\tilde{S}_i)$ is the sum of the number of accesses to t_i multiplied by the page-hit write cycle time $T_{p/w}$ and the average page miss count multiplied by the page miss time $T_{p/m}$, so that

$$T(\tilde{S}_i) = \varepsilon_i T_{p/w} + \eta(s_i, d_i, \varepsilon_i, V) T_{p/m}$$

Finally, if $\tilde{S}_i = \langle \langle r_i : \theta_j, w_j : \theta_j \rangle : (\varepsilon_j / \theta_j) \rangle$ then $T(\tilde{S}_i)$ is the sum of the number of accesses to stream t_i (t_j) multiplied by the sum of the page-hit read and page-hit write cycle times and the sum of the average page miss counts for read and write operations multiplied by the page miss time $T_{p/m}$, so that

$$T(\tilde{S}_i) = \varepsilon_i (T_{p/r} + T_{p/w}) + (\eta(s_i, d_i, \varepsilon_i, V) + (\varepsilon_j / \theta_j) \rho(s_j, d_j, \theta_j)) T_{p/m}$$

From the preceding analysis, the time to complete an iteration of the access sequence $\tilde{S}$ is the sum of the times required to complete each component sequence; i.e.

$$T_{tot} = \sum_{\tilde{S}_i \in \tilde{S}} T(\tilde{S}_i)$$

Then the average time per data item accessed T_{avg} is the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_{tot}}{\sum_{t_i \in S} (\varepsilon_i \gamma_i)}$$

The effective memory bandwidth BW, measured in megabytes per second, is the number of bytes of relevant data transferred per iteration divided by the time to complete all accesses; i.e.

$$BW = \frac{10^3 \sum_{t_i \in S} (\varepsilon_i \gamma_i d_i)}{T_{tot}}$$

4.4 Simulation Results

For a single module of page-mode components, access ordering can significantly increase effective memory bandwidth over that achieved by the ‘natural’ sequence of references for a given computation. In this context, the natural reference sequence is the sequence that results from a straight-forward translation of the loop source code.

To illustrate the improvement in performance achieved via access ordering, and to validate performance models, simulation and analytic results are presented for a set of benchmark

scientific kernels. Recall that for both modeling and simulation, the processor is assumed sufficiently fast so that there is always an outstanding request. Thus, results represent maximum achievable bandwidth.

The parameters of the single-module memory are defined in Table 1; sizes are in bytes and times are in nanoseconds. These parameters are representative of the node memory system for the Intel IPSC/860, as detailed in [Moye91].

Table 1 Module Parameters (Single - Page)

Parameter	Value
w	8
p	4096
$T_{p/r}$	50
$T_{p/w}$	75
$T_{p/m}$	200

Table 2 presents simulation results comparing effective bandwidth achieved by the natural versus ordered access sequence for a range of scientific kernels. For access ordering, the depth of loop unrolling is 4 in all cases.

The *daxpy* computation is the double-precision version of the axpy computation discussed earlier. Similarly *dvaxpy* is the double-precision version of the *vaxpy* (vector axpy) computation

$$\forall i \qquad y_i \leftarrow a_i x_i + y_i$$

The remaining computations in Table 2 are selections from the Livermore Loops [Mcma90], with all vectors defined as double-precision. This set of benchmark kernels is used in all subsequent performance evaluations.

Access ordering improves performance over the natural access sequence for the given computations from 102% to 149%. Note that for LL-24 only a single vector is referenced so that no reordering is performed.

Table 2 Natural vs Ordered Performance (Single - Page)

Computation	Natural BW	Ordered BW	% Increase
daxpy	41.7	87.1	108.9
dvaxpy	38.8	85.1	119.3
LL-1	31.0	73.7	137.7
LL-3	32.0	79.8	149.4
LL-4	32.0	79.3	147.8
LL-5	31.0	73.7	137.7
LL-7	31.2	75.1	140.7
LL-11	30.5	70.9	132.5
LL-12	30.5	71.0	132.8
LL-20	31.3	75.6	141.5
LL-21	41.0	82.9	102.2
LL-22	30.8	72.6	135.7
LL-24	158.5	158.5	0.0

Table 3 compares performance of ordered accesses for the benchmark computations as calculated analytically and measured via simulation; again, loops are unrolled to a depth of 4. Note that in all cases analytic and simulation results differ by less than 1%, validating the accuracy of the performance model.

Table 3 Analytic vs Simulation Results (Single - Page)

Computation	Analysis		Simulation	
	T_{avg}	BW	T_{avg}	BW
daxpy	91.9	87.1	91.9	87.1
dvaxpy	94.0	85.1	94.0	85.1
LL-1	108.6	73.7	108.6	73.7
LL-3	100.3	79.8	100.3	79.8
LL-4	100.9	79.3	100.8	79.3
LL-5	108.6	73.7	108.6	73.7
LL-7	106.5	75.1	106.5	75.1
LL-11	112.8	70.9	112.7	70.9
LL-12	112.8	70.9	112.8	71.0
LL-20	105.9	75.6	105.9	75.6
LL-21	96.6	82.9	96.6	82.9
LL-22	110.3	72.5	110.3	72.6
LL-24	50.4	158.8	50.5	158.5

4.5 Summary

This chapter develops optimal access ordering algorithms for a single module of uniform-access and page-mode components. Performance models are derived for the maximum effective memory bandwidth achievable by a given computation.

As uniform-access components are insensitive to the sequence of memory requests, any ordering that preserves dependence is optimal. Ordering is trivial and a performance model is derived in a straight-forward fashion.

For page-mode components, an optimal access sequence must minimize page miss count for a given computation while maintaining dependencies. The access ordering algorithm derived results in a sequence consisting of (potentially intermixed) access sets arranged so as to maximize reduction in page overhead achievable via intermixing and wrap-around adjacency. The algorithm has a time complexity exponential in the number of streams, though the stream count N tends to be small and effective optimizations exist.

Simulation results are presented for a system constructed from page-mode components. Access ordering is shown to significantly increase effective memory bandwidth over that achieved by the ‘natural’ sequence of reference for a set of benchmark scientific kernels. The performance model is demonstrated to be accurate.

5 Sequentially Interleaved Architecture

This chapter derives access ordering algorithms and performance predictors for a sequentially interleaved memory system as depicted in Figure 8. Sequential interleaving is the ‘standard’ parallel memory storage scheme whereby for an m module system, word a maps to module $a \bmod m$.

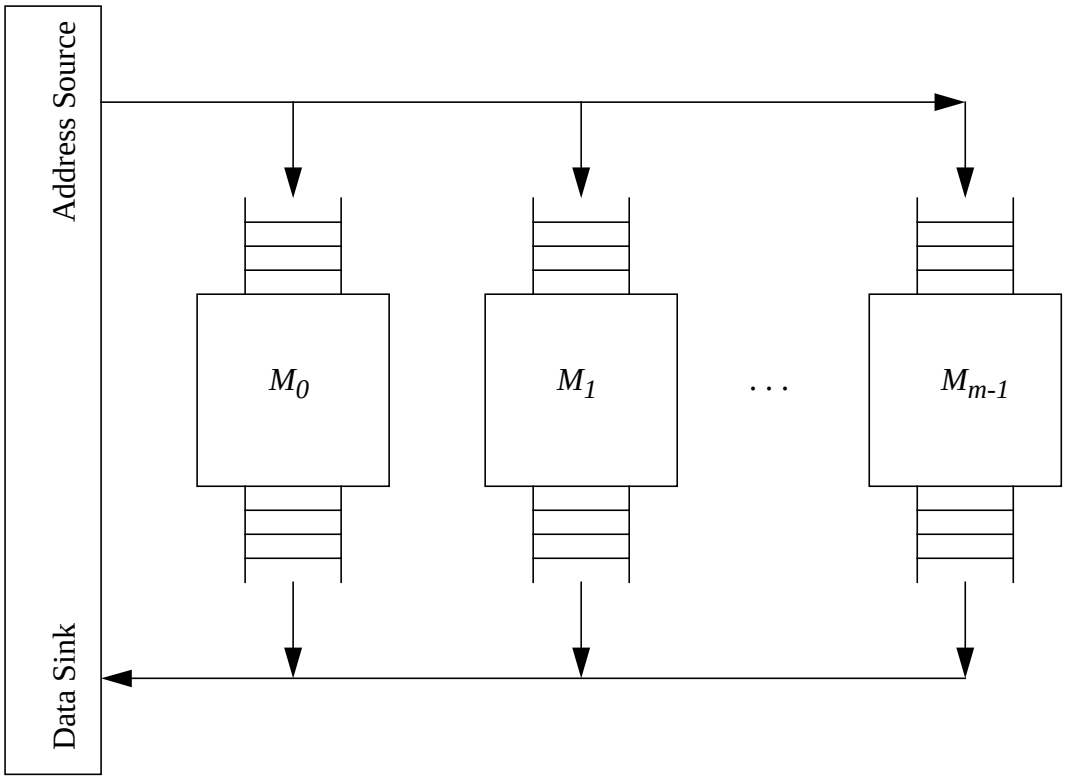

Figure 8 Sequentially Interleaved Architecture

The interleaved memory system is defined to function as follows. Access requests are directed to the appropriate module, as determined by the storage scheme. If input buffer space is available then the request is queued, otherwise the memory system blocks until a

buffer slot is freed. Access requests are serviced at a module in the order queued, with data from read requests placed in the module's output buffer.

Note that in a parallel memory system, accesses may not complete in the order of request. Read accesses are assumed tagged so that data may be returned in the requested order. The details of such a tagging scheme are not important to the analysis presented here, and as such are not defined. It is sufficient to assume that results can be returned at the rate satisfied. Recall that in modeling maximum effective bandwidth, the request rate is assumed sufficient such that performance is limited by the memory. These are common assumptions in the study of parallel memory systems.

Section 5.1 discusses the problem space for efficient utilization of sequentially interleaved memory. Analytic results characterizing the interaction of a single stream with an interleaved architecture are presented in section 5.2. Section 5.3 extends the basic MAP notation to simplify expressing access sequences for parallel memory systems. Finally, sections 5.4 and 5.5 derive ordering algorithms and performance predictors for a sequentially interleaved system under the assumption of unknown and known stream alignments, respectively.

5.1 Problem Dimensions

In general, to efficiently utilize an interleaved memory system, stream accesses must be ordered so as to

- maximize concurrency and
- minimize page overhead, when applicable.

Ordering accesses to maximize concurrency requires knowledge of stream alignment so that nonconflicting module references may be scheduled to proceed in parallel. In the

absence of alignment information, accesses can be ordered to increase the likelihood of concurrency.

Techniques for minimizing page overhead come directly from analytic results derived in chapter 4 for a single memory module. Page miss count at module M_k is minimized for a given iteration if elements of a stream stored at that module are referenced consecutively without an intervening access to M_k. For two streams that implement a read-modify-write, page miss count may further be reduced via intermixing and wrap-around adjacency.

Optimal effective memory bandwidth results from an access sequence that minimizes completion time for all accesses in a loop. Such a sequence may require a trade-off between maximum concurrency and minimum page overhead. To illustrate, consider ordering accesses for the computation

$$
\begin{aligned}
&\forall i \\
&\{ \\
&\quad e_i \leftarrow fn1\,(q_i, x_i, y_i, z_i) \\
&\quad f_i \leftarrow fn2\,(q_i, x_i, y_i, z_i) \\
&\quad g_i \leftarrow fn3\,(q_i, x_i, y_i, z_i) \\
&\quad q_i \leftarrow fn4\,(q_i) \\
&\quad x_i \leftarrow fn5\,(x_i) \\
&\quad y_i \leftarrow fn6\,(y_i) \\
&\quad z_i \leftarrow fn7\,(z_i) \\
&\}
\end{aligned}
$$

Assume that the 4 read streams and 7 write streams are of sufficient stride such that each successive access results in a page miss, the exception being a write immediately following a read of the same vector element at a given module. Furthermore, assume that the

number of modules is 2 and that strides are even, so that accesses from each stream are serviced by a single module. The depth of loop unrolling equals 1.

Figure 9 demonstrates the time to initiate accesses in successive iterations of the above computation for three different access orderings, given that memory references hitting in the current page require 1 time unit ($T_{p/r}$, $T_{p/w}$) and a page miss incurs a 3 time unit penalty ($T_{p/m}$). Stream accesses in Figure 9 are labeled by vector, with read and write streams subscripted with *r* and *w* respectively.

Figure 9(a) depicts an ordering that minimizes page miss count via intermixing; for all 4 read-modify-write operations, read accesses immediately precede corresponding writes. Figure 9(b) depicts an ordering that maximizes concurrency by initiating all read accesses prior to the first write; in doing so, write accesses at module M_1 are completely overlapped with those at M_0. Finally, Figure 9(c) depicts an optimal solution that balances minimizing page overhead and maximizing concurrency to achieve the minimum completion time.

For parallel memory systems constructed from page-mode components optimal bandwidth can result from a sequence that is neither regular nor intuitive, as demonstrated by the example above.

5.2 Single Stream Module Interaction

To develope access ordering algorithms, analytic results are first required that characterize the interaction of a single stream with an interleaved memory architecture. In particular, it is necessary to model the mapping of accesses to modules and the effective stride of access at a given module. In doing so, an additional restriction is placed on data item, word and page sizes: *d*, *w*, and *p* are assumed to be powers of 2.

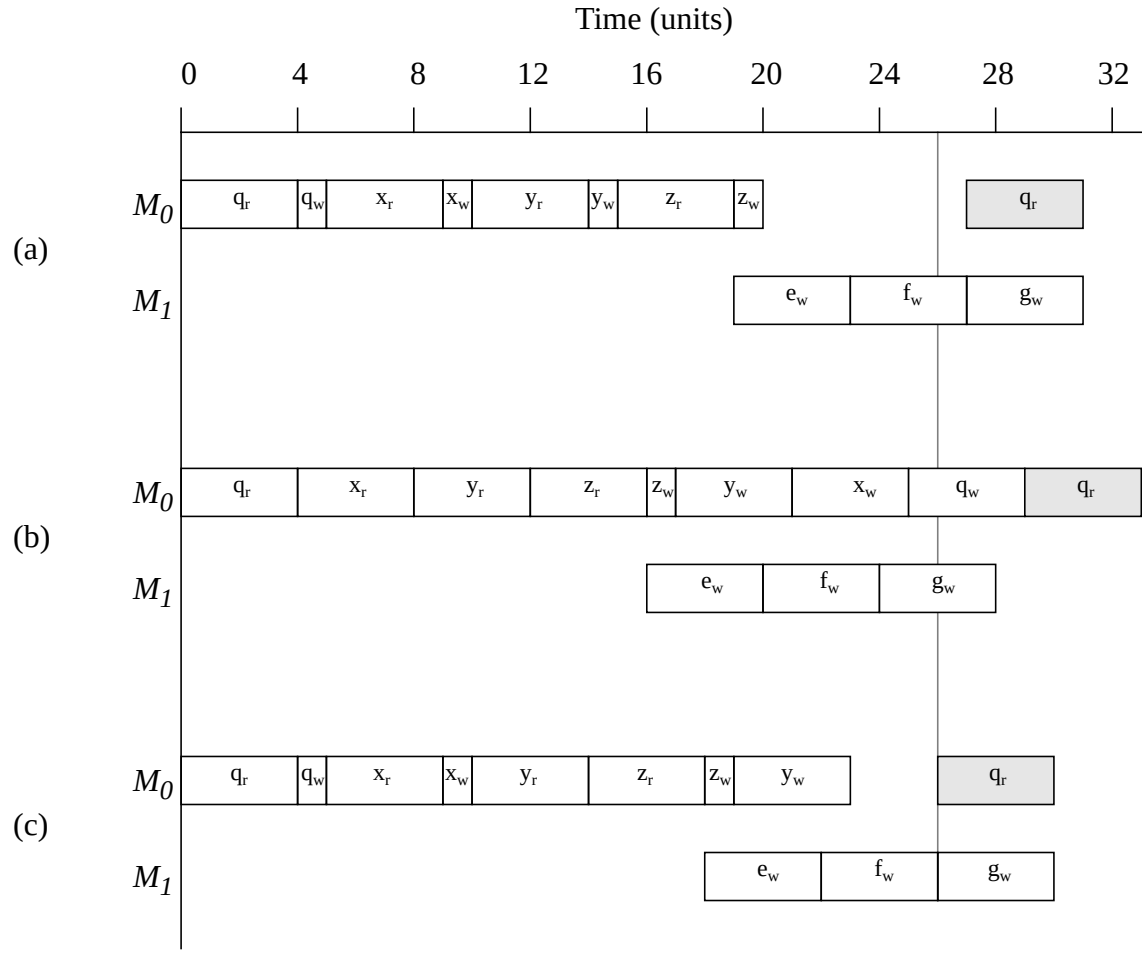

Figure 9 Minimizing Completion Time

5.2.1 Access Mapping

For an *m* module interleaved memory, the mapping of stream accesses to modules is characterized by the number of modules accessed and the distribution of accesses across those modules. If stream alignment is known, then it can also be determined to which modules stream accesses map.

Given streams *S* and $t_i \in S$, an *access cycle* is defined as a minimal set of consecutive accesses from stream t_i such that the first access in each adjacent cycle references a word containing similarly aligned data items at the same module. Access cycle length is the

least common multiple of the number of bytes traversed per access, $s_i d_i$, and the number of bytes across all modules mw; i.e. $lcm(s_i d_i, mw)$. Then the number of stream accesses per access cycle is

$$\frac{lcm(s_i d_i, mw)}{s_i d_i} = \frac{mw}{gcd(s_i d_i, mw)} \tag{1}$$

If the number of bytes traversed per access is a multiple of the word size, i.e. $w \mid s_i d_i$, then each access references a word containing similarly aligned data items. The number of modules referenced is equal to the number of stream accesses per cycle, as computed in equation (1), and reduces to

$$\frac{m}{gcd(\frac{s_i d_i}{w}, m)}$$

Each module is referenced exactly once per cycle, resulting in a sequence of module accesses periodic in the number of modules referenced.

Now consider the case where the number of bytes traversed per access is not a multiple of the word size. By definition $s_i d_i > w$ so that, in computing the number of accesses per cycle from equation (1), the $gcd(s_i d_i, mw)$ must be a power of 2 less than *w.* Thus the number of accesses per cycle is a multiple of *m*, and references are uniformly distributed across all *m* modules *on a per cycle basis*.

Note that each module is not necessarily referenced exactly once for each *m* consecutive stream accesses. Figure 10 depicts a single access cycle for a 4 module system, a word size of 4 bytes and a data size of 1 byte referenced at a stride of 6; aligned as shown, each set of 4 accesses maps to 3 modules.

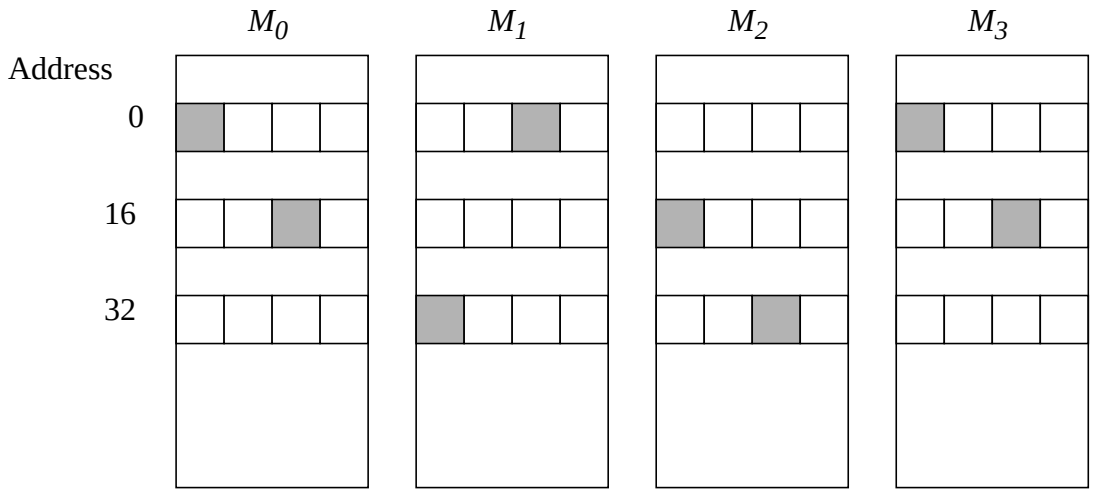

Figure 10 Access Mapping Diagram

Lemma 5.1: Given streams S and $t_i \in S$, accesses from stream t_i are distributed uniformly across a number of modules defined by:

$$\mu_i = \begin{cases} \dfrac{m}{gcd(\dfrac{s_i d_i}{w}, m)} & \text{when } w \mid s_i d_i \\ m & \text{otherwise} \end{cases}$$

Furthermore, when the number of bytes traversed per access is a multiple of the word size, i.e. $w \mid s_i d_i$, then the sequence of modules accessed by t_i has a period of μ_i.

