Access Ordering Algorithms for a Multicopy Memory

Steven A. Moyer

IPC-TR-92-013

December 18, 1992

Access Ordering Algorithms for a Multicopy 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. A *multicopy* memory is proposed here as a parallel memory system consisting of replicated data that increases the potential for access concurrency and reduced page overhead for systems of page-mode DRAMs. Simulation results indicate that a multicopy system can provide increased bandwidth over an equivalent interleaved memory for computations with a high read to write access ratio; nearly an order of magnitude better performance is achieved in some benchmarks. Furthermore, multicopy access can be implemented with a minimal increase in hardware complexity as part of a heterogeneous interleaved-multicopy memory architecture.

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
σ	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^{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:

ε_i number of accesses per loop iteration

θ_i intermix factor

Properties of stream t_i for a multicopy architecture:

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

$\hat{\xi}_i$ module stride

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.

A *multicopy* memory is proposed here as a parallel memory system consisting of m modules of replicated data such that if $*(M_k, a)$ represents the contents of address a at module M_k, then $*(M_0, a) = \ldots = *(M_{m-1}, a)$. A multicopy memory system increases the potential for access concurrency, as maximum concurrency is achievable for all strides of reference. Furthermore, for systems of page-mode DRAMs, page overhead can be more effectively amortized by directing stream accesses to a smaller number of modules.

Access ordering algorithms [Moye92c] are developed that exploit a multicopy memory. Access ordering is a loop optimization that reorders non-caching accesses to better utilize

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

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 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 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’.

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 a multicopy 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.

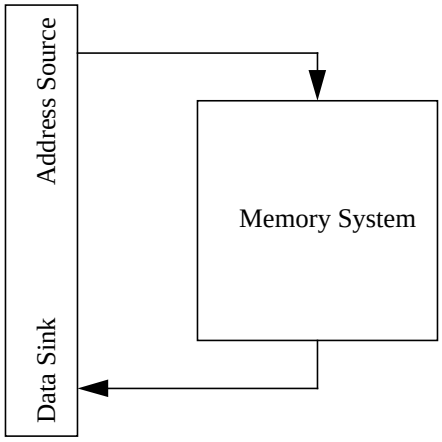

Figure 1 General System Model

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

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

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

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

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:

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

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

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 necessary to characterize the contributions of this work; a more complete survey of all relevant topics can be found in [Moye92c].

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 imple-

mented 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)

σ = 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 α_i accesses from A_i until all accesses in $A_1, \ldots, A_n$ have been initiated. If fewer than α_i accesses remain in A_i, then only these accesses are issued. When all accesses specified in A_i have been initiated A_i is dropped from the pattern.

   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^{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 implementing 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 a parallel memory system, 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

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

- 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 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^{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 anti-dependent 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 a multicopy 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 a multicopy 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 ε_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 θ_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 Multicopy Architecture Analysis

Access ordering algorithms and performance predictors are now derived for a multicopy memory system as depicted in Figure 3. A *multicopy* memory is a proposed parallel memory architecture consisting of *m* modules of replicated data such that if $*(M_k, a)$ represents the contents of address *a* at module M_k, then $*(M_0, a) = \ldots = *(M_{m-1}, a)$.

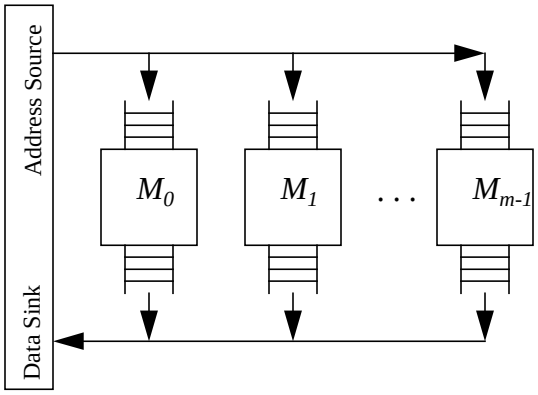

Figure 3 Multicopy Architecture

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

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.

