# Access Ordering Algorithms for an Interleaved Memory

Steven A. Moyer

IPC-TR-92-012

December 18, 1992

# Access Ordering Algorithms for an Interleaved Memory

Steven A. Moyer

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, Virginia 22903

(*sam2y@virginia.edu*)

Superscalar processors are well suited for meeting the demands of scientific computing, given sufficient memory bandwidth. Employing parallel memory modules increases the bandwidth available; however, storage schemes devised to reduce module conflict for vector computers are not suitable for scalar computation. *Access ordering* is a compilation technique that increases effective bandwidth by reordering references to exploit the underlying memory system. Ordering algorithms are derived in this report for a sequentially interleaved memory architecture.

The author wishes to gratefully acknowledge the work of the WM Architecture Group at the University of Virginia, the UVA Academic Enhancement Program, NASA grant NAG-1-242, and NSF grants MIP-9114110 and CDA-8922545-01.

# Table of Contents

# 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 |
| $\sigma$ | number of data items referenced per functional iteration |

**MAP notation**:

| | |
|---|---|
| $a_i$ | access to the next element of stream $t_i$ |
| $a_i^k$ | $k^{\text{th}}$ access from $t_i$ for a given access sequence iteration |
| $S$ | set of all streams in a given MAP |
| $N$ | number of streams in $S$ |
| $V$ | number of different vectors referenced by streams in $S$ |
| $b$ | depth of loop unrolling |

**Performance measures**:

| | |
|---|---|
| $T_{avg}$ | average time per access |
| $BW$ | processor-memory bandwidth |

**General properties of stream $t_i$**:

$\varepsilon_i$ number of accesses per loop iteration

$\theta_i$ intermix factor

**Properties of stream $t_i$ for a sequentially interleaved architecture**:

$\mu_i$ number of modules referenced

$Z_i$ set of modules referenced

$\xi_i$ module stride

$\psi_i$ maximum number of accesses serviced at any module for a given iteration

**Modeling functions**:

$\gamma(s, d)$ average number of data items per word

$\phi(s, d)$ average number of data items per page

$\eta(s, d, c, V)$ 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 average page miss count of write stream

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

# 1 Introduction

Superscalar† pipelined processors are well suited for meeting the demands of scientific computing, singly and as components of parallel machines. However, studies demonstrate that for such applications, performance is limited by the processor-memory bandwidth [Lee90, Moye91].

For vector computers, parallel memory modules are employed to increase effective bandwidth through concurrent processing of memory requests. Research into parallel memory systems is generally directed towards developing storage schemes, i.e. mappings of addresses to memory locations, that reduce module conflict and hence increase concurrency. Proposed storage schemes include the use of a prime number of modules [LaVo82], skewed storage [BuKu71, HaJu87], and dynamic address transformations [Harp89, Rau91]. Note that these techniques are dependent on a relatively long sequence of references to a single vector.

Scalar processors executing scientific codes generate an interleaved sequence of references to a set of vector operands. Thus, simply applying a given storage scheme is unlikely to produce maximum concurrency in a parallel memory system. Furthermore, the performance of individual modules of modern DRAM components is sensitive to the sequence of requests; this issue is not addressed in previous parallel memory studies.

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 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 effective memory bandwidth by fetching extraneous data for non-unit strides. Thus, as

---

†. Both superscalar and VLIW architectures are suited for scientific applications and place similar demands on the memory system.

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* [Moye92b] is a compiler technology 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.

In this report, access ordering algorithms are developed for a sequentially interleaved memory architecture. Interleaving is the most prevalent parallel memory storage scheme whereby for an $m$ module system, word $a$ maps to module $(a \bmod m)$.

## 1.1 Background

This work builds on previous analytic results derived for a single module memory system [Moye92a]. To make this document self-contained, the necessary analysis from that report is repeated here. Readers familiar with previous work may skip immediately to the analysis of an interleaved architecture presented in section 5; note: there is an important addition to the MAP access sequence definition presented in 3.1.

## 1.2 General System Model

Access ordering algorithms presume a dedicated memory system driven by a single scalar processor, 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 uniprocessors 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 con-

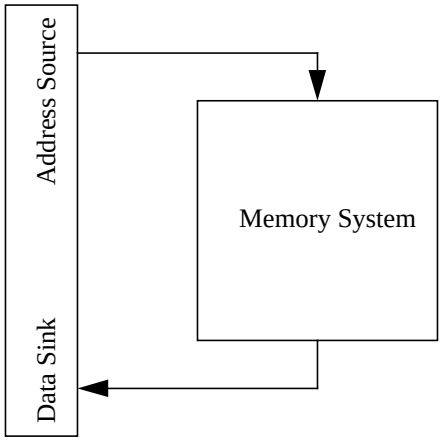


**Figure 1 General System Model**

trolled 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 section 6.

## 1.3 Access Ordering Observation

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

Consider a single module of *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].

Figure 2(a) illustrates the ‘natural’ reference sequence for a straight-forward translation of the *vaxpy*, vector axpy, computation

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

†. Note that a DRAM page should not be confused with a virtual memory page; this is an unfortunate overloading of terms.

For modest size vectors, elements $a_i$, $x_i$, and $y_i$ are likely to reside in different pages, so that alternating accesses to each incurs the page miss overhead; memory references likely to page miss are highlighted in Figure 2.

In the loop of Figure 2(a), 3 page misses occur for every 4 references; a different ordering can result in every reference generating a page miss. 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; in this case 3 misses occur for every 8 references. In reducing page miss count, processor-memory bandwidth is increased significantly.

```
loop:
   load a
   load x
   load y
   stor y
   jump loop
```

(a)

```
loop:
   load a
   load a
   load x
   load x
   load y
   load y
   stor y
   stor y
   jump loop
```

(b)

**Figure 2 Vaxpy Code**

## 1.4 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 that 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:

$$\forall i \qquad y_i \leftarrow ax_i + y_i$$

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 $\bar{y}$.

In this report, the computation domain for which access ordering algorithms are developed is further restricted to the class of vectorizable loops. Since vectorizable loops contain no loop-carried dependencies, excepting ignorable input dependence and self-antidependence cycles [Wolf89], reordering accesses within an unrolled loop is simplified. Note that recurrence relations can often be eliminated through streaming optimizations [BeDa91], so that algorithms developed here are actually applicable to a superset of the vectorizable loops.

## 1.5 Memory Device Types

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. Ordering algorithms are derived here for each of the two major memory component 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 1.3; 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.

The system word size is defined by $w$. 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.

## 1.6 Performance Modeling

For a given computation, access ordering results in code that generates a well-defined sequence of vector references. Consequently, for each 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. In developing such models, it is assumed that the processor is sufficiently fast so that performance is limited by the memory system. Thus performance models represent maximum effective bandwidth.

# 2 Previous Work

Access ordering spans a number of interrelated topics from compiler optimizations to performance modeling. The following sections provide the minimal level of context neces-

sary to characterize the contributions of this work; a more complete survey of all relevant topics can be found in [Moye92b].

