Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance?

Vinodh Cuppu and Bruce Jacob
Dept. of Electrical & Computer Engineering
University of Maryland, College Park
{ramvinod,blj}@eng.umd.edu

ABSTRACT

Given a fixed CPU architecture and a fixed DRAM timing specification, there is still a large design space for a DRAM system organization. Parameters include the number of memory channels, the bandwidth of each channel, burst sizes, queue sizes and organizations, turnaround overhead, memory-controller page protocol, algorithms for assigning request priorities and scheduling requests dynamically, etc. In this design space, we see a wide variation in application execution times; for example, execution times for SPEC CPU 2000 integer suite on a 2-way ganged Direct Rambus organization (32 data bits) with 64-byte bursts are 10–20% lower than execution times on an otherwise identical configuration that uses 32-byte bursts. This represents two system configurations that are relatively close to each other in the design space; performance differences become even more pronounced for designs further apart.

This paper characterizes the sources of overhead in high-performance DRAM systems and investigates the most effective ways to reduce a system's exposure to performance loss. In particular, we look at mechanisms to increase a system's support for concurrent transactions, mechanisms to reduce request latency, and mechanisms to reduce the "system overhead"—the portion of the primary memory system's overhead that is not due to DRAM latency but rather to things like turnaround time, request queueing, inefficiencies due to read/write request interleaving, etc. Our simulator models a 2GHz, highly aggressive out-of-order uniprocessor. The interface to the memory system is fully non-blocking, supporting up to 32 outstanding misses at both the level-1 and level-2 caches and split-transaction busses to all DRAM banks.

1 INTRODUCTION

As many recent studies have shown, the memory system is one of the primary bottlenecks in current systems. Further, a number of studies show that, within the memory system, the memory bus accounts for a substantial portion of the primary memory's overhead. For example, Schumann reports that, in Alpha workstations, 30–60% of primary memory latency is attributable to system overhead rather than to latency of DRAM components [22]. Brown and Seltzer cite memory-bus turnaround as responsible for a factor-of-two difference between predicted execution time and actual measured execution time on a Pentium Pro system [1]. Cuppu, et al. demonstrate the inability of a 128-bit 100MHz (1.6 GB/s) memory bus to keep up with high-performance DRAMs [5]. Bryg, et al. estimate that 20–30% of the Hewlett-Packard memory bus bandwidth is lost to dead cycles in back-to-back read/write transactions [2].

There are a number of paths developers and researchers have taken to reducing the overhead of the primary memory system. These have largely been divided into approaches that are focussed on the DRAM component and those that are focussed on the system or bus component. For example, a simple DRAM-oriented approach has been to increase DRAM bandwidth. This is the tack taken by the PC industry recently, with the widespread shift from 800 MB/s PC100 SDRAM systems to 1.1 GB/s PC133, 1.6 GB/s Direct Rambus, and 2.1 GB/s DDR266 SDRAM systems. This brings the memory bandwidth of the PC up to that of traditional RISC workstations, such as several UltraSPARC and Alpha models, and to within an order of magnitude of many server-class machines.

Another approach is to reduce DRAM latency. DRAM vendors have recently announced numerous core variations that improve access time. For example, Enhanced Memory System's ESDRAM improves performance over regular SDRAM by adding an SRAM cache for the full row buffer, thereby allowing precharge to begin immediately after an access and DRAM writes to go directly to the core without destroying read locality [5, 7, 6]. Fujitsu's FCRAM subdivides each internal bank by activating only a portion of each word line, thereby reducing capacitance on the word access and improving access time over that of standard SDRAM to roughly 30ns [9, 10]. MoSys takes this a step further and subdivides the on-chip storage into a large number of very small banks (on the order of 32KB each), reducing the access time of the DRAM core to nearly that of SRAM [39, 19, 10]. Several vendors have placed large amounts of SRAM onto the DRAM die, in addition to the row buffers, in an attempt to reduce latency. For example, NEC'S VCDRAM places a set-associative SRAM buffer on the die that holds an implementation-defined number of sub-page (typically 10–100), where a sub-page is a subset of the bits activated by a column access and is on the order of 16–32 bytes [6, 10].

Recent studies show that these DRAM-oriented approaches do reduce application execution time [5, 7]. However, focussing on the DRAM alone is not enough; we note that, even with zero-latency DRAM access, the overhead of the primary memory system would not reduce to zero, because bus transactions still require time. To begin with, there is the obvious time to transfer addresses and data over the bus to and from the DRAM subsystem. In addition, factors such as turnaround time, queueing delays, and inefficiencies due to asymmetric read/write request shapes on an in-order bus all add together to produce a sizable overhead. In multiprocessor systems, the overhead is even larger, due to arbitration and cache coherency protocols—moreover, many uniprocessor systems share the memory bus with graphics chips in an organization that effectively makes the uniprocessor system behave like a multiprocessor.

In addition to efforts aimed at improving DRAM devices, we must also improve the connection between the CPU and the DRAM devices—we must improve the overhead of the memory bus. There are a number of approaches one can take, including changing the

Copyright © 2001 IEEE. Published in the *Proceedings of the 28th International Symposium on Computer Architecture*, June 30–July 4, 2001 in Göteborg, Sweden. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

nature of the request stream, increasing the available concurrency in the memory system, decreasing the latency of the memory system, and attacking system overhead that is due to queueing, turnaround, etc. An example of the first type of optimization is access reordering, which compacts sparse data into densely-packed bus transactions, thus reducing both the number of bus transactions and, potentially, the duration of each. This has been explored by many in academia, including McKee, et al. and the Impulse group at Utah [22, 21, 24, 4, 15].

Another approach is to increase available concurrency on the memory channel. This can be done by supporting concurrent access to different, independent DRAM banks via the same channel or by replicating resources and providing multiple, independent channels to different DRAM banks—or by a combination of the two. Concurrent access on the same channel is usually through pipelined requests or split-transaction requests [23, 21, 2], the primary difference being that split-transaction busses allow requests to be handled and responses delivered to the requestor/s out-of-order. Note that concurrency on the same channel can be provided at either the bus-organization level or the DRAM-interface level, or both. For example, the Direct Rambus device interface is fully pipelined, supporting up to four concurrent requests [21]. Concurrency over multiple DRAM channels has been common in high-end server-class machines for some time, but the advent of narrow, high-speed DRAM interfaces has made it economically feasible to provide multiple DRAM channels in even the value-end of the market. For example, the Sony Playstation 2 has two Direct Rambus channels on board [8], and the Compaq Alpha 21364 will support four or more Direct Rambus channels [12].

The third approach is to decrease the latency on the channel. This is not the latency due to the access time of the DRAM core; it is the latency that is due to stalling for memory-bus resources. For example, if two read requests arrive at the memory system back-to-back, the second must stall until the first one gives up the data bus, increasing its latency by the duration of the first. Another example: if a read request to a particular DRAM bank arrives while the bank is precharging its bit-lines, the request must stall. Holding all else constant, narrow channels tend to have a higher number of databus-related stalls than wide channels, because the end-to-end request time is longer than on a wide channel. Data bursts of long duration tend to result in a higher number of stalls, but the burst length also affects how well the precharge time is hidden, and longer bursts expose less of the precharge overhead to later requests.

The last of the bus-oriented approaches is that of addressing system overhead, including bus turnaround time, dead cycles on in-order busses due to asymmetric read/write shapes, queueing overhead, coalescing requests that are queued (when possible), and dynamic re-prioritization of requests. Parameters that affect this overhead include the shapes of read and write requests (delaying the data packet for a write makes the two symmetric and therefore easier for a memory controller to schedule) and the queue size (larger queues offer more opportunity for delaying write requests until read traffic has subsided, and requests that sit in the queue for a longer time have an increased probability of being coalesced or cancelled).