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

The following sections discuss the problem space for efficient utilization of a multicopy memory and notation is developed for expressing the mapping of read accesses to modules. Access ordering algorithms and performance predictors are derived for a multicopy system of uniform-access and page-mode components, respectively. The effectiveness of a multicopy architecture and accuracy of performance predictors are demonstrated via simulation.

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.

For a multicopy memory, ordering reads to maximize concurrency is a matter of distributing accesses uniformly across modules. Write accesses are broadcast to all modules so that concurrency is not an issue.

Techniques for minimizing page overhead come directly from analytic results derived in section 4 for a single memory module. Page overhead for a given stream is minimized if elements of that stream are referenced consecutively from a single module on each iteration. 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 usually requires a trade-off between minimizing page overhead and maximizing concurrency.

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

5.2 Module Access Notation

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

5.3 Multicopy Storage and Uniform-access Components

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

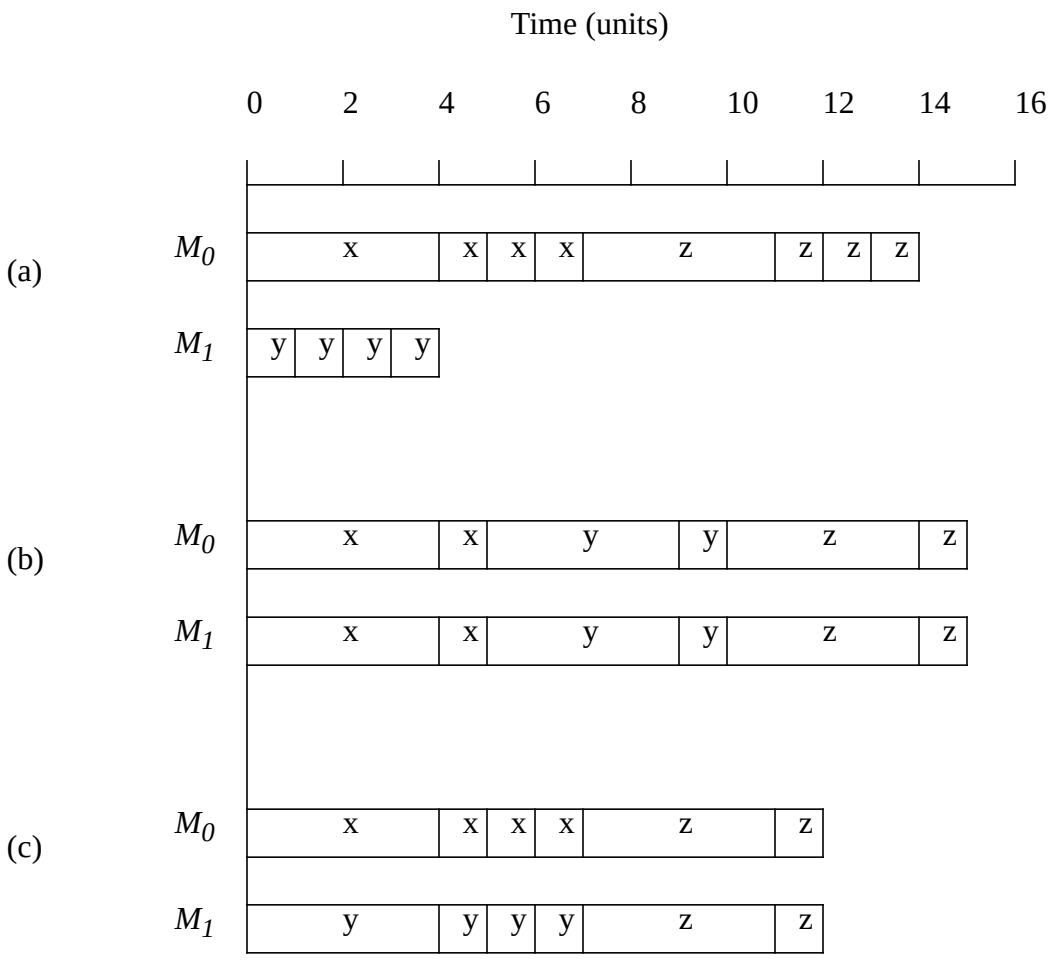

Figure 4 Minimizing Completion Time

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

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

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

Then an optimal access sequence is

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

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

5.3.1 Performance Predictor

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

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

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

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

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

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

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, i.e.

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

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

