Eager Writeback - a Technique for Improving Bandwidth Utilization

Hsien-Hsin Lee Gary Tyson
ACAL, Department of EECS
University of Michigan
Ann Arbor, Michigan 48109
{linear,tyson}@eecs.umich.edu

Matthew Farrens
Department of Computer Science
University of California, Davis
Davis, California 95616
farrens@cs.ucdavis.edu

June 14, 1999

Abstract

Modern high-performance processors utilize multi-level cache structures to help tolerate the increasing latency (measured in processor cycles) of main memory. These caches employ either a *writeback* or a *write-through* strategy to deal with store operations. Write-through caches propagate data to more distant memory levels at the time each store occurs, which requires a very large bandwidth between the memory hierarchy levels. Writeback caches can significantly reduce the bandwidth requirements between caches and memory by marking cache lines as *dirty* when stores are processed and writing those lines to the memory system only when that dirty line is evicted. This approach works well for many applications (e.g. SPEC95), but for graphics applications that experience significant numbers of cache misses due to streaming data, writeback cache designs can degrade overall system performance by clustering bus activity when dirty lines contend with data being fetched into the cache. In this paper we present a new technique called *Eager Writeback*, which re-distributes and balances traffic by writing dirty cache lines to memory prior to their eviction. This reduces the likelihood of writing dirty lines that impede the loading of cache miss data. Eager Writeback can be viewed as a compromise between write-through and writeback policies, in which dirty lines are written later than write-through, but prior to writeback. We will show that this approach can reduce the large number of writes seen in a write-through design, while avoiding the performance degradation caused by clustering bus traffic of a writeback approach.

1 Introduction

Caches are used extensively in high-performance processors to help minimize the average memory access time and thus improve the performance of a program. Caches accomplish this by exploiting the spatial and temporal locality of references exhibited by most programs, and are very effective in reducing memory bus traffic by intercepting and handling most of the read requests generated by the processor.

However, caches must deal with both reads *and* writes to memory. Support for writes (stores) tends to be simple – on a store the data item is either written into both the cache and through the cache hierarchy to the memory (referred to as a *write-through* policy), or it is written into the cache exclusively and the data item is written out to memory only when the cache line is evicted (known as a *writeback* policy.)

Caches employing a write-through policy generate memory traffic every time a store occurs in the program. Since it would largely defeat the purpose of having a cache if the processor had to block on each store until the write completed, write-through caches use a structure known as a *store buffer* or *write buffer*[8] to buffer writes to memory. Whenever a write occurs, the data item is written into both the cache and this structure, allowing the processor to continue executing without blocking (until the store buffer becomes full). The store buffer will send its contents to memory as soon as the bus is idle.

Writeback caches, on the other hand, generate memory traffic much less frequently. When a store occurs in a writeback cache the data value is written into the corresponding line in the cache, which is then marked *dirty*. Writes to memory occur only when a line marked *dirty* is evicted from the cache (usually due to a cache miss) in order to make room for the incoming data item.

Whenever there are many consecutive misses (caused by context switches, or working set changes, or by certain graphics applications, for example) the writeback cache can find itself blocked waiting for a dirty line to be written to memory. This is the same problem faced by the write-through cache, and can be dealt with in much the same manner by adding a writeback buffer. However, there are certain classes of programs which suffer from memory delay penalties that even a large writeback buffer cannot eliminate. For example, many newer applications (e.g. D graphics ultimedia) have enormous incoming data streams. In these programs, the stream of incoming data items can cause many con cache misses and trigger the eviction of many dirty lines. This dirty writeback traffic must compete for available memory bandwidth with the arriving data, and often impedes the delivery of the data to the cache. For programs where overall performance is bound by memory bandwidth, this competition for bandwidth can have a substantial negative impact.

In this paper we propose a modification to the writeback policy which spreads out memory activity by selectively writing some dirty lines to memory whenever the bus is free, instead of waiting until that line in the cache is replaced. This early writing of dirty lines to the memory system reduces the potential impact of bursty reference streams, and can effectively re-distribute and balance the memory bandwidth and improve system performance.

Background

As discussed in the introduction, caches that employ a writeback policy reduce memory traffic by delaying the transfer of data to memory as long as possible. Most modern microprocessors using a writeback cache policy incorporate a writeback (or cast-out) buffer, which is used as temporary storage space for holding dirty cache lines while the data request that caused the eviction is serviced. Upon eviction, a dirty cache line is deposited into the writeback buffer, which usually has the highest bus scheduling priority among all types of non-read bus transactions. the writeback buffer fills up, subsequent dirty line replacements cannot take place. As a result, their corresponding data demand fetch operations cannot be committed into the cache, and the processor pipeline stalls waiting for the dependent data.