The obvious question is *which of these factors is the most important*? Which contributes the most to the overhead of the primary memory system, and which can be best exploited to reduce application execution time? In this paper, we quantify these overheads, investigate methods to address the overheads, and measure the effectiveness of each method by varying the degree to which it is used (e.g. by varying the bandwidth of a DRAM channel, number of banks, etc.). We find that it is essential to support concurrency in the DRAM system, but not to the extent that it adversely affects transaction latency. Methods that improve concurrency at the expense of latency include using multiple channels and small burst lengths.

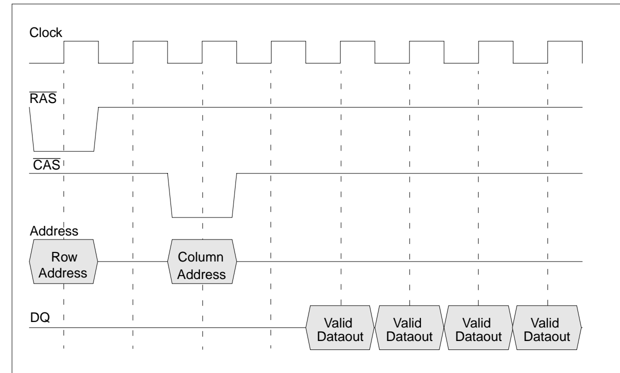

Figure 1: SDRAM Read Operation Clock Diagram. SDRAM contains a writable register for the request length, allowing high-speed column access.

Methods that increase available concurrency without affecting latency include using multiple independent banks per channel. All in all, we find that, by appropriately sizing the system parameters, one can improve execution time by 30% over the average case and by 50% over poorly performing organizations such as a 32-bit, 800 MHz memory channel, four independent DRAM banks, and a burst size of 32 bytes (which would at first glance seem to be a perfectly capable high-performance configuration).

Our simulator is based on SimpleScalar 3.0a and models a fast (simulated as 2GHz), highly aggressive out-of-order uniprocessor [15]. The interface to the memory system is non-blocking, supporting up to 32 outstanding misses at both the level-1 and level-2 caches. The memory-system bus supports pipelined and split transactions. We model 1, 2, and 4 independent 800MHz channels with data widths of 8 bits, 16 bits, 32 bits and 64 bits each, representing memory bandwidths from 800 MB/s to 25.6 GB/s. We focus on the more memory-intensive applications in the SPEC CPU 2000 integer suite. As one would expect, our results are dependent on our choices of system parameters and benchmarks studied.

2 BACKGROUND

A Random Access Memory (RAM) that uses a single transistor-capacitor pair for each binary value (bit) is referred to as a Dynamic Random Access Memory or DRAM. The operation of the traditional DRAM was defined several decades ago, and as this has become a performance bottleneck, a number of evolutionary and revolutionary changes have been made [26]. The performance of many of these DRAMs in the context of a fixed bus architecture has been studied recently [5, 7, 6], and more detail on the nature of DRAM operation can be found there and elsewhere [26, 18].

2.1 Access Timing at the DRAM Device Level

Most DRAM in use today are synchronous: they run off an external clock derived from the bus. A timing diagram for a typical Synchronous DRAM is shown in Figure 1. SDRAM devices typically have a programmable register that holds a bytes-per-request value. SDRAM may therefore return the bytes for a large request over several cycles. Note that there is only one burst width: all reads and writes use the same transaction granularity.

The timing diagrams of all modern DRAMs look similar, with the most noticeable differences being the clock frequencies and the number of cycles for each phase of the transaction. For example, the read-transaction timing diagram for Direct Rambus, which is considered a radical departure from traditional DRAMs, is shown in Figure 2. The soonest that the bank can be precharged and reacti-

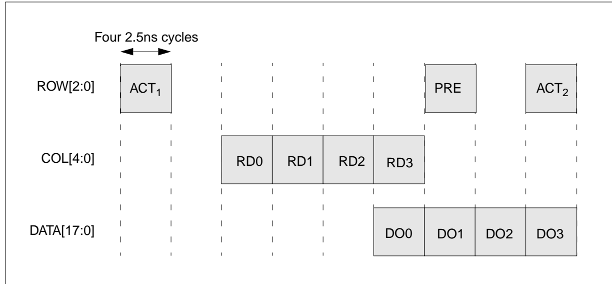

Figure 2: Direct Rambus Read Clock Diagram. Direct Rambus DRAMs transfer on both edges of a fast clock and can handle multiple simultaneous requests to different banks.

vated is also shown, though other banks can be activated sooner. ACT1 and ACT2 are row activate commands, PRE is the precharge command, RD0 to RD3 are read commands and DO0 to DO3 are the corresponding dataouts. Direct Rambus uses a 400 Mhz 3-byte channel (2 for data, 1 for addresses/commands). Direct Rambus parts transfer on both clock edges, implying a maximum bandwidth of 1.6 Gbytes/s. Because DRDRAM partitions the bus into different components, four transactions can simultaneously utilize the different portions of the DRDRAM interface (the overlap is four and not three because control information can be embedded in column address packets) [21].

2.2 Access Timing at the DRAM System Level

At the system level, the request timing is slightly different. In PC architectures and most workstation architectures, an external memory controller is responsible for sending the row and column requests directly to the DRAM, pictured in the top half of Figure 3. This is a relatively simple example, in which the output pins of the DRAM are connected directly to the data pins of the CPU (e.g., the Alpha server described by Schumann [22]). This modularity makes more of the CPU's pins available for data (as opposed to putting the memory controller on the CPU), thereby increasing the CPU's potential pin bandwidth; it allows simpler implementation of multiprocessor systems (including uniprocessor systems with external graphics processors), as it provides a single point of contact for the DRAM system; it provides upward compatibility to the system by allowing it to use future DRAMs without significant redesign (only the lowest-level memory controller needs to be redesigned) [14]; and it is essential in server configurations with large amounts of memory on large numbers of DIMMS, as the capacitance of such a system

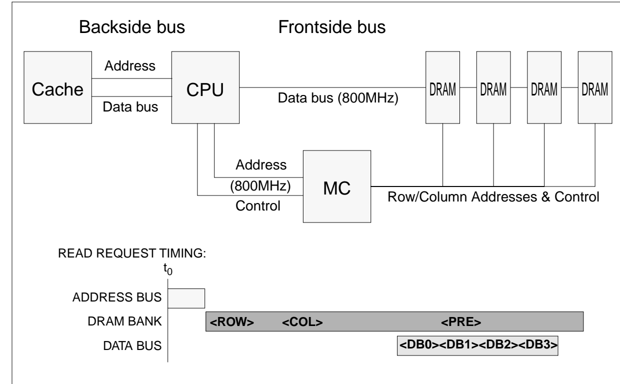

Figure 3: Typical System Organization for Uniprocessor. Figure adapted from [22]. This is the system modeled in this paper; note the lack of AGP, which would tend to overestimate system performance.

usually requires that the DIMMs be placed on separate, multiplexed, busses [22, 14].

One of the side-effects of the modular design is that the timing of operations changes slightly. The most obvious change is that the address is sent to the memory controller all at once, and it is not until the following cycle that the DRAM bank is activated with the row-address command. The bottom half of Figure 3 shows how a Direct Rambus-like access timing fits within the framework of a full DRAM system.

