Indexing Memory Banks to Maximize Page Mode Hit Percentage and Minimize Memory Latency

Tomas Rokicki

June 10, 1996

Abstract. Many memory controllers today take the page mode gamble—they leave DRAM pages open on the chance that the next access will be on that same page. If it is, a row access delay is saved; if it is not, a row precharge delay may be incurred. Most studies have concentrated on whether this gamble is worthwhile, and how much it saves if it is. This report instead concentrates on how to organize your memory system so that the gamble succeeds as often as possible. Indeed, on sample traces, we show that the techniques originated here increase the page mode hit rate significantly; for a typical memory system organization and a small set of traces, the page mode hit rate increased from an average of 49% to an average of 87%. We also present results showing that inexpensive memory-side prefetching can further reduce the latency of memory reads—in our tests, half of the read data can be on the way to the bus drivers before the read is issued by the processor.

Keywords. memory controller, pagemode, indexing, addressing, performance, prefetching.

Contents

1 Introduction

As processor clock and instruction rates increase much faster than memory system access times, memory system design is becoming increasingly critical. In the past ten years, typical processor speeds have increased from 10 MHz to over 200 MHz—a factor of more than twenty. Memory chip access delays, on the other hand, have decreased from 200 ns to 60 ns—a decrease of about a factor of three. Actual memory access delays at the processor pins have decreased by less than this. Thus, the number of processor cycles per memory access delay has increased from 2 to 12 in the past ten years—and this trend shows no signs of abating.

To make things worse, older processors typically required multiple cycles for a single instruction, while current processors can typically execute multiple instructions in a single cycle. Thus, the potential number of instructions per memory access is increasing faster than the typical number of cycles per memory access [WM95].

Even software trends appear to conspire against performance. Much existing software emphasizes static allocation and iteration through arrays, techniques that allow scheduling and prefetching techniques to minimize the impact of memory latency. But study of more recent object-oriented programs show that they spend much of their time chasing pointers—behavior that foils most latency-hiding techniques. For these more recent programs, memory latency has a higher impact on performance.

Increasingly larger caches have mitigated the impact of this disparity in performance by putting expensive, fast memory closer to the processor. As the size of these caches increase, so does the access time, cost, and power requirements. The impact on performance of doubling the cache size decreases as the size of the cache grows; we are quickly entering the region of diminishing returns. Indeed, with the larger on-chip caches of current processors, the "cacheless" PC is returning.

While the basic architecture of processors has been undergoing some dramatic transformations over the past decade or so, the basic architecture of high-capacity memory chips has remained fairly constant. Page mode DRAMs have been available for at least ten years. In page mode, memory cells within a certain portion of the memory array can be accessed much more quickly than other cells. By exploiting page mode, not only can the average memory system latency be reduced, but the bandwidth per memory chip pin can be significantly enhanced. While most studies have focused on what page mode hit rate can typically be attained and what impact this has on performance, we focus on how the memory system organization can maximize the page mode hit rate. Our two major techniques, contiguous pages and cache-effect interleaving, can significantly improve the page mode hit rate and thus the overall performance of the memory system.

2 DRAM Architecture

The basic DRAM memory cell is a single-transistor cell with a small capacitor. The contents of the memory cell is indicated by the presence or absence of charge on the capacitor. The cells are organized into a grid of rows and columns, as shown in Figure 1. A typical four million bit memory array might have 2,048 rows of 2,048 columns each. The distinction between rows and columns is

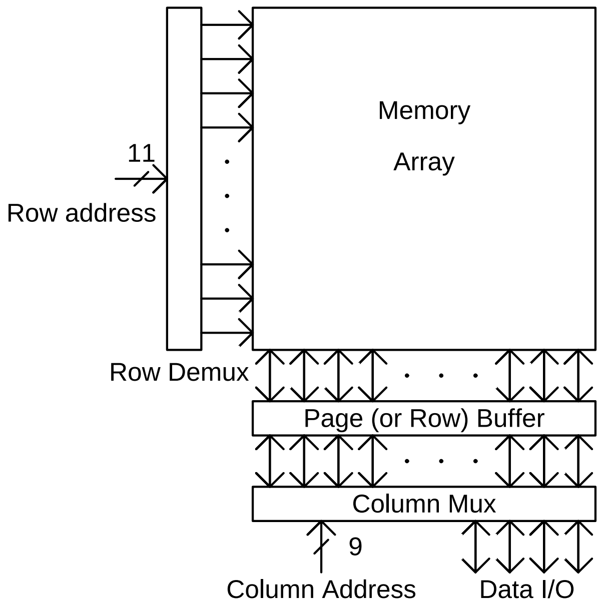

Figure 1: DRAM memory organization. This is a 4M array with 2048 rows of 2048 bits, with four-bit data input and output.

significant because of the way a memory access proceeds.

At the bottom of the memory array is a single row of static cells called the "page buffer". This row acts as a temporary staging area for both reads and writes. A typical DRAM access consists of a row access, one or more column accesses, and a precharge.

The row access (also called a page opening) is performed by presenting the row address bits to the row demultiplexor to select a row. The entire contents of that row are then transferred into the page buffer. This transfer is done in parallel, and it empties all memory cells in that row of their contents. The transfer is done by driving whatever charge exists in each row capacitor down to a set of amplifiers that load the page buffer. For current DRAMs, this operation takes approximately 50 ns.

Next, the column address bits are presented to select a particular column or set of columns, and the data is either read from or written to the page buffer. In this phase of the access, the page buffer acts as a small static RAM. The typical access delay for this operation is very short; 10 ns is typical. Several consecutive accesses can be made to the page to access different columns; for instance, the entire row can be read out very quickly. Many DRAMs have multiple data input and output pins, so four or eight bits might be read out in each 10 ns. For a typical four bit wide DRAM, a page of 2,048 bits (or 256 bytes) can be read out in 512 accesses or 5.12 μs. This is 50 megabytes per second—from a single chip. It is easy to see how a few DRAM chips in parallel can yield very high bandwidth.

The final phase of access is precharge (or page closing). Remember that the row access destroyed the contents of the row in the memory array. Before another row can be accessed, the contents in the page buffer must be transferred back to the memory array; this process is called precharge. Note that no addresses are required; the currently open row is latched during the page opening and remembered as long as the page is open internally within the DRAM. The typical precharge latency is 30 ns.

In addition to the normal read and write accesses, DRAMs also support (and require) refresh cycles. The small capacitors that make up each memory cell suffer from leakage, and after a short period of time, their charge will drain away. To prevent the loss of data, each row must be precharged—opened and closed—at a certain minimum rate. The size of the capacitors and leakage allowed is balanced with the size of the array in such a way that the number of refresh cycles required is a small fraction of the total bandwidth of the DRAM; typically, DRAMs are engineered such that refreshing the rows in order at one row per 60 microseconds is sufficient to maintain the data. The main impact of this on our page mode study is that pages cannot be kept open indefinitely; they must be closed in order to refresh other rows.

The most common type of DRAMs are now called 'asynchronous' DRAMs; these DRAMs do not have a clock input. Rather, there are complex timing constraints among the various signals and addresses that must be satisfied in order for the DRAM to operate properly. Each asynchronous DRAM has a single logical 'array' of memory, or bank. The two main control pins for these DRAMs are RAS and CAS. To open a row, RAS is asserted (typically, lowered); to close a row, RAS is deasserted. To access a column CAS is asserted; to access another column, CAS must be deasserted and reasserted.

Recently, synchronous DRAMs (SDRAMs) have been introduced. These DRAMs accept a clock input, and almost all timing delays are specified with respect to this clock. In addition, these DRAMs usually have two or four different logical arrays of memory (or banks) that can operate independently. Rather than use separate RAS and CAS wires for each bank, a sequence of "commands" is sent to the DRAM synchronously; these are interpreted as page opening, column access, and page closing commands internally. These chips have one or two additional address bits used for bank selection.

The first major benefit of SDRAMs is pipelining; it is possible for a sequence of column reads or writes to be fed into the chip faster than they could be satisfied serially. Thus, the time to load the address and data onto the chip, perform the column access, and send the result out of the chip from different accesses are overlapped. A second major benefit is the use of the multiple banks to hide the precharging. While one bank is being accessed, another bank can be refreshed or precharged in the background.