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

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

5.4 Multicopy Storage and Page-mode Components

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

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

5.4.1 A Base Access Sequence

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

$$\{\, [A_0, \ldots, A_{m-1}]\,, w_{N_r+1}{:}\varepsilon_{N_r+1}, \ldots, w_N{:}\varepsilon_N\} \tag{1}$$

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

5.4.1.1 Request Buffering

For a multicopy 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 $[A_0, \ldots, A_{m-1}]$. Since individual access times vary, the sequence $[A_0, \ldots, A_{m-1}]$ provides maximum 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. For this analysis, buffering is assumed sufficient so that $[A_0, \ldots, A_{m-1}]$ results in maximum performance for that sequence.

5.4.2 A Module Reference Model

For $A_0, \ldots, A_{m-1}$ defined in the base access sequence (1), assume that references from each read stream $t_i \in S$ are distributed uniformly among some number of sequences, and hence modules, $\hat{\mu}_i$. Furthermore assume all accesses from t_i in a sequence A_k are arranged consecutively. Such a sequence arises from the mapping heuristic derived in section 5.4.4.

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

To illustrate, 4 accesses from a stream t_a and 2 from a stream t_b are mapped to a 2 module system as depicted in Figure 5. For the references of Figure 5(a), elements of stream t_a are accessed alternately from each module so that the observed stride at a given module is $2\,(t_a.s)$. Such a mapping results from sequence $[\{r_{(a,0)}:2\},\ \{r_{(a,1)}:2, r_{(b,1)}:2\}]$.

Alternatively, references depicted in Figure 5(b) access consecutive elements of t_a at each module so that the observed stride is $t_a.s$. Such a mapping results from the sequence $[\{r_{(a,0)}:2\},\ \{r_{(b,1)}:2, r_{(a,1)}:2\}]$.

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

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

$$\hat{\xi}_i = \hat{\mu}_i\,(t_i.s)$$

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

5.4.3 Greedy 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 (1), potentially reducing completion time for all accesses. Note that in this context, intermixing refers to read