It is possible to alleviate this problem somewhat by using existing cache hardware. *onking o caches* have been proposed by roft[6] which use a set of *status ho registers* (MSH tos) manage several outstanding cache misses. When a cache miss occurs in a non-blocking cache, it is allocated an empty MSH try. en nce the MSH try is allocated, processor execution can continue. If none of the MSH are available (i.e. a structural hazard [5] exists due to resource con ice), processor will have to block until an MSH try becomes free.

y adding data fields to the MSH ould, it is possible to use them to temporarily store returning cache lines. This would allow fetched data to be immediately forwarded to the appropriate destination registers, and help overcome the situation where the cache cannot be written to because the writeback buffer is full. However, this scenario delays MSH deallocation and can lead to processor stalls on a cache miss because of no free MSH Figure 1 illustrates a non-blocking cache organization.

In addition, in a modern computer system memory bandwidth is not exclusively dedicated to the host processor. There are often multiple agents on the bus (such as graphics accelerators or multiple processors) issuing requests to memory over a short period of time. A typical system architecture of a contemporary PC system is illustrated in Figure .

In a contemporary PC platform with an Accelerated Graphics Port (AGP) interface running a graphics-centric application, for example, the graphics accelerator shares system memory bandwidth

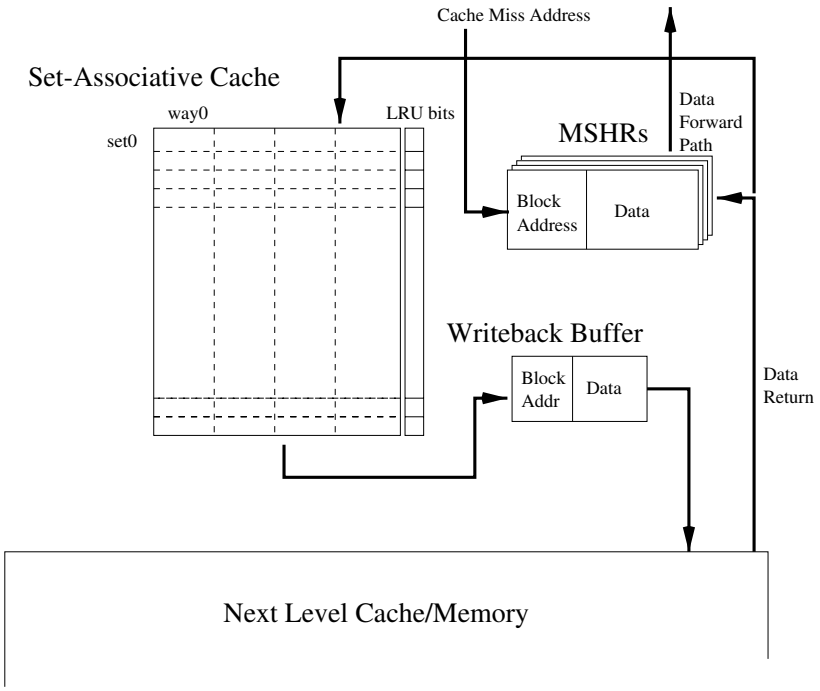

Figure 1 Architectural block diagram of non-blocking caches.

with the host processor by constantly retrieving graphics commands and texture maps from the system memory. As is shown in Figure , the same system memory serves as the instruction and data repository for both the processor and graphics accelerator.

In a common 3D graphics application, for instance, the processor reads instructions and triangle vertices, processes and then stores them with rendering state commands back into AGP memory space. The graphics accelerator then reads these commands out of AGP memory for rasterization. In addition to the command traffic, the graphics accelerator also reads a large amount of texture data (which constitutes the major portion of AGP traffic on the bus). These textures are mapped onto polygon surfaces to increase the visual realism of computer-generated images. In the future, with richer content 3D graphics applications or graphics accelerators with enhanced quality features such as bi-linear or tri-linear interpolation, AGP command and data bandwidth demands for graphics accelerators will undoubtedly be even greater than they are now.

Current cache designs have difficulty in efficiently managing the flow of data in and out of the cache hierarchy in these data intensive applications. Buffering techniques, including write buffers and MSHRs can help, but do not alleviate the problems of clustering bus traffic caused by writeback data. In the next section we introduce a new technique designed to distribute the writes of dirty blocks to times when the bus is idle.

