Access Ordering Algorithms for a Single Module Memory

Steven A. Moyer

IPC-TR-92-002

Revised: December 18, 1992

Access Ordering Algorithms for a Single Module 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. A number of compiler algorithms have been developed that schedule loop accesses so as to overlap memory latency with computation, reducing load/store interlock delay. However, none of these algorithms address the memory bandwidth issue directly. *Access ordering* is a compilation technique presented here that addresses the memory bandwidth problem in the context of scientific computing. Access ordering algorithms are derived for a single module 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

1 Introduction 1
- 1.1 General System Model 2
- 1.2 Access Ordering Observation 3
- 1.3 Computation Domain 5
- 1.4 Memory Device Types 6
- 1.5 Performance Modeling 7

2 Previous Work 7
- 2.1 Stream Detection 7
- 2.2 Access Scheduling Techniques 8

3 Model Access Pattern 9
- 3.1 MAP Notation 9
- 3.2 Definitions and Assumptions 10
- 3.3 Wide Word Restrictions 11
- 3.4 Stream Interaction Restriction 12
- 3.5 MAP Dependence Relations 13
  - 3.5.1 Output and Input Dependence 14
  - 3.5.2 Antidependence 14
  - 3.5.3 Data Dependence 14
  - 3.5.4 Dependence Rules 15
  - 3.5.5 Other Dependencies 15

4 Single Module Architecture Analysis 16
- 4.1 Minimizing Page Overhead 16
  - 4.1.1 Intermixing 18
    - 4.1.1.1 Intermix Factor 19
  - 4.1.2 Wrap-around Adjacency 20
- 4.2 Single Module of Uniform-access Components 21
  - 4.2.1 Performance Predictor 22
- 4.3 Single Module of Page-mode Components 23
  - 4.3.1 Example Problem 24
  - 4.3.2 Performance Predictor 25

5 Simulation Results 27

6 Implementation Issues 29
- 6.1 Relieving Register Pressure 30
- 6.2 Pipelined Processors and Bus Bandwidth 31
- 6.3 Combining Caching and Non-Caching Memory Access 33
- 6.4 Relaxation of the Stream Interaction Restriction 35
  - 6.4.1 Self-Antidependence Cycles 35
  - 6.4.2 Overlapping Read Address Spaces 36
  - 6.4.3 Access Ordering and Vectorizable Computations 36

7 Conclusions 36

Appendix A 38

Bibliography 42

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

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

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

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

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

In general purpose scalar computing, the addition of cache memory is often a sufficient solution to the memory latency and bandwidth problems given the spatial and temporal locality of reference exhibited by most codes. For scientific computations, vectors are normally too large to cache. Iteration space tiling [CaKe89, Wolf89] can partition problems into cache-size blocks, however tiling often creates cache conflicts [LaRW91] and the technique is difficult to automate. Furthermore, only a subset of the vectors accessed will generally be reused and hence benefit from caching. Finally, caching may actually reduce the effective memory bandwidth achieved by a computation by fetching extraneous data

1. Both superscalar and VLIW architectures incorporate concurrent functional units and thus place similar demands on the memory system.

for non-unit strides. Thus, as noted by Lam *et al* [LaRW91], ‘while data caches have been demonstrated to be effective for general-purpose applications..., their effectiveness for numerical code has not been established’.

Access ordering [Moye92] is a compiler technology that addresses the memory bandwidth problem for scalar processors executing scientific codes. Access ordering is a loop optimization that reorders non-caching accesses to better utilize memory system resources. For a given computation, memory architecture, and memory device type, an access ordering algorithm determines a well-defined interleaving of vector references that maximizes effective bandwidth. Consequently, analytic models of performance can also be derived.

In this report, access ordering algorithms and performance models are derived for a single module memory system. Future reports will present ordering algorithms for parallel memory systems.

The following sections introduce access ordering and define the scope of the work presented here.

1.1 General System Model

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

The processor is presumed to implement a non-caching load instruction, ala Intel’s i860 [Inte89], allowing the sequence of requests observed by the memory system to be controlled via software. For access ordering, all memory references are assumed to be non-caching. Combining caching and non-caching accesses is discussed with other implementation issues in section 6.

