Memory Access Scheduling

Scott Rixner[1], William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{rixner, billd, ujk, pmattson, jowens}@cva.stanford.edu

Abstract

The bandwidth and latency of a memory system are strongly dependent on the manner in which accesses interact with the "3-D" structure of banks, rows, and columns characteristic of contemporary DRAM chips. There is nearly an order of magnitude difference in bandwidth between successive references to different columns within a row and different rows within a bank. This paper introduces memory access scheduling, *a technique that improves the performance of a memory system by reordering memory references to exploit locality within the 3-D memory structure. Conservative reordering, in which the first ready reference in a sequence is performed, improves bandwidth by 40% for traces from five media benchmarks. Aggressive reordering, in which operations are scheduled to optimize memory bandwidth, improves bandwidth by 93% for the same set of applications. Memory access scheduling is particularly important for media processors where it enables the processor to make the most efficient use of scarce memory bandwidth.*

1 Introduction

Modern computer systems are becoming increasingly limited by memory performance. While processor performance increases at a rate of 60% per year, the bandwidth of a memory chip increases by only 10% per year making it costly to provide the memory bandwidth required to match the processor performance [14] [17]. The memory bandwidth bottleneck is even more acute for media processors with streaming memory reference patterns that do not cache well. Without an effective cache to reduce the bandwidth demands on main memory, these media processors are more often limited by memory system bandwidth than other computer systems.

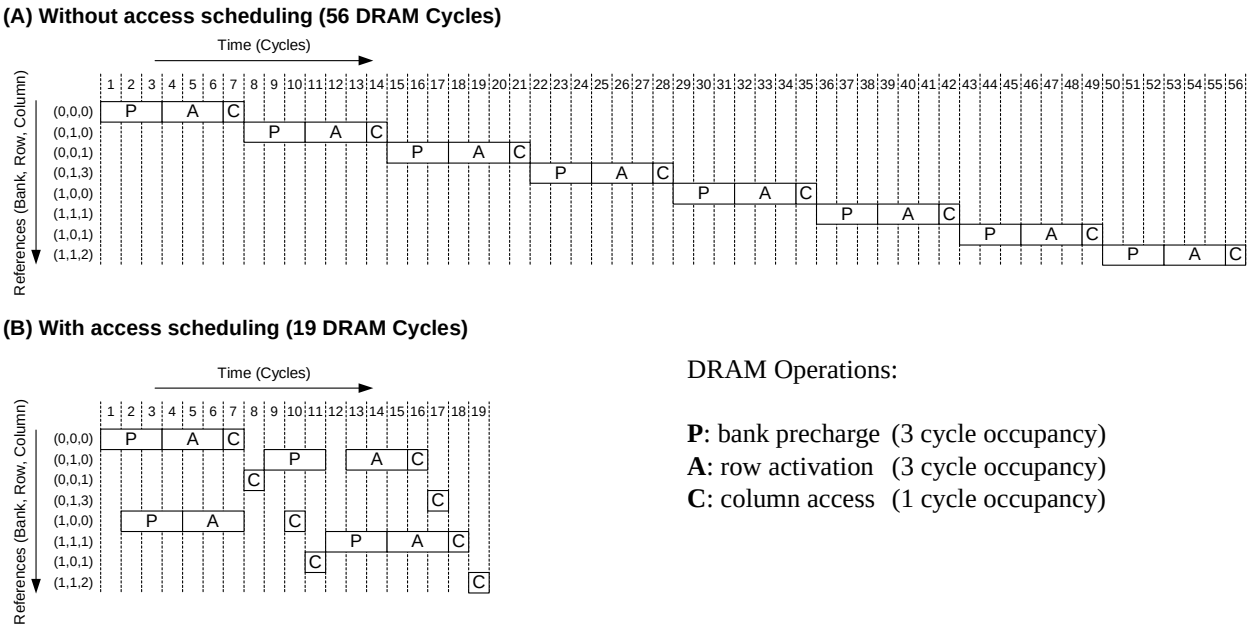
To maximize memory bandwidth, modern DRAM components allow pipelining of memory accesses, provide several independent memory banks, and cache the most recently accessed row of each bank. While these features increase the peak supplied memory bandwidth, they also make the performance of the DRAM highly dependent on the access pattern. Modern DRAMs are not truly random access devices (equal access time to all locations) but rather are three-dimensional memory devices with dimensions of bank, row, and column. Sequential accesses to different rows within one bank have high latency and cannot be pipelined, while accesses to different banks or different words within a single row have low latency and can be pipelined.

The three-dimensional nature of modern memory devices makes it advantageous to reorder memory operations to exploit the non-uniform access times of the DRAM. This optimization is similar to how a superscalar processor schedules arithmetic operations out of order. As with a superscalar processor, the semantics of sequential execution are preserved by reordering the results.

This paper introduces *memory access scheduling* in which DRAM operations are scheduled, possibly completing memory references out of order, to optimize memory system performance. The several memory access scheduling strategies introduced in this paper increase the sustained memory bandwidth of a system by up to 144% over a system with no access scheduling when applied to realistic synthetic benchmarks. Media processing applications exhibit a 30% improvement in sustained memory bandwidth with memory access scheduling, and the traces of these applications offer a potential bandwidth improvement of up to 93%.

To see the advantage of memory access scheduling, consider the sequence of eight memory operations shown in Figure 1A. Each reference is represented by the triple (bank, row, column). Suppose we have a memory system utilizing a DRAM that requires 3 cycles to precharge a bank, 3 cycles to access a row of a bank, and 1 cycle to access a column of a row. Once a row has been accessed, a new column access can issue each cycle until the bank is precharged. If these eight references are performed in order, each requires a pre-

1. Scott Rixner is an Electrical Engineering graduate student at the Massachusetts Institute of Technology.



DRAM Operations:

P: bank precharge (3 cycle occupancy)
A: row activation (3 cycle occupancy)
C: column access (1 cycle occupancy)

Figure 1. Time to complete a series of memory references without (A) and with (B) access reordering.

charge, a row access, and a column access for a total of seven cycles per reference, or 56 cycles for all eight references. If we reschedule these operations as shown in Figure 1B they can be performed in 19 cycles.