Eager Writeback

3.1 Overview

To address the performance drawbacks of a conventional writeback policy, we are proposing a new technique called *Eager Writeback*. The fundamental idea behind Eager Writeback is to write dirty cache lines to the next level of the memory hierarchy and clear their dirty bits earlier than in a conventional writeback cache design, in order to better distribute bandwidth utilization and alleviate memory bus congestion. If dirty cache lines are written to memory when the bus is less congested,

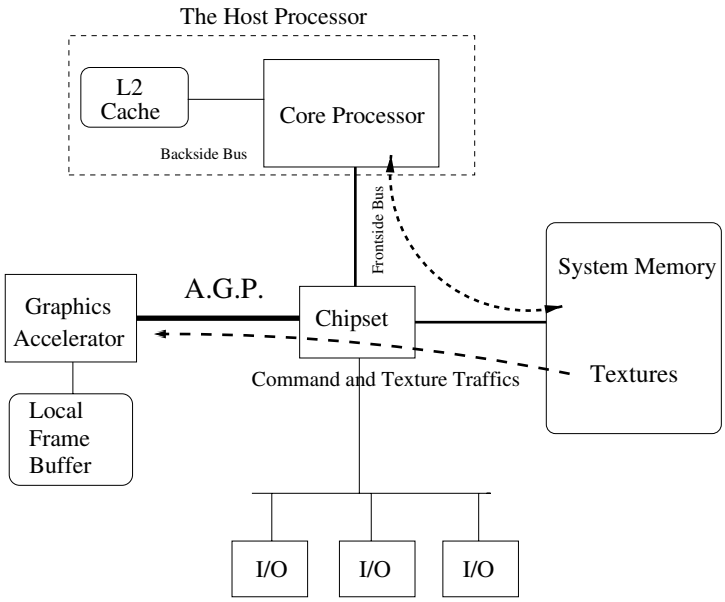

Figure A PC system architecture with AGP.

then there will be fewer dirty lines that require eviction during peak memory activity.

In essence, we are speculating that certain dirty lines will not be re-written before eviction and thus there is no need to wait until eviction time to perform the cache line write. An Eager Writeback will never impact the correctness of the architectural state even if the operation that triggers it was wrongly speculated - if our speculation is incorrect and we write too often, we approach the limiting case of write-through cache behavior. If we do not speculate often enough, we approach writeback cache behavior. However, in either case we do not violate any correctness constraints. In the worst case incorrect speculation may lead to excessive memory traffic, by consistently writing and cleaning lines in the cache that are then quickly marked dirty again.

In order to select the best trigger because an eager writeback, we examined the probability of rewriting a dirty line in a set-associative cache when it was in a given state (M U through L U) for the well-known SPEC95 benchmarks[4] and four applications from the lesser-known benchmark suite[The benchmark suite consists of four applications representing different graphics algorithms. D M, a popular video game, uses a polygon-based rendering algorithm. is a public domain ray tracing package developed for generating photo-realistic images on a computer. The third application is an animation viewer which processes an MPEG-1 data stream to display an animated sequence. The final application, xlock, renders a D polygonal object on the screen.

Our results indicate that lines that have been marked dirty and reach the L U (Least Recently Used) state in a 4-way set-associative cache are rarely written to again before they are evicted. In Figure and Figure 4, we show the probability of a line that was marked dirty being written to again as it moves from the M U (Most Recently Used) state to the L U state for both L1 and L caches. The graph on the left in Figure for example, shows that in the L1 cache the average probability of a dirty line in the L U state being re-written is 0.15, while the similar probability for a dirty line in the M U state is 0.95. The probabilities of re-dirtying lines in the L U state are even lower in the L cache - in fact, close to 0 as shown in the graphs on right of Figure and Figure 4.

These figures indicate there are some programs (such as and *su or*) that have a fairly high probability of writing to dirty lines after they have entered the L U state. In order to further evaluate this, we looked at the ratio of the number of times a dirty line in the L U state is written to normalized to the number of times a dirty line in the M U state is written to. The results are