Despite these differences, SDRAM organization is still very similar to asynchronous DRAM organization; in fact, memory controllers for asynchronous DRAMs have been supporting multiple banks and hiding precharge for years.

For the examples in this paper, we will assume we are using 4Mbit asynchronous DRAMs with 2,048 rows of 2,048 bits each, with four-bit wide data inputs and outputs. The analysis hold exactly for

16Mbit SDRAMs with 4 banks, each organized in the same way as the asynchronous DRAMs. The essential results hold for virtually any current common DRAM configuration.

3 Memory System Architecture

Memory chips can be organize in memory systems in a variety of ways. The width and speed of the system bus must typically be matched. Usually system busses are both faster and wider than DRAM chips. Typically enough DRAM chips are cascaded in parallel to match the width of the data bus; if we are using a 16-byte wide data bus, we would need 32 4-bit wide DRAM chips to match this bus. If our cache line size is 64 bytes, we will generally use four bus clock ticks for each reference. If our data bus is clocked faster than the DRAMs, we can use a small pipelining FIFO to match the speeds. In this case, we need several sequential accesses to each chip to access an entire cache line; using our example values, we need four sequential accesses. Another alternative is to use an even wider memory bus and multiplex it at high speeds onto our data bus. For simplicity, we shall assume a small pipelining FIFO in this discussion.

If we require 32 4-bit wide DRAM chips in parallel to match the width of the bus, the minimum memory increment is 32 4Mbit chips, which is 16 megabytes. Typically, when addressing more memory, in order to keep physical memory contiguous, we will use high-order physical address bits to select different groups of memory chips. Since our memory bus is 16 bytes wide, we will ignore the least significant four address bits. Since our cache lines are 64 bytes long, we need the next two address bits as column index bits that we can cycle to obtain an entire cache line. The next seven bits can be assigned to the remaining column address bits, and the next eleven bits to the row address bits. This is a total of 24 bits, which correctly matches our computed 16 megabyte memory size. We can thus use address bits from the 25th bit on up to select what group of chips should be used. The first 32 chips will be selected if addresses are within the first 16 megabytes of the physical address range; the second 32 chips will be selected if addresses are within the next 16 megabytes, and so on. This is the simplest and most common way to select different groups of memory chips.

Consider what happens to the DRAM pages. For the DRAM chips that are accessed in parallel to form a single cache line, all of the page buffers in the various chips combine to form a larger logical page buffer. Each chip in our example has a 2,048 bit, or 256 byte, row. Since we are using 32 chips in parallel, we end up with a 8,192 byte logical page. If we use the low-order address bits to index the columns, two accesses that differ only in the lower 13 bits of the physical address will be on the same DRAM page.

Each collection of 32 parallel DRAM chips has its own set of page buffers. Thus, we can have multiple banks in our memory system. If we use high-order address bits to select banks, as described previously, then we have one 8K DRAM page for the first 16M of physical memory, another 8K page for the next 16M of physical memory, and so on.

If we are using SDRAMs with bank select bits, we might treat the internal banks as just collections of relatively independent chips, and use high-order address bits as our bank select bits. For the purposes of this paper, there is no difference between memory banks that are derived from collections of chips addressed independently, and memory banks that are derived from bank select inputs to specific SDRAM chips.

4 The Page Mode Gamble

We finally have enough background to discuss the page mode gamble. In our hypothetical system, a typical cache line read will occur as follows. First, the appropriate bank is selected and the page is opened. This takes approximately 50 ns. Next, four 16-byte chunks are read from the page mode buffer; this takes approximately 40 ns, and gives us a complete cache line. Finally, the page is closed; this takes 30 ns. The total time was 120 ns. The time before the first word was read was 60 ns (page open plus first column access); on systems that implement critical word first, this can be the most important latency value.

Now let us imagine a controller that gambles that successive references to the same memory bank will access the same row (page). We call this a page hit. In such a memory controller, the page is not closed after an access. Instead, the page is only closed when an access to that bank is seen that is for a different page.

If the memory controller is correct, and the subsequent access is indeed for the same page, then the critical word latency is shortened to just 10 ns—a significant savings. If the memory controller is wrong, then it must pay a penalty—it must precharge the old page before it can open a new page—so the total access is 30 ns plus 50 ns plus 10 ns, or 90 ns, quite a bit more than our previous value of 60 ns.

If p is the probability that the next access is on the same page, then our average critical word latency is $10p + 90(1 - p)$, or $90 - 80p$; this function decreases as p increases. The point at which the gamble pays off is when this value is equal to our old value of 60; this occurs when $p = 0.375$. Thus, we do not have guess correctly very often in order to win this gamble.

Let us assume that requests are fed to the memory system as fast as they can be consumed. Each page new page requires an open and a close, which together require 80 ns. Each cache line access from an open page requires 40 ns. Thus, an average access requires $40 + 80(1 - p)$ ns of bank time for a page mode controller, and 120 ns of bank time for a basic controller. Clearly, the page mode controller is capable of extracting significantly more bandwidth per bank and thus per chip; for a page hit percentage of only 0.5, the page mode controller obtains 50% more bandwidth per chip. (SDRAM technology cuts this back somewhat since two or four banks share the same chip pins, and thus opening and closing of one bank can overlap accessing a different bank in a non-page-mode controller.)

If we have a split-transaction data bus that allows us to queue up multiple requests, the bursty nature of memory requests might mean that we can peek ahead in the request queue to determine if the next access is a page hit even while the current request is being processed. In this case, we can always “guess” correctly.

The simplest controllers might keep only a single page open, instead of a page per bank. We still need only a little look-ahead or a moderate page hit percentage to decrease the average memory

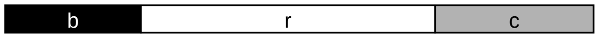

Figure 2: The *brc* bank indexing mechanism. This is the most common organization.

latency. We will show later how much higher the page hit percentage can be if we keep track of which page in each bank is opened. Since each bank is 32 megabytes for our hypothetical example, we need just a simple register, comparator, and state machine for each 32 megabytes—a little bit of hardware can go a long way.

5 Mapping Physical Address Bits to Memory Chip Pins

In the previous section, we discussed one organization of a memory system, with the least-significant address bits assigned to the column bits, the next address bits assigned to the row bits, and the remaining address bits assigned to bank selection schemes. While this is the most commonly seen organization, it is not the organization that maximizes the page mode hit rate. In this section, we will discuss alternative organizations.

In order to get an entire cache line out of the memory quickly, it is clear that column accesses should be used for the subsequent words in the cache line. Thus, with a 16-byte wide memory but 64-byte cache line, we will always want to use two of the column bits to cycle as we obtain the cache line. Also, with a 64-byte cache line, the low order six bits of the address are essentially ignored.

If we use the 4M chips described earlier, we have seven remaining column address bits. We have 11 row address bits. The remaining bits are used to index any internal banks and select groups of chips. We shall assume for simplicity that we have a power of two number of groups of chips, and that some of the address bits are demultiplexed to select the groups of chips. These demultiplexed address bits thus function as bank selection bits in exactly the same way (for the purposes of this paper) as the bank select bits in the SDRAM chips, so we shall lump the two types of bank select bits together in our subsequent discussion.

The mapping of the physical address bits from the processor or bus to the row, column, and bank select bits is our primary interest in this report; we refer to this mapping as the bank indexing scheme. The organization we have discussed so far we call *brc*, because the bits are assigned from most significant to least significant to the bank select bits, row select bits, and column select bits. This is shown in Figure 2. But there is nothing requiring this organization. In fact, so long as cache lines remain contiguous (so page mode hits can still be assumed for subsequent word accesses within a cache line), the partitioning of physical address bits among row, column, and bank select is totally arbitrary, so long as only as many address bits are used as there is physical memory to support.

On the other hand, to maximize page mode hit rate, one requirement is clear. The column bits should be less significant than the row bits, to maximize the preservation of spatial locality in

Figure 3: The *rcb* bank indexing mechanism (cache line interleaving). Hurts the page mode hit rate, but certain memory systems may have justification for using something similar to this.

Figure 4: The *rbc* bank indexing mechanism (page mode interleaving). Improves the page mode hit rate significantly, but we can still do better.

physical addressing to column addressing. This is why caches are organized to map adjacent words to a line, using the lowest address bits to select words and bytes within a line.