The following section discusses the characteristics of modern DRAM architecture. Section 3 introduces the concept of memory access scheduling and the possible algorithms that can be used to reorder DRAM operations. Section 4 describes the streaming media processor and benchmarks that will be used to evaluate memory access scheduling. Section 5 presents a performance comparison of the various memory access scheduling algorithms. Finally, Section 6 presents related work to memory access scheduling.

2 Modern DRAM Architecture

As illustrated by the example in the Introduction, the order in which DRAM accesses are scheduled can have a dramatic impact on memory throughput and latency. To improve memory performance, a memory controller must take advantage of the characteristics of modern DRAM.

Figure 2 shows the internal organization of modern DRAMs. These DRAMs are three-dimensional memories with the dimensions of bank, row, and column. Each bank operates independently of the other banks and contains an array of memory cells that are accessed an entire row at a time. When a row of this memory array is accessed (*row activation*) the entire row of the memory array is transferred into the bank's row buffer. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. While a row is active in the row buffer, any number of reads or writes (*column accesses*) may be performed, typically with a throughput of one per cycle. After completing the available column accesses, the cached row must be written back to the memory array by an explicit operation (*bank precharge*) which prepares the bank for a subsequent row activation. An overview of several different modern DRAM types and organizations, along with a performance comparison for in-order access, can be found in [4].

For example, the 128Mb NEC μPD45128163 [13], a typical SDRAM, includes four internal memory banks, each composed of 4096 rows and 512 columns. This SDRAM may be operated at 125MHz, with a precharge latency of 3 cycles (24ns) and a row access latency of 3 cycles (24ns). Pipelined column accesses that transfer 16 bits may issue at the rate of one per cycle (8ns), yielding a peak transfer rate of 250MB/s. However, it is difficult to achieve this rate on non-sequential access patterns for several reasons. A bank cannot be accessed during the precharge/activate latency, a single cycle of high impedance is required on the data pins when switching between read and write column accesses, and a single set of address lines is shared by all DRAM operations (bank precharge, row activation, and column access). The amount of bank parallelism that is exploited and the number of column accesses that are made per row access dictate the sustainable memory bandwidth out of such a DRAM, as illustrated in Figure 1 of the Introduction.

A memory access scheduler must generate a schedule that conforms to the timing and resource constraints of these modern DRAMs. Figure 3 illustrates these constraints for the NEC SDRAM with a simplified bank state diagram and a table of operation resource utilization. Each DRAM operation makes different demands on the three DRAM resources: the internal banks, a single set of address lines, and a single set of data lines. The scheduler must ensure that

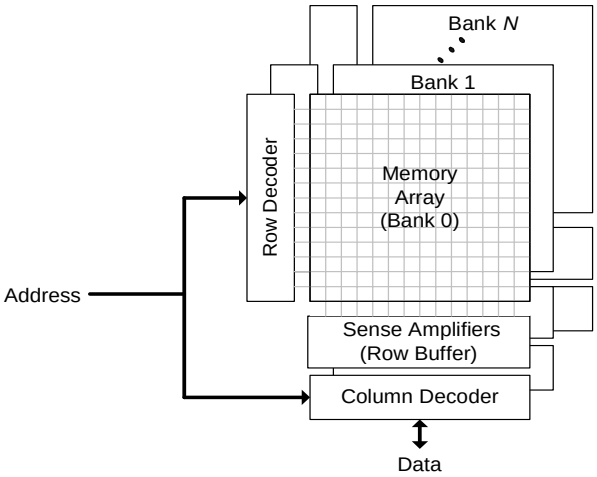


Figure 2. Modern DRAM organization.

the required resources are available for each DRAM operation it schedules.

Each DRAM bank has two stable states: IDLE and ACTIVE, as shown in Figure 3A. In the IDLE state, the DRAM is precharged and ready for a row access. It will remain in this state until a row activate operation is issued to the bank. To issue a row activation, the address lines must be used to select the bank and the row being activated, as shown in Figure 3B. Row activation requires 3 cycles, during which no other operations may be issued to that bank, as indicated by the utilization of the *bank* resource for the duration of the operation. During that time, however, operations may be issued to other banks of the DRAM. Once the DRAM's row activation latency has passed, the bank enters the ACTIVE state, during which the contents of the selected row are held in the bank's row buffer. Any number of pipelined column accesses may be performed while the bank is in the ACTIVE state. To issue either a read or write column access, the address lines are required to indicate the bank and the column of the active row in that bank. A write column access requires the data to be transferred to the DRAM at the time of issue, whereas a read column access returns the requested data three cycles later. Additional timing constraints not shown in Figure 3, such as a required cycle of high impedance between reads and writes, may further restrict the use of the data pins.

The bank will remain in the ACTIVE state until a precharge operation is issued to return it to the IDLE state. The precharge operation requires the use of the address lines to indicate the bank which is to be precharged. Like row activation, the precharge operation utilizes the *bank* resource for 3 cycles, during which no new operations may be issued to that bank. Again, operations may be issued to other banks during this time. After the DRAM's precharge latency, the bank is returned to the IDLE state and is ready for a new row activation operation. Frequently, there are also timing constraints that govern the minimum latency between a column access and a subsequent precharge operation. DRAMs typically also support column accesses with automatic precharge, which implicitly precharges the DRAM bank as soon as possible after the column access.

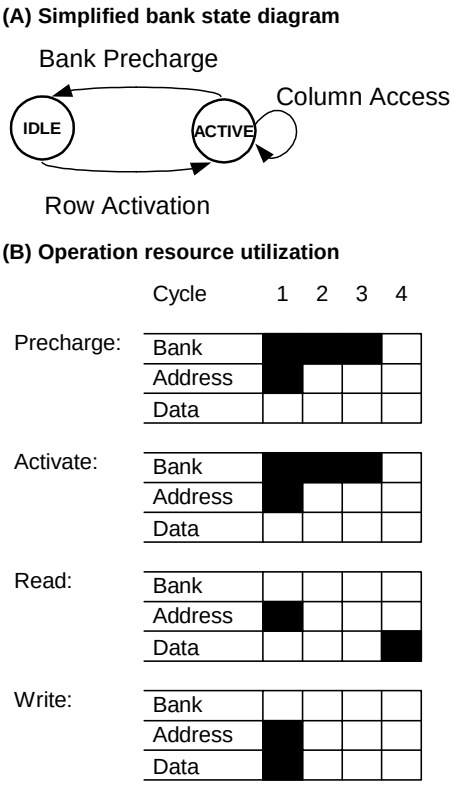


Figure 3. Simplified state diagram and resource utilization governing access to an internal DRAM bank.