We model the data bus at 800 MHz with different widths. The address bus is modeled as a 1-byte 800 MHz channel that requires 10ns to transmit a 64-bit read-address packet. Figure 4 shows examples of the timings used in this study, which are similar to those reported in the literature [2, 14, 22]. The figure presents numbers for burst widths equal to 32, 64, and 128 bytes on a 32-bit 800 MHz channel. As mentioned previously, a burst is the smallest atomic transaction size—all read and write requests are processed as an integral number of bursts, and the bursts of different requests may be multiplexed in time over the same channel. We model the bus turnaround time as a constant number of bus cycles; for this study, we simulated both 1 cycle and 0 cycles.

Figure 5 illustrates how back-to-back requests can be overlapped in time on a split-transaction bus, similar to the behavior of existing busses [2]. Back-to-back reads can be pipelined, provided they require different banks. Back-to-back read/write pairs can be similarly pipelined, but it is also possible to nestle writes "inside of" reads, as shown in Figures 5(b), provided the conditions support it. This last feature is only possible because the asymmetric nature of

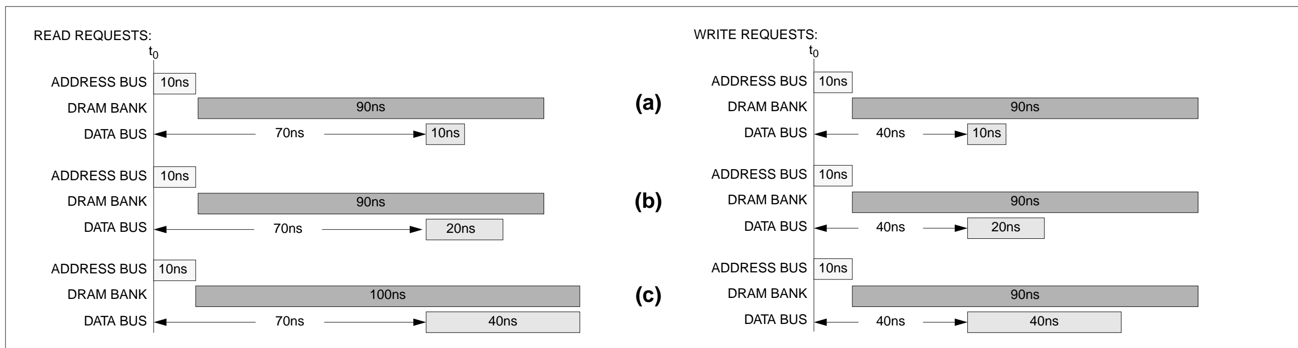

Figure 4: Bus and bank occupancies for 800MHz channel. Each DRAM request requires the address bus, the data bus, and whatever bank it is destined for. The shape of these request blocks is dependent on the burst widths. Figures are shown for burst-widths equal to (a) 8 times the bus width (e.g. 8 bytes over a 1-byte channel), (b) 16 times the bus width, and (c) 32 times the bus width. One of the interesting points is that, though reads and writes are asymmetric, they become less so as the burst width increases.

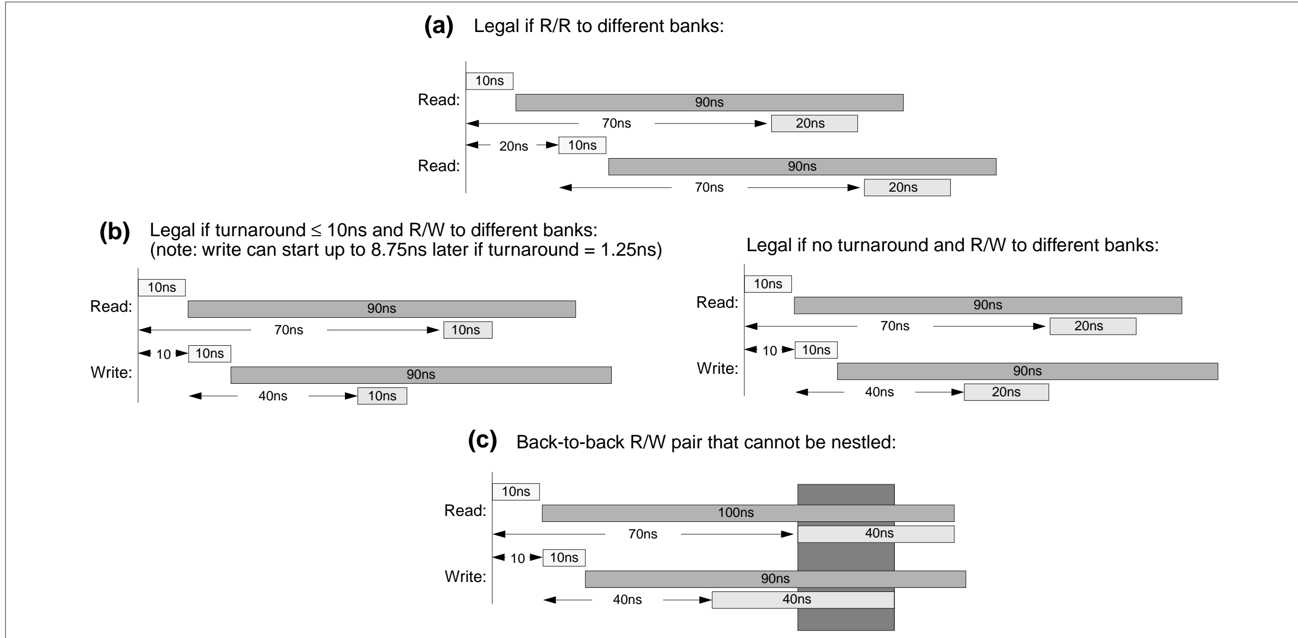

Figure 5: Concurrency within a single channel. If two concurrent reads require different banks, they can be pipelined across the address and data bus as shown in (a). Writes can be nestled inside of reads, provided the bus turnaround time is low and the burst width is small (b). However, for some burst sizes, reads and writes cannot be nestled (c).

read/write requests, and it is only possible on a split-transaction bus. Note that, though reads and writes are asymmetric, they look less so as the burst width increases and the time that the data bus is held grows large.

2.3 Burst Ordering and Coalescing, Bit Addressing, and Page Policy

Our burst ordering is critical-burst first, non-critical bursts second, and writes last. Queued bursts can be coalesced if later requests target the same memory region. The bit assignments are chosen to exploit page mode and maximize the degree of memory concurrency achieved by the application. The page policy is close-page auto-precharge [7], and we assume SRAM buffering of one DRAM page in the style of ESDRAM [10]. These are not new concepts; they are characteristics of many high-performance memory busses, and they are described in more detail in the following paragraphs.

If a burst is smaller than the level-2 cache line size, there are a number of options for the ordering of the burst-sized blocks that make up the request. In this study, the block containing the critical word is always fetched first and takes priority over any other block in the queue, unless that block also contains a critical word. Write requests are always given lowest priority and tend to stack up in the queue until all the reads drain from the queue. If a low-priority read request is still in the queue when the CPU executes a data load to the same address as its data block, that low-priority read request is dynamically promoted to a high-priority read request over all other low-priority requests in the queue. Queued requests are coalesced or cancelled if later requests are to the same address. For example, if a queued write is followed by a read to the same address, the read returns the data from the write block rather than generate another request. If a queued read is followed by another read to the same address, they are coalesced into one request.