## 2.1 Stream Detection

Access ordering algorithms derived in this report presuppose the existence of compiler techniques to detect stream-oriented computations. Benitez and Davidson [BeDa91] describe a technique for detecting streaming opportunities, including those in recurrence relations. Callahan *et al* [CaCK90] present a technique called *scalar replacement* that detects redundant accesses to subscripted variables in a loop, often transforming a more complex sequence of references to a vector into a single access stream. Finally, as 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, referred to here 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; however, the effective memory bandwidth is not considered. 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.

# 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. The Model Access Pattern notation used to denote specific reference sequences is defined below, along with a set of general definitions and assumptions applicable to all computations. Access ordering in the presence of wide words is also discussed. Finally, a restriction is placed on stream interaction to simplify optimality results.

## 3.1 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 $t_i = (v, s, d, m) : \sigma$ where

- $v$ = vector to be accessed = stream starting address
- $s$ = stride of access
- $d$ = data type size
- $m$ = access mode, read($r$) or write($w$)
- $\sigma$ = number of data items accessed in a single functional iteration

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

let $a_i$ denote access to the 'next' element of the stream $t_i$, then

1. $\{a_i\}$ is an access sequence.
2. $\{A_1, \ldots, A_n\}$ 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. $\{A:c\}$ is an access sequence where $A$ is an access sequence and $c$ is a positive integer; $A$ is repeated $c$ consecutive times.
4. $[A_1, \ldots, A_n \mid \alpha_1, \ldots, \alpha_n]$ 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 $\alpha_i$ accesses from $A_i$ until all accesses in $A_1, \ldots, A_n$ have been initiated. If fewer than $\alpha_i$ 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.

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

In discussing a particular MAP

- stream parameters are referred to by dot notation, e.g. $t_i.s$ is stride, and
- $a_i^k$ refers to the $k^{\text{th}}$ access from $t_i$ for a given access sequence iteration.

For visual clarity, $\{a_i\}:c \equiv \{a_i:c\}$ and extraneous brackets are omitted when the meaning is unambiguous. When the access mode is known, an access is denoted as $r_i$ or $w_i$ for $t_i.m = r$ or $t_i.m = w$, respectively.

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

$$\forall i \qquad y_i \leftarrow ax_i + y_i$$

Three access streams are generated defined by the tuples $t_x = (x, s_x, d_x, r):1$, $t_{y_r} = (y, s_y, d_y, r):1$, and $t_{y_w} = (y, s_y, d_y, w):1$. The 'natural' access sequence imple-

menting the axpy computation is: $\{r_x, r_{y_r}, w_{y_w}\}$, specifying one read from each of $t_x$ and $t_{y_r}$, followed by one write from $t_{y_w}$, per loop iteration.

The above notation affords convenient specification of accesses to parallel memory modules. For example, given an interleaved system and known stream alignment†, if sequence $A_i$ represents requests to module $M_i$, then $[A_0, \ldots, A_{m-1}]$ specifies an access sequence that references each module with period $m$‡ and provides for concurrency among accesses from different streams.

## 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 MAP,
- $N = |S|$, i.e. for a given MAP the total number of access streams is $N$, and
- $V$ = number of unique $t_i.v$ such that $t_i \in S$, i.e. for a given MAP the number of vectors accessed is $V$.

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

- $t_i.d \mid w$, i.e. for all streams in $S$ word size is a multiple of the data size,
- access stream $t_i$ begins at an address divisible by $t_i.d$, i.e. data is aligned, and
- stride of access $t_i.s$ is positive; the stream interaction restriction defined below allows this assumption without loss of generality.

## 3.3 Wide Word Restrictions

For completeness, it is desirable to accommodate wide word access in ordering algorithms and performance models; a typical example being a 32-bit value referenced from a 64-bit word. To fully utilize wide words, and simplify modeling, several minor restrictions are

†. Alignment assumed known with respect to modules; i.e. the first module referenced by each stream.

‡. Module reference sequence has period $m$ if all modules service the same number of accesses per iteration.

placed on stream parameters and code generation for a computation. Prior to presenting these restrictions, the following definition is made:

For access stream $t_i$ with $s = t_i.s$ and $d = t_i.d$, the average number of data items per word is

$$\gamma(s, d) = \begin{cases} 1 & \text{when} \quad \frac{w}{sd} \leq 1 \\ \frac{w}{sd} & \text{when} \quad \frac{w}{sd} > 1 \end{cases}$$

Then for the set of streams *S* of a given MAP, it is assumed that for all $t_i \in S$

- access stream $t_i$ begins at an address divisible by *w*, i.e. streams are word aligned, and
- the average number of data items per word $\gamma(s, d)$ is an integer, implying that each word accessed contains exactly the same number of data items.

Access ordering employs loop unrolling to increase the number of stream accesses within a loop that can be reordered, as discussed in section 1.3; *b* is defined to be the depth of unrolling. To maximize wide word utilization, an access ordering algorithm must insure that for a given computation, the depth of loop unrolling is such that the number of data items referenced from each stream per iteration is a multiple of the number of data items per word; i.e. for stream $t_i$ with $\sigma = t_i.\sigma$, $\gamma(s, d) \mid b\sigma$. Note that in the most common case of one data item per word per stream, *b* can be any positive integer.

Given the above restrictions, each access to stream $t_i$ references exactly $\gamma(s, d)$ data items, with the number of accesses per loop iteration defined by

$$\varepsilon_i = \frac{b\sigma}{\gamma(s, d)}$$

Wide word access is accommodated in a natural, intuitive, and optimal fashion. Each stream access is guaranteed to reference a different word, and the number of data items per word is constant.

## 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 access streams $t_i, t_j \in S$, $t_i.v \neq t_j.v$ implies that the streams have non-intersecting address spaces; in particular, streams reference no pages in common. When $t_i.v = t_j.v$ stream parameters are identical except in mode, where by definition $t_i.m \neq t_j.m$.

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. $t_i.v \neq t_j.v$, 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. $t_i.v = t_j.v$, 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 $k^{\text{th}}$ accesses from each of $t_i$ and $t_j$ reference the same data item and hence the same page.

Strict adherence to the stream interaction restriction limits the applicability of access ordering algorithms. However, this limited problem domain is still large and encompasses many interesting computations. Furthermore, under the stream interaction restriction, optimality results are obtained for single module access and concurrency is more easily managed in parallel memory systems. Relaxation of this restriction for applying ordering algorithms to the set of vectorizable loops is discussed in section 6.

## 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 $t_i.v = t_j.v$, 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 $t_i.v = t_j.v$ 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.

### 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.

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 streams 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 preloaded 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 Analysis

Prior to examining an interleaved system, techniques are first presented for minimizing page overhead at a single module of page-mode DRAMs. Complete ordering algorithms for a single module system are not derived; only the tools necessary for analyzing an interleaved system of page-mode components are developed.

## 4.1 Minimizing Page Overhead

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. $t_i.v \neq t_j.v$ for all $t_j \in S$, minimum page overhead is achieved by performing a sequence of accesses $a_i$ without an intervening access to a second vector $a_j$. 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 access stream $t_i$ with $s = t_i.s$ and $d = t_i.d$, the average number of data items per page is

$$\phi(s, d) = \begin{cases} 1 & \text{when} \quad \frac{p}{sd} \leq 1 \\ \frac{p}{sd} & \text{when} \quad \frac{p}{sd} > 1 \end{cases}$$

Then arranging accesses from $t_i$ as $\{\ldots, a_i{:}c, \ldots\}$, the average per iteration page miss count is

$$\eta(s, d, c, V) = \begin{cases} \frac{c\gamma(s, d)}{\phi(s, d)} & \text{when} \quad V = 1 \\ 1 + \frac{(c-1)\,\gamma(s, d)}{\phi(s, d)} & \text{when} \quad V \geq 2 \end{cases}$$

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 the number of data items referenced divided by the number of data items 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 data items to access, $(c-1)\,\gamma(s, d)$, divided by the number of data items per page.