The shared address and data resources serialize access to the different DRAM banks. While the state machines for the individual banks are independent, only a single bank can perform a transition requiring a particular shared resource each cycle. For many DRAMs, the bank, row, and column addresses share a single set of lines. Hence, the scheduler must arbitrate between precharge, row, and column operations that all need to use this single resource. Other DRAMs, such as Direct Rambus DRAMs (DRDRAMs) [3], provide separate row and column address lines (each with their own associated bank address) so that column and row accesses can be initiated simultaneously. To approach the peak data rate with serialized resources, there must be enough column accesses to each row to hide the precharge/activate latencies of other banks. Whether or not this can be achieved is dependent on the data reference patterns and the order in which the DRAM is accessed to satisfy those references. The need to hide the precharge/activate latency of the banks in order to sustain high bandwidth cannot be eliminated by any DRAM architecture without reducing the precharge/activate latency, which would likely come at the cost of decreased bandwidth or capacity, both of which are undesirable.

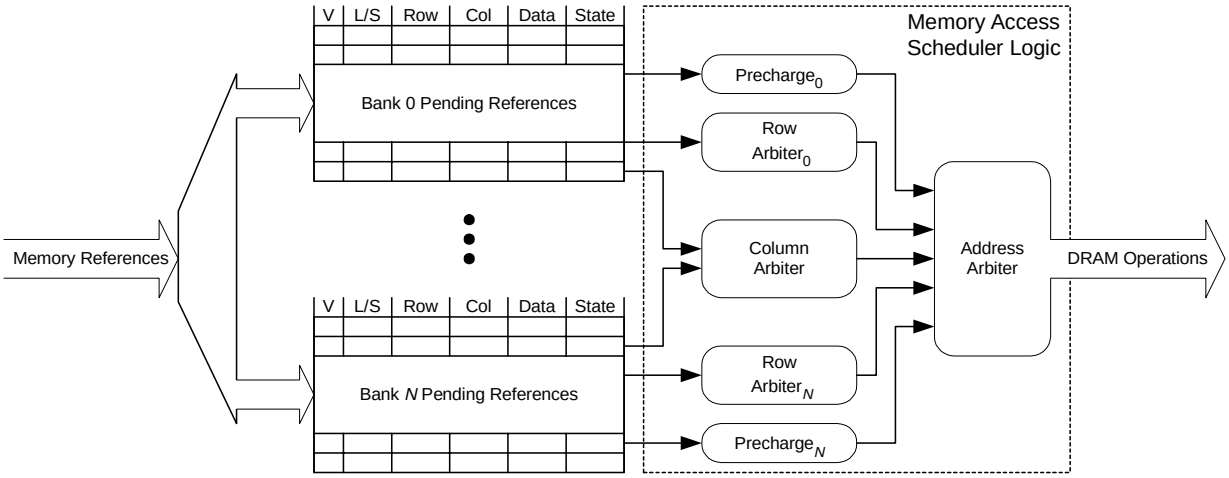


Figure 4. Memory access scheduler architecture.

3 Memory Access Scheduling

Memory access scheduling is the process of ordering the DRAM operations (bank precharge, row activation, and column access) necessary to complete the set of currently pending memory references. Throughout the paper, the term *operation* denotes a command, such as a row activation or a column access, issued by the memory controller to the DRAM. Similarly, the term *reference* denotes a memory reference generated by the processor, such as a load or store to a memory location. A single reference generates one or more memory operations depending on the schedule.

Given a set of pending memory references, a memory access scheduler may chose one or more row, column, or precharge operations each cycle, subject to resource constraints, to advance one or more of the pending references. The simplest, and most common, scheduling algorithm only considers the oldest pending reference, so that references are satisfied in the order that they arrive. If it is currently possible to make progress on that reference by performing some DRAM operation then the memory controller makes the appropriate access. While this does not require a complicated access scheduler in the memory controller, it is clearly inefficient, as illustrated in Figure 1 of the Introduction.

If the DRAM is not ready for the operation required by the oldest pending reference, or if that operation would leave available resources idle, it makes sense to consider operations for other pending references. Figure 4 shows the structure of a more sophisticated access scheduler. As memory references arrive, they are allocated storage space while they await service from the memory access scheduler. In the figure, references are initially sorted by DRAM bank. Each pending reference is represented by six fields: valid (V), load/store (L/S), address (Row and Col), data, and whatever additional state is necessary for the scheduling algorithm. Examples of state that can be accessed and modified by the scheduler are the age of the reference and whether or not that reference targets the currently active row. In practice, the pending reference storage could be shared by all the banks (with the addition of a bank address field) to allow dynamic allocation of that storage at the cost of increased logic complexity in the scheduler.

As shown in Figure 4, each bank has a precharge manager and a row arbiter. The precharge manager simply decides when its associated bank should be precharged. Similarly, the row arbiter for each bank decides which row, if any, should be activated when that bank is idle. A single column arbiter is shared by all the banks. The column arbiter grants the shared data line resources to a single column access out of all the pending references to all of the banks. Finally, the precharge managers, row arbiters, and column arbiter send their selected operations to a single address arbiter which grants the shared address resources to one or more of those operations.

The precharge managers, row arbiters, and column arbiter can use several different policies to select DRAM operations, as enumerated in Table 1. The combination of policies used by these units, along with the address arbiter's policy, determines the memory access scheduling algorithm. The address arbiter must decide which of the selected precharge, activate, and column operations to perform subject to the constraints of the address line resources. As with all of the other scheduling decisions, the *in-order* or *priority* policies can be used by the address arbiter to make this selection. Additional policies that can be used are those that select precharge operations first, row operations first, or column operations first. A column-first scheduling policy would reduce the latency of references to active rows, whereas a precharge-first or row-first scheduling policy would increase the amount of bank parallelism.

If the address resources are not shared, it is possible for both a precharge operation and a column access to the same bank to be selected. This is likely to violate the timing constraints of the DRAM. Ideally, this conflict can be handled by having the column access automatically precharge the bank

Table 1. Scheduling policies for the precharge managers, row arbiters, and column arbiter.