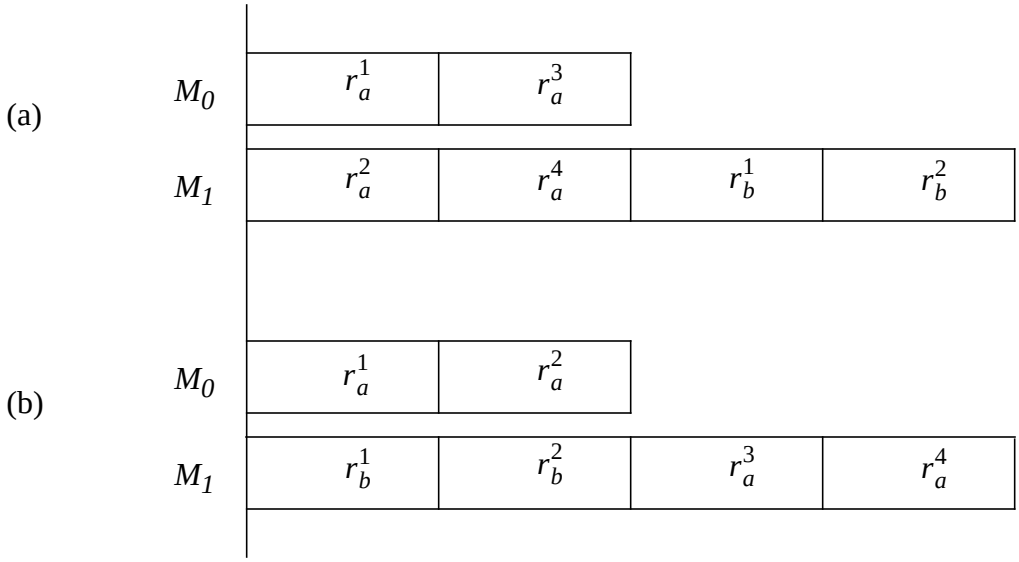

Figure 5 Dependence of Module Stride on Reference Pattern

accesses immediately preceding corresponding write accesses at a given module; read and write operations are not interleaved.

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

Intermixing and wrap-around adjacency are employed in a greedy fashion by choosing prior to access mapping the two pair of streams most likely to benefit from these relationships. From streams *S*, $t_{r\text{-}wadj}$ and $t_{w\text{-}wadj}$ are a pair of read and write streams, respectively, designated to be mapped for wrap-around adjacency. Similarly, $t_{r\text{-}imix}$ and $t_{w\text{-}imix}$ are designated to be mapped for intermixing. All else being equal, streams with the smallest stride have the lowest average page miss count per access and hence the most to gain; $t_{r\text{-}wadj}$ ($t_{w\text{-}wadj}$) and $t_{r\text{-}imix}$ ($t_{w\text{-}imix}$) are chosen accordingly. For *S* consisting of fewer

than two read-modify-write operations, $t_{r\text{-}wadj}$ and $t_{r\text{-}imix}$ are chosen in that order, with one or both remaining undefined.

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

5.4.4 Read Mapping Heuristic

For read sequences $A_0, \ldots, A_{m-1}$ of the base sequence (1), an optimal mapping of read accesses to modules usually requires a trade-off between minimum page overhead and maximum concurrency as discussed in section 5.1. Minimizing completion time results in a balanced load of accesses such that if $T(A_k)$ is the time to complete accesses in the sequence A_k, then

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

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

The Read Mapping Heuristic (RMH) derived below approximates an optimal solution as follows. For a read stream t_i, accesses are mapped uniformly to a number of modules $\hat{\mu}_i$ proportional to the ratio of the minimum time to complete accesses to t_i at a single module and the minimum time to complete all read accesses at a single module. Essentially, each stream is assigned resources proportional to the amount of work to be completed; over-allocation limits the amortization of page overhead while under-allocation limits concurrency. Load balancing is performed in a greedy fashion by mapping accesses from t_i to the $\hat{\mu}_i$ modules with the minimum load. Page overhead is minimized at each module as references to t_i are initiated consecutively.

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

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

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

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

$$\Delta = \sum_{\substack{t_i \in S \\ t_i.m = r}} \Gamma_i(t_i.s, \varepsilon_i)$$

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

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

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

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

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

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

Figure 6 presents the complete read mapping heuristic (RMH). To summarize, for each read stream $t_i \in S$

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

5.4.4.1 RMH Performance

Table 1 compares results of the RMH with an optimal mapping of read accesses as determined via exhaustive search. The general form of the problem mapped is

$$\forall i \qquad y(i) = fn(x_1(i), \ldots, x_n(i)) \tag{2}$$

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

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

if  R ≤ m

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

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

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

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

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

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

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

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

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

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

Figure 6 Read Mapping Heuristic (RMH)

Table 1 contains ratios of RMH to optimal performance, where performance is defined as the average time to complete *all read operations* for a given iteration. Only read accesses are considered to avoid skewing results in favor of the RMH, as write accesses in the general problem (2) take the same time regardless of mapping.