Let Z_i represent the set of modules to which stream t_i maps. If the number of modules accessed by t_i is less than m, then Z_i is only defined if stream alignment is known. For stream t_i aligned to base module M_{B_i}, the set of modules referenced is

$$Z_i = \begin{cases} \{M_0, \ldots, M_{m-1}\} & \text{when} \quad \mu_i = m \\ \{M_j \mid j = (B_i + k\dfrac{s_i d_i}{w}) \bmod m, \quad 0 \le k \le \mu_i - 1\} & \text{when} \quad \mu_i < m \end{cases}$$

In computing Z_i, if the number of modules referenced is less than m then Z_i is the first μ_i modules accessed starting from base module M_{B_i}.

Theorem 5.2: Given streams S and $t_i, t_j \in S$, $Z_i \cap Z_j = \emptyset$, or $Z_i \cap Z_j = Z_i$, or $Z_i \cap Z_j = Z_j$.

Proof: Follows directly from the fact that for any $t_i \in S$, the number of modules referenced μ_i must be a power of 2. □

Thus, two streams either reference no modules in common, i.e. are *nonconflicting,* or one stream accesses a subset of the modules accessed by the other.

5.2.2 Module Stride

To apply functions modeling page overhead derived in chapter 4 for a single module system to individual modules of an interleaved system requires deriving the *module stride* for a given stream. Module stride is defined as the stride of reference for a given stream as observed at a particular module.

Given streams S and $t_i \in S$, if the number of bytes traversed per access is a multiple of the word size, i.e. $w \mid s_i d_i$, then at a module M_k referenced by t_i the observed stride of access is constant; recall that t_i references modules in a sequence periodic in the number of modules accessed. Module stride is computed as the product of the number of modules accessed and the actual stride, divided by the total number of modules; i.e.

$$\frac{\mu_i s_i}{m} = \frac{s_i}{gcd(\frac{s_i d_i}{w}, m)}$$

Now consider the case where the number of bytes traversed per access, $s_i d_i$, is not a multiple of the word size. From the analysis of access mapping, the number of accesses per module per access cycle is

$$\frac{w}{gcd(s_i d_i, mw)} \tag{2}$$

The number of bytes traversed per module per cycle is the cycle length divided by the number of modules; i.e.

$$\frac{lcm(s_i d_i, mw)}{m} = \frac{(s_i d_i)\, w}{gcd(s_i d_i, mw)} \tag{3}$$

So the average number of bytes traversed per access is the ratio of (3) to (2), or $s_i d_i$, resulting in an average module stride of s_i.

Figure 11 depicts a single access cycle, plus a portion of the adjacent cycle, for a 4 module system with a word size of 4 bytes and a data size of 1 byte referenced at a stride of 5; strides between individual accesses at a given module take on values of 3 and 11, resulting in an average module stride of 5.

Lemma 5.3: Given streams S and $t_i \in S$, the average stride of access observed at all modules referenced, i.e. the module stride, is computed as

$$\xi_i = \begin{cases} \dfrac{s_i}{gcd(\frac{s_i d_i}{w}, m)} & \text{when } w \mid s_i d_i \\ s_i & \text{otherwise} \end{cases}$$

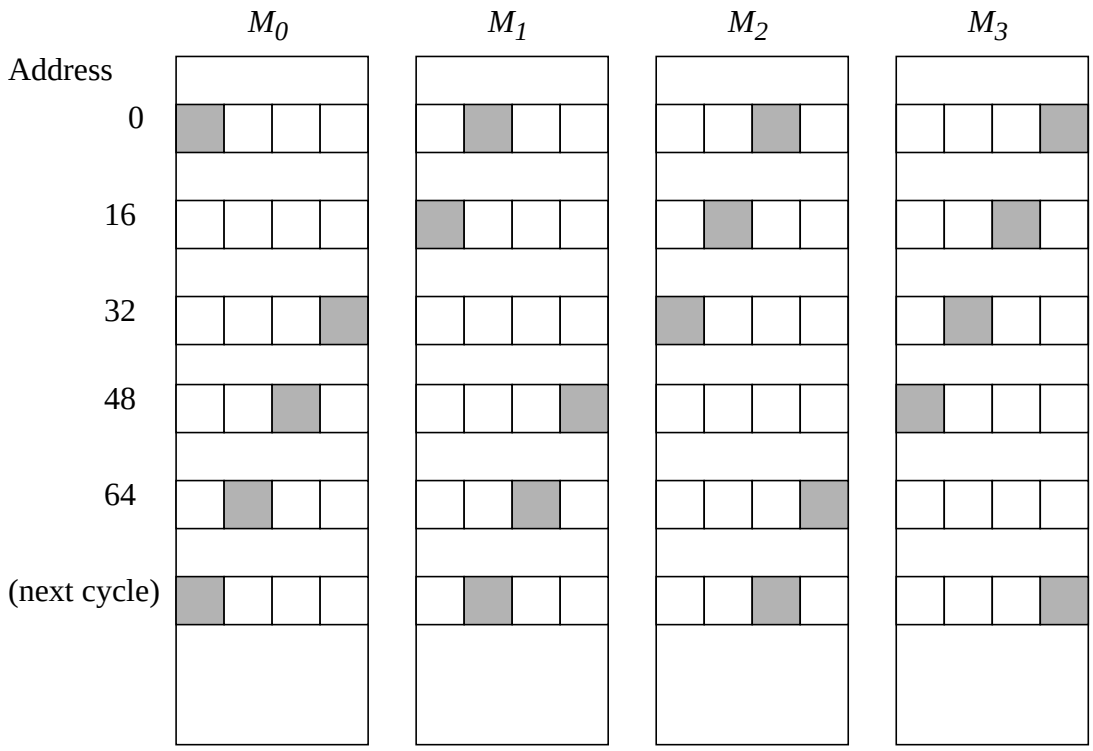

Figure 11 Module Stride Diagram

For an interleaved system, analytic results for access mapping and module stride completely characterize the interaction of a single access stream with the memory architecture. Note that streams for which the number of bytes traversed per access, sd, is a multiple of the word size reference a sequence of modules periodic in the number of modules accessed with a constant module stride. Access ordering and performance modeling are significantly complicated by streams that do not posses this property. Fortunately, such streams rarely occur in practice. Thus, for the remainder of this chapter, all streams are assumed defined such that $w \mid sd$.

5.3 Extended MAP Notation

To facilitate the specification of a MAP access sequence that effectively utilizes a parallel memory system, the recursive sequence definition of section 3.1 is augmented with the additional clause

4. $\langle [A_1, \ldots, A_n \mid \alpha_1, \ldots, \alpha_n] \rangle$ is an access sequence where $A_1, \ldots, A_n$ are access sequences and $\alpha_1, \ldots, \alpha_n$ are positive integers. $A_1, \ldots, A_n$ are performed left to right in a modified round-robin fashion, with α_i accesses from A_i until all accesses in $A_1, \ldots, A_n$ have been initiated. If fewer than α_i accesses remain in A_i, then only these accesses are issued. When all accesses specified in A_i have been initiated, A_i is dropped from the pattern.

To illustrate, the access sequence notation

$$\langle [r_i{:}5, w_j{:}3 \mid 2, 2] \rangle$$

defines the linear sequence of references

$$\langle r_i, r_i, w_j, w_j, r_i, r_i, w_j, r_i \rangle$$

A *strict round-robin* selection of accesses from each of the sequences $A_1, \ldots, A_n$ is achieved when $\alpha_1 = \ldots = \alpha_n = 1$. For visual clarity, strict round-robin selection is denoted simply as $\langle [A_1, \ldots, A_n] \rangle$.

The above notation affords convenient specification for controlling access to parallel modules. For example, given an interleaved system and known stream alignment, each sequence A_i can represent accesses serviced at module M_i such that a strict round-robin selection of accesses from $A_0, \ldots, A_{m-1}$ results in concurrency among accesses from different streams.

5.4 Access Ordering Algorithms for Unknown Alignments

For a sequentially interleaved memory, access ordering algorithms and performance predictors are derived based on the assumption that stream alignments, with respect to modules or each other, are unknown. In the absence of alignment information an optimal solution can not be derived for the general case, as knowledge of stream alignment is required to schedule nonconflicting references to proceed in parallel. However, accesses can be ordered to increase the likelihood of concurrency.

Sections 5.4.1 and 5.4.2 develop ordering algorithms and performance predictors for systems of uniform-access and page-mode components, respectively. The effectiveness of access ordering and accuracy of performance models are demonstrated via simulation in 5.4.3. Section 5.4.4 summarizes results.

5.4.1 Interleaved Storage and Uniform-access Components

For an *m* module interleaved system of uniform-access components, an access ordering algorithm need only maximize module concurrency. As a general optimal ordering can not be derived, a heuristic solution is presented below.

Recall that a stream t_i references μ_i modules in a sequence with period μ_i. If $\mu_i = m$, i.e. all modules are referenced, then maximum concurrency is achieved in performing all accesses to t_i consecutively for a given iteration. If $\mu_i < m$ and the number of consecutive references to t_i exceeds μ_i, then $m - \mu_i$ modules are potentially idle for the time required to initiate accesses to t_i.

For a set of *N* independent streams, consider a sequence that interleaves a number of accesses from each stream equal to the number of modules referenced (or the number of accesses remaining, whichever is smaller); e.g.

$$\langle [a_1{:}\varepsilon_1, \ldots, a_N{:}\varepsilon_N \mid \mu_1, \ldots, \mu_N] \rangle$$

The above sequence maximizes concurrency for a stream t_i by issuing sets of (at most) μ_i consecutive accesses to that stream, the maximum number that can proceed in parallel. Furthermore, sets of accesses from each stream are interleaved to increase the likelihood of concurrency among accesses from different streams. In the absence of alignment information, no sequence can guarantee greater concurrency.

For a general set of streams *S*, accesses are performed in two phases: a read phase and a write phase. By the stream interaction restriction, streams associated with each phase are independent. If streams t_1 through t_{N_r} are read streams and t_{N_r+1} through t_N are write streams, then the access sequence employed is

$$\tilde{S} = \langle [r_1{:}\varepsilon_1, \ldots, r_{N_r}{:}\varepsilon_{N_r} \mid \mu_1, \ldots, \mu_{N_r}],\ [w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N \mid \mu_{N_r+1}, \ldots, \mu_N] \rangle$$

In the sequence above, accesses from each phase are ordered to maximize concurrency for individual streams and increase the likelihood of concurrency among accesses from different streams. Dependencies are maintained as all read accesses are initiated prior to the first write.

5.4.1.1 Performance Predictor

Given streams *S* and access sequence $\tilde{S}$ as defined above, a performance predictor is derived for the average time per data item accessed T_{avg} and processor-memory bandwidth BW. Because alignments are unknown, it must be assumed that accesses from different streams can not be serviced concurrently. Thus, the models represent a lower bound on performance.

The maximum number of accesses from stream t_i serviced at any module for a given iteration, ψ_i, is the ceiling of the number of accesses per iteration divided by the number of modules accessed; i.e.

$$\psi_i = \left\lceil \frac{\varepsilon_i}{\mu_i} \right\rceil$$

For a read stream t_i, if the number of streams N is greater than one then the time to complete all accesses for a given iteration is the maximum number of references at any given module ψ_i multiplied by the uniform-access read cycle time $T_{u/r}$. For the special case of $N = 1$, the average time to complete all reads is the product of the number of accesses ε_i and the average time per access; i.e.

$$\varepsilon_i \frac{T_{u/r}}{\mu_i}$$

Let T_r be the time required to complete all read accesses for a given iteration. Then T_r is computed as the sum of the times to complete accesses for each individual read stream, i.e.

$$T_r = \begin{cases} \varepsilon_i \dfrac{T_{u/r}}{\mu_i} & \text{when } N = 1 \text{ and for } t_i \in S,\ m_i = r \\ \displaystyle\sum_{\substack{t_i \in S \\ m_i = r}} \psi_i T_{u/r} & \text{when } N \geq 2 \end{cases}$$

T_w is defined as the time to complete all write accesses for a given iteration and is computed analogously to T_r, so that

$$T_w = \begin{cases} \varepsilon_i \dfrac{T_{u/w}}{\mu_i} & \text{when } N = 1 \text{ and for } t_i \in S,\ m_i = w \\ \displaystyle\sum_{\substack{t_i \in S \\ m_i = w}} \psi_i T_{u/w} & \text{when } N \geq 2 \end{cases}$$

Then the average time per data item accessed T_{avg} is the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_r + T_w}{\displaystyle\sum_{t_i \in S} \varepsilon_i \gamma_i}$$

The effective memory bandwidth BW, in megabytes per second, is the number of bytes of relevant data transferred per iteration divided by the time to complete all access; i.e.

$$BW = \frac{10^3 \displaystyle\sum_{t_i \in S} \varepsilon_i \gamma_i d_i}{T_r + T_w}$$

5.4.2 Interleaved Storage and Page-mode Components

For an interleaved memory constructed from page-mode components, optimal performance results from an access sequence that balances maximizing concurrency with minimizing page overhead to achieve minimum completion time. In the absence of alignment information, a general optimal ordering algorithm can not be derived. A heuristic solution is presented below.

In the sections that follow, an access strategy is first developed for a set of independent streams. Intermixing and wrap-around adjacency are then employed to reduce page overhead for computations implementing read-modify-write operations. Finally, a general ordering algorithm is presented and a performance predictor derived for the ordered accesses.

5.4.2.1 A General Access Strategy

Consider a set of N independent streams S. By Theorem 4.2, page miss count at module M_k is minimized when elements of stream $t_i \in S$ stored at that module are referenced consecutively without an intervening access to M_k. Then for streams S, page overhead is minimized by performing all accesses to each stream consecutively for a given iteration, as in the sequence

$$\langle a_1{:}\varepsilon_1, \ldots, a_N{:}\varepsilon_N \rangle$$

Alternatively, as in the ordering algorithm derived in 5.4.1, potential concurrency can be maximized by interleaving a number of accesses from each stream equal to the number of modules referenced; e.g.

$$\langle\, [a_1{:}\varepsilon_1, \ldots, a_N{:}\varepsilon_N \mid \mu_1, \ldots, \mu_N] \,\rangle$$

The above sequences address conflicting requirements. The first minimizes page miss count at the cost of potentially decreased concurrency. The second increases potential concurrency at the cost of increased page miss count.

In choosing a general method of access, the following observations are made. First, the most common stride of access is 1. At a stride of 1, or any stride that results in all modules being referenced, performing accesses to each stream consecutively results in maximum concurrency and minimum page miss count. Conversely, interleaving accesses from different streams results in maximum page miss count without an increase in concurrency. Second, in the absence of alignment information, interleaving references can not guarantee concurrency among accesses from nonconflicting streams.

Based on these observations it is concluded that performing all accesses to each stream consecutively constitutes a better access strategy than an interleaved sequence of references. Essentially, a guaranteed minimization of page overhead for all streams is chosen over a potential increase in concurrency for nonconflicting streams. Thus, the access ordering algorithm derived below specifies an access sequence consisting of (potentially intermixed) access sets $\langle a_i : \varepsilon_i \rangle$, $t_i \in S$.

5.4.2.2 Intermixing and Wrap-around Adjacency

For streams S with one or more pair of streams that implement a read-modify-write, access sets can be ordered to reduce page overhead via intermixing and wrap-around adjacency. In the absence of alignment information, it must be assumed that all streams in S are conflicting. Thus, ordering access sets to exploit intermixing and wrap-around adjacency is analogous to that for a single module system as discussed below.

Consider a read stream t_i and write stream t_j implementing a read-modify-write, i.e. $t_i, t_j \in S$ and $v_i = v_j$. Wrap-around adjacency results when accesses from t_i and t_j occur at the beginning and end of a sequence, respectively. Within a given iteration, writes to t_j

reference the same vector elements read from t_i so that for each subsequent iteration, reads from t_i proceed as if no other vector is referenced.

The effect of wrap-around adjacency for an interleaved system is analogous to that for a single module system, and reduction in read stream page miss count is modeled by the function $wadj(s, d, c, V)$ derived in 4.1.2. In employing this function for an interleaved system, $wadj(s, d, c, V)$ must model the reduction in page overhead achieved *at the module servicing the greatest number of accesses*. Stride s is module stride and the number of accesses c is the maximum number at any module; for read stream t_i, $s = \xi_i$ and $c = \psi_i$. The number of vectors V is the number referenced by all streams in S.

Note that for an interleaved system, more than one pair of streams may exhibit wrap-around adjacency. This can occur when two or more sets of streams implementing a read-modify-write are nonconflicting. However, in the absence of alignment information, it is assumed that every pair of streams conflict so that wrap-around adjacency benefits at most one.

Intermixing reduces page overhead for write operations by interleaving accesses from a pair of streams implementing a read-modify-write. Recall that for a single module, the general intermix sequence as derived in section 4.1.1 is

$$\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle \tag{4}$$

For an interleaved system, the above sequence is modified to maximize concurrency as well as minimize page overhead. However, the pattern of access observed at individual modules is still that of the general intermix sequence.

For read stream t_i and write stream t_j, if the number of modules accessed equals one then the optimal intermix sequence and intermix parameters are those derived in Theorem 4.6 for a single module system.

If the number of modules accessed by read stream t_i and write stream t_j is greater than one, then an optimal intermix sequence must maximize concurrency and minimize page overhead. Recall that for the general intermix sequence, the intermix parameter *c* must be a multiple of the intermix factor to optimize wide word access and maintain data dependence. Then if the number of modules accessed by t_j (t_i) is a multiple of the intermix factor θ_j, i.e. $\theta_j \mid \mu_j$, the optimal intermix sequence is

$$\langle \ldots, [r_i{:}\varepsilon_i, w_j{:}\varepsilon_j \mid \mu_i, \mu_j], \ldots \rangle$$

Page miss count for write operations is 0, as corresponding read and write accesses occur alternately at each module referenced. Concurrency is maximized as the number of consecutive accesses to t_i and t_j is equal to the number of modules accessed (or the number of accesses remaining, whichever is smaller). Data dependence is maintained as the number of consecutive accesses to each stream is a multiple of the intermix factor. Note that *individual modules* observe the general intermix sequence (4); intermix parameter *c* is 1, as read and write operations are initiated alternately at each module referenced, and *h* is ψ_j at the module servicing the maximum number of accesses.

If the number of modules accessed by read stream t_i and write stream t_j is greater than one but not a multiple of the intermix factor θ_j, then the intermix sequence employed is

$$\langle \ldots, \langle r_i{:}\varepsilon_i, w_j{:}\varepsilon_j \rangle, \ldots \rangle \tag{5}$$

Concurrency is maximized for each stream as each of the μ_j modules referenced is accessed with period μ_j. However, page overhead is not guaranteed to be minimal.

By definition, if intermixing reduces the page miss count for write operations then minimum page overhead is achieved when the intermix parameter *c* is equal to the intermix factor θ_j. As discussed above, if θ_j divides the number of modules referenced μ_j then

accesses to t_i and t_j can be issued so that page miss count for write stream t_j is 0 and concurrency is maximized. Otherwise, interleaving sets of θ_j accesses from each of t_i and t_j minimizes page overhead but may result in some of the μ_j modules referenced remaining idle with each set of θ_j reads and writes. Thus, optimal intermix performance results in a trade-off between minimum page overhead and maximum concurrency.

In the intermix sequence (5), concurrency is chosen over page overhead in a potentially suboptimal solution; however, for small strides, the additional page overhead for performing all ε_j (ε_i) read and write accesses consecutively is minimal. Again, note that *individual modules* observe the general intermix sequence (4); intermix parameter *c* is ψ_j for the module servicing the maximum number of accesses, and *h* is 1.