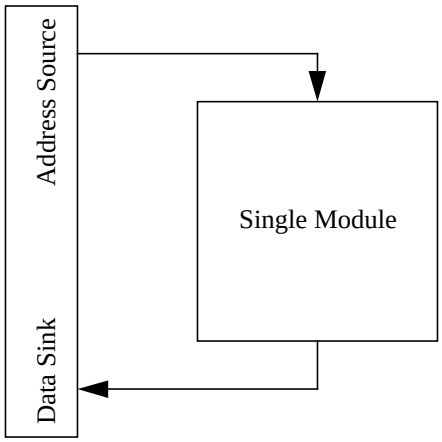

Figure 1 Single Module Architecture

1.2 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 memory system constructed from *page-mode* DRAMs. Page-mode DRAMs operate as if implemented with a single on-chip cache line, referred to as a *page*[1]. An access that does not fall within the address range of the current DRAM page forces a new page to be accessed, requiring significantly more time to service than an access that 'hits' the cached page. Thus, the effective bandwidth is sensitive to the sequence of requests. Nearly all DRAMs currently manufactured implement a form of page-mode operation [Quin91].

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

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

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.

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

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

Figure 3 depicts the effective memory bandwidth, in megabytes per second, versus depth of loop unrolling for the vaxpy computation, given page miss and hit access times of 160 and 40 nanoseconds respectively. For the curve labeled ‘Natural’ the loop body of Figure 2(a) is essentially replicated the appropriate number of times, as is standard practice. For the curve labeled ‘Ordered’, accesses have been arranged as per Figure 2(b); in doing so a performance gain of nearly 110% is realized at a depth of 4.

As noted above, access ordering employs loop unrolling to increase the number of accesses within a given loop that can be reordered. However, loop unrolling creates register pressure and has traditionally been limited by register resources. Techniques that utilize cache memory to mimic vector registers, thereby relieving processor register pressure and effectively increasing register set size, are discussed in section 6.

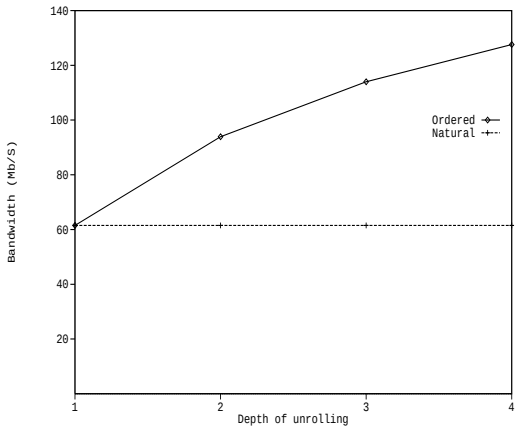

Figure 3 Vaxpy Performance

1.3 Computation Domain

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

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

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

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

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

1.4 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. For the single module architecture depicted in Figure 1, ordering algorithms are derived 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 1.2; static-column and fast page-mode DRAMs are the common examples of this device type. The performance of page-mode components is parameterized by

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

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.

1.5 Performance Modeling

For a given computation, access ordering results in code that generates a well-defined sequence of vector references. Consequently, for each access ordering algorithm, an analytic model of effective memory bandwidth can be derived.

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

Developing an access ordering algorithm for a given memory architecture and device type provides a unique opportunity to derive a precise analytic model of memory system performance for a large and important class of computations. In developing such models for a single module architecture, 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 [Moye92].

2.1 Stream Detection

Access ordering algorithms derived in this report presuppose the existence of compiler techniques to detect stream-oriented computations. Benitez and Davidson [BeDa91] describe a technique for detecting streaming opportunities, including those in recurrence relations. Callahan *et al* [CaCK90] present a technique called *scalar replacement* that detects redundant accesses to subscripted variables in a loop, often transforming a more

complex sequence of references to a vector into a single access stream. Finally, as stream-oriented computations reference vector operands, well known vectorization techniques are applicable, such as those described by Wolfe [Wolf89].

2.2 Access Scheduling Techniques

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

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

Access ordering and access scheduling are fundamentally different. Access scheduling techniques allow load/store architectures to better tolerate memory latency; however, the effective memory bandwidth is not considered. Note that access ordering and access scheduling are complementary. Access ordering can first be applied to a computational kernel to obtain an ordering of load/store instructions that maximizes effective bandwidth. Access scheduling can then be applied to reduce interlock delay while maintaining the specified load/store instruction order.