Note that the average page miss count per access, $\eta(s, d, c, V)/c$, is either constant or inversely proportional to $c$. In the later case, separating the $c$ accesses must increase the per reference page overhead. Consequently, minimum page overhead is achieved when accesses are grouped by stream.

**Theorem 1:** Given stream $t_i \in S$ such that $t_i$ does not participate in a read-modify-write, i.e. $t_i.v \neq t_j.v$ for all $t_j \in S$, minimum average page overhead is achieved by the access sequence $\{\ldots, a_i{:}\varepsilon_i, \ldots\}$.

### 4.1.1 Intermixing

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

Consider the *general intermix sequence*

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

that generates the string of references

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

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 average page miss count for the read stream is unchanged. However, the sequence of accesses $w_j^{(k-1)c+1}$ through $w_j^{kc}$, $1 \leq k \leq h$, suffers a page miss only when $r_i^{(k-1)c+1}$ and $r_i^{kc}$ reference a different page.

For write stream $t_j$ with $s = t_j.s$ and $d = t_j.d$, the average page miss count in performing each set of $c$ write accesses in the intermix sequence $\{\ldots, \{r_i{:}\,c, w_j{:}\,c\}{:}\,h, \ldots\}$ is derived in Appendix A.1 as

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

Thus, the total average page miss count in performing all $ch$ write operations for a given iteration is $h\rho(s, d, c)$. The general intermix sequence $\{\ldots, \{r_i{:}\,c, w_j{:}\,c\}{:}\,h, \ldots\}$ is optimal, as demonstrated in Appendix A.2.

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

$$imix(s, d, c, h, V) = \begin{cases} \rho(s, d, ch) - h\rho(s, d, c) & \text{when} \quad V = 1 \\ \eta(s, d, ch, v) - h\rho(s, d, c) & \text{when} \quad V \ge 2 \end{cases}$$

It can be shown algebraically that $imix(s, d, c, h, V) > 0$, i.e. intermixing reduces write access page miss count, if $c = 1$ or $((c-2)h + 1)\gamma(s, d)sd < p$. Therefore, when $imix(s, d, c, h, V) > 0$ the average page miss count in performing each set of $c$ write accesses, $\rho(s, d, c)$, is directly proportional to $c$. Thus, choosing $c$ as small as possible minimizes write page overhead.

#### 4.1.1.1 Intermix Factor

For the general intermix sequence, the values of the *intermix parameters c* and *h* that minimize page overhead 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 efficiently utilizing wide word access, when applicable. 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 computation 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 $t_j$ with $s = t_j.s$, $d = t_j.d$ and $\sigma = t_j.\sigma$, the intermix factor is computed as

$$\theta_j = \begin{cases} 1 & \text{when } t_j \text{ is not data dependent on } t_i \\ \dfrac{lcm(\sigma, \gamma(s, d))}{\gamma(s, d)} & \text{otherwise} \end{cases}$$

From the derivation of $\varepsilon_j$ in section 3.3, it can be seen that the number of accesses to stream $t_j$ per loop iteration is a multiple of the intermix factor $\theta_j$; i.e. $\theta_j \mid \varepsilon_j$. Thus, intermix parameters $c = \theta_j$ and $h = \varepsilon_j / \theta_j$ minimize page overhead if $imix(s, d, c, h, V) > 0$; otherwise, intermixing increases page overhead and is therefore not employed.

**Theorem 2:** For read stream $t_i$ and write stream $t_j$ that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $t_i.v = t_j.v$, minimum average page overhead for write stream $t_j$ is achieved by the general intermix sequence $\{\ldots, \{r_i : c, w_j : c\} : h, \ldots\}$ with $c = \theta_j$ and $h = \varepsilon_j / \theta_j$ if $imix(s, d, c, h, V) > 0$. Page overhead for read stream $t_i$ is unaffected by intermixing and equivalent to that achieved by the access sequence $\{\ldots, r_i : \varepsilon_i, \ldots\}$.

Though intermixing minimizes page overhead, the resulting sequence may not be amenable for execution on pipelined processors; this issue is discussed further in section 6.

### 4.1.2 Wrap-around Adjacency

Given read stream $t_i$ and write stream $t_j$ that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $t_i.v = t_j.v$, it is often possible to reduce the average page miss count of the read stream via wrap-around adjacency. 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.

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

Note that in the special case where $t_i$ and $t_j$ are the only streams in a computation, the intermix sequence $\{\{r_i{:}c, w_j{:}c\}{:}h\}$ also results in wrap-around adjacency.

Since $r_i^{\varepsilon_i}$ and $w_j^{\varepsilon_j}$ reference the same location, then for a given 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. In terms of page overhead the read stream proceeds as if no other vector is accessed, so that page miss count is computed by $\eta(s, d, c, V)$ where $V = 1$.

Then, for a wrap-around adjacent read stream $t_i$ with $s = t_i.s$ and $d = t_i.d$, the average per iteration page miss count is

$$\omega(s, d, c) = \frac{c\gamma(s, d)}{\phi(s, d)}$$

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

$$wadj(s, d, c, V) = \eta(s, d, c, V) - \omega(s, d, c)$$

For a given read stream wrap-around adjacency results in minimum possible page overhead, as the read stream proceeds without page thrashing.

**Theorem 3:** For read stream $t_i$ and write stream $t_j$ that specify a read-modify-write, i.e. $t_i, t_j \in S$ and $t_i.v = t_j.v$, minimum average page overhead for read stream $t_i$ is achieved via wrap-around adjacency.

# 5 Interleaved Architecture Analysis

Access ordering algorithms and performance predictors are now derived for a sequentially interleaved memory system as depicted in Figure 3. Sequential interleaving is the ‘standard’ parallel memory storage scheme whereby for an $m$ module system, word $a$ maps to module $a \bmod m$.

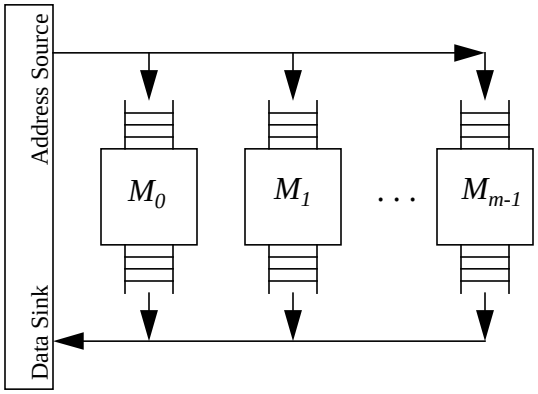


**Figure 3 Sequentially Interleaved Architecture**

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.

The following sections discuss the problem space for efficient utilization of sequentially interleaved memory and present analytic results characterizing the interaction of a single stream with an interleaved architecture. Access ordering algorithms and performance pre-

dictors are then derived under the assumption of unknown and known stream alignments, respectively.