Table 1 RMH / Optimal Performance Ratios

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

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

5.4.5 Access Ordering Algorithm

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

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

The complete access ordering algorithm consists of the following steps:

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

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

5.4.6 Example Problem

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

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

and generates the three streams defined by $t_x = (x, s_x, d_x, r):1$, $t_{y_r} = (y, s_y, d_y, r):1$, and $t_{y_w} = (y, s_y, d_y, w):1$.

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

The initial step identifies streams for intermixing and wrap-around adjacency, as discussed in 5.4.3. For the axpy computation, $t_{y_r} = t_{r\text{-}wadj}$ and $t_{y_w} = t_{w\text{-}wadj}$; $t_{r\text{-}imix}$ and $t_{w\text{-}imix}$ are undefined.

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

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

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

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

Then the time to complete all read accesses Δ is

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

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

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

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

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

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

$$\{ [r_{(y_r, M_0)} : 2, r_{(x, M_1)} : 2], \{w_{y_w} : 2\} \}$$

representing the linear sequence of references

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

Figure 7 depicts a typical iteration of the above sequence, assuming an access to the current page requires 1 time unit $(T_{p/r}, T_{p/w})$ and a page miss incurs an additional 2 time unit penalty $(T_{p/m})$.

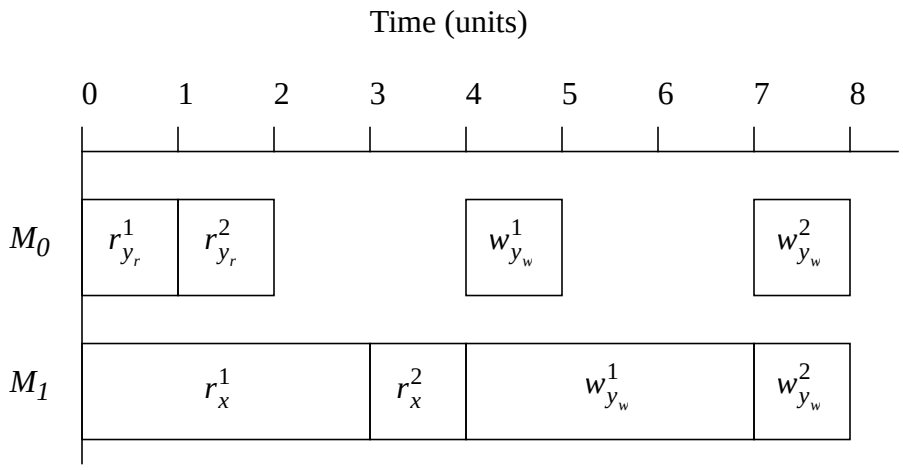

Figure 7 Multicopy Example

5.4.7 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. Recall that as a result of the module reference model developed in section 5.4.2, performance models represent estimated performance rather than bounds.

Functions modeling page overhead derived in section 4 for a single module system are applicable to accesses at individual modules of a multicopy 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 a multicopy system, stride *s* is module stride.

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

represents the write access set $\{w_i : \varepsilon_i\}$; recall that writes are broadcast so that each module services all ε_i accesses from write stream t_i.

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

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

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 $\omega_k(\hat{\xi}_i, t_i.d, c)$ models page overhead. Otherwise, $\eta_k(\hat{\xi}_i, t_i.d, c, V)$ is the applicable model where the number of vectors V is the number accessed at module M_k. For clarity, functions modeling page overhead are subscripted with the module number to denote context. Note that for a wrap-around adjacent access set, the page overhead $\omega_k(\hat{\xi}_i, t_i.d, c)$ is an upper-bound representative of the overhead at the module servicing the $\hat{\mu}_i^{th}$ access from read stream t_i; this effect is a consequence of distributed reads and broadcast writes.