3 Model Access Pattern

For deriving access ordering algorithms and performance models, it is useful to define a notation for expressing sequences of requests generated by stream-oriented computations. The Model Access Pattern notation used to denote specific reference sequences is defined below, along with a set of general definitions and assumptions applicable to all computations. Access ordering in the presence of wide words is also discussed. Finally, a restriction is placed on stream interaction to simplify optimality results.

3.1 MAP Notation

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

An *access stream* is defined by the tuple $t_i = (v, s, d, m) : \sigma$ where

v = vector to be accessed = stream starting address

s = stride of access

d = data type size

m = access mode, read(r) or write(w)

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.

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.

3.2 Definitions and Assumptions

The following definitions complement the MAP notation:

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

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

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

3.3 Wide Word Restrictions

For completeness, it is desirable to accommodate wide word access in ordering algorithms and performance models; a typical example being a 32-bit value referenced from a 64-bit word. To fully utilize wide words, and simplify modeling, several minor restrictions are 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 1.2; *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 $\sigma = t_i.\sigma$, $\gamma(s, d) \mid b\sigma$.

Then for stream t_i, *b* must be a multiple of the least common multiple of the number of data items referenced per computation iteration and the number of data items per word divided by the number of data items per computation iteration, i.e.

$$b = k' \left(\frac{lcm(\sigma, \gamma(s, d))}{\sigma}\right) \qquad k' \in Z^+$$

So, for the set of N streams S of a given MAP, the depth of loop unrolling is restricted such that

$$b = k\left(lcm(\frac{lcm(t_1.\sigma, \gamma(t_1.s, t_1.d))}{t_1.\sigma}, \ldots, \frac{lcm(t_N.\sigma, \gamma(t_N.s, t_N.d))}{t_N.\sigma})\right) \qquad k \in Z^+$$

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 differ-

ent 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. Relaxation of this restriction for applying ordering algorithms to the set of vectorizable loops is discussed in section 6.

Many loops can be transformed to adhere to the stream interaction restriction, if optimal ordering is desired. A number of these transformations are discussed in [Moye92].

3.5 MAP Dependence Relations

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

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

3.5.1 Output and Input Dependence

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

3.5.2 Antidependence

The stream interaction restriction states that two streams referencing the same vector do so with stream parameters that differ only in access mode. Thus, antidependence is limited to loop-independent antidependence between corresponding components of a read stream t_i and write stream t_j implementing a read-modify-write. So, if $t_i.v = t_j.v$, then w_j^k is antidependent on r_i^k; notationally $r_i^k \ \bar{\delta} \ w_j^k$.

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

3.5.3 Data Dependence

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

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

3.5.4 Dependence Rules

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

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

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

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

3.5.5 Other Dependencies

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

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

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

Analysis techniques [BeDa91, CaCK90] can transform the ‘natural’ pattern of access to vector $\bar{v}$ to a simple stream requiring one access per iteration; two values of $\bar{v}$ are preloaded prior to entering the loop, and each successive value accessed is carried in a regis-

ter for two iterations. The loop-carried input dependence created in the transformation has no affect on the ordering of memory access instructions.

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

4 Single Module Architecture Analysis

Access ordering algorithms and performance predictors are now derived for a single module memory architecture. Systems constructed from both uniform-access and page-mode components are considered. Optimal effective memory bandwidth is achieved in both cases.

Techniques for minimizing page overhead are first developed. Ordering algorithms and performance models are then derived for systems of uniform-access and page-mode components, respectively.

4.1 Minimizing Page Overhead

Consider a single module of page-mode components. For access ordering to generate a reference sequence for a given computation that achieves optimal effective memory bandwidth, page overhead resulting from accesses that page miss must be minimized. 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 inter-

vening 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}, $1 \le k \le h$, suffers a page miss only when $r_i^{(k-1)c+1}$ and r_i^{kc} reference a different page.

For write stream t_j 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

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

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; alternating read and write accesses can force scalar-mode (non-pipelined) arithmetic operations. However, intermixing is justified if the additional access time resulting from a sub-optimal reference sequence exceeds the additional cost of performing scalar-mode computation. This is discussed in detail 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.