It is less clear how bank bits and row bits should be interspersed. In addition, some systems interleave banks at a cache line or lower level, in order to distribute memory traffic among banks. For instance, the Convex Exemplar uses a crossbar to connect four groups of processors to four physically separate memory banks. In order to balance the memory load among the four memory banks, sequential cache lines come from different memory banks. This organization is called *rcb*, as shown in Figure 3. We shall show that this reduces the page mode hit rate somewhat. On the other hand, since the bottleneck was assumed to be the bandwidth of the crossbar, it is a good engineering decision to maximize that by balancing the traffic.

One interesting way to organize the mapping is called *rbc*, shown in Figure 4, because the row bits are more significant than (some of the) bank bits. This organization is called 'page mode interleaving' and it is the subject of at least one patent (05051889).

There are many other organizations. Indeed, in this paper we will show that cache-oriented *rbrbc* as shown in Figure 5 improves the page mode hit percentage very significantly. Precisely which bits to select for bank indexing will be discussed at length.

6 Experiments

We used a simple model of a page mode memory controller on traces of several large applications to investigate the impact of various bank indexing schemes. We used four application traces. All traces included both instruction and data traffic. The first trace was a 142 million reference NASTRAN trace. The second was a 149 million reference Pro Engineering trace. The third was a 98 million reference Verilog simulation run trace. Each of these first three were virtual memory address traces generated by ptrace on a PA-RISC platform. The final trace was a 39 million reference physical address trace of TPC running on a 486 platform.

Since the memory system does not see every reference, but only those that miss in the cache or cause a cast-out, we filtered the traces by a cache model. The cache model we used was a 4 megabyte, 64-byte line, 4-way associative cache. We did not warm up the cache. The resulting set

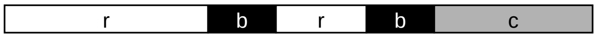

Figure 5: The *rbrbc* bank indexing mechanism. The best way to index memory banks.

of references (misses and cast-outs) we call a "miss trace".

To simulate a multiprocessor trace, we interleaved in a round-robin style the four miss traces, after offsetting the traces physically by adding a distinct large value to each that made sure there was no shared page among the applications. We used this as our fifth trace.

We assumed a 32-byte physical address. For simplicity, we neglected modeling the physical to virtual address translation; we will discuss the potential impact of this in a later section.

We ran simulations with the number of banks varying from 2 to 256 (and thus the number of bank bits from 1 to 8). We used values as described so far: 6 cache line index bits, 7 column index bits, 1 to 8 bank index bits, and 18 to 11 row bits, to total to 32 bits. Note that using additional row bits (which function as 'tag' bits) approximated a non-existent virtual to physical address mapping to reduce the size of the physical address space by mapping down the extra bits.

We could not try all possible combinations; assuming all row bits are more significant than all column bits, there are more than 2.5 million possible bank indexing combinations. At first, we just ran all bank indexing schemes that satisfied the regular expression $b^*r^*b^*c^*b^*$ (that is, some number of bank bits followed by all row bits followed by some number of bank bits followed by all column bits followed by some number of bank bits). Analysis of the results obtained led us to try all combinations of $b^*r^*b^*r^*b^*c^*b^*$ where the number of bank bits in positions to the right of the column bits were less than or equal to two. There are a total of $3,114$ different bank index schemes represented by these constraints.

7 Results and Analysis

We begin by presenting results showing those bank indexing schemes that maximized the page hit percentage for each trace and on average. These results are shown in Figure 6. First we will discuss what the bit patterns look like, and why, and next we will present the actual page hit percentage figures.

First, notice that for all of the best bank indexing schemes, the column address bits were the least significant bits. None of the best bank indexing schemes used any bank indexing bits less significant than the column bits. This indicates that interleaving is suboptimal.

Next, notice that almost all of the best bank indexing schemes have two subfields of bank indexing bits. The first subfield is adjacent to and more significant than the column indexing bits; the second subfield is coincident with some less-significant bits of the cache tag bits. This pattern holds without exception for the average results and for the 8-bank-bit results. The two phenomena behind these results we call 'contiguous pages' and 'cache-effect interleaving'.

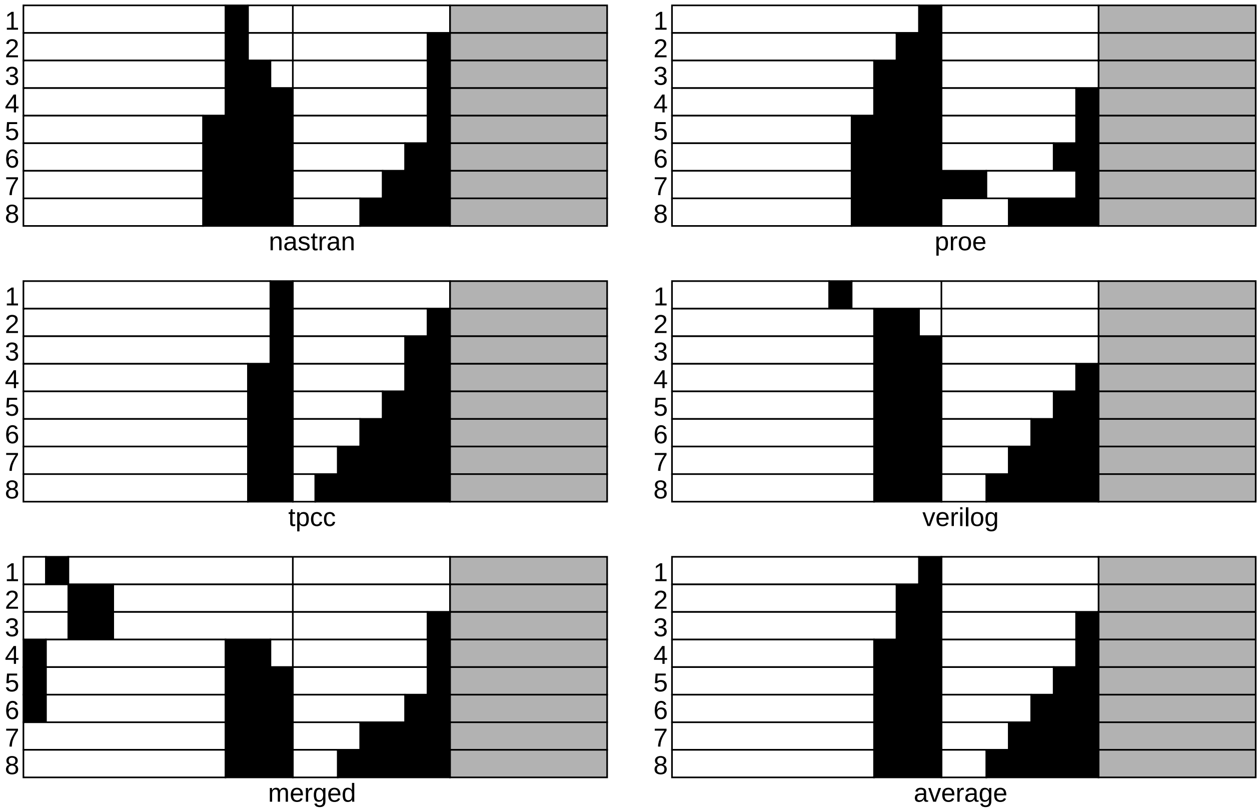

Figure 6: The best bit patterns for each of the traces and on average. Gray indicates the bit was used for column addressing, black indicates the bit was used for bank indexing, and white means the bit was used for row addressing. The number on the left is the total number of bank bits. The black line near the center of the addresses is the border between the cache tag and index bits. The average results are the bank indexing schemes that had the highest average page hit percentage over all five traces.

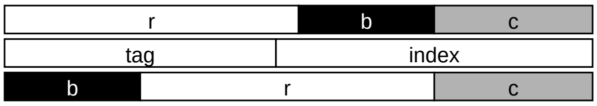

Figure 7: The *brc* and *rbc* schemes and their relationship to the cache index and tag bits.

To illustrate the contiguous page effect, consider what happens when some of the bits adjacent to the column bits are row bits. What this means in physical address space terms is that consecutive DRAM pages are in the same bank, but different rows. Remember that DRAM pages are moderately large (8K is typical). If n bits adjacent to the column bits are row bits, then 2^n pages from the same bank are adjacent. For instance, if n is seven, then 128 pages from the same bank are contiguous; this is a full megabyte.