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

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

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

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

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

Then the time to complete all read accesses in an iteration of the base sequence (1) is the maximum time to complete read operations at any module, so that

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

Note the tacit assumption in computing T_r is that buffering is sufficient so that read accesses proceed without access gaps that result in increased completion, as discussed in section 5.4.1.1.

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

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

And the time to complete all write operations in an iteration of the base sequence is

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

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

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

From the above, the average time per 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.

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

5.5 Simulation Results

For a multicopy memory system there is no ‘natural’ mapping of accesses to modules. Thus the quality of the access ordering algorithm is best captured by comparison with an optimal reference sequence; such a comparison is presented in section 5.4.4.1. For reference sequences generated by the ordering algorithm, simulation results are presented to validate the accuracy of the performance models.

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

5.5.1 Performance Predictors

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

Table 2 Module Parameters (Both)

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

Table 3 compares performance of ordered accesses as calculated analytically and measured via simulation for a range of scientific kernels. For all computation the depth of loop unrolling is 4 and data is double-precision.

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.

For the computations and conditions modeled, analytic and simulation results differ by less than 1%. Two exceptions are highlighted. Recall from section 5.4.1.1 that in modeling performance for read operations, buffering is assumed sufficient so that accesses proceed at the maximum rate. For both cases noted, a non-buffered system results in access gaps that reduce performance; for a buffer size of 1, simulated performance achieves the predicted bandwidth.

Table 3 Analytic vs Simulation Results (Both)

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

5.5.2 Evaluation of Multicopy Performance

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

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

Table 4 Multicopy vs Interleaved (4:1)

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

Table 4 presents simulation results comparing bandwidth delivered by a 4 module multicopy system with buffer depth 1 to an equivalent interleaved system for the set of benchmark kernels; the depth of loop unrolling is 4 for all computations. Module parameters are those of Table 2 with a page miss versus hit cycle time ratio of 4:1, typical of current DRAMs. For the interleaved system, access ordering is performed assuming known alignment [Moye92b] to achieve maximum bandwidth.

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

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

Table 5 Module Parameters (Page)

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

Assume a 4 module multicopy system with buffer depth 1 and an equivalent interleaved system. Module parameters are defined in Table 5 with a page miss-hit cycle time ratio of

10:1. Table 6 presents simulation results comparing bandwidth achieved for the set of benchmark computations; depth of loop unrolling is 4 in all cases.

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

Table 6 Multicopy vs Interleaved (10:1)

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

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

5.5.3 Evaluation of Multicopy Cost

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

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

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

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

5.6 Summary

Access ordering algorithms are derived for a proposed multicopy architecture. Performance predictors are developed for the effective memory bandwidth achieved by ordered accesses.

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

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

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

6 Implementation Issues

Addressing all the implementation issues associated with access ordering for a multicopy memory 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 [Moye92c].

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.

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

A *multicopy* memory is proposed here as a parallel memory system consisting of m modules of replicated data such that if $*(M_k, a)$ represents the contents of address a at module M_k, then $*(M_0, a) = \ldots = *(M_{m-1}, a)$. A multicopy memory system increases the

potential for access concurrency, as maximum concurrency is achievable for all strides of reference. Furthermore, for systems of page-mode DRAMs, page overhead can be more effectively amortized by directing stream accesses to a smaller number of modules.

Access ordering algorithms are developed that exploit a multicopy memory. Access ordering is a loop optimization that reorders non-caching accesses to better utilize memory system resources. For a multicopy 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 indicate that a multicopy system of page-mode components can provide increased bandwidth over an equivalent interleaved memory for computations with a high read to write access ratio; nearly an order of magnitude better performance is achieved in some benchmarks. Furthermore, multicopy access can be implemented with a minimal increase in hardware complexity as part of a heterogeneous interleaved-multicopy memory architecture.

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

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 Algorithms for an Interleaved Memory”, University of Virginia, IPC-TR-92-012, 1992.

[Moye92c] 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.