## 5.1 Problem Dimensions

Three features of current parallel memory systems can be exploited to increase processor-memory bandwidth: module concurrency, page-mode operation (if applicable), and wide-word access. Note that wide-word access is managed optimally via conditions specified in section 3.3.

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 section 4 for a single memory module. Page overhead 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 overhead 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.

## 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.

### 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.

For deriving these characteristics, an *access cycle* is defined as a minimal set of consecutive stream accesses such that the first access in each adjacent cycle references a similarly aligned data item at the same module. For a stream with stride of access $s$ and data size $d$, access cycle length is the least common multiple of the number of bytes traversed per access $sd$ and the number of bytes across all modules $mw$; i.e. $lcm(sd, mw)$. Then the number of data items accessed per cycle is

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

If the number of data items per word is greater than one, i.e. $\gamma(s, d) > 1$, then stream accesses proceed sequentially across all modules so that the number of modules accessed is $m$. By assumption streams are word aligned, the number of data items per word is constant, and each access references exactly those data items within a word. Thus, the sequence of modules referenced has a period of $m$.

If the number of data items per word equals one, i.e. $\gamma(s, d) = 1$, and the number of bytes traversed per access is a multiple of the word size, i.e. $w \mid sd$, then each access is to the same relative position in a different word from the previous reference. The number of modules accessed is equal to the number of data items accessed per cycle, as computed in equation (1), and reduces to

$$\frac{m}{gcd(\frac{sd}{w}, m)}$$

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

Finally, consider the case where the number of data items per word equals one and the number of bytes traversed per access is not a multiple of the word size. In computing the number of accesses per cycle from equation (1), since $mw$ is a power of 2 then the $gcd(sd, mw)$ is also a power of 2. Given that $\gamma(s, d) = 1$ then by definition $w < sd$, so that $gcd(sd, mw)$ must be a power of 2 less than *w.* Thus the number of accesses per cycle is a multiple of *m*, so that 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 4 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.

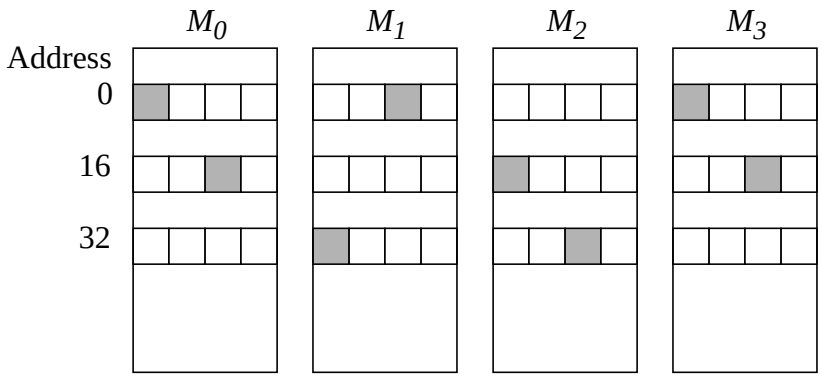


**Figure 4 Access Mapping Diagram**

From the preceding analysis, for stream $t_i$ with $s = t_i.s$ and $d = t_i.d$, accesses are distributed uniformly across a number of modules defined by

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

Furthermore, when the number of data items per word is greater than one or the number of bytes traversed per access is a multiple of the word size, then the sequence of modules accessed by $t_i$ has a period of $\mu_i$.

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

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

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

Since the number of modules accessed by a stream is a power of 2, then for any pair of streams $t_i$ and $t_j$ it must be true that $Z_i \cap Z_j = \varnothing$, or $Z_i \cap Z_j = Z_i$, or $Z_i \cap Z_j = Z_j$. 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 section 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 distance between consecutive accesses from the same stream observed at a particular module.

For a stream $t_i$ with $s = t_i.s$ and $d = t_i.d$, if the number of data items per word is greater than one, then by assumption $t_i$ is word aligned and the number of data items per word is constant. For a given module $M_k$, accesses from $t_i$ reference consecutive words so that the stride between data items accessed is *s*.

When the number of data items per word equals one and the distance between stream accesses is a multiple of the word size, i.e. $w \mid sd$, then for a module $M_k$ the distance between consecutive accesses from the same stream is constant. Module stride is com-

puted as the number of modules accessed multiplied by the actual stride and divided by the total number of modules; i.e.

$$\frac{\mu_i s}{m} = \frac{s}{gcd(\frac{sd}{w}, m)}$$

Finally, if the number of data items per word equals one and the distance between stream accesses is not a multiple of the word size, then from the analysis of access mapping the number of accesses per module per cycle is

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

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

$$\frac{lcm(sd, mw)}{m} = \frac{sdw}{gcd(sd, mw)} \tag{3}$$

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

Figure 5 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.

From the above results, for stream $t_i$ with $s = t_i.s$ and $d = t_i.d$, the module stride is computed as

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

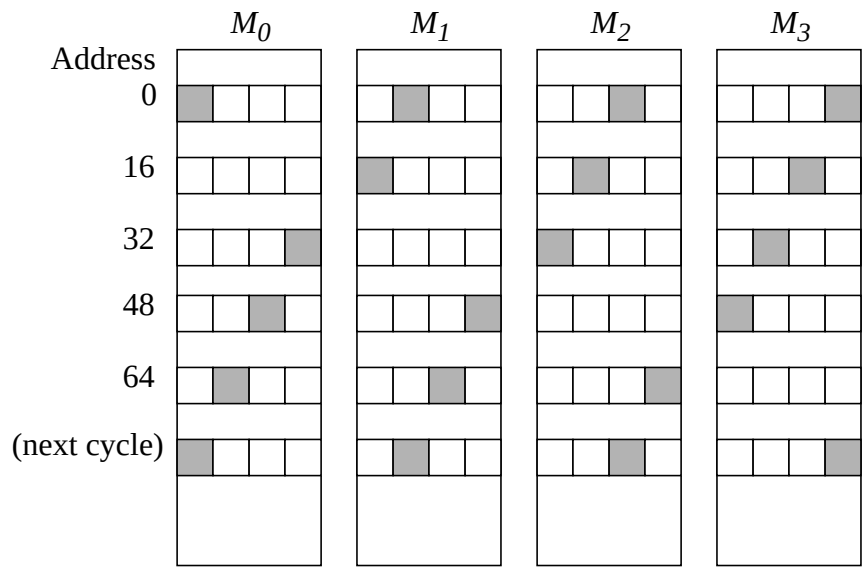


**Figure 5 Module Stride Diagram**

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. Streams for which

- the number of data items per word is greater than one or
- the number of bytes traversed per access 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 either property. Fortunately, such streams rarely occur in practice. For the remainder of this analysis, all streams are assumed to exhibit one of the two properties listed above.

## 5.3 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.

Ordering algorithms and performance predictors are derived below for systems of uniform-access and page-mode components, respectively. The effectiveness of access ordering and accuracy of performance models are demonstrated via simulation.

### 5.3.1 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 developed.

Recall that a stream $t_i$ references $\mu_i$ modules in a sequence with period $\mu_i$. 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 $\mu_i$, 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.

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

The above sequence maximizes concurrency for a stream $t_i$ by issuing sets of (at most) $\mu_i$ 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.