4.2 Single Module of Uniform-access Components

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

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

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

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

4.2.1 Performance Predictor

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

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

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

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

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

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

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, resulting in

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

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

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

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

4.3 Single Module of Page-mode Components

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

For streams not involved in a read-modify-write, grouping accesses by stream results in minimum page overhead (Theorem 1). Given two streams that implement this operation, further reduction in page overhead may be achieved for write and read accesses by intermixing (Theorem 2) and wrap-around adjacency (Theorem 3), respectively.

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

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

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

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

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

4.3.1 Example Problem

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

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

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

Antidependence exists between corresponding elements of read stream t_{y_r} and write stream t_{y_w}, and data dependence exists between corresponding elements of t_{y_r} and t_{y_w}, and t_x and t_{y_w}. In the notation presented in section 3.5, $r_{y_r}^k \ \bar{\delta} \ w_{y_w}^k$, $r_{y_r}^k \ \delta \ w_{y_w}^k$, and $r_x^k \ \delta \ w_{y_w}^k$ for $1 \le k \le \varepsilon_{y_r} = \varepsilon_{y_w} = \varepsilon_x$.

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

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

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

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

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

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

The gross reduction in page overhead is calculated as $wadj(t_{y_w}.s, t_{y_w}.d, \varepsilon_{y_w}, 2)$.

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

4.3.2 Performance Predictor

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

Let *P* represent the access sequence over the set of streams *S*. Then *P* is composed of some number of component sequences P_i, where the subscript is defined to be that of the stream referenced; for an intermix sequence the subscript is defined to be that of the read stream. Each P_i must be in the form of

- a read access set $\{r_i{:}\,\varepsilon_i\}$,
- a write access set $\{w_i{:}\,\varepsilon_i\}$, or
- an intermix sequence $\{\{r_i{:}\,c, w_j{:}\,c\}{:}\,h\}$.

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

$$T(P_i) = \varepsilon_i T_{p/r} + \begin{cases} \omega(t_i.s, t_i.d, \varepsilon_i) T_{p/m} & \text{when } t_i \text{ is wrap-around adjacent} \\ \eta(t_i.s, t_i.d, \varepsilon_i, V) T_{p/m} & \text{otherwise} \end{cases}$$

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

$$T(P_i) = \varepsilon_i T_{p/w} + \eta(t_i.s, t_i.d, \varepsilon_i, V) T_{p/m}$$

Finally, if $P_i = \{\{r_i{:}\,c, w_j{:}\,c\}{:}\,h\}$ then $T(P_i)$ is the sum of the number of accesses to stream t_i (t_j) multiplied by the sum of the page-hit read and page-hit write cycle times and the sum of the average page overheads for read and write operations multiplied by the page miss time $T_{p/m}$, so that

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

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

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

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

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

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

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

5 Simulation Results

For a single module of page-mode components, access ordering can significantly increase effective memory bandwidth over that achieved by the ‘natural’ sequence of references for a given computation. In this context, the natural reference sequence is the sequence that results from a straight-forward translation of the loop code. To illustrate the improvement in performance achieved via access ordering, and to validate performance models, simulation and analytic results are presented for a number of common scientific kernels.

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

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

Table 1 Module Parameters

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

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

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

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

The remaining computations in Table 2 are selections from the Livermore Loops [Mcma90], with all vectors defined as double-precision.

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

Table 3 compares performance of ordered accesses for the computations of Table 2 as calculated analytically and measured via simulation; again, loops are unrolled to a depth of 4. Note that in all cases analytic and simulation results differ by less than 1%, validating the accuracy of the performance model.