The effect of intermixing for an interleaved system is analogous to that for a single module system, and reduction in write stream page miss count is modeled by the function $imix(s, d, c, h, V)$ derived in section 4.1.1. In employing this function for an interleaved system, $imix(s, d, c, h, V)$ must model the reduction in page miss count achieved *at the module servicing the greatest number of accesses*. Stride *s* is module stride so that for write stream t_j, $s = \xi_j$. The intermix parameters *c* and *h* are dependent on the intermix sequence, and are derived in the preceding analysis. The number of vectors *V* is the number referenced by all streams in *S*.

Note that for an interleaved system, two or more pair of streams may benefit from intermixing when each write stream is data dependent on each read stream. This can occur when sets of streams implementing read-modify-writes are nonconflicting. For example, if streams t_{y_r} and t_{y_w} and streams t_{x_r} and t_{x_w} are two pairs of corresponding read and write streams, with t_{y_w} and t_{x_w} each data dependent on t_{y_r} and t_{x_r}, then both t_{y_w} and t_{x_w} can benefit from intermixing if they are nonconflicting. However, it is assumed that every pair of streams conflict. Thus, for this example, at most one of t_{y_w} and t_{x_w} may be considered to benefit from intermixing.

5.4.2.3 Access Ordering Algorithm

For a set of streams *S* with no pair of streams implementing a read-modify-write, ordering is trivial. Let t_1 through t_{N_r} be read streams and t_{N_r+1} through t_N be write streams. An access sequence that minimize page overhead while preserving dependence is

$$\tilde{S} = \langle r_1{:}\varepsilon_1, \ldots, r_{N_r}{:}\varepsilon_{N_r}, w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N \rangle$$

For streams *S* with one or more pair of streams implementing a read-modify-write, an access sequence is defined by an ordering algorithm analogous to that derived in 4.3 for a single module system:

> Determine the total ordering of access sets $\langle a_i{:}\varepsilon_i \rangle$, $t_i \in S$, that maximizes the reduction in page overhead achievable via intermixing and wrap-around adjacency and that maintains the partial ordering of access sets defined by the dependence relations. Reduction in page overhead for a particular ordering is calculated by the functions $wadj(s, d, c, V)$ and $imix(s, d, c, h, V)$ as discussed in 5.4.2.2.

Though the algorithm is exponential in the number of streams in *S*, the stream count *N* tends to be small, dependencies reduce the number of total orderings, and access sets not involved in a read-modify-write may be coalesced by mode.

As the ordering algorithm presented above is completely analogous to that for a single module of page-mode components, the example problem of section 4.3.1 serves to illustrate its application; only intermix parameters differ, as discussed in section 5.4.2.2.

5.4.2.4 Performance Predictor

For a set of streams *S* and an access sequence $\tilde{S}$ defined by the algorithm above, a performance predictor is derived for the average time per data item accessed T_{avg} and the processor-memory bandwidth BW. As alignments are unknown, it is assumed that accesses from different streams do not exhibit concurrency. Thus, the models represent a lower bound on performance.

Functions modeling page overhead derived in chapter 4 for a single module system are applicable to accesses at individual modules of an interleaved system. Recall that in general, average page miss count is modeled by the function $\eta(s, d, c, V)$. For a read stream that is wrap-around adjacent, average page miss count is modeled by the function $\omega(s, d, c)$. Finally, for an intermixed write stream, average page miss count is modeled by the function $\rho(s, d, c)$. Note that in employing these functions for an interleaved system, stride s is module stride.

Access sequence $\tilde{S}$ is composed of some number of component sequences $\tilde{S}_i$, where the subscript is defined to be that of the stream referenced; for an intermix sequence the subscript is defined to be that of the read stream. Each $\tilde{S}_i$ must be in the form of

- a read access set $\langle r_i : \varepsilon_i \rangle$,
- a write access set $\langle w_i : \varepsilon_i \rangle$, or
- an intermix sequence $\langle \langle r_i : c, w_j : c \rangle : h \rangle$ or $\langle [r_i : \varepsilon_i, w_j : \varepsilon_j \mid \mu_i, \mu_j] \rangle$.

If $\tilde{S}_i = \langle r_i : \varepsilon_i \rangle$ and the number of streams N is greater than one then $T(\tilde{S}_i)$, the time to complete the sequence $\tilde{S}_i$, is the sum of the maximum number of references at any module ψ_i multiplied by the page-hit read cycle time $T_{p/r}$ and the average page miss count at that module multiplied by the page miss time $T_{p/m}$. For the special case of $N = 1$, the average time to complete all reads is the product of the number of accesses ε_i and the average time per access so that

$$T(\tilde{S}_i) = \begin{cases} \varepsilon_i \left(\dfrac{T_{p/r} + \eta(\xi_i, d_i, 1, V) T_{p/m}}{\mu_i} \right) & \text{when } N = 1 \\ \psi_i T_{p/r} + \omega(\xi_i, d_i, \psi_i) T_{p/m} & \text{when } N \geq 2 \text{ and } t_i \text{ wrap-around adj.} \\ \psi_i T_{p/r} + \eta(\xi_i, d_i, \psi_i, V) T_{p/m} & \text{when } N \geq 2 \text{ and } t_i \text{ not wrap-around adj.} \end{cases}$$

As discussed in section 5.4.2.2, at most one read access set may be considered wrap-around adjacent and must be the first access set in the sequence $\tilde{S}$. Note that in the page overhead modeling function $\eta(s, d, c, V)$ the number of vectors V is the number referenced by all streams in S, as it is assumed that all access sets conflict.

Similarly, if $\tilde{S}_i = \langle w_i : \varepsilon_i \rangle$ and the number of streams N is greater than one then $T(\tilde{S}_i)$ is the sum of the maximum number of references at any module ψ_i multiplied by the page-hit write cycle time $T_{p/w}$ and the average page miss count at that module multiplied by the page miss time $T_{p/m}$. For the special case of $N = 1$, the average time to complete all writes is the product of the number of accesses ε_i and the average time per access so that

$$T(\tilde{S}_i) = \begin{cases} \varepsilon_i \left(\dfrac{T_{p/w} + \eta(\xi_i, d_i, 1, V) T_{p/m}}{\mu_i} \right) & \text{when } N = 1 \\ \psi_i T_{p/w} + \eta(\xi_i, d_i, \psi_i, V) T_{p/m} & \text{when } N \geq 2 \end{cases}$$

Finally, if $\tilde{S}_i$ is one of the two possible intermix sequences $\langle \langle r_i : c, w_j : c \rangle : h \rangle$ or $\langle [r_i : \varepsilon_i, w_j : \varepsilon_j \mid \mu_i, \mu_j] \rangle$ then the pattern of reference observed at individual modules is the general intermix sequence $\langle \langle r_i : c, w_j : c \rangle : h \rangle$. Intermix parameters c and h are derived in 5.4.2.2 for the module servicing the maximum number of accesses.

Then $T(\tilde{S}_i)$ is the sum of the maximum number of accesses ε_i (ε_j) at any module multiplied by the sum of the page-hit read and page-hit write cycle times and the sum of the average page miss count for read and write operations at that module multiplied by the page miss time $T_{p/m}$ so that

$$T(\tilde{S}_i) = \psi_i (T_{p/r} + T_{p/w}) + (\eta(\xi_i, d_i, \psi_i, V) + h\rho(\xi_j, d_j, c)) T_{p/m}$$

From the preceding analysis, the time to complete an iteration of the access sequence $\tilde{S}$ is the sum of the times required to complete each component sequence $\tilde{S}_i$; i.e.

$$T_{tot} = \sum_{\tilde{S}_i \in \tilde{S}} T(\tilde{S}_i)$$

Then the average time per data item accessed T_{avg} is the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_{tot}}{\sum_{t_i \in S} \varepsilon_i \gamma_i}$$

The effective memory bandwidth BW is the number of bytes of relevant data transferred per iteration divided by the time to complete all access; i.e.

$$BW = \frac{10^3 \sum_{t_i \in S} \varepsilon_i \gamma_i d_i}{T_{tot}}$$

All times are in nanoseconds and bandwidth is measured in megabytes per second.

5.4.3 Simulation Results

For an interleaved memory system, access ordering can significantly increase effective memory bandwidth over that achieved by the natural sequence of references through better management of concurrency and minimization of page overhead. This is true even for the case when stream alignment is unknown. To illustrate the improvement in performance achieved via access ordering, and to validate performance models, simulation and analytic results are presented for a range of scientific kernels.

5.4.3.1 Uniform-access Components

Results are first presented for a non-buffered 4 module interleaved system of uniform-access components. Module parameters are defined in Table 4, with sizes in bytes and times in nanoseconds. These parameters are typical of commercially available SRAMs.

Table 4 Module Parameters (Interleaved - Uniform)

Parameter	Value
w	8
$T_{u/r}$	50
$T_{u/w}$	50

Table 5 presents simulation results comparing effective memory bandwidth achieved by the natural versus ordered access sequence for the benchmark scientific kernels defined in section 4.4. For access ordering, the depth of loop unrolling is 4. Vectors for all computations are double-precision and aligned to module M_0.

Access ordering improves performance over the natural access sequence for the given computations from 100% to 256%; the exception being LL-24 that references only a single vector.

Table 6 compares performance of ordered accesses as calculated analytically and measured via simulation. Recall that analytic results represent a lower bound. For the computations and conditions modeled, analytic results accurately predict performance; however, this is not necessarily the case.

Table 5 Natural vs Ordered Performance (Interleaved - Uniform)

Computation	Natural BW	Ordered BW	% Increase
daxpy	239.8	640.0	166.9
dvaxpy	213.2	640.0	200.2
LL-1	239.8	640.0	166.9
LL-3	319.4	640.0	100.4
LL-4	319.4	640.0	100.4
LL-5	239.8	640.0	166.9
LL-7	213.2	640.0	200.2
LL-11	319.4	640.0	100.4
LL-12	319.4	640.0	100.4
LL-20	180.0	640.0	255.6
LL-21	239.8	640.0	166.9
LL-22	199.9	640.0	220.2
LL-24	640.0	640.0	0.0

5.4.3.2 Page-mode Components

Simulation results are presented for a non-buffered 2 module interleaved system of page-mode components. Module parameters are defined in Table 7 and are representative of the IPSC/860 node memory system.

Table 8 presents simulation results comparing effective memory bandwidth achieved by the natural versus ordered access sequence for the set of benchmark kernels. Depth of loop unrolling is 4, data is double-precision, and all vectors are aligned to module M_0.

Table 6 Analytic vs Simulation Results (Interleaved - Both)

Computation	Uniform-access		Page-mode	
	Analysis BW	Simulation BW	Analysis BW	Simulation BW
daxpy	640.0	640.0	127.9	127.9
dvaxpy	640.0	640.0	121.8	121.8
LL-1	640.0	640.0	100.9	100.9
LL-3	640.0	640.0	106.5	106.5
LL-4	640.0	640.0	106.3	106.1
LL-5	640.0	640.0	100.9	101.0
LL-7	640.0	640.0	102.3	102.3
LL-11	640.0	640.0	98.3	98.3
LL-12	640.0	640.0	98.3	98.3
LL-20	640.0	640.0	102.7	102.7
LL-21	640.0	640.0	124.7	123.4
LL-22	640.0	640.0	99.9	99.9
LL-24	640.0	640.0	317.5	316.9

For this system, access ordering improves performance over the natural access sequence for the given computations from 60% to 189%. Once again LL-24 is the exception as only a single vector is referenced.

Table 6 compares effective memory bandwidth for ordered accesses as calculated analytically and measured via simulation. Once again, analytic results represent a lower bound on performance. However, for the computations and conditions modeled, analytic and simulation results differ by less than 1%. Note that as a result of start-up transients in the

Table 7 Module Parameters (Interleaved - Page)

Parameter	Value
w	8
p	4096
$T_{p/r}$	50
$T_{p/w}$	75
$T_{p/m}$	200

simulation, measured performance falls below the theoretical lower bound for several computations.

5.4.4 Summary

Section 5.4 develops access ordering algorithms for a interleaved system of uniform-access and page-mode components under the assumption that stream alignment is unknown. Performance predictors are derived for the effective memory bandwidth achieved by ordered accesses.

For a system of uniform-access components, access ordering attempts to maximize module concurrency. The algorithm divides references into two phases: a read phase and a write phase. Accesses from each phase are ordered to maximize concurrency for individual streams and increase the likelihood of concurrency among accesses from different streams. Ordering is trivial, with a time complexity linear in the number of accesses. Performance predictors assume that accesses from different streams can not be serviced concurrently, and thus represent a lower bound.

For a system of page-mode components, the access ordering algorithm results in a sequence consisting of (potentially intermixed) access sets arranged to maximize reduction in page overhead achievable via intermixing and wrap-around adjacency. No attempt

Table 8 Natural vs Ordered Performance (Interleaved - Page)

Computation	Natural BW	Ordered BW	% Increase
daxpy	48.0	127.9	166.5
dvaxpy	42.7	121.8	185.2
LL-1	48.0	100.9	110.2
LL-3	63.9	106.5	66.7
LL-4	63.9	106.1	66.0
LL-5	48.0	101.0	110.4
LL-7	42.7	102.3	139.6
LL-11	60.9	98.3	61.4
LL-12	60.9	98.3	61.4
LL-20	35.6	102.7	188.5
LL-21	77.3	123.4	59.6
LL-22	39.0	99.9	156.2
LL-24	315.1	316.9	0.6

is made to increase potential concurrency. The access ordering algorithm has a time complexity exponential in the number of streams. Again performance predictors assume no concurrency between access from different streams and thus represent a lower bound.

Simulation results are presented for interleaved systems of both uniform-access and page-mode components. Access ordering is shown to significantly increase effective memory bandwidth over that achieved by the natural sequence of reference for the set of benchmark kernels. Performance models are validated.

Recall that modules in an interleaved system may be buffered, as depicted in Figure 8. Buffering potentially improves performance by allowing accesses from nonconflicting streams to be initiated under conditions that would otherwise result in the processor blocking on a busy module; i.e. buffering may increase concurrency among accesses from different streams. The effect of buffering on reference sequences generated by the ordering algorithms presented above is not studied here.

5.5 Access Ordering Algorithms for Known Alignments

For a sequentially interleaved memory system, access ordering algorithms and performance predictors are derived based on the assumption that stream alignments are known at compile time. In this context, stream alignment refers to the module that services the first access from a given stream. Note that if relative alignment is known, one stream can be assumed aligned to a specific module with the remaining streams aligned appropriately; relative alignment is sufficient to completely define module contention between accesses from different streams. For a system of page-mode components, no assumption is made concerning stream alignment with respect to pages.

Section 5.5.1 presents results for the optimal access of independent streams used in the general ordering algorithms. Sections 5.5.2 and 5.5.3 derive access ordering algorithms and performance predictors for systems of uniform-access and page-mode components, respectively. The effectiveness of access ordering and accuracy of performance models are demonstrated via simulation in section 5.5.4. Section 5.5.5 summarizes results.

5.5.1 Optimal Access of Independent Streams

Given a set *S* of independent streams, knowledge of stream alignment allows for the specification of an access sequence that results in optimal effective memory bandwidth. A methodology for generating such a sequence is presented below.

To derive an optimal access sequence, the depth of loop unrolling *b*, chosen in mapping logical streams $S^{(L)}$ to physical streams *S*, is restricted to values such that on each successive loop iteration the first access from each stream references the same module as the first access from the previous iteration. Restricting *b* in this manner guarantees a repetitive sequence of module references per stream per loop iteration.

Lemma 5.4: For stream $t_i \in S$, if the number of accesses per iteration ε_i is a multiple of the number of modules referenced μ_i, i.e. $\mu_i \mid \varepsilon_i$, then on each successive loop iteration t_i references exactly the same set of modules in exactly the same sequence, with each module servicing exactly ψ_i accesses.

Proof: By Lemma 5.1, stream t_i references μ_i modules in a sequence with period μ_i. Therefore, if $\mu_i \mid \varepsilon_i$ then each set of ε_i accesses must reference exactly the same set of μ_i modules in exactly the same sequence with $\psi_i = \varepsilon_i / \mu_i$ accesses per module. □

Loop Unrolling Restriction: For a set *S* of independent streams, to derive an optimal access sequence the depth of loop unrolling *b*, chosen in forming *S*, is restricted to values such that for all $t_i \in S$, $\mu_i \mid \varepsilon_i$.

For most scientific codes, the number of accesses per iteration from a given stream equals the depth of loop unrolling. Given a set of streams *S*, if $\varepsilon_i = b$ for all $t_i \in S$ then *b* is restricted to a multiple of the maximum number of modules accessed by any stream; i.e. $b = k\,(max(\mu_i)) \leq km$ for all $t_i \in S$ and $k \in Z^+$.

In the context of scalar microprocessor systems, the number of modules in an interleaved memory is expected to be modest. Thus while the loop unrolling restriction potentially results in a large value of *b*, for most codes this is not the case.

For *N* independent streams *S* and a loop depth *b* satisfying the loop unrolling restriction, an optimal access sequence is derived as follows. Consider the mapping of stream

accesses to modules that results from a single loop iteration when all accesses from each stream are initiated consecutively, as in the sequence

$$\langle a_1 : \varepsilon_1, \ldots, a_N : \varepsilon_N \rangle$$

At each module $M_0, \ldots, M_{m-1}$, the relative sequence of accesses serviced can be represented by $A_0, \ldots, A_{m-1}$, respectively. Sequences $A_0, \ldots, A_{m-1}$ are relative in the sense that the order in which stream accesses are serviced is specified, not the particular stream accesses in a given loop iteration. For example, $A_0 = \langle a_1 : \psi_1, a_3 : \psi_3, a_4 : \psi_4 \rangle$ specifies that module M_0 satisfies ψ_1 accesses from stream t_1, followed by ψ_3 accesses from stream t_3 followed by ψ_4 accesses from stream t_4; the specific accesses serviced from each of the three steams, e.g. a_1^k, is alignment dependent. Note that $A_0, \ldots, A_{m-1}$ are constant for all iterations as a result of the loop unrolling restriction.

Figure 12 presents the Module Sequence Algorithm (MSA) for defining the sequences $A_0, \ldots, A_{m-1}$ that result from a consecutive access sequence. The algorithm defines $A_0, \ldots, A_{m-1}$ by mapping streams in decreasing order of number of modules accessed; i.e. t_i is mapped prior to t_j if $\mu_i > \mu_j$.

Lemma 5.5: Given streams S and sequences $A_0, \ldots, A_{m-1}$ derived via the Module Sequence algorithm, each round robin selection of accesses from $A_0, \ldots, A_{m-1}$, i.e. the set of accesses formed by taking the 'next' access from each sequence $A_0, \ldots, A_{m-1}$, has the property that for each stream t_i referenced: there are exactly μ_i accesses from t_i, and accesses from t_i do not conflict, i.e. do not reference a module referenced by any other access in the set.

Proof: Located in Appendix B.1. □

$A_0 = \ldots = A_{m-1} = \varnothing$

```
// for each stream t_i in the set of streams S selected
// in decreasing order of number of modules referenced.

for all t_i ∈ S selected in decreasing order of μ_i

   // for each module M_j accessed by t_i

   for all M_j ∈ Z_i

      // concatenate accesses from t_i to the sequence A_j

      A_j ← ⟨A_j, ⟨a_i: ψ_i⟩⟩
```

Figure 12 Module Sequence Algorithm

Theorem 5.6: Given a set S of independent streams and sequences $A_0, \ldots, A_{m-1}$ derived via the Module Sequence algorithm, $\tilde{S} = \langle [A_0, \ldots, A_{m-1}] \rangle$ is an optimal access sequence.

Proof: Accesses from $A_0, \ldots, A_{m-1}$ are initiated round robin. By Lemma 5.5, for each round robin sequence of accesses it is observed that

- for each stream t_i referenced exactly μ_i accesses are initiated, maximizing concurrency for stream t_i,
- accesses from a stream t_i do not conflict, maximizing concurrency between accesses from different streams, and
- the total number of accesses is equal to the number of modules that service the remaining accesses, maximizing module utilization.

Furthermore, for a given loop iteration, accesses from a stream t_i are serviced consecutively at each module referenced, minimizing page overhead when applicable. □

To illustrate, an optimal access sequence is derived for a set of three read streams $S = \{t_x, t_y, t_z\}$. For each stream data size equals word size, stride of access is 2 and the number of accesses per iteration is equal to the depth of loop unrolling; i.e. $\varepsilon_x = \varepsilon_y = \varepsilon_z = b$. Assume a 4 module interleaved system with stream t_x aligned to module M_3, and streams t_y and t_z aligned to module M_0. Then each stream accesses 2 modules, so that by the loop unrolling restriction b is a multiple of 2.