To maximize the effectiveness of page mode, memory is divided into DRAM page-sized chunks. To maximize request concurrency, the lowest-order bits after the DRAM page offset choose the DRAM channel, the next bits choose the bank, and the highest-order bits choose the row. Address bits are assigned so that the most significant bits identify the smallest-scale component in the system, and the least significant bits, which should change most often from request to request, identify the largest-scale component in the system. Simultaneous requests to adjacent DRAM-page-sized blocks in the memory system will go out on two different DRAM channels if available, and the requests that make up a cache-block fill will go to the same DRAM page. Both exploit concurrency to the greatest degree possible because sequential addresses are striped across entire DRAM channels, and the requests that make up a cache-block fill can exploit a DRAM's page mode.

The DDR2 working group is leaning toward a close-page auto-precharge policy. This makes sense as systems go to larger amounts of DRAM; to fully exploit an open-page mode, a memory controller needs to retain information on every open page in the DRAM system, which can get expensive. Therefore, we assume a close-page auto-precharge policy in our simulations. Our address-bit assignments will direct adjacent cache-fill requests to the same DRAM page, which in a normal close-page policy would encounter an intervening precharge cycle. Nonetheless, we do not stall in this situation; the use of a DRAM's page mode is possible (and *only* possible) because we expect that the DRAM device will have ESDRAM-style page buffering [10]: one full page of SRAM storage per internal bank. The benefit of such buffering is that it allows a memory controller to implement a close-page auto-precharge policy without destroying any read locality for future requests to the same DRAM page. This does, however, incur the penalty of keeping track of all the open pages.

2.4 CPU Model

To obtain accurate timing of memory requests in a dynamically reordered instruction stream, we integrated our code into SimpleScalar 3.0a, an execution-driven simulator of an aggressive out-of-order processor [4]. Our simulated processor is eight-way superscalar; its simulated cycle time is 0.5ns (2GHz clock). Its L1 caches are split 64KB/64KB; both are 2-way set associative; both have 64-byte line-

Table 1: SPEC CPU 2000 Integer Benchmarks

Benchmark	Description
164.gzip	gzip uses Lempel-Ziv coding (LZ77) to compress input files. It is a popular data compression program. The benchmark run compresses and decompresses a large, already-compressed input file in memory at different blocking factors.
175.vpr	VPR is a FPGA placement and routing program that implements a technology-mapped circuit.
176.gcc	176.gcc is based on gcc version 2.7.2.2 that generates code for the Motorola 88100 processor. It runs with many optimization flags enabled.
181.mcf	MCF is a single-depot vehicle scheduling program for public mass transportation.
197.parser	Parser chops the input sentences into words and performs certain parsing functions.
253.perlbmk	Perlbmk is a cut-down version of Perl v5.005_03. The benchmark run generates email messages from a set of random components and converts them to HTML.
255.vortex	Vortex is an object-oriented database transaction benchmark. The benchmark builds and manipulates three separate, but interrelated, databases.
256.bzip2	Bzip2 is based on bzip2 version 0.1, which is a data-compression utility. Like gzip, the benchmark run compresses and decompresses a large, already-compressed input file in memory with different blocking factors.
300.twolf	TimberWolfSC is a placement and global routing package which uses simulated annealing as a heuristic to find very good solutions for row-based standard cell designs.

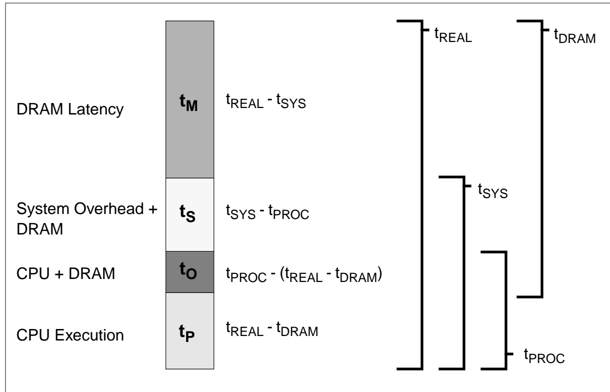

Figure 6: Definitions for execution-time breakdowns. The results of several simulations are used to show time spent in the memory system vs. time spent processing vs. the amount of memory latency hidden by the CPU.

sizes and 1-cycle access time. Its L2 cache is unified 1MB, 4-way set associative, writeback, has a 128-byte linesize and a 10-cycle access time. The L1 and L2 caches are both lockup-free, and both allow up to 32 outstanding requests at a time. For our lockup-free cache model, a load instruction that misses the L2 cache is blocked until it obtains a miss status holding register (MSHR) [15], and it holds the MSHR *only until the critical burst of data returns* (remember that the atomic unit of transfer between the CPU and DRAM system is a burst). This scheme frees up the MSHR relatively quickly, allowing subsequent load instructions that miss the L2 cache to commence as soon as possible and assumes that the cache tags can be checked for the subsequently arriving blocks without disturbing cache traffic. We model this optimization to put the highest possible pressure on the physical memory system—it represents the highest rate at which the processor can generate concurrent memory accesses given the number of available MSHRs.

2.5 Timing Calculations

Much of the DRAM access time is overlapped with instruction execution. To determine the degree of overlap, we run a second simulation with perfect primary memory (no overhead). Similar to the methodology in [5], we partition the total application execution time into three components: T_P T_M and T_O which correspond to time spent processing, time spent stalling for memory, and the portion of time spent in the memory system that is successfully overlapped with processor execution. In addition, we divide the memory-system overhead into T_S which indicates the system overhead that is irrespective of the latency of DRAM devices. In this paper, time spent “processing” includes all activity above the primary memory system, i.e. it contains all processor execution time and L1 and L2 cache activity. Let T_{REAL} be the total execution time for the realistic simulation and let T_{DRAM} be the portion of the total execution time that is spent in the DRAM system; let T_{PROC} be the execution time with a perfect DRAM device and bus organization (modeled by a 1-cycle L2 cache miss); let T_{SYS} be the execution time with a perfect DRAM device but normal bus organization. Then we have the following:

- $T_M = T_{REAL} - T_{SYS}$
- $T_S = T_{SYS} - T_{PROC}$
- $T_O = T_{PROC} + T_{DRAM} - T_{REAL}$
- $T_P = T_{REAL} - T_{DRAM}$

The relationships between the different time parameters are illustrated in Figure 6.

3 EXPERIMENTAL RESULTS

This paper focusses on the relationships between concurrency, latency, and system overheads on the memory bus. We characterize these overheads, and we investigate methods that reduce the system’s exposure to bus- and system-related performance loss. The simulations cover the following range of parameters:

Bus speed:	800 MHz
Bus width:	{1, 2, 4, 8 bytes}
Channels:	{1, 2, 4}
Banks/channel:	{1, 2, 4, 8}
Queue size:	{infinite, 0, 1, 2, 8, 16, 32 requests per channel}
Turnaround:	{0, 1 cycle}
R/W shapes:	{symmetric, asymmetric}

In addition, Table 1 lists the SPEC CPU 2000 integer benchmarks that we simulated. Note that these are all reasonable values that are found in physical high-performance systems; none of the configurations simulated could be considered straw-men or intentionally ham-

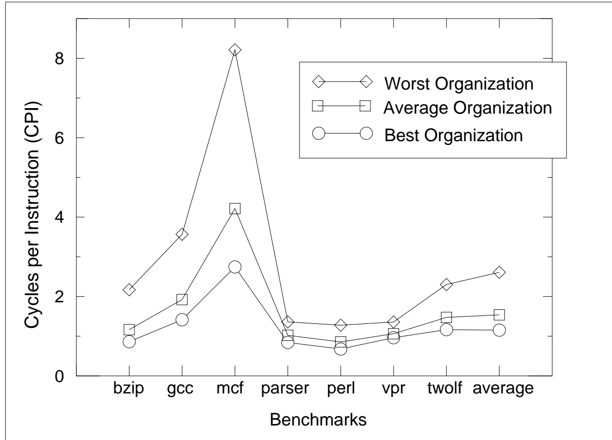

Figure 7: The range of execution times for the benchmarks studied.

strung. Therefore, any variations that we see in performance are very significant.