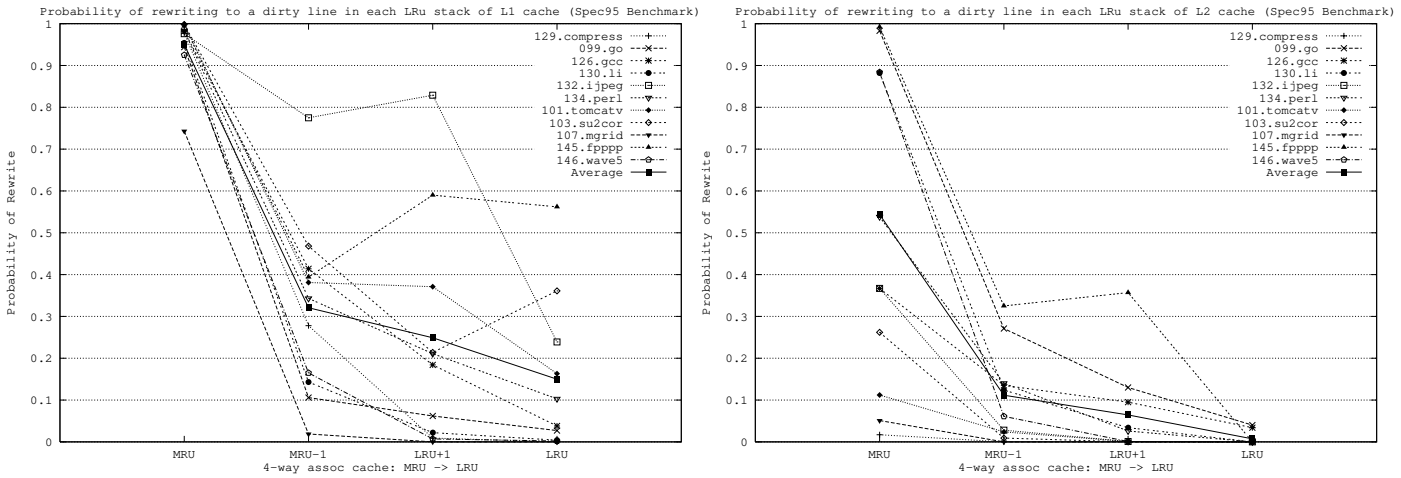

Figure Probability of writing to a dirty line in each L U stack of L1 and L hesa(SPEC95)

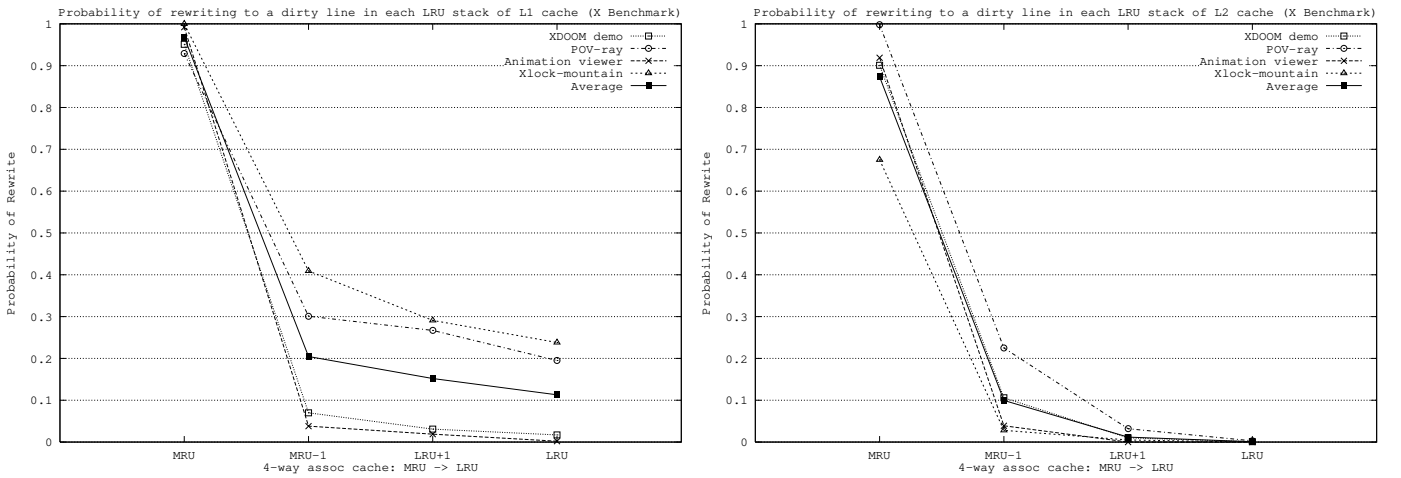

Figure 4 Probability of writing to a dirty line in each L U stack of L1 and L hesa(bmar