For $b = 2$, assume the MSA defines the following sequences: $A_0 = \langle r_y, r_z \rangle$, $A_1 = \langle r_x \rangle$, $A_2 = \langle r_y, r_z \rangle$ and $A_3 = \langle r_x \rangle$. The resulting optimal access sequence

$$\tilde{S} = \langle [A_0, A_1, A_2, A_3] \rangle = \langle [\langle r_y, r_z \rangle, \langle r_x \rangle, \langle r_y, r_z \rangle, \langle r_x \rangle] \rangle$$

defines the linear sequence of references

$$\langle r_y(A_0, M_0), r_x(A_1, M_3), r_y(A_2, M_2), r_x(A_3, M_1), r_z(A_0, M_0), r_z(A_2, M_2) \rangle$$

The above sequence is annotated to illustrate both the round robin selection of accesses and the specific mapping of accesses to modules as determined by alignment; e.g. $r_x(A_1, M_3)$ specifies r_x chosen from sequence A_1 generates a reference to module M_3. Note that in the general case of mapping $\langle [A_0, \ldots, A_{m-1}] \rangle$ to a linear sequence of references, a particular access a_i selected from a relative sequence A_k does not necessarily specify a reference to module M_k; a_i may in fact specify access to any module in Z_i, the set of all modules referenced by stream t_i. This is demonstrated in the example above for accesses to stream t_x.

5.5.1.1 Request Buffering

For an interleaved system, modules may be buffered as depicted in Figure 8. Ordering accesses as above results in a sequence that references each module at most once per

round robin selection of accesses from $\langle [A_0, \ldots, A_{m-1}] \rangle$. If individual accesses require an equal amount of time to complete, then the sequence $\langle [A_0, \ldots, A_{m-1}] \rangle$ achieves optimal effective memory bandwidth without the need for request buffering. This is the case for a system of uniform-access components and streams of the same mode.

If individual access times vary, then the sequence $\langle [A_0, \ldots, A_{m-1}] \rangle$ provides optimal bandwidth only if buffering is sufficient to eliminate *access gaps* that result in increased completion time for all accesses in a loop. An *access gap* is defined as a period of time during which a module is idle due to the memory system blocking on a busy module. Such is the case for an interleaved system of page-mode components. For this analysis, buffering is assumed sufficient so that the sequence $\langle [A_0, \ldots, A_{m-1}] \rangle$ results in optimal performance.

5.5.2 Interleaved Storage and Uniform-access Components

For an interleaved system of uniform-access components, an access ordering algorithm need only maximize module concurrency. Unfortunately, in the presence of dependencies, determining an access sequence that maximizes concurrency is NP-complete with a time complexity exponential in the number of accesses; this result is obtained by restriction to precedence constrained scheduling [GaJo79]. As an optimal solution is intractable, a heuristic solution is presented below.

In defining an access sequence for streams S, accesses are performed in two phases: a read phase and a write phase. By the stream interaction restriction, streams associated with each phase are independent. Thus, an optimal access sequence can be derived for each phase based on the results of section 5.5.1.

For streams S, S_r is defined as the subset of all read streams and S_w the subset of all write streams; loop depth b is assumed to satisfy the loop unrolling restriction defined in 5.5.1.

Sequences $P_0, \ldots, P_{m-1}$ and $Q_0, \ldots, Q_{m-1}$ are defined by the MSA for S_r and S_w, respectively. Then the access sequence employed is

$$\tilde{S} = \langle [P_0, \ldots, P_{m-1}], [Q_0, \ldots, Q_{m-1}] \rangle$$

In the above sequence, accesses associated with each phase are ordered to maximize concurrency, resulting in optimal effective memory bandwidth for that phase. However, the aggregate solution is likely suboptimal as potential concurrency among read and write accesses is not exploited. Dependencies are maintained as all read accesses are performed prior to any writes.

5.5.2.1 Performance Predictor

For a set of streams S and an access sequence $\tilde{S}$ as defined above, a performance predictor is derived for the average time per data item accessed T_{avg} and effective processor-memory bandwidth BW.

Let T_r define the time required to complete all read accesses for a given loop iteration. From the sequence $\tilde{S}$, $P_0, \ldots, P_{m-1}$ represent the relative sequences of read operations serviced at modules $M_0, \ldots, M_{m-1}$ respectively. As accesses proceed concurrently at all modules, the time to complete all reads is equal to the time to complete accesses at the module servicing the greatest number of reads. Let $|P_i|$ define the number of read operations in the sequence P_i. Then T_r is the maximum number of accesses at any module multiplied by the uniform-access read cycle time $T_{u/r}$; i.e.

$$T_r = max(|P_0|, \ldots, |P_{m-1}|) T_{u/r}$$

T_w is defined as the time to complete all write operations for a given iteration and is computed analogously to T_r, so that

$$T_w = max(|Q_0|, \ldots, |Q_{m-1}|)T_{u/w}$$

An upper bound on the time to complete all accesses in a given iteration, and hence a lower bound on performance, is the sum of the time to complete all read and write accesses; i.e.

$$T_{tot} = T_r + T_w$$

Note that T_{tot} is an upper bound as it assumes no concurrency among read and write operations at the boundaries between the read and write phases of the sequence $\tilde{S}$. An exact model of performance can not be expressed as a closed form equation.

From the above, the average time per data item accessed T_{avg} is computed as the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_{tot}}{\sum_{t_i \in S} \varepsilon_i \gamma_i}$$

The effective memory bandwidth BW is the number of bytes of relevant data transferred per iteration divided by the time to complete all access; i.e.

$$BW = \frac{10^3 \sum_{t_i \in S} \varepsilon_i \gamma_i d_i}{T_{tot}}$$

All times are assumed to be in nanoseconds and bandwidth is measured in megabytes per second.

5.5.3 Interleaved Storage and Page-mode Components

For an interleaved system constructed from page-mode components, optimal performance results from an access sequence that balances maximizing concurrency with minimizing page overhead to achieve minimum completion time. Determining such a sequence is NP-complete with a time complexity exponential in the number of accesses; this result is obtained by restriction to precedence constrained scheduling [GaJo79]. As an optimal solution is intractable, a heuristic solution analogous to that derived in 5.5.2 is presented below.

In the sections that follow, a base access sequence is first developed for computations that do not specify a read-modify-write. Intermixing and wrap-around adjacency are then employed to reduce page overhead for computations implementing this operation. The general access ordering algorithm is presented and a performance predictor is derived.

5.5.3.1 A Base Access Sequence

In defining an access sequence for streams *S*, accesses are performed in two phases: a read phase and a write phase. As streams associated with each phase are independent, an optimal access sequence can be derived for a phase based on the results of section 5.5.1.

For streams *S*, S_r is defined as the subset of all read streams and S_w the subset of all write streams; loop depth *b* is assumed to satisfy the loop unrolling restriction defined in 5.5.1. Sequences $P_0, \ldots, P_{m-1}$ and $Q_0, \ldots, Q_{m-1}$ are defined by the MSA for S_r and S_w, respectively. Then the base access sequence employed is

$$\tilde{S}_B = \langle [P_0, \ldots, P_{m-1}], [Q_0, \ldots, Q_{m-1}] \rangle$$

In the above sequence, accesses associated with each phase are ordered to maximize concurrency and minimize page overhead. Again, the aggregate solution is likely suboptimal

as potential concurrency among read and write accesses is not exploited. Dependencies are maintained as all read accesses are performed prior to any writes.

5.5.3.2 Intermixing and Wrap-around Adjacency

For streams *S* implementing a read-modify-write, intermixing and wrap-around adjacency may reduce page overhead in each phase of the base sequence $\tilde{S}_B$, potentially reducing completion time for all accesses. Note that in this context, intermixing refers to read accesses immediately preceding corresponding write accesses at a given module; read and write operations are not interleaved so that accesses associated with each phase remain separate.

In deriving the base sequence $\tilde{S}_B$, sequences P_0, …, P_{m-1} and Q_0, …, Q_{m-1} are defined via the MSA by mapping streams in decreasing order of number of modules referenced. Intermixing and wrap-around adjacency are employed by choosing a legal mapping order such that one or more pair of streams benefits from these relationships.

5.5.3.3 Access Ordering Algorithm

For a set of stream *S* with no pair of streams implementing a read-modify-write, the access sequence employed is the base sequence $\tilde{S}_B$; i.e. $\tilde{S} = \tilde{S}_B$. Access within each phase of $\tilde{S}$ can be mapped in any order that satisfies the requirements of the MSA.

If *S* contains one or more pairs of streams implementing a read-modify-write, then an access sequence $\tilde{S}$ in the form of the base sequence $\tilde{S}_B$ is derived as follows:

> Given streams *S* with read streams S_r and write streams S_w, determine the legal order for mapping elements of S_r and S_w to form P_0, …, P_{m-1} and Q_0, …, Q_{m-1}, respectively, in the base sequence $\tilde{S}_B$ that results in the minimum completion time for all accesses in a given loop iteration.

Note that for a given ordering of stream mappings, simply computing reduction in page overhead at a particular module is not sufficient as reduction in completion time for all accesses is not guaranteed. Thus for each ordering, the average time to complete an iteration of the sequence must be computed as derived below in section 5.5.3.5.

Determining the order for stream mappings that results in minimum average completion time is exponential in the number of streams in *S*. However, as stated previously for access ordering algorithms with similar time complexity, the stream count *N* tends to be small. Furthermore, the number of legal mappings may be severely restricted by the requirements of the MSA. Finally, page overhead is only affected by the relative mapping order of streams involved in read-modify-writes, again reducing the number of mapping orders that need be considered. The result is an efficient algorithm.

5.5.3.4 Example Problem

The following example illustrates the application of the ordering algorithm defined above. Consider the vaxpy computation

$$\forall i \qquad y_i \leftarrow a_i x_i + y_i$$

that generates the four streams $S = \{t_a, t_x, t_{y_r}, t_{y_w}\}$, where $t_a = (a, s_a, d_a, r) : \varepsilon_a$, $t_x = (x, s_x, d_x, r) : \varepsilon_x$, $t_{y_r} = (y, s_y, d_y, r) : \varepsilon_{y_r}$, and $t_{y_w} = (y, s_y, d_y, w) : \varepsilon_{y_w}$.

For each vector assume data size equals word size and stride of access is defined by $s_a = 1$ and $s_x = s_y = 2$. Assume a 2 module interleaved system with all streams aligned to module M_0. The loop depth *b* is chosen to be 2, so that $\varepsilon_a = \varepsilon_x = \varepsilon_{y_r} = \varepsilon_{y_w} = 2$.

Recall that the MSA maps streams in decreasing order of number of modules accessed. Thus the ordering algorithm considers two legal forms of the base sequence $\tilde{S}_B$:

- $\langle [\langle r_a, r_x, r_x, r_{y_r}, r_{y_r} \rangle, \langle r_a \rangle], [\langle w_{y_w}, w_{y_w} \rangle, \langle\ \rangle] \rangle$ and
- $\langle [\langle r_a, r_{y_r}, r_{y_r}, r_x, r_x \rangle, \langle r_a \rangle], [\langle w_{y_w}, w_{y_w} \rangle, \langle\ \rangle] \rangle$.

In the first sequence write accesses benefit from intermixing, as the corresponding read accesses immediately precede at module M_0. In the second sequence intermixing is not exploited. Note that wrap-around adjacency can not occur as accesses to stream t_a must be initiated first at both modules, since $\mu_a = 2$ and $\mu_x = \mu_{y_r} = \mu_{y_w} = 1$.

Thus the access ordering algorithm for the vaxpy computation results in the first of the two sequences listed above, generating the corresponding linear sequence of references

$$\tilde{S} = \langle r_a, r_a, r_x, r_x, r_{y_r}, r_{y_r}, w_{y_w}, w_{y_w} \rangle$$

5.5.3.5 Performance Predictor

For a set of streams S and an access sequence $\tilde{S}$ defined as above, a performance predictor is derived for the average time per data item accessed T_{avg} and the processor-memory bandwidth BW.

Functions modeling page overhead derived in chapter 4 for a single module system are applicable to accesses at individual modules of an interleaved system. Recall that in general, average page miss count is modeled by the function $\eta(s, d, c, V)$. For stream accesses that are wrap-around adjacent or intermixed, average page miss count is modeled by the functions $\omega(s, d, c)$ and $\rho(s, d, c)$ respectively. In employing these functions for an interleaved system, stride s is module stride and the number of accesses c is the number at each module; i.e. for a stream t_i, $s = \xi_i$ and $c = \psi_i$.

In the sequence $\tilde{S}$, $P_0, \ldots, P_{m-1}$ represent the sequences of read accesses serviced at modules $M_0, \ldots, M_{m-1}$. Each P_k serviced at module M_k is composed of some number of component sequences $P_{(i,k)}$, where the first subscript i is defined to be that of the stream referenced. Thus, $P_{(i,k)}$ represents the read access set $\langle r_i : \psi_i \rangle$. Similarly, Q_k is the sequence of write accesses serviced at M_k and $Q_{(i,k)}$ represents the write access set $\langle w_i : \psi_i \rangle$.

The time required to complete all accesses in the sequence $P_{(i,k)}$ is the sum of the number of accesses ψ_i multiplied by the page-hit read cycle time $T_{p/r}$ and the average page miss count multiplied by the page miss time $T_{p/m}$; i.e.

$$T(P_{(i,k)}) = \psi_i T_{p/r} + \begin{cases} \omega_k(\xi_i, d_i, \psi_i) T_{p/m} & \text{when } P_{(i,k)} \text{ is wrap-around adjacent} \\ \eta_k(\xi_i, d_i, \psi_i, V) T_{p/m} & \text{otherwise} \end{cases}$$

Note that in modeling page miss count, conditions that determine appropriate use of modeling functions *must be applied in the context of the module accessed.* $P_{(i,k)}$ is wrap-around adjacent if there exists a $Q_{(j,k)}$ such that read stream t_i and write stream t_j implement a read-modify-write, $P_{(i,k)}$ is the first access set in P_k and $Q_{(j,k)}$ is the last access set in Q_k; then $\omega_k(\xi_i, d_i, \psi_i)$ correctly models page overhead. Otherwise, $\eta_k(\xi_i, d_i, \psi_i, V)$ is the applicable model where the number of vectors V is the number accessed at module M_k. For clarity, functions modeling page overhead are subscripted with the module number to denote context.

Similarly, the time required to complete all accesses in the sequence $Q_{(i,k)}$ is the sum of the number of accesses ψ_i multiplied by the page-hit write cycle time $T_{p/w}$ and the average page miss count multiplied by the page miss time $T_{p/m}$, so that

$$T(Q_{(i,k)}) = \psi_i T_{p/w} + \begin{cases} \rho_k(\xi_i, d_i, \psi_i) T_{p/m} & \text{when } Q_{(i,k)} \text{ is intermixed} \\ \eta_k(\xi_i, d_i, \psi_i, V) T_{p/m} & \text{otherwise} \end{cases}$$

In this context, $Q_{(i,k)}$ is intermixed if there exists a $P_{(g,k)}$ such that read stream t_g and write stream t_i implement a read-modify-write, $P_{(g,k)}$ is the last access set in P_k and $Q_{(i,k)}$ is the first access set in Q_k.

From the preceding analysis, the time to complete all read operations in the sequence P_k is the sum of the time to complete all accesses in each component sequence; i.e.

$$T(P_k) = \sum_{P_{(i,k)} \in P_k} T(P_{(i,k)})$$

Then the time to complete all read accesses in an iteration of the sequence $\tilde{S}$ is the maximum time to complete read operations at any module, so that

$$T_r = max(T(P_0), \ldots, T(P_{m-1}))$$

Similarly, the time to complete all write operations in the sequence Q_k is the sum of the time to complete all accesses in each component sequence; i.e.

$$T(Q_k) = \sum_{Q_{(i,k)} \in Q_k} T(Q_{(i,k)})$$

And the time to complete all write operations in an iteration of the sequence $\tilde{S}$ is

$$T_w = max(T(Q_0), \ldots, T(Q_{m-1}))$$

Note the tacit assumption in computing T_r and T_w is that buffering is sufficient so that each phase of the sequence proceeds without access gaps that result in increased completion time for that phase; this is discussed in section 5.5.1.1.

An upper bound on the time to complete all accesses in a given iteration, and hence a lower bound on performance, is the sum of the time to complete all read and write accesses so that

$$T_{tot} = T_r + T_w$$

T_{tot} is an upper bound as it assumes no concurrency among read and write operations at the boundaries between the read and write phases. An exact model of performance can not be expressed as a closed form equation. Note that T_{tot} is the value used by the access ordering algorithm in determining the order for stream mappings that results in minimum completion time.

From the above, the average time per data item accessed T_{avg} is computed as the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_{tot}}{\sum_{t_i \in S} \varepsilon_i \gamma_i}$$

The effective memory bandwidth BW, in megabytes per second, is the number of bytes of relevant data transferred per iteration divided by the time to complete all access; i.e.

$$BW = \frac{10^3 \sum_{t_i \in S} \varepsilon_i \gamma_i d_i}{T_{tot}}$$

5.5.4 Simulation Results

For scientific kernels previously simulated, vector strides are such that all *m* modules in a sequentially interleaved system are referenced by each stream for any $m = 2^n$. Thus ordering algorithms do not benefit from alignment information as all streams are conflicting. Simulation and analytic results for algorithms derived under the assumption of known alignment are identical to those presented in section 5.4.3 for algorithms that assume alignment is unknown.

Consider again the vaxpy computation

$$\forall i \qquad y_i \leftarrow a_i x_i + y_i$$

that generates the four streams $S = \{t_a, t_x, t_{y_r}, t_{y_w}\}$, where $t_a = (a, s_a, d_a, r) : \varepsilon_a$, $t_x = (x, s_x, d_x, r) : \varepsilon_x$, $t_{y_r} = (y, s_y, d_y, r) : \varepsilon_{y_r}$, and $t_{y_w} = (y, s_y, d_y, w) : \varepsilon_{y_w}$.

For each vector assume data size equals word size, thus $\varepsilon_a = \varepsilon_x = \varepsilon_{y_r} = \varepsilon_{y_w} = b$, and stride of access is defined by $s_a = 1$ and $s_x = s_y = 2$. Assume a non-buffered 4 module system of page-mode components with module parameters as previously defined in Table 7. Streams t_a and t_x are aligned to module M_0 and streams t_{y_r} and t_{y_w} are aligned to module M_1.

Table 9 presents simulation and analytic results comparing performance of the vaxpy computation ordered under the assumptions of known and unknown alignment for a range of loop depths *b*. Assuming known alignment, access ordering improves performance over the natural reference sequence from 96% to 216%; under the assumption of unknown alignment performance is improved from 49% to 139%. Note that for unknown alignment the performance predictor is below the effective bandwidth achieved, as all streams are incorrectly assumed to be conflicting.

For this example knowledge of stream alignment allows accesses from nonconflicting streams to be scheduled to proceed concurrently, resulting in increased performance over the case where alignment is unknown.

Table 9 Simulation and Analytic Results (Interleaved - Page)

Algorithm	*b*	Simulation		Analysis	% Increase
		Natural *BW*	Ordered *BW*	Ordered *BW*	
Unknown Alignment	4	93.0	138.3	127.9	48.7
	8	93.0	192.9	182.5	107.4
	12	93.0	222.0	212.9	138.7
Known Alignment	4	93.0	182.4	182.5	96.1
	8	93.0	254.9	255.0	174.1
	12	93.0	293.5	293.9	215.6

5.5.5 Summary

Section 5.5 develops access ordering algorithms for an interleaved system of uniform-access and page-mode components under the assumption that alignment is known. Performance predictors are derived for the effective memory bandwidth achieved by ordered accesses.

For a system of uniform-access components, the ordering algorithm divides accesses into two phases: a read phase and a write phase. Accesses associated with each phase are ordered to maximize concurrency, resulting in optimal effective memory bandwidth for that phase. The aggregate solution is likely suboptimal, as potential concurrency among read and write accesses is not exploited. Ordering is trivial with a time complexity of $O(N\,(lg(N))\,)$ where N is the number of streams, representing the implied sort in the

MSA. Performance predictors assume no concurrency at the boundaries between read and write phases and thus represent a lower bound.