3.1 The Bottom Line

Figure 7 shows the range of performance one sees when varying these parameters and includes the average, best-case, and worst-case execution times with an infinite request queue. We see that there is a 3x difference between the worst-performing configurations and the best-performing configurations. As later graphs will show, the best- and worst-case configurations are not outliers—there are usually quite a few configurations at or near the top and bottom, and they tend to be the same configurations from benchmark to benchmark.

Now that we see a significant variation in performance over what seems a very reasonable set of system parameters, the question is *what is causing these differences*? Because several studies have suggested bus turnaround to be a significant source of overhead, one might think of looking at turnaround first. Figure 8 shows the effect of reducing turnaround time in a 32-entry request queue to 0 cycles, which amounts to a reduction of roughly 5% of system-related overhead. Note that these results represent a large (but finite) queue size which allows writes to be delayed; however, turnaround only accounts for 5–10% overhead in a queue-less system.

What is more significant is the number of banks attached to the memory channel. This alone yields a factor of 1.2x to 2x variation in performance within a bandwidth configuration, and it suggests that support for concurrency is much more important than turnaround time. We also see that the system overhead can be 10–40% of the primary memory latency for some configurations and some benchmarks (the primary memory latency is everything above "perfect" CPU execution and includes the top two components of each bar). This system overhead is reduced considerably by additional banks. These results suggest that there is much to be gained by adding support for concurrency. The following sections look at this in more depth. First, we will look at latency effects; then we will look at concurrency effects; lastly, we will look at overhead related to the request queue.

3.2 Fine-Tuning the Transaction Burst Length

The burst length is the minimum number of bytes transferred on any given request and so represents the minimum amount of time the data bus can be held. If two requests are interleaved, they are done so at the burst-length granularity. As mentioned previously, the burst length affects transaction latency in several ways: later instructions must stall until earlier transactions have finished using the address and data busses, which gives preference to smaller bursts, but shorter bursts expose more of the precharge overhead to later requests. Smaller burst lengths allow faster access to critical data chunks when multiple requests are serviced within a short window of time. However, if the interleaved requests go to the same bank, each will incur the precharge penalty. Larger burst lengths reduce the apparent asymmetry between read and write request shapes and amortize the cost of bus turnaround over a larger number of bytes transferred. They also reduce a cache block's stall time due to precharge delay. However, with large bursts the address and data busses take longer to become available for queued requests, which reduces the amount of overlap between program execution and memory latency. We measure the performance effects of burst lengths of 32, 64, and 128 bytes; across even this small range of burst lengths, while keeping all other parameters constant, execution time varies by 10–30%, as shown in Figure 9.

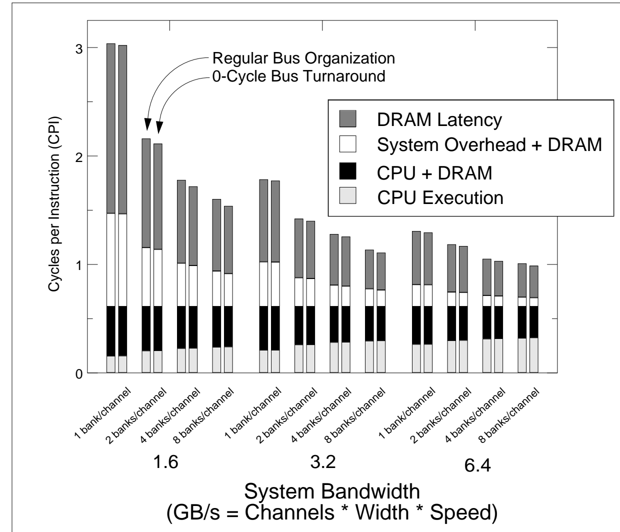

Figure 8: The effect of decreasing turnaround and increasing banks, Benchmark = BZIP, 32-byte burst, 16-bit bus, 32-entry request queue.

The figure shows a number of things. To begin with, there are some obvious trade-offs related to burst size and channel width that can affect execution time significantly within a constant system organization. Configurations with wider channels (32/64 data bits) have optimal performance with larger bursts. Those with narrow channels (8 data bits) have optimal performance with smaller bursts. Those with medium channels (16 data bits) have optimal performance with medium bursts.

If concurrency were all-important, we would find smaller bursts to be better, because this would allow us to interleave the bursts of multiple read/write requests. Instead, we see that the optimal burst width scales with the bus width. This provides us with an indication of the degree of concurrency seen in the memory system. This is shown in Figure 10. The optimal burst size is not a constant; it varies strongly with the width of the channel considered and corresponds to a fixed number of data transfers per burst, chosen so that the ratio of time spent in the RAS/CAS/data/precharge cycle to the time spent transferring data across the data bus is between 4:1 and 2:1. This means that the uniprocessor applications that we are studying are managing to sustain a degree of concurrency on each channel that is between two and four simultaneous requests. This conclusion is supported by Figure 11, which shows that increasing the number of banks per channel up to eight is not a waste of time, suggesting that these applications can exploit such degrees of concurrency. Note, however, that as the number of channels increases (the right side of