Policy	Arbiters	Description
in-order	precharge, row, and column	A DRAM operation will only be performed if it is required by the oldest pending reference. While used by almost all memory controllers today, this policy yields poor performance compared to policies that look ahead in the reference stream to better utilize DRAM resources.
priority	precharge, row, and column	The operation(s) required by the highest priority ready reference(s) are performed. Three possible priority schemes include: *ordered*, older references are given higher priority; *age-threshold*, references older than some threshold age gain increased priority; and *load-over-store*, load references are given higher priority. Age-threshold prevents starvation while allowing greater reordering flexibility than ordered. Load-over-store decreases load latency to minimize processor stalling on stream loads.
open	precharge	A bank is only precharged if there are pending references to other rows in the bank and there are no pending references to the active row. The open policy should be employed if there is significant row locality, making it likely that future references will target the same row as previous references did.
closed	precharge	A bank is precharged as soon as there are no more pending references to the active row. The closed policy should be employed if it is unlikely that future references will target the same row as the previous set of references.
most pending	row and column	The row or column access to the row with the most pending references is selected. This allows rows to be activated that will have the highest ratio of column to row accesses, while waiting for other rows to accumulate more pending references. By selecting the column access to the most demanded row, that bank will be freed up as soon as possible to allow other references to make progress. This policy can be augmented by one of the priority schemes described above to prevent starvation.
fewest pending	column	The fewest pending policy selects the column access to the row targeted by the fewest pending references. This minimizes the time that rows with little demand remain active, allowing references to other rows in that bank to make progress sooner. A weighted combination of the fewest pending and most pending policies could also be used to select a column access. This policy can also be augmented by one of the priority schemes described above to prevent starvation.

upon completion, which is supported by most modern DRAMs.

4 Experimental Setup

Streaming media data types do not cache well, so they require other types of support to improve memory performance. In a stream (or vector) processor, the stream transfer bandwidth, rather than the latency of any individual memory reference, drives processor performance. A streaming media processing system, therefore, is a prime candidate for memory access scheduling. To evaluate the performance impact of memory access scheduling on media processing, a streaming media processor was simulated running typical media processing applications.

4.1 Stream Processor Architecture

Media processing systems typically do not cache streaming media data types, because modern cache hierarchies cannot handle them efficiently [10]. In a media computation on long streams of data, the same operations are performed repeatedly on consecutive stream elements, and the stream elements are discarded after the operations are performed. These streams do not cache well because they lack temporal locality (stream elements are usually only referenced once) and they have a large cache footprint, which makes it likely that they will interfere with other data in the cache. In many media processing systems, stream accesses bypass the cache so as not to interfere with other data that does cache well. Many streams are accessed sequentially, so prefetching streams into the cache can sometimes be effective at improving processor performance [15]. However, this is an inefficient way to provide storage for streaming data because address translation is required on every reference, accesses are made with long addresses, tag overhead is incurred in the cache, and conflicts may evict previously fetched data.

The Imagine stream processor [16] employs a 64KB stream register file (SRF), rather than a cache, to capture the reference locality of streams. Entire streams are transferred between the DRAMs and the SRF. This is more efficient than a cache because a single instruction, rather than many explicit instructions, can be used to transfer a stream of data to or from memory.

Stream memory transfers (similar to vector memory transfers) are independent operations that are isolated from computation. Therefore, the memory system can be loading streams for the next set of computations and storing streams for the previous set of computations while the current set of computations are occurring. A computation cannot commence until all of the streams it requires are present in the stream register file. The Imagine streaming memory system consists of a pair of address generators, four interleaved memory bank controllers, and a pair of reorder buffers that place stream data in the SRF in the correct order. All of

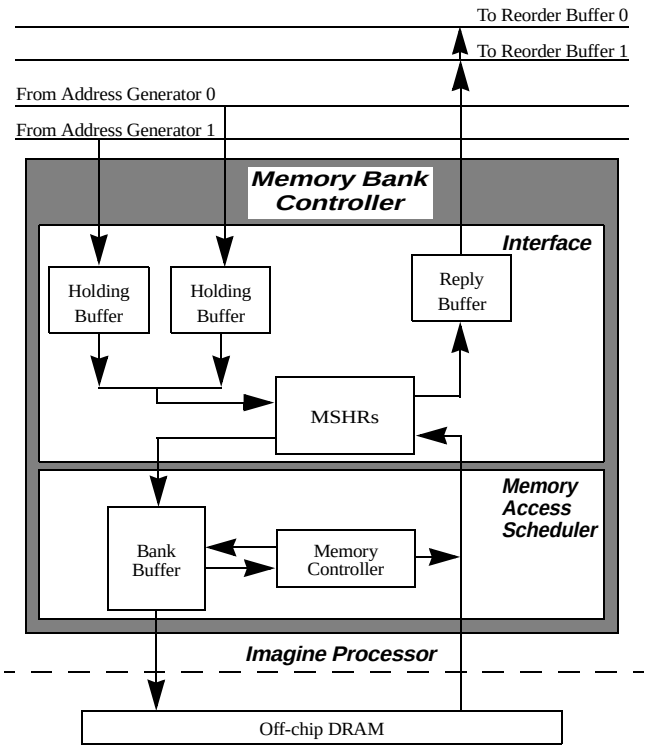


Figure 5. Memory bank controller architecture.

these units are on the same chip as the Imagine processor core.

The address generators support three addressing modes: constant stride, indirect, and bit-reversed. The address generators may generate memory reference streams of any length, as long as the data fits in the SRF. For constant stride references, the address generator takes a base, stride, and length, and computes successive addresses by incrementing the base address by the stride. For indirect references, the address generator takes a base address and an index stream from the SRF and calculates addresses by adding each index to the base address. Bit-reversed addressing is used for FFT memory references and is similar to constant stride addressing, except that bit-reversed addition is used to calculate addresses.