For a system of page-mode components, ordering is performed analogous to the uniform-access case. However intermixing and wrap-around adjacency are employed to reduce page overhead in each phase, potentially reducing completion time for all accesses. The ordering algorithm has a time complexity exponential in the number of streams.

Recall that modules in an interleaved system may be buffered, as depicted in Figure 8. The tacit assumption for systems of page-mode components is that buffering is sufficient so that each phase of the sequence proceeds without access gaps that result in increased completion time for that phase; this is discussed in section 5.5.1.1. If buffering is not sufficient, performance is degraded and performance predictors are no longer guaranteed to represent a lower bound.

6 Multicopy Architecture

This chapter derives access ordering algorithms and performance predictors for a multicopy memory system as depicted in Figure 13. A *multicopy* memory is proposed here as a parallel memory system consisting of *m* modules of replicated data such that if $*(M_k, a)$ represents the contents of address *a* at module M_k, then $*(M_0, a) = \ldots = *(M_{m-1}, a)$.

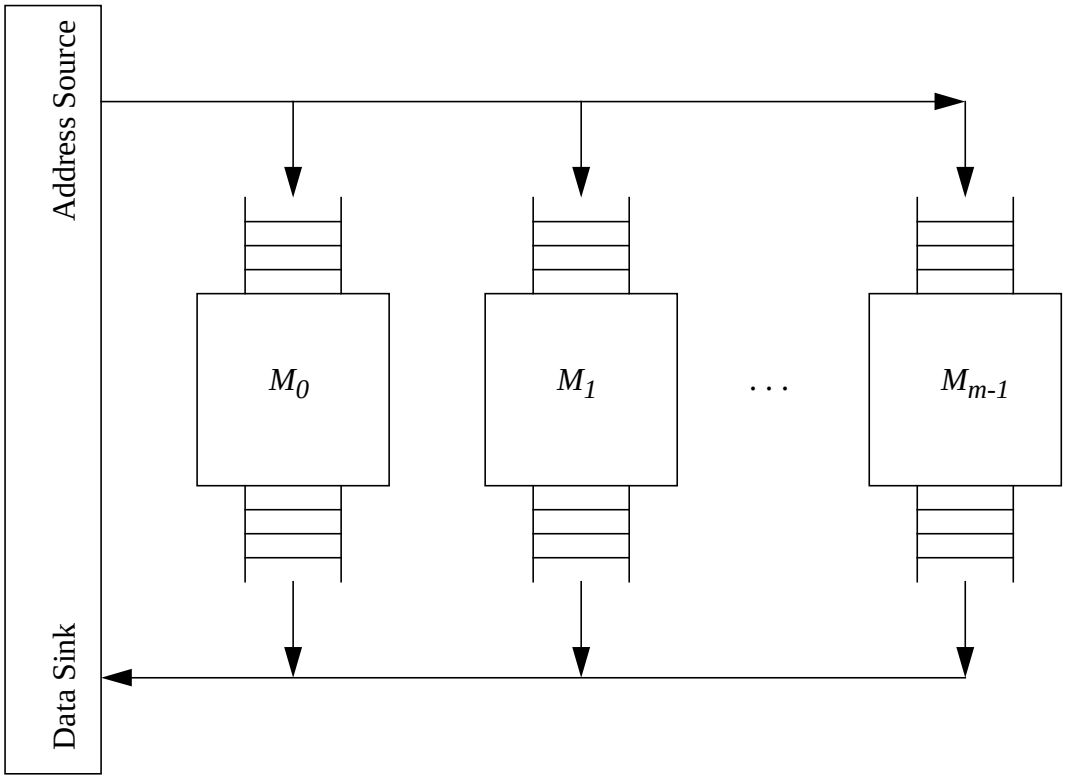

Figure 13 Multicopy Architecture

The multicopy architecture is defined to function as follows. Read accesses specify the module to which the request is to be directed. If input buffer space is available then the request is queued at the appropriate module, otherwise the memory system blocks until a buffer slot is freed. Write accesses are broadcast to all modules to maintain consistency among copies. If the input buffer is full at one or more modules, the memory system blocks until the appropriate buffer slots are freed; all writes are queued simultaneously.

Access requests are serviced at a module in the order queued, with data from read requests placed in the module's output buffer.

Recall that for parallel memory systems it is assumed read accesses are tagged and that data may be returned in the requested order at the rate satisfied. In modeling maximum effective bandwidth, the request rate is assumed sufficient such that performance is limited by the memory. These assumptions are identical to those for the sequentially interleaved architecture analyzed in chapter 5.

A multicopy memory system increases the potential for read access concurrency, as maximum concurrency is achievable for all strides of reference. Furthermore, for systems of page-mode components, read stream page overhead can be more effectively amortized by directing stream accesses to a smaller number of modules. However write operations must be broadcast to maintain coherence, serializing an otherwise parallel operation. Thus it is intuitive that the relative performance of a multicopy system is dependent on a high read to write ratio; simulation results verify this to be the case.

Section 6.1 discusses the problem space for efficient utilization of a multicopy memory. Notation is developed in section 6.2 for expressing the mapping of read accesses to modules. Sections 6.3 and 6.4 derive access ordering algorithms and performance predictors for a multicopy system of uniform-access and page-mode components, respectively. The effectiveness of a multicopy architecture and accuracy of performance models are demonstrated via simulation in section 6.5. Section 6.6 summarizes results.

6.1 Problem Dimensions

In general, to efficiently utilize a multicopy system, stream accesses must be ordered to

- maximize concurrency and
- minimize page overhead, when applicable.

Ordering reads to maximize concurrency is a matter of distributing accesses uniformly across modules. Write accesses are broadcast to all modules so that concurrency is not an issue.

Techniques for minimizing page overhead come directly from analytic results derived in chapter 4 for a single memory module. Page miss count for a given stream is minimized if elements of that stream are referenced consecutively from a single module on each loop iteration. For two streams that implement a read-modify-write, page miss count may further be reduced via intermixing and wrap-around adjacency.

Optimal effective memory bandwidth results from an access sequence that minimizes completion time for all accesses in a loop. As with a sequentially interleaved memory, such a sequence usually requires a trade-off between minimizing page overhead and maximizing concurrency.

To illustrate, consider mapping onto a 2 module system accesses from the three read streams t_x, t_y and t_z. Assume all streams are stride 1 with 4 accesses per stream per iteration. Figure 14 demonstrates the time to complete a typical loop iteration for three different mappings of accesses to modules given that an access to the current page requires 1 time unit $(T_{p/r})$ and a page miss incurs a 3 time unit penalty $(T_{p/m})$. Figure 14(a) depicts a mapping that results in the minimum page miss count, with all accesses from a given stream serviced by a single module. Figure 14(b) depicts a mapping that maximizes concurrency for a given stream by distributing accesses evenly across all modules. Finally, Figure 14(c) depicts an optimal solution that balances minimizing page overhead and maximizing concurrency.

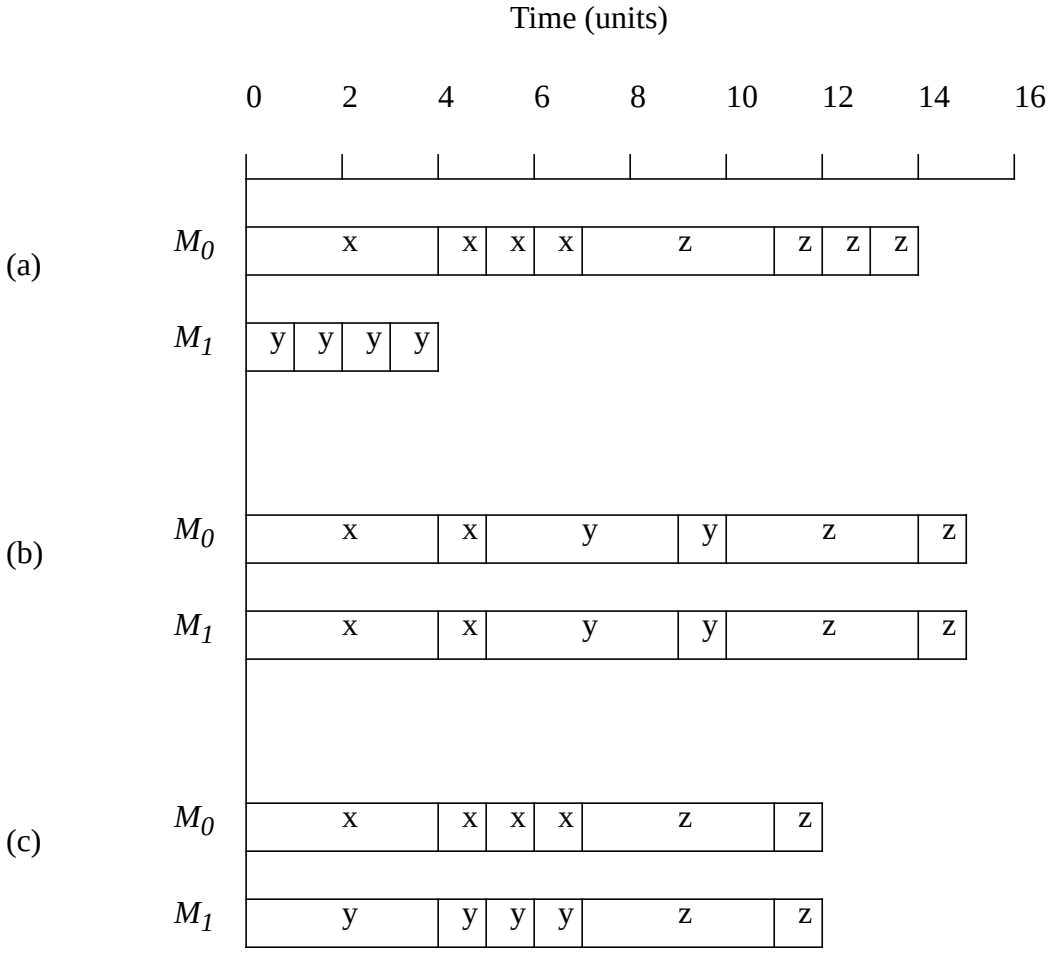

Figure 14 Minimizing Completion Time

6.2 Module Access Notation

To facilitate the specification of an access sequence that maps read accesses to specific modules, the MAP notation developed in section 3.1 is augmented as follows. For individual read accesses, $r_{(i, M_k)}$ denotes access to the next element of stream t_i from module M_k. This notation augments the previous definition of r_i with the specification of the module to which the access is directed.

6.3 Multicopy Storage and Uniform-access Components

Deriving an optimal access ordering algorithm for a multicopy system of uniform-access components is trivial. Concurrency is maximized and dependence maintained by distributing read accesses uniformly across all modules and initiating all reads prior to the first write.

For streams *S*, let t_1 through t_{N_r} be read streams and t_{N_r+1} through t_N be write streams. Let *R* be the total number of read accesses, so that

$$R = \sum_{\substack{t_i \in S \\ m_i = r}} \varepsilon_i$$

Then *m* sequences $A_0, \ldots, A_{m-1}$ are defined such that the *R* accesses are evenly distributed among the sequences. That is, $m - (R \bmod m)$ of the sequences contain $\lfloor R/m \rfloor$ reads, with the remaining $(R \bmod m)$ sequences containing $\lfloor R/m \rfloor + 1$ read accesses. Furthermore, accesses in sequence A_k are tagged for service at module M_k.

Then an optimal access sequence is

$$\tilde{S} = \langle [A_0, \ldots, A_{m-1}], w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N \rangle$$

The above sequence maximizes concurrency among read accesses while maintaining dependencies. Note that module buffering is not required to achieve optimal bandwidth, as read access times are uniform and read requests are initiated across modules in a strict round-robin sequence.

6.3.1 Performance Predictor

For streams S and an access sequence $\tilde{S}$ defined as above, a performance predictor is derived for the average time per data item accessed T_{avg} and processor-memory bandwidth BW.

Let T_r be the time required to complete all read accesses. Then T_r is the time to complete accesses at the module servicing the greatest number of requests, that is

$$T_r = \begin{cases} \left\lfloor \frac{R}{m} \right\rfloor T_{u/r} & \text{when} \quad R \bmod m = 0 \\ \left(\left\lfloor \frac{R}{m} \right\rfloor + 1 \right) T_{u/r} & \text{when} \quad R \bmod m \geq 1 \end{cases}$$

$$= \left\lceil \frac{R}{m} \right\rceil T_{u/r}$$

Similarly, let T_w be the time to complete all write accesses. By definition every write request generates a memory access that is serviced at all modules, so that

$$T_w = \sum_{\substack{t_i \in S \\ m_i = w}} \varepsilon_i T_{u/w}$$

Then the average time per data item accessed T_{avg} is the time to complete all accesses in a given iteration divided by the number of data items referenced, i.e.

$$T_{avg} = \frac{T_r + T_w}{\sum_{t_i \in S} \varepsilon_i \gamma_i}$$

And the effective memory bandwidth BW is the number of bytes of relevant data transferred per iteration divided by the time to complete all accesses, so that

$$BW = \frac{10^3 \sum_{t_i \in S} \varepsilon_i \gamma_i d_i}{T_r + T_w}$$

All times are in nanoseconds and sizes in bytes, with bandwidth measured in megabytes per second.

6.4 Multicopy Storage and Page-mode Components

For a multicopy system of page-mode components, optimal performance results from a sequence that balances maximizing concurrency with minimizing page overhead to achieve minimum completion time. Determining such a sequence is NP-complete with a time complexity exponential in the number of accesses; this result is obtained by restriction to multiprocessor scheduling [GaJo79]. As an optimal solution is intractable, a heuristic solution is presented below.

In the sections that follow, a base access sequence and module reference model are developed. Intermixing and wrap-around adjacency are then discussed for computations implementing a read-modify-write. A heuristic is developed that determines the order and mapping for read operations. Finally, a general ordering algorithm is presented and a performance predictor derived.

6.4.1 A Base Access Sequence

For streams *S*, let t_1 through t_{N_r} be read streams and t_{N_r+1} through t_N be write streams. Then the base access sequence employed is

$$\tilde{S}_B = \langle [A_0, \ldots, A_{m-1}], w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N \rangle$$

In the above, the sequences $A_0, \ldots, A_{m-1}$ specify read operations for streams t_1 through t_{N_r} where accesses in A_k are directed to module M_k. The read sequences $A_0, \ldots, A_{m-1}$ are defined by a mapping heuristic that attempts to minimize completion time. Write accesses are grouped by stream to minimize page overhead; recall that writes are broadcast so that concurrency is not an issue. Write accesses follow reads, maintaining dependence relations. Intermixing and wrap-around adjacency are employed at the boundary of read and write accesses in a greedy fashion.

6.4.1.1 Request Buffering

For a multicopy system, modules may be buffered as depicted in Figure 13. Ordering accesses as above results in a sequence that references each module at most once per round robin selection of accesses $\langle [A_0, \ldots, A_{m-1}] \rangle$. Since individual access times vary, the sequence $\langle [A_0, \ldots, A_{m-1}] \rangle$ provides maximum bandwidth only if buffering is sufficient to eliminate access gaps that result in increased completion time for all accesses in a loop. Recall that an access gap is defined as a period of time during which a module is idle due to the memory system blocking on a busy module. For this analysis, buffering is assumed sufficient so that $\langle [A_0, \ldots, A_{m-1}] \rangle$ results in maximum performance for that sequence.

6.4.2 A Module Reference Model

For $A_0, \ldots, A_{m-1}$ defined in the base access sequence $\tilde{S}_B$, assume that references from each read stream $t_i \in S$ are distributed uniformly among some number of sequences, and hence modules, $\hat{\mu}_i$. Furthermore, assume all accesses from t_i in a sequence A_k are arranged consecutively. Such a sequence arises from the mapping heuristic derived in section 6.4.4.

Mapping accesses as above minimizes page miss count for references serviced at a given module. However, the absolute page miss count is dependent on the overall pattern of reference.

To illustrate, 4 accesses from a stream t_a and 2 from a stream t_b are mapped to a 2 module system as depicted in Figure 15. For the references of Figure 15(a), elements of stream t_a are accessed alternately from each module so that the observed stride at a given module is $2s_a$. Such a mapping results from the sequence $\langle [\langle r_{(a,0)} : 2\rangle, \langle r_{(a,1)} : 2, r_{(b,1)} : 2\rangle] \rangle$.

Alternatively, references depicted in Figure 15(b) access consecutive elements of t_a at each module so that the observed stride is s_a. Such a mapping results from the sequence $\langle [\langle r_{(a,0)} : 2\rangle, \langle r_{(b,1)} : 2, r_{(a,1)} : 2\rangle] \rangle$.

For accesses from the same stream, page miss count at a given module is a function of the distance between individual references. As demonstrated above, this distance is dependent on the overall pattern of reference and as such can not be expressed as a closed form equation.

For the remainder of this discussion, the observed stride of reference for accesses from a read stream t_i serviced at a given module is modeled as the product of the stream stride and the number of modules referenced; i.e.

$$\hat{\xi}_i = \hat{\mu}_i s_i$$

The module stride $\hat{\xi}_i$ is the stride that results if elements of t_i are accessed alternately from each of the $\hat{\mu}_i$ modules referenced. Thus performance models represent estimated performance rather than bounds.

Note that modules accessed $\hat{\mu}_i$ and module stride $\hat{\xi}_i$ are analogous to parameters derived in chapter 5 for a sequentially interleaved architecture and are represented by the same symbols augmented with a hat (^).

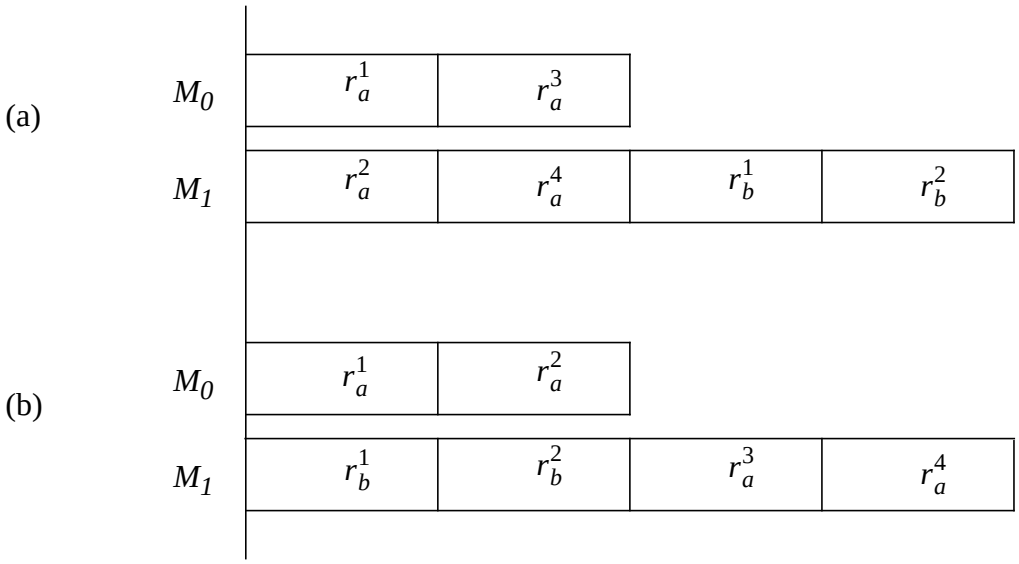

Figure 15 Dependence of Module Stride on Reference Pattern

6.4.3 Greedy Intermixing and Wrap-around Adjacency

For streams *S* implementing a read-modify-write, intermixing and wrap-around adjacency may reduce page miss count in each phase of the base sequence $\tilde{S}_B$, potentially reducing completion time for all accesses. Note that in this context, intermixing refers to read accesses immediately preceding corresponding write accesses at a given module; read and write operations are not interleaved.

Because the base sequence separates accesses by mode and because write accesses are broadcast, at most two pairs of streams may benefit from intermixing and wrap-around adjacency. Furthermore, intermixing generally reduces the time for writes to complete only if corresponding read accesses reference all modules.