If this megabyte happens to be a hot spot for an application (such as the stack, or a root B-tree page, or a symbol lookup hash table) then we will have many accesses to the same bank—but to different pages. Thus, for this large hotspot, we are thrashing among the pages within a a single bank.

If some bits adjacent to the column address are used to index banks, then we are distributing our hot spot across multiple memory banks. With multiple memory banks, we can hold open multiple pages for the single hot spot. This helps to distribute our data accesses among banks, utilizing all banks and increasing the page hit rate.

Put another way, hot spots that fit within a single page are handled best with column bits that are at the least-significant physical address bit positions, because only a single page needs to be opened for these hot spots, and it remains open as long as only that hot spot is accessed. Hot spots that span multiple pages are handled best by distributing those hot spots across multiple banks, so that the multiple pages can be held open simultaneously; this is the effect of using bank index bits that are adjacent to and immediately more significant than the column index bits.

The other effect, 'cache-effect interleaving', is even more important, as the results for few banks show. To explain what is happening, consider the standard *brc* and *rbc* schemes and a cache. We are using a 4 megabyte, 64-byte line, 4-way associative cache, so the lower six bits of the address are used for cache line offset, the next fourteen bits are used for cache indexing, and the high twelve bits are used for cache tags. In Figure 7 we see the superposition of the *brc* scheme, the *rbc* scheme, and the cache tag and index bits.

Let us imagine that a particular reference misses in the cache, and the victim line chosen to be replaced is dirty. In this case, we will generate both a read and a write to the memory system, and they will be adjacent or nearly so (depending on any write buffering done in the processor). Since both of these memory addresses belong to the same cache associativity set, their index bits will be the same, but their tag bits will differ. The tag bits that are most likely to differ are the least significant tag bits, while the most significant tag bits are likely to be the same. This is because the hot spots in an application tend to be fairly contiguous. It is also true for sequential access

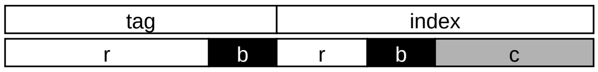

Figure 8: An optimal scheme and its relationship to the cache index and tag bits.

patterns that are larger than the size of the cache.

Referring back to Figure 7, we see the least significant tag bits in both the *brc* and *rbc* schemes are row bits. Thus, for the cast-out line and its replacement, the column and bank index bits will be the same, but the row bits will be different—so the two references are to different pages within the same bank. This means that at least one of the references will be a page miss. If we have a sequence of misses and cast-outs, spatial locality within a page will mean that most of the references will be misses—we will thrash, alternating between the pages used for cast-outs and the pages used for replacement lines within a single bank.

For many workloads, cast-outs make up 30% or more of the memory traffic, so this is a huge portion of our workload. And each cast-out will cause the replacement line to miss in the page as well, so our page miss percentage will probably be below 40% if we do not take this effect into account.

The way to solve this problem is simple—just select some of the bank index bits to coincide with the low-order bits of the cache tags, as in Figure 8. This way, the cast-outs and their replacement references will go to different banks, and thus separate pages can be held open for the cast-outs and their replacements. This effect is so dominant that for almost all of our traces, when only a few bank index bits are available, the best bank indexing scheme assigns them to low-order bits of the cache tags, as shown in Figure 6.

Taking advantage of cache-effect interleaving has additional beneficial effects. First, by eliminating almost all interleaved reads and writes to the same bank, write buffering becomes simpler; it is possible, with almost no performance loss, to fully sequentialize writes and reads to the same bank, since any bank at one given time will be seeing almost entirely reads or almost entirely writes. Without cache-effect interleaving, the frequency of a write followed by a read to the same bank is significantly higher, and the hardware cost of checking the write buffer for a following read, while small, certainly complicates the memory controller.

The second minor advantage of cache-effect interleaving has to do with SDRAMs. These chips support highly pipelined reads and highly pipelined writes, allowing a transfer per cycle from among the various banks. However, these chips also have a significant read after write or write after read penalty (typically, 30 ns in each case). By grouping reads and writes to the same SDRAM group, by using bank select bits that select different physical chips rather than internal bank select bits for cache-effect interleaving, the frequency of read after write or write after read is reduced.

Some memory controllers support memory-side write buffers to minimize the latency of writes and following requests to the same bank. Read requests must be checked against the contents of these buffers. With cache-effect interleaving, the amount of read/write interleaving to a particular bank is significantly constrained. Thus, there may be very little performance impact of delaying reads until all reads pending for that bank are completed, thus eliminating the write-buffer check.

If we consider the average results, which are representative of a mixed workload, we see that the optimal bank indexing schemes take advantage of both of these effects, first populating the cache-effect interleave bits, followed by some page-mode interleaving bits.

Figure 9 shows the quantitative results showing how much gain is attainable using such carefully designed bank indexing schemes. Table 1 summarizes the average results.

Bank bits	brc	rbc	optimal
0	0.250	0.250	0.250
1	0.254	0.288	0.403
2	0.303	0.338	0.513
3	0.346	0.378	0.633
4	0.353	0.411	0.718
5	0.362	0.431	0.765
6	0.366	0.448	0.800
7	0.419	0.463	0.834
8	0.495	0.690	0.866

Table 1: The attainable page hit percentage for different bank indexing schemes.

The first thing to note is that the curves for the nastran, proe, verilog, and average traces look very similar. In each graph, the optimal scheme is much better than either *brc* or *rbc*. Also, the difference between the optimal (the best bank indexing scheme for that particular trace) and the average-optimal (the best bank indexing scheme on average over all of the traces) is negligible. Finally, there is a significant jump in performance of the *rbc* scheme at eight bits; this is because the number of bank bits has increased to the extent that one of them now coincides with the least significant cache tag bit.

The results for tpcc and the merged trace deserve more attention. The tpcc trace is unique in that its total memory usage is significantly less than the size of the cache—so almost all of the memory traffic is cold-start traffic. For this traffic, which is mostly sequential, having just a single page open at a time led to almost a 60% page hit percentage. Because there were few flushes or conflict misses, the effect of cache-effect interleaving was measurable but small. The merged trace also showed interesting behavior. For small numbers of banks, the applications interfered with each other significantly. For larger number of banks, the curves closely resembled the "normal" curves.

From the table above, we see that for a typical memory system with 64 banks, the average page hit percentage ranged from 36.6% for the typical *brc* scheme, to an improve 44.8% for the page-interleaved *rbc* scheme, to an impressive 80.0% for the optimal scheme. These results make it clear that carefully indexing the memory banks can have a significant impact on the page hit percentage and thus on the average memory latency.