Figure 5 shows the architecture of the memory bank controllers.[2] References arriving from the address generators are stored in a small holding buffer until they can be processed. Despite the fact that there is no cache, a set of registers similar in function to the *miss status holding registers* (MSHRs) of a non-blocking cache [9] exist to keep track of in-flight references and to do read and write coalescing. When a reference arrives for a location that is already the target of another in-flight reference, the MSHR entry for that reference is updated to reflect that this reference will be satisfied by the same DRAM access. When a reference to a location that is not already the target of another in-flight reference arrives, a new MSHR is allocated and the reference is sent to the *bank buffer*. The bank buffer corresponds directly to the pending reference storage in Figure 4, although the storage for all of the internal DRAM banks is combined into one 32 entry buffer. The memory controller schedules DRAM accesses to satisfy the pending references in the bank buffer and returns completed accesses to the MSHRs. The MSHRs send completed loads to the reply buffer where they are held until they can be sent back to the reorder buffers. As the name implies, the reorder buffers receive out of order references and transfer the data to the SRF in order.

In this streaming memory system, memory consistency is maintained in two ways: conflicting memory stream references are issued in dependency order and the MSHRs ensure that references to the same address complete in the order that they arrive. This means that a stream load that follows a stream store to overlapping locations may be issued as soon as the address generators have sent all of the store's references to the memory banks.

For the simulations, it was assumed that the processor frequency was 500 MHz and that the DRAM frequency was 125 MHz.[3] At this frequency, Imagine has a peak computation rate of 20GFLOPS on single precision floating point computations and 20GOPS on 32-bit integer computations. Each memory bank controller has two external NEC μPD45128163 SDRAM chips attached to it to provide a column access width of 32 bits, which is the word size of the Imagine processor. These SDRAM chips were briefly described earlier and a complete specification can be found in [13]. The peak bandwidth of the SDRAMs connected to each memory bank controller is 500MB/s, yielding a total peak memory bandwidth of 2GB/s in the system.

4.2 Benchmarks

The experiments were run on a set of microbenchmarks and five media processing applications. Table 2 describes the microbenchmarks above the double line, and the applications below the double line. For the microbenchmarks, no computations are performed outside of the address generators. This allows memory references to be issued at their maximum throughput, constrained only by the buffer storage in the memory banks. For the applications, the simulations were run both with the applications' computations and without. When running just the memory traces, dependencies were maintained by assuming the computation occurred at the appropriate times but was instantaneous. The applications results show the performance improvements that can

2. Note that these are external memory banks, each composed of separate DRAM chips in contrast to the internal memory banks within each DRAM chip.

3. This corresponds to the expected clock frequency of the Imagine stream processor and the clock frequency of existing SDRAM parts.

Table 2. Benchmarks.

Name	Description
Unit Load	Unit stride load stream accesses with parallel streams to different rows in different internal DRAM banks.
Unit	Unit stride load and store stream accesses with parallel streams to different rows in different internal DRAM banks.
Unit Conflict	Unit stride load and store stream accesses with parallel streams to different rows in the same internal DRAM banks.
Constrained Random	Random access load and store streams constrained to a 64KB range.
Random	Random access load and store streams to the entire address space.
FFT	Ten consecutive 1024-point real Fast Fourier Transforms.
Depth	Stereo depth extraction from a pair of 320x240 8-bit grayscale images.[a]
QRD	QR matrix decomposition of a 192x96 element matrix.[b]
MPEG	MPEG2 encoding of three frames of 360x288 24-bit color video.
Tex	Triangle rendering of a 720x720 24-bit color image with texture mapping.[c]

a. *Depth* performs depth extraction using Kanade's algorithm [8]. Only two stereo images are used in the benchmark, as opposed to the multiple cameras of the video-rate stereo machine.

b. *QRD* uses blocked Householder transformations to generate an orthogonal Q matrix and an upper triangular R matrix such that Q·R is equal to the input matrix.

c. *Tex* applies modelview, projection, and viewport transformations on its unmeshed input triangle stream and performs perspective-correct bilinear interpolated texture mapping on its generated fragments. A single frame of the SPECviewperf 6.1.1 Advanced Visualizer benchmark image was rendered.

be gained by using memory access scheduling with a modern media processor. The application traces, with instantaneous computation, show the potential of these scheduling methods as processing power increases and the applications become entirely limited by memory bandwidth.

5 Experimental Results

A memory controller that performs no access reordering will serve as a basis for comparison. This controller performs no access scheduling, as it uses an in-order policy, described in Table 1, for all decisions: a column access will only be performed for the oldest pending reference, a bank will only be precharged if necessary for the oldest pending reference, and a row will only be activated if it is needed by the oldest pending reference. No other references are considered in the scheduling decision. This algorithm, or slight variations such as automatically precharging the bank when a cache line fetch is completed, can commonly be found in systems today.

The gray bars of Figure 6 show the performance of the benchmarks using the baseline in-order access scheduler. Unsurprisingly, *unit load* performs very well with no access scheduling, achieving 97% of the peak bandwidth (2GB/s) of the DRAMs. The 3% overhead is the combined result of infrequent precharge/activate cycles and the start-up/shut-down delays of the streaming memory system.

The 14% drop in sustained bandwidth from the *unit load* benchmark to the *unit* benchmark shows the performance degradation imposed by forcing intermixed load and store references to complete in order. Each time the references switch between loads and stores a cycle of high impedance must be left on the data pins, decreasing the sustainable bandwidth. The *unit conflict* benchmark further shows the penalty of swapping back and forth between rows in the DRAM banks, which drops the sustainable bandwidth down to 51% of the peak. The random benchmarks sustain about 15% of the bandwidth of the *unit load* benchmark. This loss roughly corresponds to the degradation incurred by performing accesses with a throughput of one word every seven DRAM cycles (the random access throughput of the SDRAM) compared to a throughput of one word every DRAM cycle (the column access throughput of the SDRAM).

The applications' behavior closely mimics their associated microbenchmarks. The *QRD* and *MPEG* traces include many unit and small constant stride accesses, leading to a sustained bandwidth that approaches that of the *unit* benchmark. The *Depth* trace consists almost exclusively of constant stride accesses, but dependencies limit the number of simultaneous stream accesses that can occur. The *FFT* trace is composed of constant stride loads and bit-reversed stores. The bit-reversed accesses sustain less bandwidth than constant stride accesses because they generate sequences of references that target a single memory bank and then a sequence of references that target the next memory bank and so on. This results in lower bandwidth than access patterns that more evenly distribute the references across the four memory banks. Finally, the *Tex* trace includes constant stride accesses, but is dominated by texture accesses which are essentially random within the texture memory space. These texture accesses lead to the lowest sustained bandwidth of the applications. Note that for the applications, memory bandwidth corresponds directly to performance because the applications make the same number of memory references regardless of the scheduling algorithm. Therefore, increased bandwidth means decreased execution time.