In defining a MAP 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. If streams $t_1$ through $t_{N_r}$ are assumed to be read streams and $t_{N_r+1}$ through $t_N$ are write streams, then the access sequence employed is

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

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.

#### 5.3.1.1 Performance Predictor

Given a MAP for a set of streams *S* with an access sequence as defined above, a performance predictor is derived for the average time per access $T_{avg}$ 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 is the ceiling of the number of accesses $\varepsilon_i$ 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 $\psi_i$ 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 $\varepsilon_i$ 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.

$$T_r = \begin{cases} \varepsilon_i \dfrac{T_{u/r}}{\mu_i} & \text{when} \quad N = 1 \text{ and } \exists (t_i \in S) \text{ such that } t_i.m = r \\ \displaystyle\sum_{\substack{t_i \in S \\ t_i.m = r}} \psi_i T_{u/r} & \text{when} \quad N \geq 2 \end{cases}$$

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

$$T_w = \begin{cases} \varepsilon_i \dfrac{T_{u/w}}{\mu_i} & \text{when} \quad N = 1 \text{ and } \exists (t_i \in S) \text{ such that } t_i.m = w \\ \displaystyle\sum_{\substack{t_i \in S \\ t_i.m = w}} \psi_i T_{u/w} & \text{when} \quad N \geq 2 \end{cases}$$

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

$$T_{avg} = \frac{T_r + T_w}{b \displaystyle\sum_{t_i \in S} t_i.\sigma}$$

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.

$$BW = \frac{10^3 b \displaystyle\sum_{t_i \in S} [\,(t_i.d)\ (t_i.\sigma)\,]}{T_r + T_w}$$

### 5.3.2 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 computations that do not specify a read-modify-write. Intermixing and wrap-around adjacency are then employed to reduce the page overhead for computations implementing this operation. Finally, the general ordering algorithm is presented and a performance predictor derived for the ordered accesses.

#### 5.3.2.1 A General Access Strategy

Consider a set of streams *S* with no pair of streams implementing a read-modify-write. By Theorem 1, page overhead 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 *N* independent streams, page overhead is minimized by performing all accesses to each stream consecutively for a given iteration, as in the sequence

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

Alternatively, as in the ordering algorithm derived in 5.3.1, potential concurrency can be maximized by interleaving a number of accesses from each stream equal to the number of modules referenced; e.g.

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

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

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 overhead. Conversely, interleaving accesses from different streams results in maximum page overhead 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 $\{a_i : \varepsilon_i\}$, $t_i \in S$.

#### 5.3.2.2 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 access sets in a sequence 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. Wrap-around adjacency results when accesses to $t_i$ 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 is analogous to that for a single module system, and reduction in page overhead for the read stream 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

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

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 2 for a single module system.

If the number of modules accessed is greater than one, 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 maintain data dependence. Then if the number of modules accessed by $t_j$ $(t_i)$ is a multiple of the intermix factor $\theta_j$, i.e. $\theta_j \mid \mu_j$, the optimal intermix sequence is

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

Page overhead 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 $\psi_j$ $(\psi_i)$ 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 $\theta_j$, then the intermix sequence employed is

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

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

By definition, if intermixing reduces page overhead for write operations then minimum page overhead is achieved when the intermix parameter *c* is equal to the intermix factor $\theta_j$. As discussed above, if $\theta_j$ divides the number of modules referenced $\mu_j$ then accesses to $t_i$ and $t_j$ can be issued so that page overhead is 0 and concurrency is maximized. Otherwise, interleaving sets of $\theta_j$ accesses from each of $t_i$ and $t_j$ minimizes page overhead but results in some of the $\mu_j$ modules referenced remaining idle with each set of $\theta_j$ reads and writes. Thus, optimal intermix performance results in a trade-off between minimum page overhead and maximum concurrency.

For small strides, the additional page overhead for performing all $\varepsilon_j$ $(\varepsilon_i)$ read and write accesses consecutively in an intermix sequence is minimal; for strides of sufficient size such that each access will page miss, minimizing page overhead is moot. Thus, for the intermix sequence (5), concurrency is chosen over page overhead in a potentially suboptimal solution. Again, note that *individual modules* observe the general intermix sequence

(4); intermix parameter *c* is $\psi_j$ for the module servicing the maximum number of accesses, and *h* is 1.

Given an intermix sequence defined as above, the reduction in page overhead for the write stream is modeled by the function $imix(s, d, c, h, V)$ derived in section 4.1.1 for a single module system. In employing this function for an interleaved system, $imix(s, d, c, h, V)$ must model the reduction in page overhead 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.

#### 5.3.2.3 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. A MAP access sequence that minimize page overhead while preserving dependence is

$$\{r_1:\varepsilon_1, \ldots, r_{N_r}:\varepsilon_{N_r}, w_{N_r+1}:\varepsilon_{N_r+1}, \ldots, w_N:\varepsilon_N\}$$

For streams *S* with one or more pair of streams implementing a read-modify-write, a MAP access sequence is defined by the following algorithm:

Determine the total ordering of access sets $\{a_i:\varepsilon_i\}$, $t_i \in S$, 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 5.3.2.2.

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. The result is an efficient algorithm.

#### 5.3.2.4 Performance Predictor

For a MAP consisting of a set of streams *S* and an access sequence defined by the algorithm above, a performance predictor is derived for the average time per access $T_{avg}$ 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 section 4 for a single module system are applicable to accesses at individual modules of an interleaved system. Recall that in general, average page overhead is modeled by the function $\eta(s, d, c, V)$. For a read stream that is wrap-around adjacent, average page overhead is modeled by the function $\omega(s, d, c)$. Finally, for an intermixed write stream, average page overhead is modeled by the function $\rho(s, d, c)$. Note that in employing these functions for an interleaved system, stride *s* is module stride.

Let *P* represent an access sequence over the set of streams *S*. Then *P* is composed of some number of component sequences $P_i$, where the subscript is defined to be that of the stream referenced; for an interleaved sequence the subscript is defined to be that of the read stream. For accesses ordered by the algorithm of 5.3.2.3, $P_i$ must be in the form of

- a read access set $\{r_i:\varepsilon_i\}$,

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

If $P_i = \{r_i{:}\,\varepsilon_i\}$ and the number of streams $N$ is greater than one then $T(P_i)$, the time to complete the sequence $P_i$, is the sum of the maximum number of references at any module $\psi_i$ multiplied by the page-hit read cycle time $T_{p/r}$ and the average page overhead 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 $\varepsilon_i$ and the average time per access so that

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

As discussed in section 5.3.2.2, at most one read access set may be considered wrap-around adjacent and must be the first access set in the sequence *P*. 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 $P_i = \{w_i{:}\,\varepsilon_i\}$ and the number of streams $N$ is greater than one then $T(P_i)$ is the sum of the maximum number of references at any module $\psi_i$ multiplied by the page-hit write cycle time $T_{p/w}$ and the average page overhead 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 $\varepsilon_i$ and the average time per access so that

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

Finally, if $P_i$ is one of the two possible intermix sequences $\{\{r_i{:}\,c, w_j{:}\,c\}:h\}$ or $\{[r_i{:}\,\varepsilon_i, w_j{:}\,\varepsilon_j \mid \mu_i, \mu_j]\}$ then the pattern of reference observed at individual modules is the general intermix sequence $\{\{r_i{:}\,c, w_j{:}\,c\}:h\}$. Intermix parameters *c* and *h* are derived in 5.3.2.2 for the module servicing the maximum number of accesses.