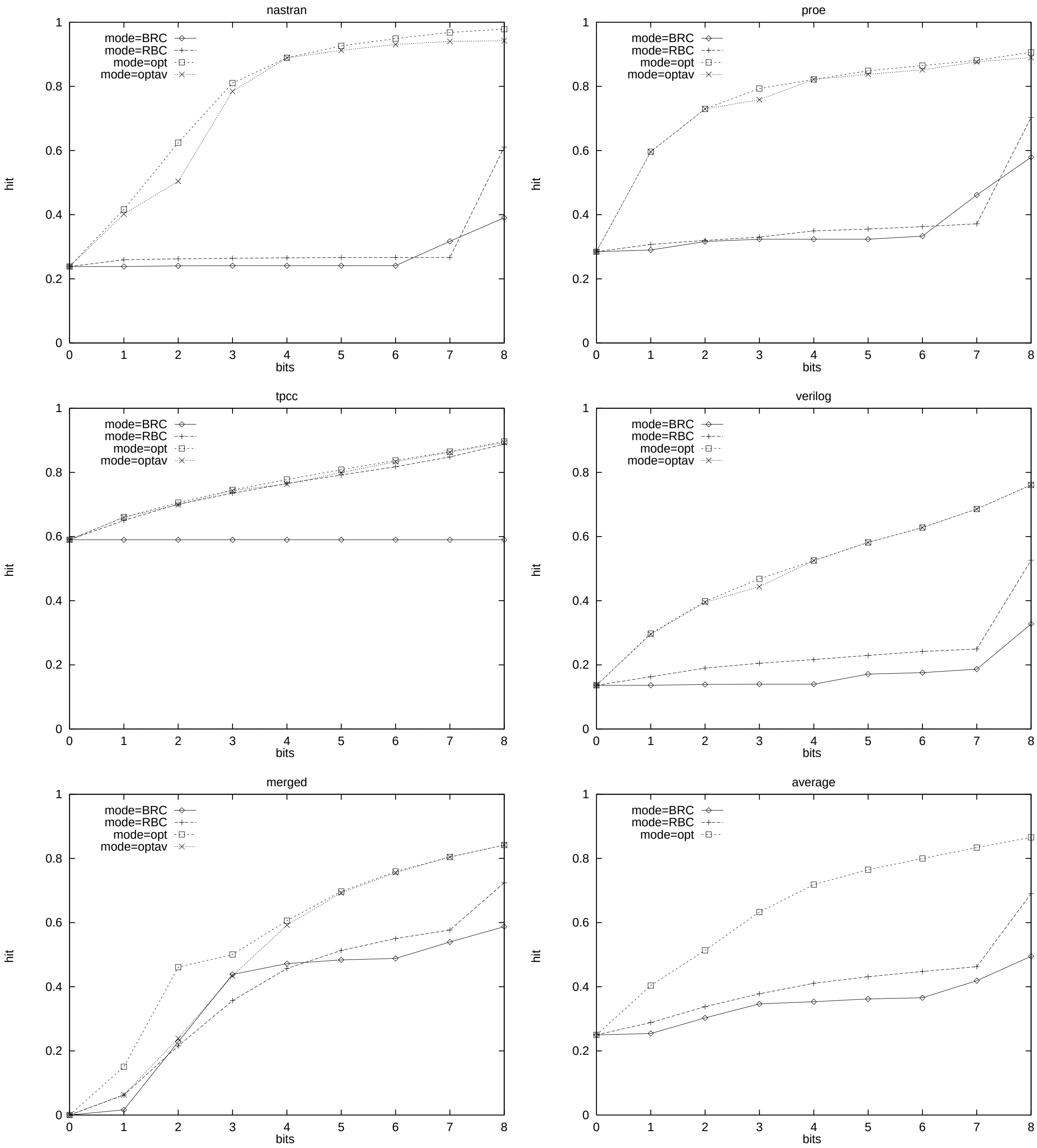

Figure 9: The quantitative impact of the bank indexing scheme. The four lines are the two 'normal' schemes, *brc* and *rbc*, followed by the optimal and finally the optimal for the average. The optimal is the scheme that works best for that particular trace. The optimal for the average is the scheme that works best on average, applied to that trace.

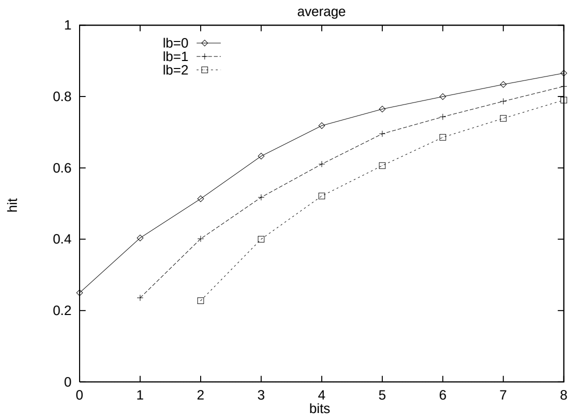

Figure 10: The impact of interleaving on the page hit percentage. The three curves show no interleaving, interleaving two banks, and interleaving four banks; each point is the optimal bank indexing scheme for that interleaving requirement. In every case, interleaving has worse performance than non-interleaving, even if the number of banks is increased by the interleave factor.

Finally, we turn our attention to cache-line interleaving across banks. In many systems, there are good reasons to consider this. In a distributed shared memory system where separate memory banks are accessed through distinct bandwidth-limited communication channels, distributing the memory load across the banks is important so that one channel does not prematurely become a bottleneck. In memory systems that do not take advantage of page mode, cache-line interleaving can be used to hide the precharge of one bank behind the access time of another bank for sequential accesses.

In a page mode system, however, sequential accesses are page-mode hits with very high probability, so precharge and access delays are automatically minimized. The problem with cache-line interleaving in a page-mode system is that most misses that show spatial locality to each other do so within a very small region—typically much smaller than a page size. This small region might be a record structure or a hash table or a stack. With interleaved cache lines, multiple pages must be opened for these hot spots, while with non-interleaved cache lines, only a single page suffices.

Our experimental data supports these conclusions. Figure 10 illustrates this; interleaving banks by cache line hurts the page hit percentage significantly. Interleaving two banks, even with twice as many banks, has a poorer page hit percentage than not interleaving at all.

If interleaving is desired to balance the memory load, it is recommended that the interleaving take place at a coarser granularity—say, four cache lines rather than one. This will help minimize the negative impact of interleaving.

8 Memory-Side Prefetching

If 87% of our accesses are page-mode accesses, anything we can do to further minimize their latency can have a large effect. One technique that can be useful is memory-side prefetching.

Consider the circuitry between a collection of DRAM chips or SIMMs and the system bus. Because DRAM chips have limited drive capability and operate relatively slowly, and because there are significant loading restrictions on most busses, there is usually a bidirectional multiplexor and buffer between the chips and the DRAM. This component is usually a bit-sliced ASIC. Typically, the bandwidth on the memory side of this component is made to match or slightly exceed the bandwidth on the bus side of the component, but it is fairly easy to increase the number of inputs to this multiplexor to increase the memory-side bandwidth. This is not typically done, because such excess bandwidth would normally go unused.

But memory accesses to the same bank of memory, especially under the optimal bank indexing schemes, show a large degree of sequentiality. That is, over half of the time, the cache line requested from a particular bank is the next sequential cache line in that bank.

Speculative memory-side prefetching can trivially be done in such a system. If, after one read is completed, there are no pending requests for that bank, continue the access to transfer the next consecutive cache line into the bus drivers. Since this is just a continuation of a page-mode access, this takes little time, and since we are using a page-mode controller that keeps the page open, this does not hurt any pending accesses either. If this is done, over half the time, a memory read request will find the data already in the bus drivers, or already on the way to the bus drivers, when the read is issued onto the bus.

In order to verify this idea, we used our page-mode hit rate traces and models to calculate the percentage of time that a memory read was the next sequential line in a page after the previous access to that bank. The results are summarized in Table 2 and graphed in Figure 11. Note that,

Bank bits	brc	rbc	optimal
0	0.262	0.262	0.262
1	0.266	0.290	0.343
2	0.313	0.328	0.417
3	0.352	0.358	0.493
4	0.357	0.383	0.536
5	0.359	0.393	0.557
6	0.361	0.399	0.570
7	0.398	0.403	0.582
8	0.430	0.507	0.585

Table 2: The attainable sequential prefetch hit rate for reads under different bank indexing schemes.

even with a poor bank indexing scheme, prefetch hit rates in excess of 35% are easy to attain. With the optimal bank indexing scheme and 256 banks, a prefetch hit rate of 58.5% can be attained—