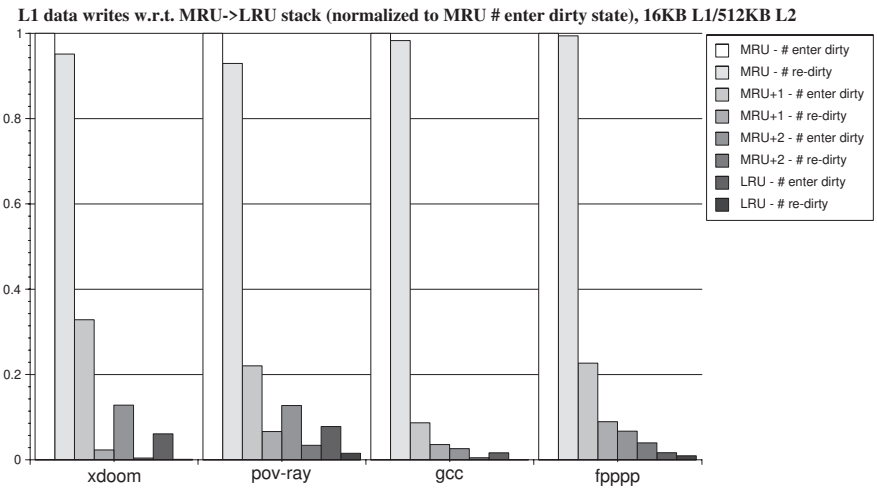

Figure 5 ormalizedber of writes and rewrites to a dirty line in each M U-L U stack

presented in Figure 5, which shows that while the probabilities may be high, the actual number of these occurrences is negligible compared to the rewriting that occurs when a line is in other states (M U, M U-1, etc.). These trends held across a wide range of cache configurations, and imply that once a line enters the L U state it becomes a prime candidate for Eager Writeback, since it has a very low occurrence of being re-written (and thus marked dirty again).

3. e i i ri e er e

There can be many different approaches to deciding when to trigger an Eager Writeback. As was shown in the previous section, one obvious candidate is to use the transition of a dirty line into the L U state as a trigger point for an Eager Writeback. For example, when a cache set is being accessed and its corresponding L U bit is being updated, the line can be checked to see if it is marked dirty. If it is, then a dirty writeback can be scheduled, and the dirty bit can be reset.

If the writeback buffer is full at this point, two approaches can be considered (a) simply abort the Eager Writeback the actual dirty writeback will take place later when the line is evicted, or (b) perform the eager writeback when an entry in the writeback buffer becomes free. This provides the ability to perform eager writeback anytime between when a line is marked L U and when it is evicted.

To provide this capability using a minimum of hardware, we chose to simulate an *Eager* which holds attempted eager writebacks which were unable to acquire writeback buffer entries. Whenever an entry in the writeback buffer becomes available, the Eager checks the cache set on the top of the queue to see if the dirty bit of the L U line in the indexed set is set. If it is, the line is moved into the writeback buffer.

An alternate implementation considered during this research was *Eager Writeback.* This implementation used a small independent state machine which autonomously polled each cache set in round-robin fashion and checked the dirty bit of its L U line, initiating eager writeback on those lines when the writeback buffer was not full. Whether eager queues or the autonomous state machine is more feasible is highly dependent on the processor and cache organization. For this study we present results for the more conservative approach which used eager queues.

ulation ramework

simulation environment was based on the SimpleScalar tool set [1], a re-targetable execution-driven simulator which models speculative and out-of-order execution. The machine employs a egister Update Unit (UU), which combines the functions of the reservation stations and the re-order buffer necessary for supporting out-of-order execution [9]. Functional unit binding, instruction dispatch and retirement all occur in the UU.

The microarchitectural parameters used in our baseline processor model are shown in Table 1. Table lists the latencies of each functional unit modeled in the simulation. A non-blocking cache structure, writeback buffer and eager queue associated with each cache level were added to the simulator for this study. The number of entries in each buffer was re-configurable from 1 to 56, and varied from simulation to simulation.