Intermixing and wrap-around adjacency are employed in a greedy fashion by choosing prior to access mapping the two pair of streams most likely to benefit from these relation-

ships. From streams *S*, $t_{r\text{-}wadj}$ and $t_{w\text{-}wadj}$ are a pair of read and write streams, respectively, designated to be mapped for wrap-around adjacency. Similarly, $t_{r\text{-}imix}$ and $t_{w\text{-}imix}$ are designated to be mapped for intermixing. All else being equal, streams with the smallest stride have the lowest average page miss count per access and hence the most to gain; $t_{r\text{-}wadj}$ ($t_{w\text{-}wadj}$) and $t_{r\text{-}imix}$ ($t_{w\text{-}imix}$) are chosen accordingly. For *S* consisting of fewer than two read-modify-write operations, $t_{r\text{-}wadj}$ and $t_{r\text{-}imix}$ are chosen in that order, with one or both remaining undefined.

Then in the base sequence $\tilde{S}_B$, the first write sequence $\langle w_{N_r+1}: \varepsilon_{N_r+1} \rangle$ specifies accesses to stream $t_{w\text{-}imix}$ and the last write sequence $\langle w_N: \varepsilon_N \rangle$ specifies accesses to $t_{w\text{-}wadj}$, if defined. Similarly, in defining $A_0, \ldots, A_{m-1}$, the read mapping heuristic insures that accesses to $t_{r\text{-}wadj}$ occur at the beginning of a sequence and accesses to $t_{r\text{-}imix}$ occur at the end, as appropriate.

6.4.4 Read Mapping Heuristic

For read sequences $A_0, \ldots, A_{m-1}$ of the base sequence $\tilde{S}_B$, an optimal mapping of read accesses to modules usually requires a trade-off between minimum page overhead and maximum concurrency as discussed in section 6.1. Minimizing completion time results in a balanced load of accesses such that if $T(A_k)$ is the time to complete accesses in the sequence A_k, then

$$|T(A_k) - T(A_l)| \leq T_{p/r} + T_{p/m} \qquad \text{for } 0 \leq k, l \leq m-1$$

That is, for any pair of modules the time required to complete all read accesses differs by no more than the maximum read access time.

The Read Mapping Heuristic (RMH) derived below approximates an optimal solution as follows. For a read stream t_i, accesses are mapped uniformly to a number of modules $\hat{\mu}_i$ proportional to the ratio of the minimum time to complete accesses from t_i at a single

module and the minimum time to complete all read accesses at a single module. Essentially, each stream is assigned resources proportional to the amount of work to be completed; over-allocation limits the amortization of page misses while under-allocation limits concurrency. Load balancing is performed in a greedy fashion by mapping accesses from t_i to the $\hat{\mu}_i$ modules with the minimum load. Page miss count is minimized at each module as references to t_i are initiated consecutively.

To compute $\hat{\mu}_i$ and perform load balancing in mapping, a model is required for the time to complete accesses to t_i at a single module. From the performance models derived in chapter 4, the time to complete c consecutive accesses to t_i at a given module is the sum of c multiplied by the page-hit read cycle time $T_{p/r}$ and the average page miss count multiplied by the page miss time $T_{p/m}$, so that

$$\Gamma_i(s, c) = cT_{p/r} + \begin{cases} \omega(s, d_i, c)T_{p/m} & \text{when } t_i = t_{r\text{-}wadj} \text{ (wrap-around adj.)} \\ \eta(s, d_i, c, V)T_{p/m} & \text{otherwise} \end{cases}$$

The function $\Gamma_i(s, c)$ is parameterized for stride s so that completion time can be computed both for all accesses to a single module where $s = s_i$, as when computing fraction of total work load to determine $\hat{\mu}_i$, and for accesses to one of $\hat{\mu}_i$ modules where $s = \hat{\xi}_i$, as when computing module load for balancing. Note that in the page miss count modeling function $\eta(s, d, c, V)$, the number of vectors V is the number referenced by all streams in S. For a multicopy system, not all modules necessarily service accesses referencing V vectors; however, for load balancing the number to be referenced is not known until mapping is complete. Thus the computed values of module load for balancing may be an over-estimate under certain conditions.

For all read streams in *S*, the minimum time to complete one iteration of accesses at a single module is

$$\Delta = \sum_{\substack{t_i \in S \\ m_i = r}} \Gamma_i(s_i, \varepsilon_i)$$

Then accesses from read stream t_i are mapped to a number of modules $\hat{\mu}_i$ computed as

$$\hat{\mu}_i = min(\varepsilon_i, max(1, \left\lfloor \frac{\Gamma_i(s_i, \varepsilon_i)}{\Delta} m + 0.5 \right\rfloor))$$

Note that the number of modules servicing t_i is rounded to the nearest integer with a lower bound of 1, as determined by the *max* function, and an upper bound of the total number of accesses to t_i, as determined by the *min* function.

For each module of the multicopy system, the load Λ_k at module M_k is the time to complete read accesses in the sequence A_k. As state previously, load balancing is performed in a greedy fashion by mapping accesses from t_i to the $\hat{\mu}_i$ modules with the minimum load. Thus the ε_i accesses to t_i are distributed uniformly by placing $\lfloor \varepsilon_i / \hat{\mu}_i \rfloor + 1$ references r_i in the $(\varepsilon_i \bmod \hat{\mu}_i)$ sequences with the minimum module loads, and $\lfloor \varepsilon_i / \hat{\mu}_i \rfloor$ references r_i in each of the remaining $\hat{\mu}_i - (\varepsilon_i \bmod \hat{\mu}_i)$ sequences. For a sequence A_k to which t_i is mapped, the load at module M_k is recomputed as

$$\Lambda_k = \Lambda_k + \Gamma_i(\hat{\xi}_i, c)$$

where $c = \lfloor \varepsilon_i / \hat{\mu}_i \rfloor$ or $c = \lfloor \varepsilon_i / \hat{\mu}_i \rfloor + 1$, as appropriate.

Figure 16 presents the complete read mapping heuristic (RMH). To summarize, for each read stream $t_i \in S$

- the number of modules to reference $\hat{\mu}_i$ is computed, and
- access are distributed uniformly to the $\hat{\mu}_i$ sequences referencing modules with the minimum loads.

6.4.4.1 RMH Performance

Table 10 compares results of the RMH with an optimal mapping of read accesses as determined via exhaustive search. The general form of the problem mapped is

$$\forall i \qquad y(i) = fn(x_1(i), \ldots, x_n(i))$$

Due to the time complexity of optimal assignment, problem sizes are small. The number of modules *m* is 2 or 4, the number of read streams is between 2 and 4 and the depth of loop unrolling *b* is between 1 and 3, inclusive; variables are chosen from a uniform random distribution.

Table 10 contains ratios of RMH to optimal performance, where performance is defined as the average time to complete *all read operations* for a given iteration. Only read accesses are considered to avoid skewing results in favor of the RMH, as write accesses in the general problem take the same time regardless of the read mapping.

Table 10 RMH / Optimal Performance Ratios

Category	Percentage of tests for which $\frac{\text{RMH performance}}{\text{Optimal performance}} \leq C$			
	$= 1.0$	≤ 1.1	≤ 1.2	≤ 1.3
S1	90	100	100	100
S1SL	77	81	89	93
SRND	71	81	91	97

```
// if the total number of read accesses R is less than the
// number of modules, assign one access to each sequence (module)

if  R ≤ m

   assign each A_i, 0 ≤ i ≤ R − 1, one read access;
else
{
   (Λ_i ← 0) and (A_i ← ∅)  for  0 ≤ i ≤ m − 1;

   // for each read stream t_i in S
   // note: t_{r-wadj} first and t_{r-imix} last, as appropriate.

   for all t_i ∈ S such that  m_i = r
   {
      compute μ̂_i;

      determine modules M_{p(1)}, ..., M_{p(μ̂_i)} with μ̂_i smallest Λ_i

      such that Λ_{p(1)} ≤ ... ≤ Λ_{p(μ̂_i)};

      // assign accesses from t_i to sequences and recompute
      // module loads.

      for (k = 1 to μ̂_i)
      {
         if  k ≤ ε_i mod μ̂_i

            c ← ⌊ε_i / μ̂_i⌋ + 1;
         else
            c ← ⌊ε_i / μ̂_i⌋;

         A_{p(k)} ← ⟨A_{p(k)}, ⟨r_{(i, M_{p(k)})} : c⟩⟩;

         Λ_{p(k)} ← Λ_{p(k)} + Γ_i(ξ̂_i, c);
      }
   }
}
```

Figure 16 Read Mapping Heuristic (RMH)

Results from 300 tests are presented, with 100 from each of 3 different categories. Category *S1* presents results for streams of stride one access. Category *S1SL* presents results from streams with a mixture of stride one and stride 'large', where large is defined as 1 data item per page. Finally, category *SRND* presents results for a mixture of strides chosen from a uniform random distribution between 1 and 1.5 (p/w) , where *p* is page size and *w* is word size. Overall, the RMH achieved optimal performance in 79% of the trials and was within 20% of optimal for 93% of the trials.

6.4.5 Access Ordering Algorithm

Recall that for streams *S*, with read streams t_1 through t_{N_r} and write streams t_{N_r+1} through t_N, the base access sequence employed is

$$\tilde{S}_B = \langle [A_0, \ldots, A_{m-1}], w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N \rangle$$

For deriving a specific access sequence $\tilde{S}$ in the form of the base sequence $\tilde{S}_B$, the complete access ordering algorithm consists of the following steps:

1. From the pairs of streams in *S* implementing a read-modify-write, if extant, choose a pair to map for wrap-around adjacency, $t_{r\text{-}wadj}$ and $t_{w\text{-}wadj}$, and a pair to map for intermixing, $t_{r\text{-}imix}$ and $t_{w\text{-}imix}$; define write accesses in the sequence $\tilde{S}$ accordingly.
2. Apply the RMH to determine read sequences $A_0, \ldots, A_{m-1}$ in the access sequence $\tilde{S}$; $t_{r\text{-}wadj}$ and $t_{r\text{-}imix}$ are mapped first and last, respectively.

The ordering algorithm is efficient, with a time complexity of $O(N_r^2 (\log N_r))$ for N_r read streams in *S*; this complexity represents the N_r sorts required for load balancing in the RMH.

6.4.6 Example Problem

For a 2 module multicopy system, an access sequence is generated for the canonical axpy operation to illustrate the ordering algorithm derived above. Recall that axpy is defined as

$$\forall i \qquad y_i \leftarrow a x_i + y_i$$

and generates the three streams $S = \{t_x, t_{y_r}, t_{y_w}\}$ where $t_x = (x, s_x, d_x, r) : \varepsilon_x$, $t_{y_r} = (y, s_y, d_y, r) : \varepsilon_{y_r}$, and $t_{y_w} = (y, s_y, d_y, w) : \varepsilon_{y_w}$.

For each vector assume that data size equals word size and stride of access is 1. The depth of loop unrolling b is 2, so that $\varepsilon_x = \varepsilon_{y_r} = \varepsilon_{y_w} = 2$.

The initial step identifies streams for intermixing and wrap-around adjacency, as discussed in 6.4.3. For the axpy computation, $t_{y_r} = t_{r\text{-}wadj}$ and $t_{y_w} = t_{w\text{-}wadj}$; $t_{r\text{-}imix}$ and $t_{w\text{-}imix}$ are undefined.

The RMH is employed to derive the read sequences A_0 and A_1 of the access sequence $\tilde{S}$. First, the number of modules to service each stream is computed. For the given stream parameters, the average times to complete accesses to t_{y_r} and t_x at a single module are

$$\Gamma_{y_r}(1, 2) = 2T_{p/r} + \omega(1, w, 2)T_{p/m} \approx 2T_{p/r}$$

$$\Gamma_x(1, 2) = 2T_{p/r} + \eta(1, w, 2, 2)T_{p/m} \approx 2T_{p/r} + T_{p/m}$$

Approximations for $\Gamma_{y_r}(1, 2)$ and $\Gamma_x(1, 2)$ derived above are used to simplify expressions in the remaining computations.

Then the time to complete all read accesses Δ is

$$\Delta = \Gamma_{y_r}(1, 2) + \Gamma_x(1, 2) = 4T_{p/r} + T_{p/m}$$

Finally, the number of modules servicing each of t_{y_r} and t_x is

$$\hat{\mu}_{y_r} = min(2, max(1, \left\lfloor \frac{\Gamma_{y_r}(1, 2)}{\Delta} 2 + 0.5 \right\rfloor)) = 1$$

$$\hat{\mu}_x = min(2, max(1, \left\lfloor \frac{\Gamma_x(1, 2)}{\Delta} 2 + 0.5 \right\rfloor)) = 1$$

The RMH load balancing criteria insures that accesses from streams t_{y_r} and t_x are placed in sequences A_0 and A_1 respectively. As $t_{y_r} = t_{r\text{-}wadj}$ accesses from t_{y_r} are mapped first, though in this example the order is irrelevant.

Thus application of the access ordering algorithm to the axpy computation defined above results in the access sequence

$$\tilde{S} = \langle [r_{(y_r, M_0)} : 2, r_{(x, M_1)} : 2], \langle w_{y_w} : 2 \rangle \rangle$$

representing the linear sequence of references

$$\langle r_{(y_r, M_0)}, r_{(x, M_1)}, r_{(y_r, M_0)}, r_{(x, M_1)}, w_{y_w}, w_{y_w} \rangle$$

Figure 17 depicts a typical iteration of the above sequence, assuming an access to the current page requires 1 time unit $(T_{p/r}, T_{p/w})$ and a page miss incurs an additional 2 time unit penalty $(T_{p/m})$.

6.4.7 Performance Predictor

For streams S and an access sequence $\tilde{S}$ defined as above, a performance predictor is derived for the average time per data item accessed T_{avg} and the processor-memory bandwidth BW. Recall that as a result of the module reference model developed in section 6.4.2, performance models represent estimated performance rather than bounds.

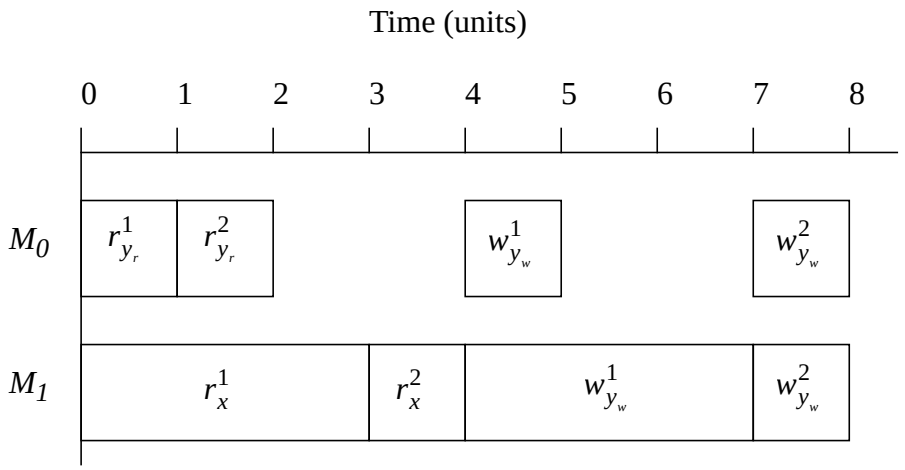

Figure 17 Multicopy Example

Functions modeling page overhead derived in chapter 4 for a single module system are applicable to accesses at individual modules of a multicopy system. Recall that in general, average page miss count is modeled by the function $\eta(s, d, c, V)$. For stream accesses that are wrap-around adjacent or intermixed, average page miss count is modeled by the functions $\omega(s, d, c)$ and $\rho(s, d, c)$ respectively. In employing these functions for a multicopy system, stride s is module stride.

Let P_k define the sequence of reads serviced by module M_k for an iteration of the access sequence $\tilde{S}$. Each P_k is composed of a number of component sequences $P_{(i,k)}$ where the first subscript i is defined to be that of the stream referenced. Thus $P_{(i,k)}$ represents the read access set $\langle r_{(i,M_k)} : c \rangle$, where $c = \lfloor \varepsilon_i / \hat{\mu}_i \rfloor$ or $c = \lfloor \varepsilon_i / \hat{\mu}_i \rfloor + 1$ as appropriate.

Similarly, Q_k is the sequence of write accesses serviced at M_k and $Q_{(i,k)}$ represents the write access set $\langle w_i : \varepsilon_i \rangle$; recall that writes are broadcast so that each module services all ε_i accesses from write stream t_i.

The time required to complete all accesses in the sequence $P_{(i,k)}$ is the sum of the number of accesses *c* multiplied by the page-hit read cycle time $T_{p/r}$ and the average page miss count multiplied by the page miss time $T_{p/m}$; i.e.

$$T(P_{(i,k)}) = cT_{p/r} + \begin{cases} \omega_k(\hat{\xi}_i, d_i, c)T_{p/m} & \text{when } P_{(i,k)} \text{ is wrap-around adj.} \\ \eta_k(\hat{\xi}_i, d_i, c, V)T_{p/m} & \text{otherwise} \end{cases}$$

Note that in modeling page miss count, conditions that determine appropriate use of modeling functions *must be applied in the context of the module accessed.* $P_{(i,k)}$ is wrap-around adjacent if there exists a $Q_{(j,k)}$ such that read stream t_i and write stream t_j implement a read-modify-write, $P_{(i,k)}$ is the first access set in P_k and $Q_{(j,k)}$ is the last access set in Q_k; then $\omega_k(\hat{\xi}_i, d_i, c)$ models page miss count. Otherwise, $\eta_k(\hat{\xi}_i, d_i, c, V)$ is the applicable model where the number of vectors *V* is the number accessed at module M_k. For clarity, functions modeling page overhead are subscripted with the module number to denote context. Note that for a wrap-around adjacent access set, the page miss count $\omega_k(\hat{\xi}_i, d_i, c)$ is an upper-bound representative of the page miss count at the module servicing $r_i^{\hat{\mu}_i}$, the $\hat{\mu}_i^{th}$ access from read stream t_i for a given iteration; this effect is a consequence of distributed reads and broadcast writes.

Similarly, the time required to complete all accesses in the sequence $Q_{(i,k)}$ is the sum of the number of accesses ε_i multiplied by the page-hit write cycle time $T_{p/w}$ and the average page miss count multiplied by the page miss time $T_{p/m}$, so that

$$T(Q_{(i,k)}) = \varepsilon_i T_{p/w} + \begin{cases} \rho_k(s_i, d_i, \varepsilon_i)T_{p/m} & \text{when } Q_{(i,k)} \text{ is intermixed} \\ \eta_k(s_i, d_i, \varepsilon_i, V)T_{p/m} & \text{otherwise} \end{cases}$$

In this context $Q_{(i,k)}$ is intermixed if there exists a $P_{(g,k)}$ such that read stream t_g and write stream t_i implement a read-modify-write, $P_{(g,k)}$ is the last access set in P_k and $Q_{(i,k)}$ is the first access set in Q_k. Note that for an intermixed access set, the page miss count $\rho_k(s_i, d_i, \varepsilon_i)$ is an upper-bound representative of the overhead at the module servicing the last reference from the corresponding read access set $r_g^{\varepsilon_g}$; again this is a consequence of distributed reads and broadcast writes.

From the preceding analysis, the time to complete all read operations in the sequence P_k is the sum of the time to complete all accesses in each component sequence; i.e.

$$T(P_k) = \sum_{P_{(i,k)} \in P_k} T(P_{(i,k)})$$

Then the time to complete all read accesses in an iteration of the sequence $\tilde{S}$ is the maximum time to complete read operations at any module, so that

$$T_r = max(T(P_0), \ldots, T(P_{m-1}))$$

Note the tacit assumption in computing T_r is that buffering is sufficient so that read accesses proceed without access gaps that result in increased completion, as discussed in section 6.4.1.1.

Similarly, the time to complete all write operations in the sequence Q_k is the sum of the time to complete all accesses in each component sequence; i.e.

$$T(Q_k) = \sum_{Q_{(i,k)} \in Q_k} T(Q_{(i,k)})$$

And the time to complete all write operations in an iteration of the sequence $\tilde{S}$ is

$$T_w = max(T(Q_0), \ldots, T(Q_{m-1}))$$

Then an estimate of the time to complete all accesses in a given iteration is the sum of the time to complete all read and write accesses so that

$$T_{tot} = T_r + T_w$$

From the above, the average time per data item accessed T_{avg} is computed as the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_{tot}}{\sum_{t_i \in S} \varepsilon_i \gamma_i}$$

The effective memory bandwidth BW, in megabytes per second, is the number of bytes of relevant data transferred per iteration divided by the time to complete all access; i.e.

$$BW = \frac{10^3 \sum_{t_i \in S} \varepsilon_i \gamma_i d_i}{T_{tot}}$$

6.5 Simulation Results

For a multicopy memory system there is no ‘natural’ mapping of accesses to modules. Thus the quality of an access ordering algorithm is best captured by comparison with an optimal reference sequence. Such a comparison is presented in section 6.4.4.1 for a system of page-mode components; for a system of uniform-access devices, access ordering results in an optimal reference sequence. For access sequences generated by each ordering algorithm, simulation results are presented to validate the accuracy of performance models.