Then $T(P_i)$ is the sum of the maximum number of accesses $\varepsilon_i$ $(\varepsilon_j)$ 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 overheads for read and write operations at that module multiplied by the page miss time $T_{p/m}$ so that

$$T(P_i) = \psi_i (T_{p/r} + T_{p/w}) + (\eta(\xi_i, t_i.d, \psi_i, V) + h\rho(\xi_j, t_j.d, c))\, T_{p/m}$$

From the preceding analysis, the time to complete an iteration of the access sequence *P* is the sum of the times required to complete each component sequence $P_i$; i.e.

$$T_{tot} = \sum_{P_i \in P} T(P_i)$$

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

$$T_{avg} = \frac{T_{tot}}{b \sum_{t_i \in S} t_i.\sigma}$$

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.

$$BW = \frac{10^3 b \sum_{t_i \in S} [\,(t_i.d)\;(t_i.\sigma)\,]}{T_{tot}}$$

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

### 5.3.3 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.

#### 5.3.3.1 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 1, with sizes in bytes and times in nanoseconds. Timing parameters are typical of commercially available SRAMs.

**Table 1 Module Parameters (Uniform)**

| Parameter | Value |
|---|---|
| $w$ | 8 |
| $T_{u/r}$ | 40 |
| $T_{u/w}$ | 40 |

Table 2 compares effective memory bandwidth achieved by the natural versus ordered access sequence for a range of scientific kernels. For all computations the depth of loop unrolling is 4, data is double-precision, and vectors are aligned to module $M_0$.

The *daxpy* and *dvaxpy* computations are double-precision versions of the axpy and vaxpy computations, respectively, discussed earlier. The remaining computations are selections from the Livermore Loops [Mcma90]. This set of scientific kernels serves as the benchmark suite for all subsequent simulations.

**Table 2 Natural vs Ordered Performance (Uniform)**

| Computation | Natural $BW$ | Ordered $BW$ | % Increase |
|---|---|---|---|
| daxpy | 299.4 | 800.0 | 167.2 |
| dvaxpy | 399.2 | 800.0 | 100.4 |
| LL-1 | 300.0 | 800.0 | 166.7 |
| LL-3 | 400.0 | 800.0 | 100.0 |
| LL-4 | 398.4 | 800.0 | 100.8 |
| LL-5 | 200.0 | 800.0 | 300.0 |
| LL-7 | 266.7 | 800.0 | 200.0 |
| LL-11 | 200.0 | 800.0 | 300.0 |
| LL-12 | 398.4 | 800.0 | 100.8 |
| LL-20 | 450.0 | 800.0 | 77.8 |
| LL-21 | 300.0 | 800.0 | 166.7 |
| LL-22 | 333.3 | 800.0 | 140.0 |

Access ordering improves performance over the natural access sequence for the given computations from 78% to 300%. Where memory bandwidth is the bottleneck, computation rate increases accordingly.

Table 3 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 3 Analytic vs Simulation Results (Both)**

| Computation | Uniform-access | | Page-mode | |
|---|---|---|---|---|
| | Analysis $BW$ | Simulation $BW$ | Analysis $BW$ | Simulation $BW$ |
| daxpy | 800.0 | 800.0 | 127.9 | 127.9 |
| dvaxpy | 800.0 | 800.0 | 121.8 | 121.8 |
| LL-1 | 800.0 | 800.0 | 100.9 | 100.9 |
| LL-3 | 800.0 | 800.0 | 106.5 | 106.5 |
| LL-4 | 800.0 | 800.0 | 106.3 | 106.1 |
| LL-5 | 800.0 | 800.0 | 100.9 | 101.0 |
| LL-7 | 800.0 | 800.0 | 102.3 | 102.3 |
| LL-11 | 800.0 | 800.0 | 98.3 | 98.3 |
| LL-12 | 800.0 | 800.0 | 98.3 | 98.3 |
| LL-20 | 800.0 | 800.0 | 102.7 | 102.7 |
| LL-21 | 800.0 | 800.0 | 124.7 | 123.4 |
| LL-22 | 800.0 | 800.0 | 99.9 | 99.9 |

#### 5.3.3.2 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 4 and are representative of the IPSC/860 node memory system [Moye91].

Table 5 compares effective memory bandwidth achieved by the natural versus ordered access sequence for the benchmark kernels. The depth of loop unrolling is 4, data is double-precision, and all vectors are aligned to module $M_0$.

For this system, access ordering improves performance over the natural access sequence for the given computations from 60% to 189%.

**Table 4 Module Parameters (Page)**

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

Table 3 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 simulation, measured performance falls below the theoretical lower bound for several computations.

### 5.3.4 Summary

Section 5.3 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.

**Table 5 Natural vs Ordered Performance (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 |

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 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 a range of scientific kernels. Performance models are validated.

Recall that modules in an interleaved system may be buffered, as depicted in Figure 3. 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.

## 5.4 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 to 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.

Results for the optimal access of independent streams are first derived for use in the general ordering algorithms. Access ordering algorithms and performance predictors are then developed for systems of uniform-access and page-mode components, respectively. The effectiveness of access ordering and accuracy of performance models are demonstrated via simulation.

### 5.4.1 Optimal Access of Independent Streams

Given a set $S$ of independent streams, knowledge of stream alignments 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.

In generating an optimal sequence of accesses, the depth of loop unrolling $b$ is restricted to values such that for each successive loop iteration, the first access to each stream refer-

ences the same module as the first access from the previous iteration. Restricting *b* in this manner guarantees a repetitive sequence of module references.

For stream $t_i$, if the number of accesses $\varepsilon_i$ is a multiple of the number of modules referenced $\mu_i$ then each module referenced services $\psi_i$ accesses per iteration in a repetitive sequence. Thus, in generating an optimal access sequence for streams *S*, *b* is restricted such that

$$\forall (t_i \in S) \qquad \mu_i \mid \varepsilon_i \tag{6}$$

For most scientific codes, the number of accesses per iteration to 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 substitution into (6) yields the restriction

$$\forall (t_i \in S) \qquad \mu_i \mid b$$

Since the number of modules referenced by a given stream is a power of 2, as derived in section 5.2.1, *b* is restricted to a multiple of the maximum number of modules accessed by any stream; i.e.

$$b = k\,(max(\mu_1, \ldots, \mu_n)) \leq km \qquad k \in Z^+$$

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 depth of loop unrolling *b* satisfying the restriction (6), 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

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

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, $A_0 = \{a_1{:}\psi_1, a_3{:}\psi_3, a_4{:}\psi_4\}$ specifies that module $M_0$ satisfies $\psi_1$ accesses from stream $t_1$, followed by $\psi_3$ accesses from stream $t_3$ followed by $\psi_4$ 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 6 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 $\mu_i > \mu_j$. Forming $A_0, \ldots, A_{m-1}$ in this manner leads to the following result, derived in Appendix B.1:

**Theorem 4:** For a set of streams $S$ and sequences $A_0, \ldots, A_{m-1}$ derived via the Module Sequence algorithm, each round robin selection of accesses from $A_0, \ldots, A_{m-1}$ has the property that for each stream $t_i$ referenced: there are exactly $\mu_i$ accesses to $t_i$, and accesses from different streams do not conflict.