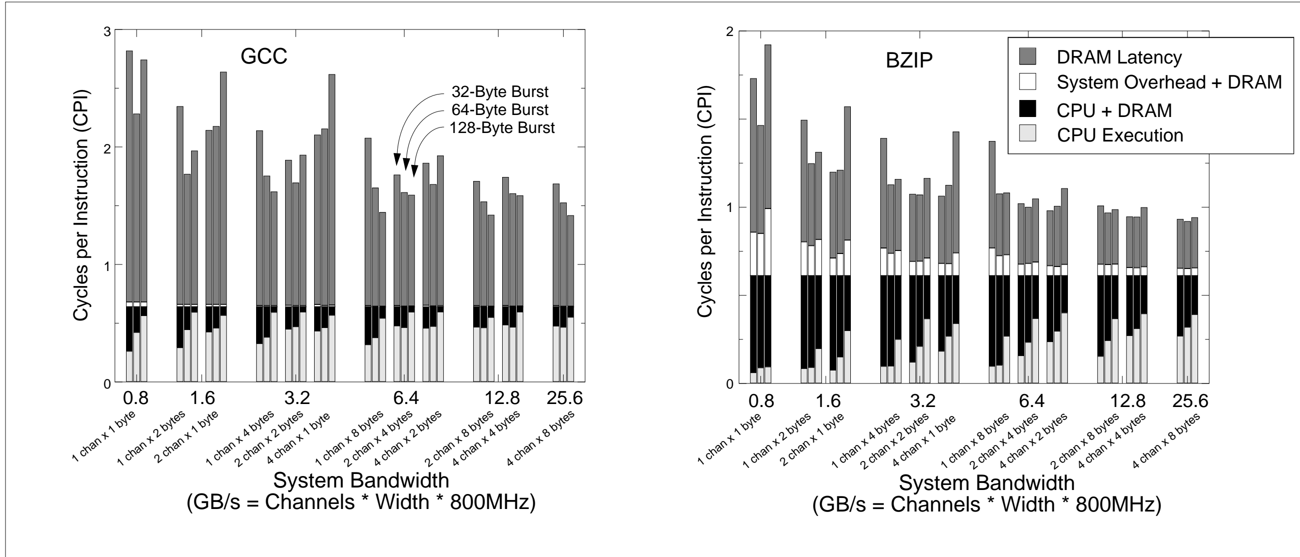

Figure 9: Burst length versus bandwidth. Configuration: 2 banks/channel

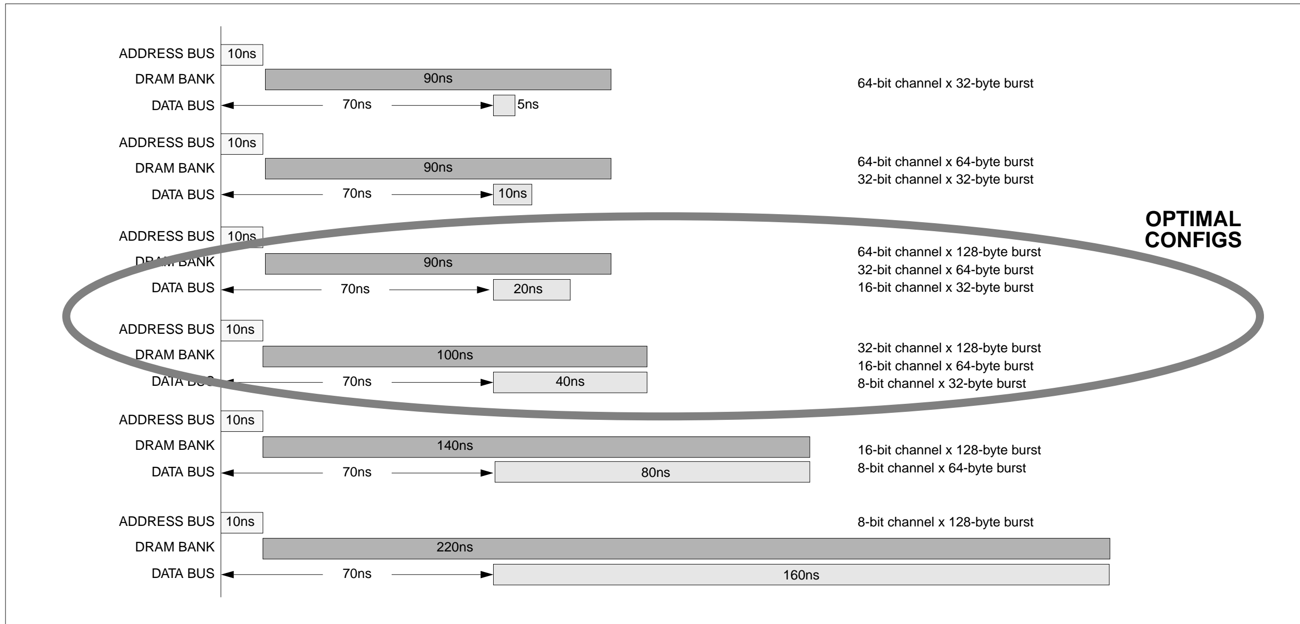

Figure 10: The range of burst widths modeled. Benchmark = BZIP, 32-byte burst, 16-bit bus.

the graphs in Figure 9), the performance variations between different burst sizes tend to decrease—as would be expected, because we evenly distribute requests over multiple channels. The conclusion: the optimal burst size reflects the degree of concurrency sustained by the application. Too many transfers per burst crowds out other requests that need to use the same data bus; too few transfers per burst allows the bank overhead (the RAS/CAS/data/precharge cycle) to dominate.

Note that the rules of thumb (wider channels want larger bursts, narrow channels want smaller bursts) do not say anything about the relative performance of different configurations—for example, which performs better: a wide channel with a large burst size or a narrow channel with a small burst size? The next section addresses this question. A related question to answer that is suggested by the current results is *to what extent should we support concurrent transactions in the memory system*? Should we do so to the detriment of transaction latency? The next section investigates this.

3.3 Increasing the Degree of Memory Concurrency

As mentioned previously, there are many ways to achieve concurrency in the DRAM system. The CPU must have a number of MSHRs to support lockup-free caches [15], which then enables concurrent outstanding requests in the primary memory system. The DRAM system must support these concurrent requests through multiple DRAM channels, or multiple independent DRAM banks per channel, or both. To get a better idea of the shape of the design space, we will now focus on one bandwidth configuration at a time. Figure 11 shows configurations modeled with bandwidths of 3.2GB/s. This is equivalent to saying “I have four 800MHz 8-bit

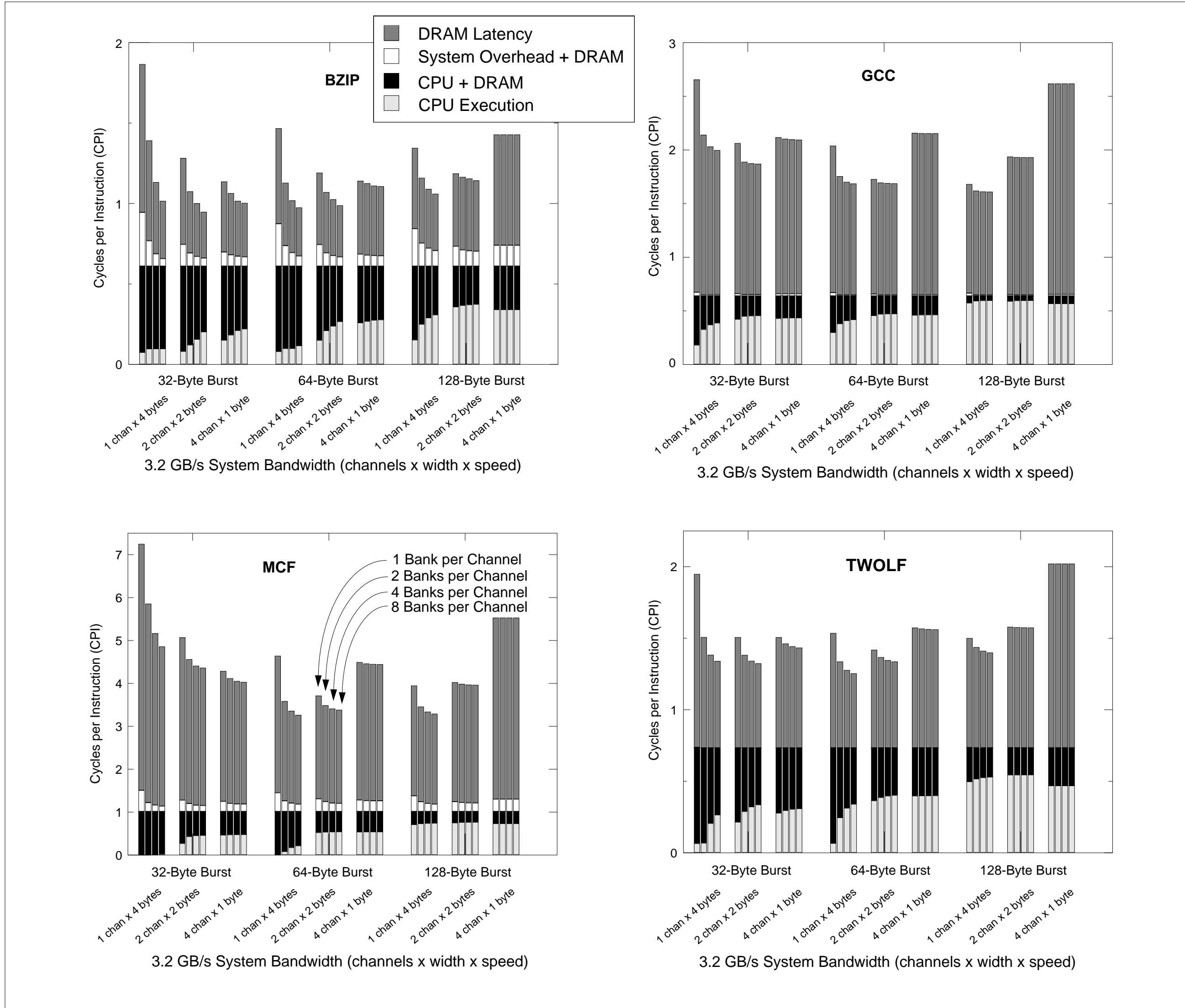

Figure 11: All configurations modeled for individual bandwidth classes.