5.1 First-ready Scheduling

The use of a very simple first-ready access scheduler improves performance by an average of over 25% on all of

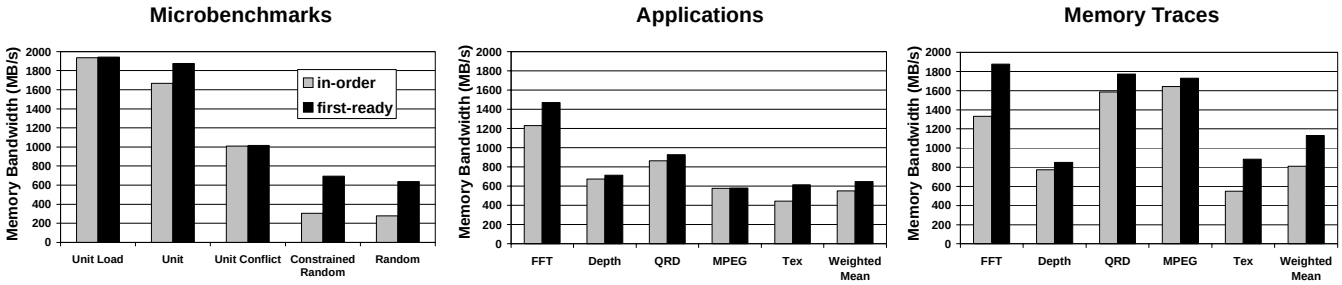


Figure 6. Sustained memory bandwidth using in-order and first-ready access schedulers (2 GB/s peak supplied bandwidth).

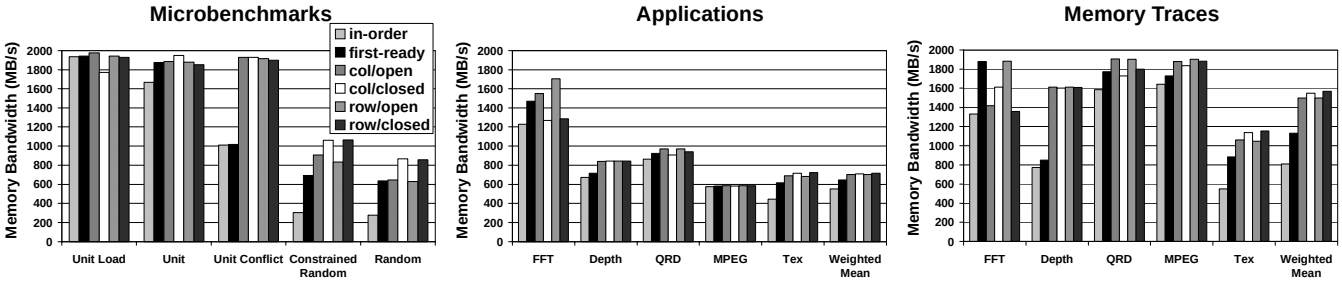


Figure 7. Sustained memory bandwidth of memory access scheduling algorithms (2 GB/s peak supplied bandwidth).

Table 3. Reordering scheduling algorithm policies.

Algorithm	Column Access	Precharging	Row Activation	Access Selection
col/open	priority (ordered)	open	priority (ordered)	column first
col/closed	priority (ordered)	closed	priority (ordered)	column first
row/open	priority (ordered)	open	priority (ordered)	row first
row/closed	priority (ordered)	closed	priority (ordered)	row first

the benchmarks. First-ready scheduling uses the ordered priority scheme, as described in Table 1, to make all scheduling decisions. The first-ready scheduler considers all pending references and schedules a DRAM operation for the oldest pending reference that does not violate the timing and resource constraints of the DRAM. The most obvious benefit of this scheduling algorithm over the baseline is that accesses targeting other banks can be made while waiting for a precharge or activate operation to complete for the oldest pending reference. This relaxes the serialization of the in-order scheduler and allows multiple references to progress in parallel.

Figure 6 shows the sustained bandwidth of the in-order and first-ready scheduling algorithms for each benchmark. The sustained bandwidth is increased by 79% for the microbenchmarks, 17% for the applications, and 40% for the application traces. As should be expected, *unit load* shows little improvement as it already sustains almost all of the peak SDRAM bandwidth, and the random benchmarks show an improvement of over 125%, as they are able to increase the number of column accesses per row activation significantly.

5.2 Aggressive Reordering

When the oldest pending reference targets a different row than the active row in a particular bank, the first-ready scheduler will precharge that bank even if it still has pending references to its active row. More aggressive scheduling algorithms are required to further improve performance. In this section, four scheduling algorithms, enumerated in Table 3, that attempt to further increase sustained memory bandwidth are investigated. The policies for each of the schedulers in Table 3 are described in Table 1. The range of possible memory access schedulers is quite large, and covering all of the schedulers examined in Section 3 would be prohibitive. These four schedulers were chosen to be representative of many of the important characteristics of an aggressive memory access scheduler.

Figure 7 presents the sustained memory bandwidth for each memory access scheduling algorithm on the given benchmarks. These aggressive scheduling algorithms improve the memory bandwidth of the microbenchmarks by 106-144%, the applications by 27-30%, and the application traces by 85-93% over in-order scheduling.