From Theorem 4 it is easily seen that given $A_0, \ldots, A_{m-1}$ as derived via the MSA, optimal access is defined by the sequence

$$[A_0, \ldots, A_{m-1}]$$

Accesses from $A_0, \ldots, A_{m-1}$ are initiated round robin, and for each round robin sequence of accesses it is observed that

- for each stream $t_i$ referenced $\mu_i$ accesses are initiated, maximizing concurrency for stream $t_i$,
- accesses from different streams do not conflict, maximizing concurrency between accesses from different streams, and

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

```
// 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 access to t_i to the sequence A_j

      A_j ← A_j ∪ {a_i: ψ_i}
```

**Figure 6 Module Sequence Algorithm**

- the total number of accesses is equal to the number of modules that service the remaining accesses, maximizing module utilization.

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

**Theorem 5:** For a set of independent streams $S$ and sequences $A_0, \ldots, A_{m-1}$ derived via the Module Sequence algorithm, $[A_0, \ldots, A_{m-1}]$ is an optimal access sequence.

To illustrate, an optimal access sequence is derived for 3 read streams $t_x$, $t_y$ and $t_z$. 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: $A_0 = \{r_y, r_z\}$, $A_1 = \{r_x\}$, $A_2 = \{r_y, r_z\}$ and $A_3 = \{r_x\}$. The resulting optimal access sequence $[A_0, A_1, A_2, A_3]$ defines the sequence of references

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

The above sequence is annotated to illustrate both the round robin selection of accesses $[A_0, A_1, A_2, A_3]$ 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 $[A_0, \ldots, A_{m-1}]$ 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 $t_x$.

#### 5.4.1.1 Request Buffering

For an interleaved system, modules may be buffered as depicted in Figure 3. Ordering accesses as above results in a sequence that references each module at most once per round robin selection of accesses $[A_0, \ldots, A_{m-1}]$. If individual accesses require an equal amount of time to complete, then the sequence $[A_0, \ldots, A_{m-1}]$ 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 $[A_0, \ldots, A_{m-1}]$ 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 $[A_0, \ldots, A_{m-1}]$ results in optimal performance.

### 5.4.2 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 a MAP 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 5.4.1.

For a set of streams *S*, $S_r$ is defined as the subset of all read streams, $S_w$ the subset of all write streams and *b* a depth of loop unrolling that satisfies the restriction defined in 5.4.1 for all streams in *S*. 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

$$\{\, [P_0, \ldots, P_{m-1}],\ [Q_0, \ldots, Q_{m-1}] \,\} \tag{7}$$

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.

#### 5.4.2.1 Performance Predictor

For a MAP consisting of streams *S* and an access sequence as defined above, a performance predictor is derived for the average time per access $T_{avg}$ 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 (7), $P_0, \ldots, P_{m-1}$ represent the relative sequence 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 (7). An exact model of performance can not be expressed as a closed form equation.

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

$$T_{avg} = \frac{T_{tot}}{b \sum_{t_i \in S} t_i.\sigma}$$

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.

$$BW = \frac{10^3 b \sum_{t_i \in S} [\,(t_i.d)\ (t_i.\sigma)\,]}{T_{tot}}$$

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

### 5.4.3 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 5.4.2 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.

#### 5.4.3.1 A Base Access Sequence

In defining a MAP 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 5.4.1.

For a set of streams *S*, $S_r$ is defined as the subset of all read streams, $S_w$ the subset of all write streams and *b* a depth of loop unrolling that satisfies the restriction defined in 5.4.1

for all streams in *S*. 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

$$\{ [P_0, \ldots, P_{m-1}], [Q_0, \ldots, Q_{m-1}] \} \quad (8)$$

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.

#### 5.4.3.2 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, 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 (8), sequences $P_0, \ldots, P_{m-1}$ and $Q_0, \ldots, Q_{m-1}$ 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.

#### 5.4.3.3 Access Ordering Algorithm

For a set of stream *S* with no pair of streams implementing a read-modify-write, a MAP access sequence is defined by the base sequence (8). Otherwise, a MAP access sequence in the form of the base sequence is derived as follows:

> Given a set of 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 $P_0, \ldots, P_{m-1}$ and $Q_0, \ldots, Q_{m-1}$, respectively, that results in the minimum completion time for all accesses in a given iteration of the base sequence.

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 5.4.3.5.

Determining the order for stream mappings that results in minimum average completion time for the base sequence (8) 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.

#### 5.4.3.4 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 four streams defined by $t_a = (a, s_a, d_a, r):1$, $t_x = (x, s_x, d_x, r):1$, $t_{y_r} = (y, s_y, d_y, r):1$, and $t_{y_w} = (y, s_y, d_y, w):1$.

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 depth of loop unrolling *b* is 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 (8):

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

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.

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

$$\{r_a, r_a, r_x, r_x, r_{y_r}, r_{y_r}, w_{y_w}, w_{y_w}\}$$

#### 5.4.3.5 Performance Predictor

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

Functions modeling page overhead derived in section 4 for a single module system are applicable to accesses at individual modules of an interleaved system. Recall that in general, average page overhead is modeled by the function $\eta(s, d, c, V)$. For stream accesses that are wrap-around adjacent or intermixed, average page overhead 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 base sequence (8), $P_0, \ldots, P_{m-1}$ 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 $\{r_i : \psi_i\}$. Similarly, $Q_k$ is the sequence of write accesses serviced at $M_k$ and $Q_{(i,k)}$ represents the write access set $\{w_i : \psi_i\}$. Recall that as a result of the loop unrolling restriction defined in 5.4.1, all modules referenced by a given stream service the same number of accesses from that stream each iteration.

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

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

Note that in modeling page overhead, 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 $\omega_k(\xi_i, t_i.d, \psi_i)$ correctly models page overhead. Otherwise, $\eta_k(\xi_i, t_i.d, \psi_i, V)$ 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 $\psi_i$ multiplied by the page-hit write cycle time $T_{p/w}$ and the average page overhead multiplied by the page miss time $T_{p/m}$, so that

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

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 base sequence (8) 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 base sequence 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 base sequence proceeds without access gaps that result in increased completion time for that phase; this is discussed in section 5.4.1.1.

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 access $T_{avg}$ is computed as the time to complete all accesses in a given iteration divided by the number of data items referenced, resulting in

$$T_{avg} = \frac{T_{tot}}{b \sum_{t_i \in S} t_i.\sigma}$$

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.

$$BW = \frac{10^3 b \sum_{t_i \in S} [\,(t_i.d)\ (t_i.\sigma)\,]}{T_{tot}}$$

### 5.4.4 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 5.3.3 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 four streams defined by $t_a = (a, s_a, d_a, r):1$, $t_x = (x, s_x, d_x, r):1$, $t_{y_r} = (y, s_y, d_y, r):1$, and $t_{y_w} = (y, s_y, d_y, w):1$.

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 non-buffered 4 module system of page-mode components with module parameters as previously defined in Table 4 for the IPSC/860 node architecture. 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 6 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 unrolling 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 schedule to proceed concurrently, resulting in increased performance over the case where alignment is unknown.