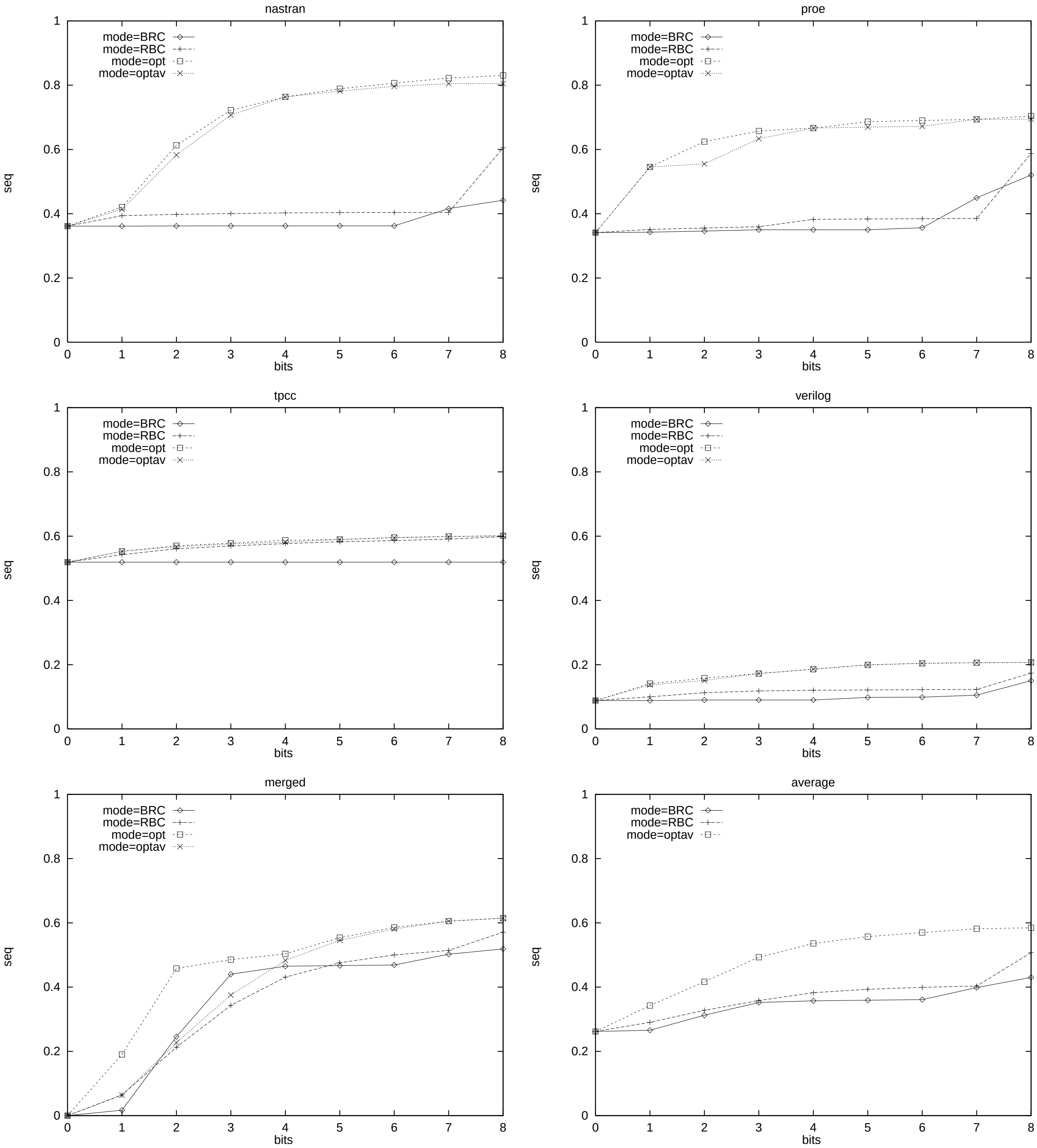

Figure 11: The frequency of success of sequential prefetching. The four lines are the two ‘normal’ schemes, *brc* and *rbc*, followed by the optimal and finally the optimal for the average. The optimal is the scheme that works best for that particular trace. The optimal for the average is the scheme that works best on average, applied to that trace.

most memory reads will have no latency whatsoever due to the memory chips; all will be due to the bus and controller.

For the applications nastran, proe, and tpcc, the prefetching success rate was good to excellent, up to 80% for nastran. For verilog, the prefetching hit rate was not very good due to the access patterns; if in a particular design prefetching does have a potential negative performance impact, prediction might be used to automatically turn off prefetching where it would not work well.

In the case of SDRAMs where there are multiple banks per chip, there may be a conflict for resources when no accesses are pending for any banks in a chip, yet there are multiple banks with previous reads. Which bank should prefetching then be done from? In our experiments, we have found that a simple single-bit predictor can give an accuracy of about 88% in whether a particular prefetch from a bank will be used. Thus, a single-bit predictor can be maintained for each bank, and prefetching done preferably from those banks for which the prediction is that the prefetched line will be used.

Memory-side prefetching is much easier than speculative prefetching; because prefetching is only done to the bus drivers, there is no impact on the cache coherency. Since prefetching might be done only when there are no pending writes, the write buffers need not be compared with the prefetching address either.

9 Impact of Cache Size

A 4 megabyte, 4-way associative cache is a rather large cache for current systems. We also evaluated cache indexing schemes for two additional cache sizes, and received nearly identical results.

The additional cache sizes we evaluated were a 512 kilobyte, 2-way associative cache and a 64 kilobyte, direct-mapped cache. The optimal bit patterns followed essentially the same pattern as those resulting for the 4 megabyte cache, except of course the cache tag bits were shifted some number of places to the right. Indeed, for a 64k direct-mapped cache, the results for the *rbc* scheme were optimal for seven or more bank index bits.

The primary reports are reported in Figure 12, and summarized in Tables 3 and 4. The primary thing to note is that a very high page hit rate is attainable with small caches as well as with large. Indeed, the page hit rate is higher for the 64k cache size, at 92%, than it was for the 4M cache size, at 87%. In addition, while the sequential prefetch hit rate decreases as the page size decreases, a non-negligible sequential page hit rate is still attainable for small cache sizes.

As the caches get smaller, the *rbc* scheme becomes closer and closer to optimal with fewer banks. This is because this scheme is equivalent to the optimal scheme, distributing the bank bits between those bits adjacent to the column bits and the cache tag bits, as the cache tag bits get closer to the column bits.

Thus, it appears that selecting a good bank indexing scheme is as important with smaller caches as it is with the large 4M cache size.

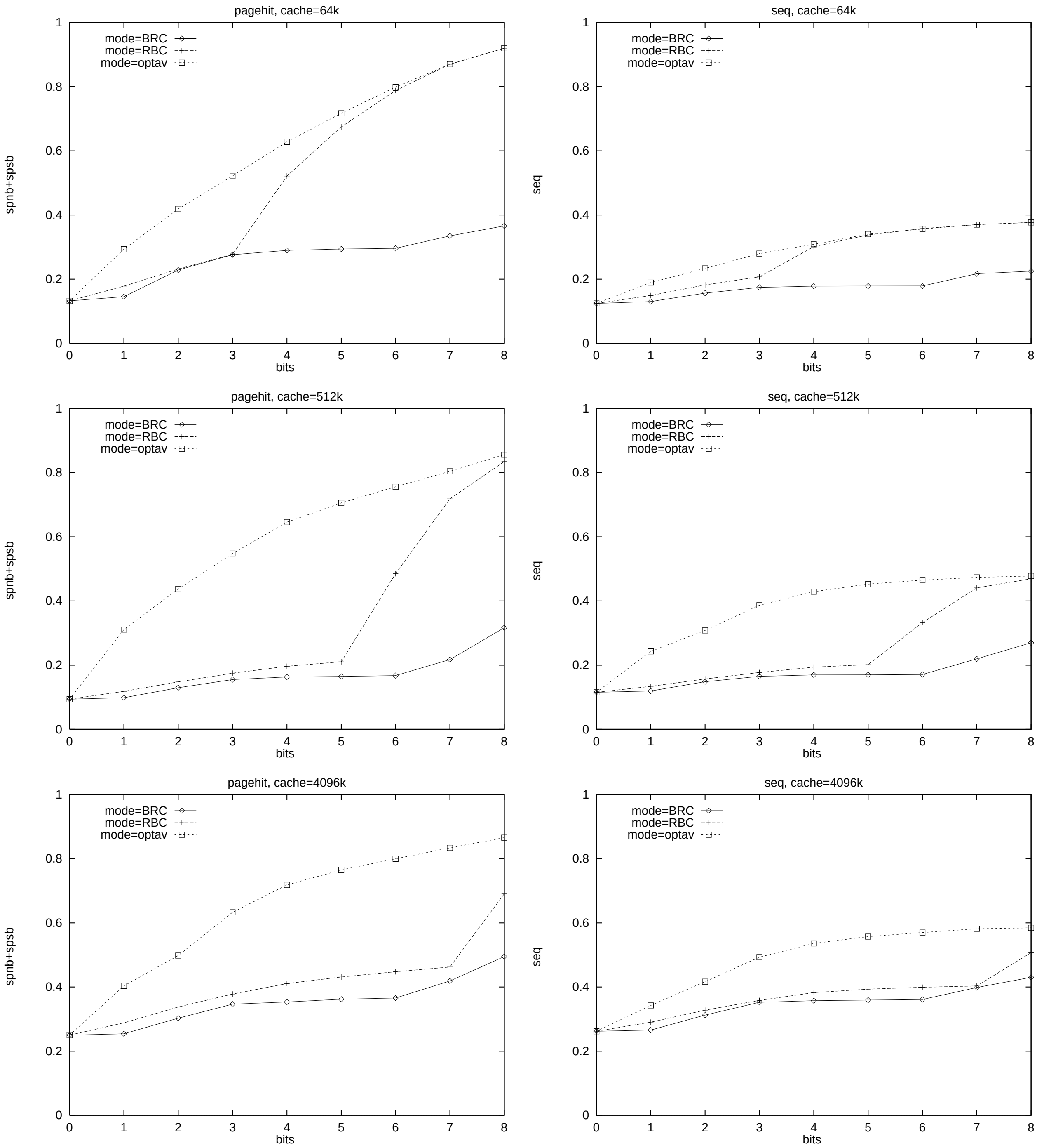

Figure 12: Page hit and sequential prefetch hit rates for different cache sizes. The three lines are the two 'normal' schemes, *brc* and *rbc*, followed by the optimal; in all cases, only the average of the five workloads is shown.