To assess the viability of a multicopy system two factors must be considered: performance and cost. Performance is evaluated relative to a sequentially interleaved memory, as interleaving is the most common parallel memory storage scheme. Cost is evaluated in terms of both hardware complexity and data space.

6.5.1 Performance Predictors

Results are first presented to validate performance predictors for the set of benchmark computational kernels defined in section 4.4. A non-buffered 2 module multicopy system of both uniform-access and page-mode components is considered; module parameters for both component types are defined in Table 11.

Table 11 Module Parameters (Multicopy - Both)

Uniform-access		Page-mode	
Parameter	Value	Parameter	Value
w	8	w	8
		p	4096
$T_{u/r}$	40	$T_{p/r}$	40
$T_{u/w}$	40	$T_{p/w}$	40
		$T_{p/m}$	120

Table 12 compares performance of ordered accesses as calculated analytically and measured via simulation; for all benchmarks, the depth of loop unrolling is 4 and vector operands are double precision. For the computations and conditions modeled, analytic and simulation results differ by less than 1%. Two exceptions are highlighted. Recall from section 6.4.1.1 that in modeling performance of read operations for a system of page-mode components, buffering is assumed sufficient so that accesses proceed at the maximum rate. For both cases noted, a non-buffered system results in access gaps that reduce performance; for a buffer size of 1, simulated performance achieves the predicted bandwidth.

Table 12 Analytic vs Simulation Results (Multicopy - Both)

Computation	Uniform-access		Page-mode	
	Analysis BW	Simulation BW	Analysis BW	Simulation BW
daxpy	240.0	240.0	171.0	170.9
dvaxpy	256.0	256.0	177.2	159.2
LL-1	240.0	240.0	171.0	170.6
LL-3	320.0	320.0	397.7	394.6
LL-4	320.0	320.0	388.6	386.4
LL-5	240.0	240.0	171.0	170.6
LL-7	256.0	256.0	152.0	152.0
LL-11	213.3	213.3	133.0	133.1
LL-12	213.3	213.3	133.0	133.1
LL-20	261.8	261.8	171.0	171.1
LL-21	240.0	240.0	161.3	156.7
LL-22	228.6	228.6	142.5	142.3
LL-24	320.0	320.0	395.4	393.1

6.5.2 Evaluation of Multicopy Performance

A multicopy system offers a number of advantages over a sequentially interleaved memory. For read streams, maximum concurrency is achievable regardless of stride and page overhead can be more effectively amortized by directing accesses from a given stream to a smaller number of modules. However, because read accesses must be tagged to reference a specific module, to fully utilize concurrency the number of read accesses in a loop must equal or exceed the number of memory modules. Furthermore, write operations are broad-

cast to all modules to maintain coherence and thus represent the serialization of an otherwise parallel operation.

For a multicopy system to deliver greater bandwidth than an equivalent interleaved memory, increases in parallelism and/or reduction in page overhead for read accesses must dominate the loss of parallelism for writes; in this context an equivalent system is one with the same number of modules, equal buffer depth, and constructed from identical memory components. Note that in all but extreme circumstances, a multicopy system of uniform-access components is not competitive as page overhead is not a concern. Thus only systems of page mode components are considered here.

Table 13 presents simulation results comparing bandwidth delivered by a 4 module multicopy system with buffer depth 1 to an equivalent interleaved system for the benchmark kernels. Module parameters are those of Table 11 with a page miss versus hit cycle time ratio of 4:1, typical of current DRAMs. For the interleaved system, access ordering is performed assuming known alignment. In all cases, the depth of loop unrolling is 4 and vector operands are double precision.

For the computations measured, vector strides are such that all *m* modules in a sequentially interleaved system are referenced by each stream for any $m = 2^n$. Thus the multicopy system can reduce page overhead for read accesses but achieves no greater parallelism. Performance results are mixed: 4 computations achieve greater bandwidth, 5 computations experience a reduction in bandwidth, and 4 computations achieve approximately the same bandwidth. Note that LL-3 and LL-4 represent dot products and do not generate write streams, thus the substantial increase in performance.

For next generation DRAMs the page miss-hit cycle time ratio will increase dramatically [NEC92]. This situation benefits a multicopy architecture as reduction in page overhead becomes even more critical to obtaining good performance, as illustrated below.

Table 13 Multicopy vs Interleaved (4:1)

Computation	4:1		%Increase
	Interleaved BW	Multicopy BW	
daxpy	266.7	199.2	(25.3)
dvaxpy	246.2	199.3	(19.0)
LL-1	200.0	199.2	(0.4)
LL-3	200.0	786.2	293.1
LL-4	200.0	751.6	275.8
LL-5	200.0	199.2	(0.4)
LL-7	200.0	227.8	13.9
LL-11	200.0	145.2	(27.4)
LL-12	200.0	145.2	(27.4)
LL-20	200.0	256.5	28.3
LL-21	266.7	188.4	(29.4)
LL-22	200.0	190.1	(4.5)
LL-24	781.7	772.8	(1.1)

Assume a 4 module multicopy system with buffer depth 1 and an equivalent interleaved system. Module parameters are defined in Table 14 with a page miss-hit cycle time ratio of 10:1. Table 15 presents simulation results comparing bandwidth achieved for the set of benchmark computations, given a loop depth of 4 and double-precision vector operands.

Relative performance of the multicopy architecture is improved: 8 computations achieve greater bandwidth than the sequentially interleaved system, 4 computations experience modest degradation of less than 15%, and only 1 computation experiences a substantial reduction in bandwidth of 21%. Note that for LL-3 and LL4, which generate no write

Table 14 Module Parameters (Multicopy - Page)

Parameter	Value
w	8
p	4096
$T_{p/r}$	10
$T_{p/w}$	10
$T_{p/m}$	90

streams, the multicopy architecture achieves nearly an *order of magnitude* more bandwidth than the equivalent interleaved system.

A multicopy architecture can substantially improve performance over an equivalent interleaved memory for computations with a high read to write ratio, as demonstrated above. Many computations exhibit this characteristic naturally; for others, intelligent use of cache memory and strip-mining or tiling techniques can increase the read-write ratio by holding modified vector elements in cache.

6.5.3 Evaluation of Multicopy Cost

A multicopy architecture can provide increased bandwidth over an equivalent interleaved memory. However, additional cost is incurred in terms of both hardware complexity and data space. Each of these issues is considered below.

The additional hardware complexity for a multicopy system is minimal. A sequentially interleaved memory distributes accesses to modules based on low-order address bits, as discussed in chapter 5. For read accesses, a multicopy architecture distributes references to modules based on high-order address bits; these bits can be set at compile time as a result of mapping as performed by the RMH. Write accesses require additional hardware for broadcast to all modules.

Table 15 Multicopy vs Interleaved (10:1)

Computation	10:1		%Increase
	Interleaved BW	Multicopy BW	
daxpy	457.1	398.8	(12.8)
dvaxpy	412.9	454.3	10.0
LL-1	320.0	397.7	24.3
LL-3	320.0	3039.7	849.9
LL-4	320.0	2681.5	738.8
LL-5	320.0	398.8	24.6
LL-7	320.0	489.6	53.0
LL-11	320.0	278.3	(13.0)
LL-12	320.0	277.4	(13.3)
LL-20	320.0	550.9	72.2
LL-21	457.1	360.3	(21.2)
LL-22	320.0	408.1	27.5
LL-24	3091.3	2894.7	(6.4)

A strict multicopy system provides only $(1/m)^{th}$ the address space of an equivalent interleaved architecture as data is replicated at all *m* modules. Note however that the hardware requirements for the two systems are very similar. It is easy to imagine a memory controller capable of implementing both schemes. In fact, given proper hardware support, multicopy and interleaved memory can be implemented concurrently by designating a portion of the interleaved address space for multicopy access.

Thus the cost of a multicopy architecture is considerably less than the functional description might imply. Building multicopy support into an interleaved architecture can provide a low cost means for increasing effective memory bandwidth for amenable computations.

6.6 Summary

This chapter develops access ordering algorithms for a proposed multicopy architecture. Performance predictors are developed for the effective memory bandwidth achieved by ordered accesses.

For a multicopy system of uniform-access components, the ordering algorithm divides accesses into two phases: a read phase and a write phase. Read accesses are distributed uniformly across modules, optimizing concurrency; write accesses are broadcast and hence proceed sequentially. Ordering is trivial and a performance predictor is derived in a straight-forward fashion. Simulation demonstrates the performance model to be accurate. In general, a multicopy system of uniform-access components does not represent a viable alternative to an equivalent sequentially interleaved architecture.

For a multicopy system of page-mode components ordering is analogous to the uniform-access case. However, mapping read accesses to modules is performed via a heuristic. Intermixing and wrap-around adjacency are employed in a greedy fashion at the boundaries of the read and write phases. The ordering algorithm has a time complexity of $O(N_r^2\,(\log N_r))$ for N_r read streams that is representative of load balancing in the RMH. Simulation demonstrates performance models for ordered accesses to be accurate.

Performance results indicate that a multicopy system of page-mode components can provide increased bandwidth over an equivalent interleaved memory for computations with a high read to write access ratio. Furthermore, multicopy access can be implemented with a minimal increase in hardware complexity as part of a heterogenous interleaved/multicopy memory architecture.

7 Implementation Issues

Access ordering algorithms derived in the preceding chapters are memory centric and do not reflect processor constraints; in particular, register file size, pipelined functional units, and bus characteristics are not considered. Furthermore, all memory references are assumed to be non-caching, even though many codes benefit from caching a subset of the vectors operands. Finally, reference sequences are assumed to adhere to the stream interaction restriction, thus limiting the applicable problem domain. Each of these implementation issues is addressed below.

7.1 Relieving Register Pressure

Access ordering employs loop unrolling to increase the number of accesses within a given loop that can be reordered, thus increasing the potential for minimizing page overhead, maximizing concurrency, and fully utilizing wide words, as applicable. However, loop unrolling creates register pressure and has traditionally been limited by register resources.

Lee [Lee91] presents a technique that employs cache memory to mimic a set of vector registers, effectively increasing register file size for vector computations. Storage is defined for a set of vectors, each of which represents a pseudo register; vector length corresponds to register size. For example, two 64-element ‘vector registers’ are defined in the *C* programming language as

```
double VectorRegister[2][64];
```

Prior to performing computations, each pseudo register element is referenced via a standard caching load instruction so that the vector register address space is likely to reside in cache memory. Note that to insure pseudo vector register elements do not conflict in cache, vector storage must not exceed cache capacity for a direct-mapped cache or $(1/n)^{\text{th}}$ cache capacity for an *n-way* set-associative cache [LaRW91].

Within a loop, vector operands are loaded into the pseudo vector registers, arithmetic operations are performed on vector register data, and vector register results are stored back to the appropriate vector elements in memory. Vector registers are loaded by first loading each vector element into a processor register via a non-caching access, and then storing the value to the appropriate vector register location in cache.

By applying the above technique, processor register pressure is relieved and the effective vector register space is limited only by the cache size.

7.2 Pipelined Processors and Bus Bandwidth

Recall that for a system constructed from page-mode components, interleaving references from a pair of streams implementing a read-modify-write can often reduce page overhead for write operations. For example, given a single memory module, read stream t_i, and write stream t_j, section 4.1.1 derives the general intermix sequence as

$$\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$$

Though intermixing minimizes page miss count for the write stream, the resulting sequence reduces data-bus bandwidth and may not be amenable for execution on a pipelined processor.

To illustrate, consider the vector scaling operation

$$\forall i \qquad y_i \leftarrow k y_i$$

that generates read stream t_{y_r} and write stream t_{y_w}; assume one data item per word. The optimal access sequence resulting in a write stream page miss count of zero is

$$\langle \langle r_{y_r}, w_{y_w} \rangle : \varepsilon_{y_r} \rangle$$

Implementing this sequence requires reading an element of the vector $\bar{y}$, performing a multiplication, then immediately storing the result. Thus multiplication must be performed in a scalar (non-pipelined) rather than pipelined mode. Furthermore, the data-bus must remain idle for one bus cycle between read and write operations to avoid interference between outgoing write data and incoming read data. Thus alternating access modes increases the number of idle cycles and hence reduces effective bus bandwidth.

If the stride of $\bar{y}$ is small then the non-interleaved sequence $\langle r_{y_r} : \varepsilon_{y_r}, w_{y_w} : \varepsilon_{y_w} \rangle$ results in a negligible increase in page miss count while maximizing bus bandwidth and allowing multiplication operations to be pipelined. If the stride of $\bar{y}$ is large, e.g. 1 data-item per DRAM page, then the additional page overhead resulting from the non-interleaved reference sequence may exceed the gains from pipelined arithmetic operations and increased bus bandwidth.

Let T_I and T_{NI} represent the times to complete one iteration of accesses for a read-modify-write operation with an interleaved and non-interleaved reference sequence, respectively. Values for T_I and T_{NI} are computed as the maximum of the bus transfer and memory access times, where bus transfer time is processor dependent and memory access time comes directly from performance models developed for each memory architecture. Let T_S and T_P represent the times to complete one iteration of arithmetic operations for a read-modify-write operation in scalar and pipelined modes, respectively. Values for T_S and T_P are processor dependent.

Then implementing scalar operations and an interleaved reference sequence achieves the maximum computation rate if

$$max(T_I, T_S) < max(T_{NI}, T_P)$$

Otherwise, maximum computation rate is obtained from pipelined arithmetic operations and a non-interleaved sequence of references. Note that this formula assumes computation is overlapped with memory latency.

Of the access ordering algorithms derived in preceding chapters, only two potentially interleave references for a read-modify-write operation: the algorithm for a single module system and the algorithm for a sequentially interleaved system under the assumption of unknown alignments. All other ordering algorithms separate read and write accesses into distinct phases, so that the interleaving of references is not an issue.

7.3 Combining Caching and Non-Caching Memory Access

Access ordering algorithms presume the use of non-caching load instructions to control via software the sequence of requests observed by the memory system and to avoid extraneous data references. However many codes generate multiple references to a subset of vector operands and hence can benefit from caching, particularly when implemented using strip-mining and tiling techniques [CaKe89, Wolf89]. Thus access ordering and caching should be used together to complement one another, exploiting the full memory hierarchy to maximize memory bandwidth.

Perhaps the simplest method for combining caching and non-caching access in a coherent fashion is to preload the cache with multiply-referenced data and utilize non-caching accesses for single-visit items. Access ordering can then be applied to all accesses in the inner loop, to maintain dependencies, with only the effect of non-caching accesses considered for maximizing memory system performance.

To illustrate, consider implementing the matrix-vector multiply operation

$$\bar{y} = (A + B)\,\bar{x}$$

where A and B are $n \times m$ matrices and $\bar{y}$ and $\bar{x}$ are vectors.

Figure 18(a) depicts code for a straight-forward implementation of the matrix-vector multiply operation. Figure 18(b) strip-mines the computation to reuse elements of $\bar{y}$; partition size is dependent on cache size and structure [LaRW91]. Elements of $\bar{y}$ are preloaded into cache memory at the appropriate loop level, and elements of *A* and *B* are referenced via non-caching loads; the reference to $\bar{x}$ is a constant within the inner loop and is preloaded into a processor register. Access ordering can now be applied to the inner loop to maximize bandwidth for references to *A* and *B*.

(**a**)
```
// Straight-forward implementation: y = (A + B)x

for j = 1 to m
   for i = 1 to n
      y[i] = y[i] + (A[i,j] + B[i,j]) * x[j];
```

(**b**)
```
// Strip-mined implementation: y = (A + B)x

for IT = 1 to n by IS
{
   preload y[IT] through y[min(n, IT + IS - 1)] into cache;

   for j = 1 to m
   {
      preload x[j] into a processor register;

      // Each element of A and B referenced exactly once via a
      // non-caching load.

      for i = IT to min(n, IT + IS - 1)
         y[i] = y[i] + (A[i,j] + B[i,j]) * x[j];
   }
}
```

Figure 18 Combining Caching and Non-Caching Access

If the pseudo vector register technique described in section 7.1 is used to relieve processor register pressure for loop unrolling, care must be taken to insure that vector registers and cached operands do not collide in cache memory; the same is true for multiple cached operands. Lam *et al* [LaRW91] analyze a technique that eliminates cache conflicts by copying data to be cached into a contiguous address space. Note that in applying this copy optimization, non-caching loads, and hence access ordering, can be used to reduce cache conflict and extraneous data movement.

By combining intelligent cache management with access ordering techniques, the full memory hierarchy is exploited to maximize effective bandwidth.

7.4 Relaxation of the Stream Interaction Restriction

Access ordering algorithms derived in preceding chapters presume access streams adhere to the stream interaction restriction, defined in section 3.4 as:

> *Stream Interaction Restriction*: For any two access streams $t_i, t_j \in S$, $v_i \neq v_j$ implies that the streams have non-intersecting address spaces; in particular, streams reference no pages in common. When $v_i = v_j$ stream parameters are identical except in mode, where by definition $m_i \neq m_j$.

Though analysis is simplified, dependence between accesses belonging to different streams is limited to two types: loop-independent antidependence and data dependence in the data flow sense.

Minor relaxation of the stream interaction restriction significantly increases the scope of computations to which the access ordering algorithms can be applied. Relaxation techniques are considered below for two special cases: self-antidependence cycles and read streams with overlapping address spaces.

7.4.1 Self-Antidependence Cycles

Some common computations exhibit a loop-carried antidependence of the form

$$\forall i \qquad y_i \leftarrow fn\,(y_{i+k}) \qquad k \in Z^+$$

Streams generated by this computation violate the stream interaction restriction by referencing overlapping, rather than identical or non-intersecting, address spaces.

For the simple self-antidependence cycle demonstrated above, common access ordering techniques, such as loop unrolling and grouping accesses by stream, can easily be applied. However, modeling page miss count and managing concurrency are more complex for streams involved in a loop-carried antidependence than for streams implementing a strict read-modify-write.

Access ordering algorithms derived in preceding chapters can accommodate streams generated by a self-antidependent computation in a suboptimal fashion by ordering accesses from each stream independently and insuring that all reads are initiated prior to the first write. A simple optimization places references from the read and write streams adjacent to potentially reduce write access page miss count, when applicable; this technique is analogous to intermixing for streams implementing a strict read-modify-write.

7.4.2 Overlapping Read Address Spaces

The stream interaction restriction states that read streams must have non-intersecting address spaces, suggesting that ordering algorithms are not applicable to common computations such as

$$\forall i \qquad y_i \leftarrow x_{3i} + x_{3i+1}$$

However, access ordering algorithms derived in preceding chapters can easily accommodate intersecting read streams in a suboptimal fashion by ordering accesses from each

stream independently. Read streams with intersecting address spaces may exhibit input dependence, however this can be ignored for non-volatile memory locations. A simple optimization places references from intersecting read streams adjacent, potentially reducing page miss count when applicable.

7.4.3 Access Ordering and Vectorizable Computations

A vectorizable loop is one with no multi-statement dependence cycles and only self-dependence cycles that are ignorable or represent known reduction or recurrence operations for which vector instructions exist; in testing if a loop is vectorizable, input dependence is ignored for non-volatile memory locations [Wolf89].

Relaxing the stream interaction restriction as discussed above allows access ordering algorithms to be applied to the class of vectorizable loops, an arguable large and interesting problem domain.

8 Conclusions

Access ordering, a loop optimization that reorders accesses to better utilize memory system resources, is a compiler technology developed in this thesis to address the memory bandwidth problem for scalar processors executing scientific codes. For a given computation, memory architecture, and memory device type, an access ordering algorithm determines a well-defined interleaving of vector references that maximizes effective bandwidth. Consequently, analytic models of performance can also be derived. Access ordering algorithms developed are applicable to a superset of the class of vectorizable loops, an arguably large and interesting problem domain.