**Table 6 Simulation and Analytic Results (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 |

### 5.4.5 Summary

Section 5.4 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 3. 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 5.4.1.1. If buffering is not sufficient, performance is degraded and performance predictors are no longer guaranteed to represent a lower bound.

# 6 Implementation Issues

Addressing all the implementation issues associated with access ordering is beyond the scope of this report. However, several important topics are briefly discussed below; a more complete treatment of these issues can be found in [Moye92b].

Access ordering employs loop unrolling which 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 vectors registers, effectively increasing register file size for vector computations. Essentially, storage is defined for a set of pseudo vector registers and placed in cache via a standard (caching) load instruction. Vector operands are loaded

into the pseudo registers, arithmetic operations are performed, and pseudo 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.

Intermixing can reduce page overhead for read-modify-writes on systems constructed from page-mode components. However, alternating read and write accesses can force scalar-mode (non-pipelined) arithmetic operations. Intermixing is justified if the additional access time resulting from a non-intermixed reference sequence exceeds the additional cost of performing scalar-mode computation.

Access ordering employs non-caching memory instructions to control the sequence of requests observed by the memory system. Though the effectiveness of cache memory for numeric codes is still the topic of much research, many codes do benefit from caching with careful application of iteration space tiling. Thus caching and access ordering should be used together as complementary techniques, caching multiply accessed blocks of data and ordering non-caching accesses to single-visit data items.

Finally, to simplify analysis and obtain optimality results, ordering algorithms derived presume access streams adhere to the stream interaction restriction. Minor relaxation of this restriction to accommodate self-antidependence cycles and read streams with intersecting address spaces allows algorithms to be applied to the set of vectorizable loops. Self-antidependence cycles are accommodated by ordering accesses from each stream independently and insuring that all reads are initiated prior to the first write. Read streams with intersecting address spaces are accommodated by simply ordering streams independently, as input dependence can be ignored for non-volatile memory locations.

# 7 Conclusions

Access ordering, a loop optimization that reorders accesses to better utilize memory system resources, is a compiler technology developed in this report to address the memory bandwidth problem for scalar processors executing scientific codes. For an interleaved

memory architecture, the access ordering algorithms developed here determine a well-defined interleaving of vector references that maximizes effective bandwidth for a given computation and memory device type. 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.

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 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. Access ordering is also complementary to caching, and is known to work well with strip-mining and tiling techniques.

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.

# Appendix A

## Intermix Sequences

### A.1 Derivation of $\rho(s, d, c)$

The function $\rho(s, d, c)$ is the average page miss count in performing each set of $c$ write accesses in the intermix sequence $\{\ldots, \{r_i{:}\, c, w_j{:}\, c\}{:}\, h, \ldots\}$, where $t_i$ and $t_j$ specify a read-modify-write operation; i.e. $t_i.v = t_j.v$.

**Case**: $\gamma(s, d) = 1$ (the number of data items per word is exactly one)

In deriving $\rho(s, d, c)$, the following observation is made: in accessing $c$ data items 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 $\{\ldots, \{r_i{:}\, c, w_j{:}\, c\}{:}\, h, \ldots\}$, the write operations $w_j^{(k-1)\,c+1}$ through $w_j^{kc}$, $1 \le k \le h$, suffer a page miss only when $r_i^{(k-1)\,c+1}$ and $r_i^{kc}$ reference a different page.

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

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

The number of starting positions resulting in the $c$ 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 touch 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 sequence of $c$ write accesses must begin with a page miss and page overhead is modeled as

$$1 + \frac{c-1}{\phi(s, d)}$$

which is one plus the remaining data items to access, $c-1$, divided by the number of data items per page.

Combining the results derived above

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

**Case**: $\gamma(s, d) > 1$ (the number of data items per word is greater than one)

Deriving $\rho(s, d, c)$ for this case is completely analogous to the previous case, with the address space spanned being $cw = c\gamma(s, d)sd$ and all accesses being word-aligned, so that

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

The two cases derived above may be combined into the single modeling function

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

## A.2 Proof of Optimal Intermix Pattern

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

**Prove**: the intermix sequence $\{\ldots, \{r_i: c, w_j: c\}: h, \ldots\}$ is the optimal interleave pattern.

**Proof**: Consider the general interleave case

$$\{\ldots, r_i: q_1, w_j: k_1, \ldots, r_i: q_n, w_j: k_n, \ldots\}$$

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

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

Then let

$$\sum_{l=1}^{\lambda} q_l = {}_qS_\lambda \qquad \text{and} \qquad \sum_{l=1}^{\lambda} k_l = {}_kS_\lambda$$

It is easily seen that for $\lambda < n$, ${}_qS_\lambda \geq {}_kS_\lambda$. If there exists a $q_l \neq k_l$ then there must exist at least one $u$ such that ${}_qS_u > {}_kS_u$, in which case

let $u_q = {}_qS_u$ and $u_k = {}_kS_u$, then

- the page miss count in performing the read sequence $\{\ldots, r_i{:}\, q_{u+1}, \ldots\}$ can be greater than in the case where ${}_qS_u = {}_kS_u$ since $w_j^{u_k}$ may access a sequentially earlier page than $r_i^{u_q}$;
- similarly, the page miss count in performing the write sequence $\{\ldots, w_j{:}\, k_{u+1}, \ldots\}$ can be greater than in the case where ${}_qS_u = {}_kS_u$ as $w_j^{u_k+1}$ may access a sequentially earlier page than $r_i^{u_q+1}$.

Thus, the minimum page miss count is achieved when ${}_qS_u = {}_kS_u$ for $u \leq n$; i.e. when $q_l = k_l$ for $1 \leq l \leq n$.

$\therefore \{\ldots, \{r_i{:}\, c, w_j{:}\, c\} : h, \ldots\}$ is the optimal intermix pattern.

QED

# Appendix B

## Module Sequence Algorithm

### B.1 Properties of the MSA

**Given**: $N$ streams $t_1, \ldots, t_N$ such $\mu_1 \geq \ldots \geq \mu_N$ 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 $\mu_i$ accesses to $t_i$, and
2. accesses from different streams 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 the sequences $A_k$ 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 mapped to sequences $A_k$ 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 $\mu_i$ 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 sequences $A_k$ 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 $A_0, \ldots, A_{m-1}$, references from different streams do not conflict.

QED

# Bibliography


[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.

[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.

[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.

[Harp89] Harper-D, “Address Transformations to Increase Memory Performance”, Proc. 1989 Intl. Conf. Parallel Processing, 1989, pp. 237-241.

[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.

[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.

[Moye92a] Moyer-S, “Access Ordering Algorithms for a Single Module Memory”, University of Virginia, IPC-TR-92-002, 1992.

[Moye92b] Moyer-S, “Access Ordering and Effective Memory Bandwidth”, Ph.D. Dissertation *in progress*, Computer Science Department, University of Virginia.

[Quin91] Quinnell-R, “High-speed DRAMs”, EDN, May 23, 1991, pp. 106-116.

[Rau91] Rau-B, “Pseudo-Randomly Interleaved Memory”, Proc. 18th Intl. Symp. Comput. Architecture, 1991, pp. 74-83.

[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.