Bank bits	64k	512k	4M
0	0.132	0.094	0.250
1	0.293	0.311	0.403
2	0.419	0.438	0.498
3	0.522	0.548	0.633
4	0.628	0.646	0.719
5	0.717	0.706	0.765
6	0.799	0.756	0.800
7	0.870	0.804	0.834
8	0.920	0.856	0.866

Table 3: The attainable page hit rates for different cache sizes.

Bank bits	64k	512k	4M
0	0.124	0.115	0.262
1	0.189	0.243	0.343
2	0.234	0.309	0.417
3	0.280	0.387	0.493
4	0.309	0.429	0.536
5	0.340	0.453	0.557
6	0.356	0.465	0.570
7	0.370	0.474	0.582
8	0.377	0.478	0.585

Table 4: The attainable sequential prefetch hit rates for different cache sizes.

10 Software Issues

Three of the traces we used in this study were virtual address traces. The fourth was a physical address trace, but for a particular operating system. The memory system sees physical address traces, which differ in most high address bits from the virtual address traces. In this section, we discuss the impact of the virtual to physical translation and present a software technique that can help improve the page hit percentage, even if a poor bank indexing scheme is used by the hardware.

It is very difficult to predict what characteristics a virtual to physical translation might have for a particular system with a particular workload. The operating system sets up the virtual to physical translation, doling out pages as they are needed to satisfy page faults. Some operating systems attempt to minimize the impact of swapping I/O on system performance by mapping or unmapping several contiguous pages on each fault, where possible. Others, on hardware that allows it, attempt to minimize TLB thrashing by keeping as many virtually contiguous pages also physically contiguous (sometimes referred to as page coloring). Some operating systems attempt to minimize cache conflicts by allocating pages contiguously, so large contiguous hot spots do not conflict with themselves by having pages that share the same cache index.

The mapping depends a lot on the previous and current system activity, and on how the operating system stores and accesses the physical free page list. If the system has sufficient RAM so that little or no swapping is done (as will be the case in high-performance systems), it is possible that almost all virtually contiguous pages will be physically contiguous as well. In addition, certain portions of the address space, such as kernel space and file buffer space, is generally set up at kernel boot time and therefore is both virtually and physically highly contiguous.

The two effects we have presented in this paper are impacted by the virtual to physical translation in different ways. The page interleaving technique works by minimizing the interference of large hot spots with themselves on the memory banks, and by distributing all hot spots evenly over the various banks. If page coloring is used for whatever reason, then the bits used for page interleaving will correlate well between physical and virtual addresses, so this technique will be preserved. If page coloring is not used, then it is unclear what correlations might exist, and fewer conclusions can be drawn as to which bits might work the best.

The second effect, cache-effect interleaving, is mostly dependent on the last-level cache itself, which is almost always physical. Thus, our conclusion that some bits from the cache tags should be used for bank indexing is independent of the virtual to physical address translation. On the other hand, we did assume that the low-order tag bits would show the most variation. For virtual to physical address translations that are highly contiguous, this will certainly be true. For processes whose pages are allocated sequentially or approximately sequentially in physical memory, this will also be true. In other cases, it may be sufficient to select other cache tag bits and get equal performance.

Indeed, it is possible for the operating system, knowing how the banks are indexed, to intentionally spread out the pages of a process so that the bits that are likely to differ physically on cast-outs and replacements are those that are most likely to differ virtually. For instance, if the highest-order physical address bits are used to index the cache, the operating system could ensure that the pages given to a particular process are distributed evenly over the banks by allocating them so they have different high-order physical address bits. In some sense, this is the converse of page coloring. Indeed, using randomization in the physical to virtual address translation for those bits that are cache tag bits can essentially mute correlations in the high-order bits, so any bits that are cache tag bits can be used to index the banks and gain the effect of cache-mode interleaving.

Of course, this will not work for those regions that are otherwise constrained to be physically contiguous (as the kernel region and file buffers sometimes are). And this also minimizes the possibility of using a single, sizable TLB entry to map a very large region of memory. But in the absence of a good bank indexing scheme, this technique might be used in an operating system to help increase the page mode hit rate even on a system that selects an unfortunate bank indexing scheme, like *brc*.

To test out this idea, we performed a hash on the top 20 bits of our original traces (assuming a 4K operating system page), ran them through our cache model, and then evaluated our bank indexing schemes against the resulting miss traces. This mimics an operating system that performs a totally random placement of virtual pages on physical pages. In such a system, there should be

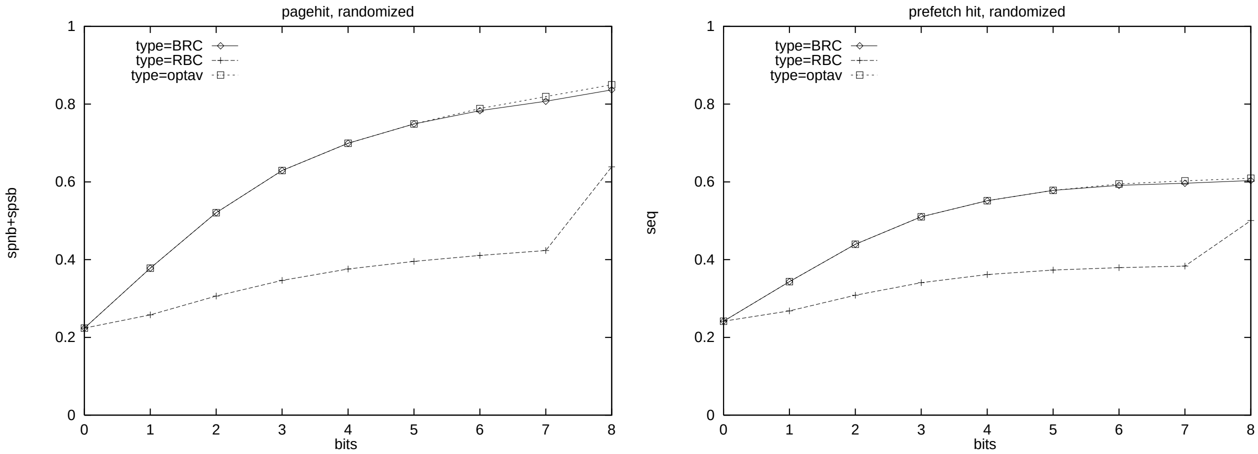

Figure 13: Page hit and sequential prefetch hit rates under a randomized virtual to physical mapping. The three lines are the two 'normal' schemes, *brc* and *rbc*, followed by the optimal; in all cases, only the average of the five workloads is shown.

no differences in correlations among the various bits used for cache tags, or among the various bits used for cache index, but more significant than the page index bits. Thus, we only evaluated bank indexing schemes with various numbers of bank bits in the cache tag region, cache index region but more significant than the column bits, and cache index region but less significant than the column bits.