Access ordering is fundamentally different from, though complementary to, access scheduling techniques that attempt to overlap computation with memory latency but do not consider the performance of the resulting access sequence. Simulation results demonstrate that for a given computation, access ordering can significantly increase effective memory bandwidth over that achieved by the natural sequence of references. Simulation results validate analytic models of performance as well.

Access ordering algorithms presume the use of non-caching load instructions to control the sequence of requests observed by the memory system and to avoid extraneous data references. For computations that benefit from caching a subset of the vector operands, access ordering is shown complementary to strip-mining and tiling techniques. By intelligent caching of multiply referenced data items, and careful ordering of non-caching references to single visit operands, the full memory hierarchy is exploited.

The following summarizes results for each of the ordering algorithms derived. Performance modeling features and applications are discussed. Finally, potential impact and future extensions of this work are considered.

8.1 Summary of Access Ordering Algorithms

Because access ordering exploits features of the underlying memory system, an ordering algorithm must be derived for each target memory architecture and device type. Three memory architectures are analyzed in the preceding chapters: single module, sequentially interleaved, and multicopy. For each architecture two memory component types are considered: uniform-access and page-mode.

Chapter 4 derives access ordering algorithms for a single module memory architecture. For uniform-access components ordering is trivial since all orderings perform equally well. For page-mode components an ordering algorithm is derived that minimizes page overhead to achieve optimal performance for a given computation. Accurate models of performance are developed in both cases. Theorems derived for the optimal access of a single memory module form the basis for analyzing parallel memory systems.

Simulation results for a range of scientific kernels demonstrate that access ordering can achieve a significant increase in effective memory bandwidth at a single module of page-mode components with a modest degree of loop unrolling. Ordered accesses achieve up to 189% more bandwidth than the natural reference sequence for the benchmark computations simulated. Analytic models predict simulation results to within 1%.

Chapter 5 derives access ordering algorithms for a sequentially interleaved memory architecture. Ordering algorithms are derived assuming both unknown and known stream alignments. If stream alignment is unknown then an optimal ordering algorithm can not be derived. If stream alignment is known then determining an optimal access sequence is NP-complete with a time complexity exponential in the number of accesses; as an optimal solution is intractable, a heuristic solution is required.

Simulation results for a range of scientific kernels demonstrate that access ordering can achieve a significant increase in effective memory bandwidth for a sequentially inter-

leaved system at a modest depth of loop unrolling. For a 4 module system of uniform-access components, ordered accesses achieve up to 256% more bandwidth than the natural reference sequence for computations simulated; for a 2 module system of page-mode components up to 189% more bandwidth is achieved. Analytic bounds on performance are validated and shown to accurately predict performance for the computations considered.

Finally, chapter 6 derives access ordering algorithms for a proposed multicopy architecture that replicates data across modules. Read accesses may be directed to any module; write accesses are broadcast to maintain coherence. Maximum concurrency for read streams is achievable for all strides of reference and page overhead can be more efficiently amortized, when applicable.

A multicopy system of uniform-access components does not represent a viable alternative to an equivalent sequentially interleaved architecture. However, simulation results indicate that a multicopy system of page-mode components can provide increased bandwidth over an equivalent interleaved memory for computations with a high read to write access ratio; an order of magnitude better performance is achieved in some benchmarks. Furthermore, multicopy access can be implemented with a minimal increase in hardware complexity as part of a heterogeneous interleaved-multicopy memory architecture.

8.2 Performance Modeling

Performance models derived for the systems above are unique in several aspects in that

- the reference sequence is not stochastic, but rather deterministically ordered for each member of a well defined class of computations,
- module performance is not assumed insensitive to the sequence of requests but is modeled to accurately reflect current memory component technology, and
- data item size is not restricted to word size, rather, wide word access is incorporated naturally into the models.

Performance modeling based on access ordering has direct application in a number of evaluation tools, in particular for

- *system evaluation* - to provide a benchmark both for cost-performance analysis of different memory systems and for matching memory performance to processor requirements, and
- *algorithm evaluation* - to provide a benchmark for algorithm selection based on effective bandwidth utilization for a given memory system.

Analytic results presented throughout this work provide a basic and extensible set of tools for capturing memory system behavior and for understanding the interaction of reference sequences with memory architecture and component characteristics.

8.3 Potential Impact and Future Work

Access ordering may impact future processor architectures and memory components. Few processors currently implement the non-caching load instruction required for access ordering. However, just as the demonstrable advantages of prefetching [CaKP91, KlLe91] led to processors with prefetch instructions, access ordering may provide the impetus for more manufacturers to implement a non-caching load. Similarly, access ordering demonstrates the true potential for page-mode memory components and provides incentive to further reduce page-hit times.

Access ordering is an original concept and the work presented forms only the initial basis; much still remains to be done. Below are topics of first-order importance for future research.

Relaxation of the stream interaction restriction to encompass the class of vectorizable loops is discussed in section 7.4. However, the more complex stream interactions that result should be incorporated formally into both ordering algorithms and performance models.

Page-mode components are modeled as if implemented with a single on-chip cache line, reflecting current DRAM technology. Future high-speed DRAMs will incorporate multiple pages, among other exploitable features. Rambus [Slat92] represents such a technology. As memory components evolve, access ordering algorithms and performance models must be developed that reflect these changes.

A number of implementation issues remain to be solved. Combining caching and non-caching loads requires detection of multiply and singly referenced vectors and the application of strip-mining and tiling techniques. Utilizing the cache to implement pseudo vector registers, thereby relieving processor register pressure, requires inserting additional accesses into the instruction stream and introduces cache management issues. Modeling the effect of an interleaved reference sequence on effective computation rate requires further formalization than provided in section 7.2. Other implementation issues are sure to arise.

Naturally, incorporating access ordering into an optimizing compiler represents the most important part of this work that remains to be done.

Appendix A

Intermix Sequences

A.1 Proof of Optimal Intermix Pattern

Given: read stream t_i and write stream t_j specifying a read-modify-write, i.e. $v_i = v_j$.

Prove: the intermix sequence $\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$ is the optimal interleave pattern.

Proof: Consider the general interleave case

$$\langle \ldots, r_i : q_1, w_j : k_1, \ldots, r_i : q_n, w_j : k_n, \ldots \rangle$$

where, by definition, r_i^k must proceed w_j^k and

$$\sum_{l=1}^{n} q_l = \sum_{l=1}^{n} k_l$$

Let

$$\sum_{l=1}^{\lambda} q_l = S(q, \lambda) \qquad \text{and} \qquad \sum_{l=1}^{\lambda} k_l = S(k, \lambda)$$

It is easily seen that for $\lambda < n$, $S(q, \lambda) \geq S(k, \lambda)$. If there exists a $q_l \neq k_l$ then there must exist at least one u such that $S(q, u) > S(k, u)$, in which case

- the page miss count in performing the read sequence $\langle \ldots, r_i : q_{u+1}, \ldots \rangle$ can be greater than in the case where $S(q, u) = S(k, u)$ since $w_j^{S(k, u)}$ may access a sequentially earlier page than $r_i^{S(q, u)}$;
- similarly, the page miss count in performing the write sequence $\langle \ldots, w_j : k_{u+1}, \ldots \rangle$ can be greater than in the case where $S(q, u) = S(k, u)$ as $w_j^{S(k, u)+1}$ may access a sequentially earlier page than $r_i^{S(q, u)+1}$.

Thus, the minimum page miss count is achieved when $S(q, u) = S(k, u)$ for $u \le n$; i.e. when $q_l = k_l$ for $1 \le l \le n$.

$\therefore \langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$ is the optimal interleave pattern. □

A.2 Derivation of $\rho(s, d, c)$

Given the intermix sequence $\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$ where t_i and t_j specify a read-modify-write operation, i.e. $v_i = v_j$, the function $\rho(s_j, d_j, c)$ is the average page miss count in performing each set of c write accesses.

Recall that if $v_i = v_j$ then stream parameters for t_i and t_j are identical except in mode; in particular, $s_i = s_j$ and $d_i = d_j$. Thus s and d are used below to denote stride and data size, respectively, for streams t_i and t_j.

In deriving $\rho(s, d, c)$, the following observation is made: in performing c accesses from a given stream the address space spanned, in bytes, is $(c-1)sd + d$.

Assume $(c-1)sd + d \le p$, then the address space spanned touches at most two pages. If p_1 is the probability that c accesses touch one page, and p_2 is the probability that two pages are touched, then

$$\rho(s, d, c) = p_1(0) + p_2(2) = 2p_2$$

That is, for the access sequence $\langle \ldots, \langle r_i : c, w_j : c \rangle : h, \ldots \rangle$, the write operations $w_j^{(k-1)c+1}$ through w_j^{kc}, $1 \le k \le h$, suffer exactly two page misses when $r_i^{(k-1)c+1}$ and r_i^{kc} reference a different page; otherwise write operations $w_j^{(k-1)c+1}$ through w_j^{kc} page-hit.

The number of d-aligned starting positions in a given page for a set of c read accesses is

$$S = \frac{p}{d}$$

The number of starting positions resulting in c read accesses touching exactly one page is

$$S_1 = \frac{p - ((c-1)sd + d)}{d} + 1$$

Then the probability that a set of c read accesses touches exactly one page is

$$p_1 = \frac{S_1}{S} = 1 - \frac{(c-1)sd}{p}$$

and the probability that two pages are touched is

$$p_2 = 1 - p_1 = \frac{(c-1)sd}{p}$$

Thus, when $(c-1)sd + d \le p$, the average page miss count in performing each set of c write accesses is

$$\rho(s, d, c) = 2p_2 = \frac{2(c-1)sd}{p}$$

When $(c-1)sd + d > p$ the address space spanned touches at least two pages, implying that each set of c write accesses must begin with a page miss. Then the average page miss count is one plus the remaining accesses, $c-1$, divided by the number of accesses per page; i.e.

$$1 + \frac{c-1}{\phi(s, d)}$$

Combining the results derived above

$$\rho(s, d, c) = \begin{cases} \dfrac{2(c-1)sd}{p} & \text{when} \quad (c-1)sd + d \le p \\ 1 + \dfrac{c-1}{\phi(s, d)} & \text{when} \quad (c-1)sd + d > p \end{cases}$$

Appendix B

Module Sequence Algorithm

B.1 Properties of the MSA

Given: N streams $t_1, \ldots, t_N$, $\mu_1 \geq \ldots \geq \mu_N$, mapped to sequences $A_0, \ldots, A_{m-1}$ via the Module Sequence algorithm.

Prove: each round-robin selection of accesses from $A_0, \ldots, A_{m-1}$ has the property that for each stream t_i referenced:

1. there are exactly μ_i accesses to t_i, and
2. accesses from t_i do not conflict.

Proof of property 1:

Let $U = \{t_j \mid Z_j \cap Z_i = Z_i, t_j \in \{t_1, \ldots, t_{i-1}\}\}$. Assume that $U \neq \varnothing$. Then there exists a $t_L \in U$ such that for all $t_j \in U$, $L \geq j$. Accesses to t_L immediately precede accesses to t_i in a sequence A_k such that $M_k \in Z_i$. If each round-robin selection of accesses from $A_0, \ldots, A_{m-1}$ that references t_L initiates exactly $\mu_L = |Z_L|$ accesses to t_L, then each subsequent round-robin selection of accesses that references t_i must initiate exactly $\mu_i = |Z_i|$ accesses to t_i.

If for all $t_j \in \{t_1, \ldots, t_{i-1}\}$ it is true that $Z_j \cap Z_i = \varnothing$, then t_i is the first stream accessed in a sequence A_k such that $M_k \in Z_i$; this is the default when $i = 1$. In this case it is easily seen that each round-robin selection of accesses that references t_i must initiate exactly $\mu_i = |Z_i|$ accesses to t_i.

$\therefore$ By induction, each round-robin selection of accesses from $A_0, \ldots, A_{m-1}$ that references t_i must initiate exactly μ_i accesses to t_i.

Proof of property 2:

Property 2 is a direct result of property 1. Since each round-robin selection of accesses that references t_i must initiate exactly $\mu_i = |Z_i|$ accesses to t_i, then a sequence A_k such that $M_k \in Z_i$ can not simultaneously specify references to any other stream.

$\therefore$ In a given round-robin selection of accesses from $A_0, \ldots, A_{m-1}$, accesses from t_i do not conflict. □

Bibliography

[AvCK87] Aven-O, Coffman-E, Kogan-Y, “Stochastic Analysis of Computer Storage”, Kluwer Academic Publishers, Norwell, MA, 1987, pp. 102-118.

[Bail87] Bailey-D, “Vector Computer Memory Bank Contention”, IEEE Trans. Comput., **36**, 3, 1987, pp. 293-298.

[BeDa91] Benitez-M, Davidson-J, “Code Generation for Streaming: an Access/Execute Mechanism”, Proc. ASPLOS-IV, 1991, pp. 132-141.

[BeRo91] Bernstein-D, Rodeh-M, “Global Instruction Scheduling for Superscalar Machines”, Proc. SIGPLAN’91 Conf. Prog. Lang. Design and Implementation, 1991, pp. 241-255.

[BuCo70] Burnett-G, Coffman-E, “A Study of Interleaved Memory Systems”, 1970 Spring Joint Computer Conference, AFIPS Conf. Proc., **36**, 1970, pp. 467-474.

[BuKu71] Budnik-P, Kuck-D, “The Organization and Use of Parallel Memories”, IEEE Trans. Comput., **20**, 12, 1971, pp. 1566-1569.

[CaCK90] Callahan-D, Carr-S, Kennedy-K, “Improving Register Allocation for Subscripted Variables”, Proc. SIGPLAN ‘90 Conf. Prog. Lang. Design and Implementation, 1990, pp. 53-65.

[CaKe89] Carr-S, Kennedy-K, “Blocking Linear Algebra Codes for Memory Hierarchies”, Proc. of the Fourth SIAM Conference on Parallel Processing for Scientific Computing, 1989.

[CaKP91] Callahan-D, Kennedy-K, Porterfield-A, “Software Prefetching”, Proc. ASPLOS-IV, 1991, pp. 40-52.

[ChKL77] Chang-D, Kuck-D, Lawrie-D, “On the Effective Bandwidth of Parallel Memories”, IEEE Trans. Comput., **26**, 5, 1977, pp. 480-489.

[ChSm86] Cheung-T, Smith-J, “A Simulation Study of the CRAY X-MP Memory System”, IEEE Trans. Comput., **35**, 7, 1986, pp. 613-622.

[CoBS71] Coffman-E, Burnett-G, Snowdon-R, “On the Performance of Interleaved Memories with Multiple-Word Bandwidths”, IEEE Trans. Comput., **20**, 12, 1971, pp. 1570-1573.

[GaJo79] Garey-M, Johnson-D, “Computers and Intractability: A Guide to the Theory of NP-Completeness”, Freeman, New York N.Y., 1979.

[GoCh84] Goodman-J, Chiang-M, “The Use of Static Column RAM as a Memory Hierarchy”, Proc. 11th Annual Intl. Symp. on Comput. Architecture, 1984, pp. 167-174.

[GuSo88] Gupta-R, Soffa-M, “Compile-time Techniques for Efficient Utilization of Parallel Memories”, SIGPLAN Not., **23**, 9, 1988, pp. 235-246.

[HaJu87] Harper-D, Jump-J, “Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme”, IEEE Trans. Comput., **36**, 12, 1987, pp. 1440-1449.

[HaLi89] Harper-D, Linebarger-D, “A Dynamic Storage Scheme for Conflict-Free Vector Access”, Proc. 16th Intl. Symp. Comput. Architecture, 1989, pp. 72-77.

[Harp89] Harper-D, “Address Transformations to Increase Memory Performance”, Proc. 1989 Intl. Conf. Parallel Processing, 1989, pp. 237-241.

[Harp91] Harper-D, “Block, Multistride Vector, and FFT Accesses in Parallel Memory Systems”, IEEE Trans. Parallel and Dist. Systems, **2**, 1, 1991, pp. 43-51.

[Hell66] Hellerman-H, “On the Average Speed of a Multiple-Module Storage System”, IEEE Trans. Comput., **15**, 8, 1966, p. 670.

[Inte89] Intel Corporation, “i860 64-Bit Microprocessor Hardware Reference Manual”, ISBN 1-55512-106-3, 1989.

[KlLe91] Klaiber-A, Levy-H, “An Architecture for Software-Controlled Data Prefetching”, Proc. 18th Annual Intl. Symp. Comput. Architecture, 1991, pp. 43-53.

[KuSt82] Kuck-D, Stokes-R, “The Burroughs Scientific Processor (BSP)”, IEEE Trans. Comput., **31**, 5, 1982, pp. 363-375.

[Lam88] Lam-M, “Software Pipelining: An Effective Scheduling Technique for VLIW Machines”, Proc. SIGPLAN’88 Conf. Prog. Lang. Design and Implementation, 1988, pp. 318-328.

[LaRW91] Lam-M, Rothberg-E, Wolf-M, “The Cache Performance and Optimizations of Blocked Algorithms”, Fourth International Conf. on Arch. Support for Prog. Langs. and Operating Systems, 1991, pp. 63-74.

[LaVo82] Lawrie-D, Vora-C, “The Prime Memory System for Array Access”, IEEE Trans. Comput., **31**, 5, 1982, pp. 435-442.

[Lee90] Lee-K, “On the Floating-Point Performance of the i860 Microprocessor”, NASA Ames Research Center, NAS Systems Division, RNR-090-019, 1990.

[Lee91] Lee-K, “Achieving High Performance on the i860 Microprocessor with Naspack Subroutines”, NASA Ames Research Center, NAS Systems Division, RNR-091-029, 1991.

[Mcma90] McMahon-F, FORTRAN Kernels: MFLOPS, Lawrence Livermore National Laboratory, Version MF443.

[Moye91] Moyer-S, “Performance of the iPSC/860 Node Architecture”, University of Virginia, IPC-TR-91-007, 1991.

[NEC92] NEC Corporation, “16Mb Synchronous DRAM”, Preliminary Data Sheet v3.1, Oct. 1992.

[OeLa85] Oed-W, Lange-O, “On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems”, IEEE Trans. Comput., **34**, 10, 1985, pp. 949-957.

[PeVa87] Peelen-T, Van de Goor-A, “Using the Page Mode of Dynamic RAMs to Obtain a Pseudo Cache”, Microprocessors and Microsystems, **11**, 9, 1987, pp. 469-473.

[Quin91] Quinnell-R, “High-speed DRAMs”, EDN, May 23, 1991, pp. 106-116.

[RaSY89] Rau-B, Schlansker-M, Yen-D, “The Cydra 5 Stride-Insensitive Memory System”, Proc. 1989 Intl. Conf. Parallel Processing, 1989, pp. 242-246.

[Rau91] Rau-B, “Pseudo-Randomly Interleaved Memory”, Proc. 18th Intl. Symp. Comput. Architecture, 1991, pp. 74-83.

[Ravi72] Ravi-C, “On the Bandwidth and Interference in Interleaved Memory Systems”, IEEE Trans. Comput., **21**, 8, 1972, pp. 899-901.

[RaWa81] Ramamoorthy-C, Wah-B, “An Optimal Algorithm for Scheduling Requests on Interleaved Memories for a Pipelined Processor”, IEEE Trans. Comput., **30**, 10, 1981, pp. 787-799.

[ShTu72] Shedler-G, Tung-C, “Locality in Page Reference Strings”, SIAM J. on Computing, **1**, 3, 1972, pp. 218-241.

[Slat92] Slater-M, “Rambus Unveils Revolutionary Memory Interface”, Microprocessor Report, March 4, 1992, pp. 15-21.

[SpDe72] Spirn-J, Denning-P, “Experiments with Program Locality”, AFIPS Conference Proc., Fall Joint Comput. Conf., **41**, 1972, pp. 611-621.

[Stev92] Stevens-R, “Computational Science Experiences on the Intel Touchstone DELTA Supercomputer”, Compcon Spring 92 Digest of Papers, 1992, pp. 295-299.

[WeSm90] Weiss-S, Smith-J, “A Study of Scalar Compilation Techniques for Pipelined Supercomputers”, ACM Trans. Math. Soft., **16**, 3, 1990, pp. 223-245.

[Wolf89] Wolfe-M, “Optimizing Supercompilers for Supercomputers”, MIT Press, Cambridge, Mass., 1989.