Rambus-like channels... what should I do? Gang them together, use them as independent channels, or a combination of the two?" Details to note:

- Though increasing the number of banks typically increases performance, it does not always do so by very much. For instance, when dealing with large bursts, there is less opportunity to interleave bursts, so the number of banks does not make much of a difference. Also, when you have multi-channel systems, this gives a degree of concurrency to the system that—to an extent—obviates the need for multiple banks. For 4-way systems, the effect of multiple banks is even less pronounced than it is for large bursts. In many cases, doubling the number of independent channels is roughly equivalent to doubling the number of banks per channel.
- However, trying to exploit concurrency by splitting a memory bus into multiple narrow channels is very risky. It is heavily dependent on burst size, because using multiple channels increases the latency of every request, compared to the request's latency on a single, wide channel. Therefore, an application must generate a large number of concurrent requests to keep those multiple channels busy, otherwise they are wasted.
- The trends are similar in all the benchmarks surveyed. The only differences are where the optimal configuration lies, but there are several configurations that are always within several percent of the globally optimal configuration (e.g. 1 channel x 4 bytes x 64-byte bursts, 2 channels x 2 bytes x 64-byte bursts, 1 channel x 4 bytes x 128-byte bursts).
- Some benchmarks are better able to take advantage of multiple narrow channels. For example, bzip has a number of configurations that are all within several percent of the performance of the optimal configuration, including several with 2-way and 4-way multiple channels. Bzip tends to have much higher traffic than the other benchmarks, and we find that when writes go to the memory system, they frequently go to channels other than those being used by read requests. Bzip is also better able to exploit multiple banks than the other benchmarks for the same reason.

We find that, in a uniprocessor setting, concurrency is very important, but it is not more important than latency. SPEC CPU 2000 inte-

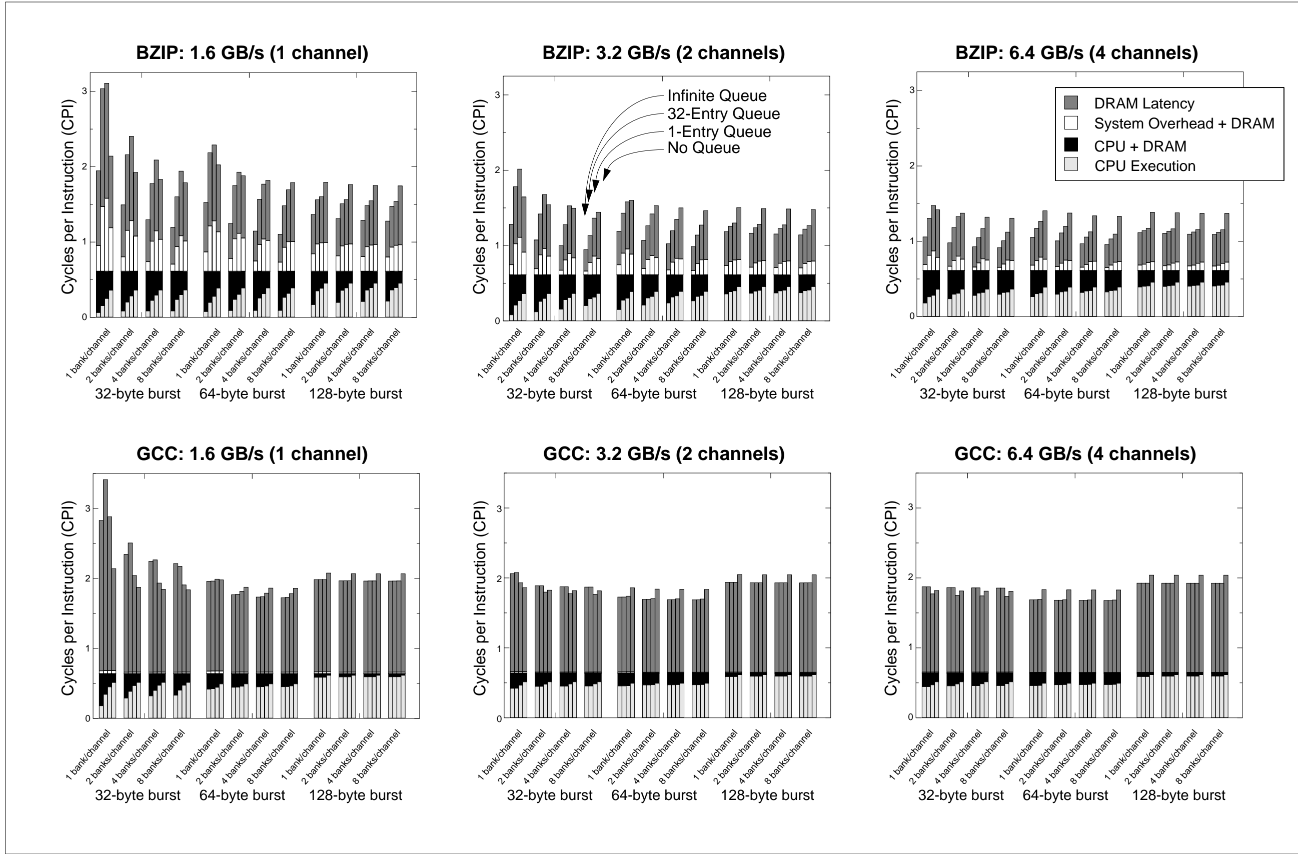

Figure 12: The effect of queue size on different benchmarks. Bus widths = 2 bytes

ger suite makes effective use of multiple independent banks per DRAM channel, suggesting that the application is producing a fair amount of concurrent transactions in the memory system. However, we find that if, in an attempt to increase support for concurrent transactions, one interleaves very small bursts or fragments the DRAM bus into multiple channels, one does so at the expense of latency, and this expense is too great for the levels of concurrency being produced (in the range of two to four). Performance can be gained by these means, but increasing performance by means of increased concurrency is best obtained through safe means such as increased banking per channel.

3.4 Increasing the Request-Queue Size

Requests can be queued at the memory-controller level when the required bus or DRAM bank is unavailable. Doing so provides three important functions: (1) the sub-blocks of different read requests can be interleaved so that the critical word of a later request can be serviced ahead of the non-critical sub-blocks of an earlier request; (2) writes can be buffered until read-burst traffic, which typically has higher priority than writes, has died down, thereby decreasing both read latency and turnaround overhead; and (3) read and write requests can be coalesced if queued long enough and there is sufficient locality in the access stream, thereby reducing the number of requests to the DRAM system. Clearly, applications with significant write activity will see more benefit from queueing. For example, Figure 12 shows results for different queue sizes for two benchmarks, bzip and gcc, that have very different write behavior: bzip has a much higher fraction of writes than gcc. In applications with high write traffic, we find that the difference in execution time between a system with an infinite queue and a system with no queue is typically 20–40%. The impact is less significant in applications with less write traffic; the difference between an infinite queue and a 0-length queue in gcc is 5–10%.