Table 2 Natural vs Ordered Performance

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

6 Implementation Issues

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

Table 3 Analytic vs Simulation Results

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

6.1 Relieving Register Pressure

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

Lee [Lee91] presents a technique that employs cache memory to mimic a set of vector registers, effectively increasing register file size for vector computations. Storage is defined for a set of vectors, each of which represents a pseudo register; vector length corresponds

to register size. For example, two 64-element ‘vector registers’ are defined in the *C* programming language as

```
double VectorRegisters[2][64];
```

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

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

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

6.2 Pipelined Processors and Bus Bandwidth

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

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

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

To illustrate, consider the vector scaling operation

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

that generates read stream t_{y_r} and write stream t_{y_w}. The optimal access sequence resulting in zero write stream page overhead is

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

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

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

Let T_I and T_{NI} represent the times to complete one iteration of accesses for a read-modify-write operation with an interleaved and non-interleaved reference sequence, respectively. Values for T_I and T_{NI} are computed as the maximum of the bus transfer and memory access times, where bus transfer time is processor dependent and memory access time comes directly from performance models developed in section 4.3.2. Let T_S and T_P represent the times to complete one iteration of arithmetic operations for a read-modify-write operation in scalar and pipelined modes, respectively. Values for T_S and T_P are processor dependent.

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

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

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

6.3 Combining Caching and Non-Caching Memory Access

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

Perhaps the simplest method for combining caching and non-caching access in a coherent fashion is to preload the cache at the appropriate loop level and then apply access ordering to the remaining non-caching loads. To illustrate, consider implementing the matrix-vector multiply operation

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

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

Figure 4(a) depicts code for a straight-forward implementation of the matrix-vector multiply operation. Figure 4(b) strip-mines the computation to reuse elements of $\bar{y}$; partition size is dependent on cache size and structure [LaRW91]. Elements of $\bar{y}$ are preloaded into cache memory at the appropriate loop level. Access ordering is then applied to the inner loop as elements of A and B are not reused and hence referenced via non-caching loads. The reference to $\bar{x}$ is a constant within the inner loop and is preloaded into a processor register.

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

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

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

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

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

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

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

Figure 4 Combining Caching and Non-Caching Access

If the pseudo vector register technique described in section 6.1 is used to relieve processor register pressure for loop unrolling, care must be taken to insure that vector registers and cached operands do not collide in cache memory; the same is true for multiple cached operands. Lam *et al* [LaRW91] analyze a technique that eliminates cache conflicts by copying data to be cached into a contiguous address space. Note that in applying this copy optimization, non-unit stride vectors can be referenced via non-caching loads to reduce extraneous data movement and wasted cache space.

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

6.4 Relaxation of the Stream Interaction Restriction

Access ordering algorithms presume access streams adhere to the stream interaction restriction, defined in section 3.4. Though analysis is simplified and optimal bandwidth is obtained, dependence between accesses belonging to different streams is limited to two types: loop-independent antidependence and data dependence in the data flow sense.

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

6.4.1 Self-Antidependence Cycles

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

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

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

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

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

6.4.2 Overlapping Read Address Spaces

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

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

However, access ordering algorithms can easily accommodate intersecting read streams in a suboptimal fashion by ordering accesses from each stream independently. Read streams with intersecting address spaces may exhibit input dependence, however this can be ignored for non-volatile memory locations. A simple optimization places references from intersecting read streams adjacent, potentially reducing page overhead when applicable.

6.4.3 Access Ordering and Vectorizable Computations

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

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

7 Conclusions

Access ordering, a loop optimization that reorders accesses to better utilize memory system resources, is a compiler technology developed in this report to address the memory bandwidth problem for scalar processors executing scientific codes. For a single module memory architecture, the access ordering algorithms developed here determine a well-defined interleaving of vector references that maximizes effective bandwidth for a given computation and memory device type. Consequently, analytic models of performance can

also be derived. Access ordering algorithms developed are applicable to a superset of the class of vectorizable loops, an arguably large and interesting problem domain.

Simulation results demonstrate that for a given computation, access ordering can significantly increase effective memory bandwidth over that achieved by the natural sequence of references. Simulation results validate analytic models of performance as well.

Access ordering is fundamentally different from, though complementary to, access scheduling techniques that attempt to overlap computation with memory latency but do not consider the performance of the resulting access sequence. Access ordering is also complementary to caching, and is shown to work well with strip-mining and tiling techniques.

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

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

Analytic results presented throughout this work provide a basic and extensible set of tools for capturing memory system behavior and for understanding the interaction of reference sequences with memory architecture and component characteristics. In particular, single module results presented here are used as the basis for deriving access ordering algorithms and performance predictors for parallel memory systems [Moye92].

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.

[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.

[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.

[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.

[Moye92] 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.

[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.