Unlike the rest of the applications, *MPEG* does not show a noticeable improvement in performance when moving to

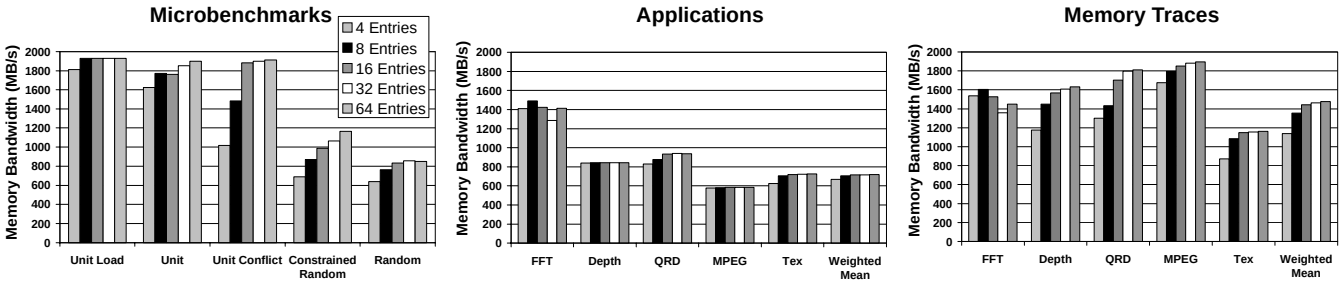


Figure 8. Sensitivity to bank buffer size.

the more aggressive scheduling algorithms. On a stream architecture like Imagine, *MPEG* efficiently captures data locality within the SRF. This makes *MPEG* compute-bound, thereby eliminating any opportunity for performance improvement by improving the memory system bandwidth. However, the performance on the memory trace can be improved by the aggressive scheduling algorithms to over 90% of the peak bandwidth of the memory system.

The use of a column-first or a row-first access selection policy makes very little difference across all of the benchmarks. There are minor variations, but no significant performance improvements in either direction, except for FFT. This has less to do with the characteristics of the scheduling algorithm than with the fact that the FFT benchmark is the most sensitive to stream load latency, and the col/open scheduler happens to allow a store stream to delay load streams in this instance.

The benchmarks that include random or somewhat random address traces favor a closed precharge policy, in which banks are precharged as soon as there are no more pending references to their active row. This is to be expected as it is unlikely that there will be any reference locality that would make it beneficial to keep the row open. By precharging as soon as possible, the access latency of future references is minimized. For most of the other benchmarks, the difference between an open and a closed precharge policy is slight. Notable exceptions are *unit load* and *FFT. Unit load* performs worse with the col/closed algorithm. This is because column accesses are satisfied rapidly, emptying the bank buffer of references to a stream, allowing the banks to be precharged prematurely in some instances. This phenomenon also occurs in the *QRD* and *MPEG* traces with the col/closed algorithm. *FFT* performs much better with an open precharging policy because of the bit-reversed reference pattern. A bit-reversed stream makes numerous accesses to each row, but they are much further apart in time than they would be in a constant stride access. Therefore, leaving a row activated until that bank is actually needed by another reference is advantageous, as it eliminates the need to reactivate the row when those future references finally arrive.

Figure 8 shows the effects of varying the bank buffer size on sustained memory bandwidth when using memory access scheduling. The row/closed scheduling algorithm is used with bank buffers varying in size from 4 to 64 entries. The *unit load* benchmark requires only 8 entries to saturate the memory system. The *unit conflict* and *random* benchmarks require 16 entries to achieve their peak bandwidth. The *unit* and *constrained random* benchmarks are able to utilize additional buffer space to improve bandwidth.

A 16 entry buffer allows all of the applications to achieve their peak memory bandwidth, which is 7% higher than with a 4 entry buffer. *Depth* and *MPEG* are not sensitive to the bank buffer size at all because they are compute-bound on these configurations. The bandwidth of *Tex* improves as the buffer size is increased from 4 to 16 entries because the larger buffer allows greater flexibility in reordering its non-strided texture references. *QRD* benefits from increasing the buffer size from 4 to 16 because it issues many conflicting stream transfers that benefit from increased reordering. For the applications' memory traces, there is a slight advantage to further increasing the buffer size beyond 16; a 16 entry buffer improves bandwidth by 27% and a 64 entry buffer improves bandwidth by 30% over a 4 entry buffer. Again, the sustainable bandwidth of *FFT* fluctuates because of its extreme sensitivity to load latency.

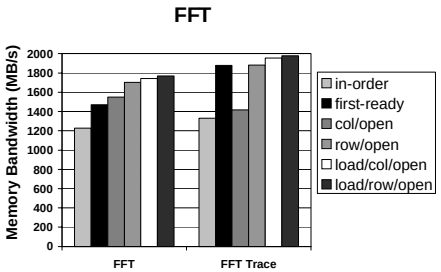


Figure 9. Sustained memory bandwidth for FFT with *load-over-store* scheduling.

To stabilize the sustainable bandwidth of *FFT*, load references must be given higher priority than store references. Write buffers are frequently used to prevent pending store references from delaying load references required by the processor [5]. As the bank buffer is already able to perform this function, the col/open and row/open scheduling algorithms can simply be augmented with a load-over-store priority scheme for their column access and row activation policies. This allows load references to complete sooner, by giving them a higher priority than store references, as described in Table 1. Figure 9 shows that with the addition of the load-over-store priority scheme, the FFT trace sus-

tains over 97% of the peak memory system bandwidth with both the load/row/open and load/col/open schedulers. Using a load-over-store policy does not affect the other applications which are not as sensitive to load latency.

6 Related Work

Stream buffers prefetch data structured as streams or vectors to hide memory access latency [7]. Stream buffers do not, however, reorder the access stream to take advantage of the 3-D nature of DRAM. For streams with small, fixed strides, references from one stream tend to make several column accesses for each row activation, giving good performance on a modern DRAM. However, conflicts with other streams and non-stream accesses often evict the active row, thereby reducing performance. McKee's Stream Memory Controller (SMC) extends a simple stream buffer to reduce memory conflicts among streams by issuing several references from one stream before switching streams [6] [12]. The SMC, however, does not reorder references within a single stream.

The Command Vector Memory System (CVMS) [2] reduces the processor to memory address bandwidth by transferring *commands* to the memory controllers, rather than individual references. A command includes a base and a stride which is expanded into the appropriate sequence of references by each off-chip memory bank controller. The bank controllers in the CVMS utilize a row/closed scheduling policy among commands to improve the bandwidth and latency of the SDRAM. The Parallel Vector Access unit (PVA) [11] augments the Impulse memory system [1] with a similar mechanism for transferring commands to the Impulse memory controller. Neither of these systems reorder references within a single stream. Conserving address bandwidth, as in the CVMS and PVA, is important for systems with off-chip memory controllers, but is largely orthogonal to memory access scheduling.