We see some interesting behavior. Though finite queues generally yield performance numbers that lie between 0-length queues and infinite queues, this is not always the case: when the number of banks is small, the bursts are short, and the number of channels is small, having a 1-entry queue or a 32-entry queue actually degrades performance over that of a system without any queueing, and in a few cases gcc displays the same behavior for an infinite queue. This is an indicator of the frequency that requests nearby in time go to the same bank. Without queueing, requests go to the memory system as soon as they arrive at the memory controller. With a small queue, writes are delayed, but the amount of time is dependent on the queue size, and the results show that when the traffic is heavy enough to expel write requests from the queue, or when traffic is light enough that the memory controller decides it is safe to dump a write request out to the DRAMs, it is very possible for those write requests to interfere with subsequent higher-priority read requests.

4 CONCLUSIONS

Careful tuning of the system-level parameters can buy a system designer 40% or more over an average system design. Ignoring the rules of the design space and building a system in an ad-hoc fashion is extremely dangerous, as it can result in a performance loss of a factor of two or more, even when looking at only a very high-performance slice of the full design space, as we do in this paper.

Bus turnaround accounts for 5–10% of the system's overhead, and it is very dependent on the queue size, as larger queues are more likely to delay writes until read traffic subsides. System overhead in general accounts for 10–40% of the primary memory latency, which echoes results from industry [2, 22]. This system overhead can be reduced in a number of ways: by increasing support for concurrency, by increasing the queue size, and by increasing the burst size.

We find that concurrency in the primary memory system is very important, even for a uniprocessor, and support for concurrent transactions improves performance by roughly a factor of two. However, that support for concurrent transactions must be bought with mechanisms that do not adversely affect individual transaction latencies; improving concurrency by subdividing the memory bus into multiple independent channels is risky, as it relies on the ability of the application to sustain a level of concurrency equal to the number of channels, otherwise the extra channels lie unused. Improving concurrency through independent banks is safe, as it supports a level of intra-channel concurrency in the form of pipelined and split transactions. Making bursts smaller to allow for interleaved transactions (which allows the critical words of later requests to go ahead of non-critical words of earlier requests) is not a good idea in general, as it also limits concurrency by exposing the bank activation-precharge cycle as a limiting factor. Large bursts amortize this cost over a larger amount of data transferred across the bus.

These results are specific to a uniprocessor environment; with multiple processors generating requests, there is likely to be a much higher sustained level of concurrency in the memory system. This would tend to favor larger numbers of channels and smaller bursts. Our future work is therefore to model the memory bus in a multiprocessor setting.

ACKNOWLEDGMENTS

Vinodh Cuppu is supported in part by NSF grant EIA-9806645 and NSF CAREER Award CCR-9983618. Bruce Jacob is supported in part by these awards, NSF grant EIA-0000439, and by Compaq and IBM.

REFERENCES

[1] A. Brown and M. Seltzer. "Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture." In *Proc. 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle WA, June 1997, pp. 214–224.

[2] W. R. Bryg, K. K. Chan, and N. S. Fiduccia. "A high-performance, low-cost multiprocessor bus for workstations and midrange servers." *The Hewlett-Packard Journal*, vol. 47, no. 1, February 1996.

[3] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0." Tech. Rep. CS-1342, University of Wisconsin-Madison, June 1997.

[4] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and et al. "Impulse: Building a smarter memory controller." In *Proc. Fifth International Symposium on High Performance Computer Architecture (HPCA'99)*, Orlando FL, January 1999, pp. 70–79.

[5] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. "A performance comparison of contemporary DRAM architectures." In *Proc. 26th Annual International Symposium on Computer Architecture (ISCA'99)*, Atlanta GA, May 1999, pp. 222–233.

[6] B. Davis, T. Mudge, and B. Jacob. "The new DRAM interfaces: SDRAM, RDRAM and variants." In *The Third International Symposium on High Performance Computing (ISHPC2K)*, Tokyo Japan, October 2000, pp. 26–31.

[7] B. Davis, T. Mudge, B. Jacob, and V. Cuppu. "DDR2 and low latency variants." In *Proc. Memory Wall Workshop at the 26th Annual Int'l Symposium on Computer Architecture*, Vancouver, Canada, May 2000.

[8] K. Diefendorff. "Sony's emotionally charged chip: Killer floating-point 'Emotion Engine' to power PlayStation 2000." *Microprocessor Report*, vol. 13, no. 5, pp. 1–11, April 1999.

[9] B. Dipert. "DRAM redesign: not just plastic surgery." *EDN*, vol. 1998, no. 14, pp. 20, July 1998.

[10] B. Dipert. "The slammin, jammin, DRAM scramble." *EDN*, vol. 2000, no. 2, pp. 68–82, January 2000.

[11] ESDRAM. *Enhanced SDRAM 1M x 16*. Enhanced Memory Systems, Inc., http://www.edram.com/products/datasheets/16M_esdram0298a.pdf, 1998.

[12] L. Gwennap. "Alpha 21364 to ease memory bottleneck: Compaq will add Direct RDRAM to 21264 core for late 2000 shipments." *Microprocessor Report*, vol. 12, no. 14, pp. 12–15, October 1998.

[13] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. "Access order and effective bandwidth for streams on a Direct Rambus memory." In *Proc. Fifth International Symposium on High Performance Computer Architecture (HPCA'99)*, Orlando FL, January 1999, pp. 80–89.

[14] T. R. Hotchkiss, N. D. Marschke, and R. M. McColsky. "A new memory system design for commercial and technical computing products." *The Hewlett-Packard Journal*, vol. 47, no. 1, February 1996.

[15] D. Kroft. "Lockup-free instruction fetch/prefetch cache organization." In *Proc. 8th Annual International Symposium on Computer Architecture (ISCA'81)*, Minneapolis MN, May 1981.

[16] S. McKee, A. Aluwihare, B. Clark, R. Klenke, T. Landon, C. Oliver, M. Salinas, A. Szymkowiak, K. Wright, W. Wulf, and J. Aylor. "Design and evaluation of dynamic access ordering hardware." In *Proc. International Conference on Supercomputing*, Philadelphia PA, May 1996.

[17] S. A. McKee and W. A. Wulf. "Access ordering and memory-conscious cache utilization." In *Proc. International Symposium on High Performance Computer Architecture (HPCA'95)*, Raleigh NC, January 1995, pp. 253–262.

[18] B. Prince. *High Performance Memories*. John Wiley and Sons, West Sussex, England, 1999.

[19] S. Przybylski. "MoSys reveals MDRAM architecture." *Microprocessor Report*, vol. 9, no. 17, pp. 17–20, December 1995.

[20] S. Przybylski. *New DRAM Technologies: A Comprehensive Analysis of the New Architectures*. MicroDesign Resources, Sebastopol CA, 1996.

[21] Rambus. *Direct RDRAM 256/288-Mbit Data Sheet*. Rambus, http://www.rambus.com/developer/downloads/rdram.256s.0060-1.1.book.pdf, 2000.

[22] R. C. Schumann. "Design of the 21174 memory controller for DIGITAL personal workstations." *Digital Technical Journal*, vol. 9, no. 2, pp. 57–70, 1997.

[23] H. S. Stone. *Microcomputer Interfacing*. Addison-Wesley Publishing Co., Reading MA, 1982.

[24] M. Swanson, L. Stoller, and J. Carter. "Increasing TLB reach using superpages backed by shadow memory." In *Proc. 25th Annual International Symposium on Computer Architecture (ISCA'98)*, Barcelona, Spain, June 1998, pp. 204–213.

[25] R. Wilson. "MoSys tries synthetic SRAM." EE Times Online, July 15, 1997, July 1997. http://www.eetimes.com/news/98/1017news/tries.html.