Under such randomization, the *brc* scheme places all bits in the cache tag region; since the correlation among the high bits has been destroyed by the randomization, using high-order tag bits is identical to using low-order tag bits. The *rbc* scheme maximizes the number of bits adjacent to the columns, and then places the remainder in the cache tag bits, as before. The results are shown in Figure 13 and in Tables 5 and 6. The results show that, in the presence of complete randomiza-

Bank bits	brc	rbc	optimal
0	0.224	0.224	0.224
1	0.378	0.258	0.378
2	0.521	0.306	0.521
3	0.629	0.347	0.629
4	0.699	0.376	0.699
5	0.749	0.395	0.749
6	0.783	0.411	0.789
7	0.807	0.423	0.819
8	0.837	0.639	0.850

Table 5: The attainable page hit rates under a randomized virtual to physical page mapping.

Bank bits	brc	rbc	optimal
0	0.242	0.242	0.242
1	0.343	0.268	0.343
2	0.439	0.308	0.439
3	0.510	0.341	0.510
4	0.552	0.362	0.552
5	0.578	0.373	0.578
6	0.591	0.379	0.595
7	0.596	0.383	0.603
8	0.603	0.501	0.609

Table 6: The attainable sequential prefetch hit rates under a randomized virtual to physical page mapping.

tion, the *brc* scheme is optimal until the number of banks exceed 64, and even then it is very close to optimal. This is because the page interleave effect is obtained by using the high-order address bits as easily as the low-order address bits, because the randomization has eliminated any essential differences in correlations among the bits. Interestingly, the *rbc* scheme performs very poorly under randomization. These two results are the converse of the results without randomization: *rbc* works poorly with randomization, while *brc* works poorly without randomization. Note also that with or without randomization, *rbc* is far from optimal; if no cache tag bits are used for bank indexing, cast-outs will almost always cause page-mode misses.

Another interesting result is that the prefetching hit rate is enhanced by randomization; with 256 banks and optimal bank indexing, randomization yields a 61% prefetch hit rate, while the unrandomized stream yielded a 59% prefetch hit rate. Again, we see a significant jump for *rbc* at 8 index bits (256 banks) because one cache tag bit is suddenly used for bank indexing.

These results are dependent on complete randomization, not just a simple page replacement policy. A simple page replacement policy will generally leave significant correlations among bits for large regions and for long periods of time. Also, if the operating system does page coloring, and the maximum size of memory region over which coloring is done is larger than a page-mode page for the memory system, then the address bits immediately more significant than the column bits should be used to index the banks, otherwise the page interleaving effect will be lost despite randomization.

We believe that the operating systems of the future will depend more and more on page coloring to minimize I/O overhead, TLB thrashing, and other effects. Thus, the virtual to physical address translation will show more and more uniformity, and the effects reported in this paper will become increasingly pronounced.

In any case, real systems will show some combination of randomization, page coloring, and direct mapping. Thus, the overall results for a real system should lie somewhere between the results for the random mapping, as described in this section, and the results for a direct mapping, as shown in the previous sections. A mapping that works for all of these different characteristics—our optimal mapping that distributes the bank index bits among the cache tag bits and the bits adjacent to the column bits—is clearly called for.

11 Performance Impact

Everything we have discussed this far was aimed at maximizing the page hit ratio. This, however, does not translate directly into performance improvement; if the bus bandwidth is the bottleneck, decreasing the memory latency will probably not have a large effect on performance. In this section, we briefly discuss the potential performance increase due to maximizing the page mode hit rate.

The impact on the memory latency at the memory chips is pretty clear; the critical word latency decreases from about 60 ns to $90 - 80p$ ns, where p is the page mode hit percentage. If we assume 256 banks, we should be able to attain $p = 0.866$, so the average critical word latency is thus 21 ns, a savings of 39 ns.

If we further assume a memory-side prefetching system, the memory-side latency of half of the requests is zero, since the data is already in the bus drivers—this reduces the average latency by another 5 ns to only 16 ns.

Let us assume we have an aggressive memory controller built-in to the CPU chip or cache controller, and its contribution to the memory latency totals 10 ns. Thus, the total average memory latency decreases from 70 ns to 26 ns. Let us further assume that critical word latency accounts for 50% of the performance of the system on a particular workload when using the non-page-mode controller. Thus, there is about 70 ns of computation for every 70 ns of memory fetch. By decreasing the average memory latency by 44 ns, we improve the performance of the system by approximately 46%.

Indeed, cacheless PCs have attracted some attention lately. (They are not really cacheless, since the CPUs have on-chip cache; rather, they simply eliminate any expensive external cache.) Using an aggressive page mode memory controller that takes advantage of the ideas in this paper could lead to a very modestly priced, yet high performance system.

In a multiprocessor system with sixteen CPUs connected with a crossbar, we might have a memory controller and interconnect latency of 100 ns in addition to the memory latency itself. If we still optimistically assume that memory latency accounts for 50% of the performance, then decreasing our average memory latency from 170 ns to 126 ns will improve performance by 15%.

Page mode optimizations have been unfairly denigrated in large-scale multiprocessing systems, since intuition would suggest that the various processors would interfere with each other to such an extent that page mode would not work well. On the contrary, since the number of memory controllers and memory banks typically scale up with the number of processors, just to balance bandwidth, it is very possible to get equally high page-mode hit rates if the banks are indexed correctly. Not using page mode in such a system is just throwing away performance.

Of course, a real system is not this simple. Real systems often show bursts of activity during which the processor is the bottleneck (such as cache-resident activity), bursts of activity during which memory system bandwidth is the bottleneck (such as copying), and bursts of activity during which memory system latency is the bottleneck (such as pointer chasing). Nonetheless, it is clear that selecting the bits to index the memory banks carefully can have a positive impact on performance for a relatively cheap investment.

12 Implementation Complexities

The cost of implementing bank indexing schemes as described here is primarily in additional complexity of the memory system. Design of the bank control hardware itself is fairly straightforward; the main key is to avoid adding hardware to the critical path. Since a bus has significant loading constraints, there is almost always an on-chip and off-chip delay between the bus pins and the memory pins; these two delays almost certainly dominate the effect of some additional gates on the chip itself to control the bank mode.

One complexity is that the memory system must support configuration. Since it is likely that processors with different cache organizations may be attached to the same memory controller design, the memory system must allow the bank index bits to be selectable from the various physical address bits available. This is very similar to the work that must currently be done anyway in order to support various types of SIMM and DIMM DRAM packaging.

Even in those systems where such a complex memory controller is considered too expensive, even a simple baseline controller should exploit page mode of at least one bank. It's a very small price to pay to improve a very critical performance metric.

13 Limitations of This Study

There are some complexities we have glossed over in this discussion. For instance, most DRAMs have a maximum page open time that cannot be exceeded; often this is of the same order of magnitude as the refresh interval. Since the refresh interval is typically 60 μs, and the total cycle time is typically 100 ns, 600 accesses can take place before the page must be closed. Thus, the impact of the maximum page open time on performance is probably minimal.

We have also not considered any particular virtual to physical mapping algorithm in detail. It is clear that this algorithm can have a large impact on the results presented here. The two effects described are preserved through any virtual to physical mapping; the primary thing the mapping impacts is which particular bits to select for each subfield. Experimentation with an existing system, both in terms of its virtual to physical address mapping and its memory controller design would be fruitful areas of investigation.

We have also not discussed the actual performance impact in any great detail, primarily because this depends on so many things. Again, experimentation with a real machine on real workloads would lead to interesting support for the conclusions of this paper.

14 Discussion

The primary results of this study are as follows:

- Maximizing the page mode hit percentage is important when designing a memory system that uses page mode.
- Page interleaving can significantly boost the page mode hit percentage.
- Taking cache effects into account can further boost the page mode hit percentage.
- Doing both synergistically maximizes the page mode hit percentage.
- Memory-side prefetching can inexpensively reduce the latency of the page-mode accesses.
- Do not interleave banks on a cache line basis if you care about page mode hit percentage. If you must balance the memory load across banks, use a coarser interleave granularity.
- The OS can help maximize the page mode hit percentage—whether the hardware supports a good bank indexing scheme or not.
- Large-scale multiprocessors can take advantage of page mode just as well as uniprocessors, since the number of banks typically scales with the number of processors.

In summary, page mode is a performance opportunity that has not yet been sufficiently exploited. We hope this paper leads to more consideration of page mode effects and memory-side prefetching in future designs.

15 References

[WM95] Wulf, Wm. A. and Sally A. McKee: Hitting the Memory Wall: Implications of the Obvious. ACM Computer Architecture News, Vol. 23, No. 1, March 1995, pp. 20–24.