The SMC, CVMS, and PVA do not handle indirect (scatter/gather) streams. These references are usually handled by the processor cache, as they are not easily described to a stream prefetching unit. However, indirect stream references do not cache well because they are large and lack both spatial and temporal locality. These references also do not typically make consecutive column accesses to the same row, severely limiting the sustainable data bandwidth when those references are satisfied in order. The memory access scheduling techniques described here work for indirect streams as well as for strided streams, as demonstrated by the improvements in the random benchmarks and the *Tex* application.

Hitachi has proposed an access optimizer for embedded DRAM as part of a system-on-a-chip and has built a test chip containing the access optimizer and some DRAM [18]. This access optimizer implements a first-ready scheduler, and is 1.5mm^2, dissipates 26mW, and runs at 100MHz in a 0.18μm process. While the more aggressive schedulers would require more logic, this should give a feel for the actual cost of memory access scheduling.

7 Conclusions

Memory bandwidth is becoming the limiting factor in achieving higher performance, especially in media processing systems. Processor performance improvements will continue to outpace increases in memory bandwidth, so techniques are needed to maximize the sustained memory bandwidth. To maximize the peak supplied data bandwidth, modern DRAM components allow pipelined accesses to a three-dimensional memory structure. Memory access scheduling greatly increases the bandwidth utilization of these DRAMs by buffering memory references and choosing to complete them in an order that both accesses the internal banks in parallel and maximizes the number of column accesses per row access, resulting in improved system performance.

Memory access scheduling realizes significant bandwidth gains on a set of media processing applications as well as on synthetic benchmarks and application address traces. A simple reordering algorithm that advances the first ready memory reference gives a 17% performance improvement on applications, a 79% bandwidth improvement for the microbenchmarks, and a 40% bandwidth improvement on the application traces. The application trace results give an indication of the performance improvement expected in the future as processors become more limited by memory bandwidth. More aggressive reordering, in which references are scheduled to increase locality and concurrency, yields substantially larger gains. Bandwidth for synthetic benchmarks improved by 144%, performance of the media processing applications improved by 30%, and the bandwidth of the application traces increased by 93%.

A comparison of alternative scheduling algorithms shows that on most benchmarks it is advantageous to employ a closed page scheduling policy in which banks are precharged as soon as the last column reference to an active row is completed. This is in part due to the ability of the DRAM to combine the bank precharge request with the final column access. There is little difference in performance between scheduling algorithms that give preference to row accesses over column accesses, except that the col/closed algorithm can sometimes close pages too soon, somewhat degrading performance. Finally, scheduling loads ahead of stores improves application performance for latency sensitive applications.

Contemporary cache organizations waste memory bandwidth in order to reduce the memory latency seen by the processor. As memory bandwidth becomes more precious, this will no longer be a practical solution to reducing memory latency. Media processing has already encountered this phenomenon, because streaming media data types do not cache well and require careful bandwidth management. As cache organizations evolve to be more conscious of memory bandwidth, techniques like memory access scheduling will be required to sustain a significant fraction of the available data bandwidth. Memory access scheduling is, therefore, an

important step toward maximizing the utilization of the increasingly scarce memory bandwidth resources.

Acknowledgments

The authors would like to thank the other members of the Imagine project for their contributions. The authors would also like to thank Kekoa Proudfoot and Matthew Eldridge for providing the *Tex* benchmark triangle trace data. The research described in this paper was supported by the Defense Advanced Research Projects Agency under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C-0037.

References

[1] CARTER, JOHN, ET AL., Impulse: Building a Smarter Memory Controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture* (January 1999), pp. 70-79.

[2] CORBAL, JESUS, ESPASA, ROGER, AND VALERO, MATEO, Command Vector Memory Systems: High Performance at Low Cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (October 1998), pp. 68-77.

[3] CRISP, RICHARD, Direct Rambus Technology: The New Main Memory Standard. *IEEE Micro* (November/December 1997), pp. 18-28.

[4] CUPPU, VINODH, ET AL., A Performance Comparison of Contemporary DRAM Architectures. In *Proceedings of the International Symposium on Computer Architecture* (May 1999), pp. 222-233.

[5] EMER, JOEL S. AND CLARK, DOUGLAS W., A Characterization of Processor Performance in the VAX-11/780. In *Proceedings of the International Symposium on Computer Architecture* (June 1984), pp. 301-310.

[6] HONG, SUNG I., ET AL., Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture* (January 1999), pp. 80-89.

[7] JOUPPI, NORMAN P., Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the International Symposium on Computer Architecture* (May 1990),pp. 364-373.

[8] KANADE, TAKEO, KANO, HIROSHI, AND KIMURA, SHIGERU, Development of a Video-Rate Stereo Machine. In *Proceedings of the International Robotics and Systems Conference* (August 1995), pp. 95-100.

[9] KROFT, DAVID, Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the International Symposium on Computer Architecture* (May 1981), pp. 81-87.

[10] LEE, RUBY B. AND SMITH, MICHAEL D., Media Processing: A new design target. *IEEE Micro* (August 1996), pp. 6-9.

[11] MATTHEW, BINU K., ET AL., Design of a Parallel Vector Access Unit for SDRAM Memory Systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture* (January 2000), pp. 39-48.

[12] MCKEE, SALLY A. AND WULF, WILLIAM A., Access Ordering and Memory-Conscious Cache Utilization. In *Proceedings of the First Symposium on High Performance Computer Architecture* (January 1995), pp. 253-262.

[13] NEC Corporation. *128M-bit Synchronous DRAM 4-bank, LVTTL Data Sheet*. Document No. M12650EJ5V0DS00, 5th Edition, Revision K (July 1998).

[14] PATTERSON, DAVID, ET AL., A Case for Intelligent RAM. *IEEE Micro* (March/April 1997), pp. 34-44.

[15] RANGANATHAN, PARTHASARATHY, ET AL., Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proceedings of the International Symposium on Computer Architecture* (May 1999), pp. 124-135.

[16] RIXNER, SCOTT, ET AL., A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the International Symposium on Microarchitecture* (December 1998), pp. 3-13.

[17] SAULSBURY, ASHLEY, PONG, FONG, AND NOWATZYK, ANDREAS, Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the International Symposium on Computer Architecture* (May 1996), pp. 90-101.

[18] WATANABE, TAKEO, ET AL., Access Optimizer to Overcome the "Future Walls of Embedded DRAMs" in the Era of Systems on Silicon. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (February 1999), pp. 370-371.