A pseudo- bus D model was used in the external memory system. This single-channel D with AM dependent banks can address up to G system memory. In the model, independent banks can be accessed simultaneously (contiguous banks share the same sense amplifier for driving data out of the AM cells) w control packets, column control packets and data packets can be pipelined and use separate busses. D AM address re-mapping [modeled] to reduce the rate of bank interference. The peak bandwidth that can be reached in our D AM model is 1.6G

A simplified uncacheable write-combinable memory [as] was implemented for the purpose of correctly simulating our benchmark behavior. Whenever a data write to an uncacheable region results in an L1

Processor Architectural Parameters	Specifications
Core frequency	1 GHz
1st Level I-Cache	2-way, 256 sets, 32B line, virtually-indexed physically-tagged
1st Level D-Cache	4-way, 128 sets, 32B line, virtually-indexed physically-tagged
2nd Level Cache	Unified, 4-way, 4096 sets, 32B line, physically-indexed physically-tagged
I- and D-TLBs	2-way, 128 sets, 32B line size, virtually-indexed virtually-tagged
Backside bus	500 MHz (i.e. half-speed L2), 64bit wide
Frontside bus	200 MHz, 64bit wide
Memory model	Rambus DRAM model (1.6GBytes/sec peak bandwidth)
Branch predictor	2-level adaptive predictor, 10-bit history, *gshare* scheme
Instr. fetch/decode/issue/commit width	8 / 8 / 8 / 8
Load/Store Queue size	32
Register update unit size	64
Memory port size	2
INT/FP ALU size	4 / 4
INT/FP MULT/DIV size	1 / 1

Table 1 Summary of the aseline Processor Model.

Processor Architectural Parameters	Cycles in Processor Clock
1st Level I- and D-Cache	3 cycles, pipelineability = every 1 cycle
2nd Level Cache	18 cycles, pipelineability = every 10 cycles
I- and D-TLBs	2 cycles, pipelineability = every 1 cycle
Backside bus arbitration	4 cycles
Frontside bus arbitration	10 cycles
RDRAM Trcd, RAS-to-CAS	20 cycles
RDRAM Tcac, CAS-to-data return	20 cycles
RDRAM Trp, Row Precharge	20 cycles
INT ALU latency/thruput	1 / 1
INT multiplier latency/thruput	3 / 1
INT divider latency/thruput	20 / 19
FP ALU latency/thruput	2 / 1
FP multiplier latency/thruput	4 / 1
FP divider latency/thruput	12 / 12

Table Latency Table (in core cycles) of Functional Units in the aseline Processor.

cache miss, the write operation will immediately request access to the bus and drive data out to the system memory directly (skipping a next-level cache look-up). ally he line writes are modeled - any partial cache line update will be treated as a full cache line write in the simulator.

For modeling multiple agents on the memory bus, a memory traffic in ector is also implemented. This in ector wa us to imitate the extra bandwidth consumed by other bus agents by configurable periodic in ection of data streams to the memory bus.

.1 e r

In order to evaluate the effectiveness of the Eager Writeback technique, we ran extensive simulations on the SPEC benchmark suite and two kernels representative of graphics applications. y conducting the analysis on small, representative kernels, we can illustrate the potential benefits of our scheme in far greater detail than can be achieved running an entire application. The first of these kernels is a small ge D metry processing kernel (*ini-ge*), which is present in most triangle-based rasterization algorithms. Two different graphics rendering configurations were simulated, one which was very simple (i.e. ambient light with no external light sources), and one which included multiple diffuse light sources. The ambient light configuration reduces the computational requirements of the

```
MINI-GEOMETRY()
  while ( frames )
        for ( ob          ects in frame )
            for ( every 4 vertices )
                /     ransformation          /
                tx = m11 * InV x    m21 * InV y    m31 * InV z    m41
                ty = m12 * InV x    m22 * InV y    m32 * InV z    m42
                tz = m13 * InV x    m23 * InV y    m33 * InV z    m43
                w = m14 * InV x    m24 * InV y    m34 * InV z    m44
                OutV rw = 1/w
                OutV tx = X offset    tx * OutV rw
                OutV ty = Y offset    ty * OutV rw
                OutV tz = tz * OutV rw
                /     e     T   ture coordinates loop          /
                OutV tu = InV u
                OutV tv = InV v
                /          tight loop          /
                ID r = ID g = ID b = 0.0
                for ( every light source )
                    dot = LDir x * InV nx    LDir y * InV ny    LDir z * InV nz
                    ID r = ID r    Ambient r    Diffuse r * dot
                    ID g = ID g    Ambient g    Diffuse g * dot
                    ID b = ID b    Ambient b    Diffuse b * dot
                    OutV cd = ((int)ID r << 24)|((int)ID g << 16)|((int)ID b << 8|α)

            /          Device drive loop          /
            for ( each transformed and lit vertex )
                /          Assumes Tip triangles          /
                /        y Copie          utV records to graphics A   P memory          /
                GfxCommand vertex − 2     OutV=vertex − 2
                if (even − numberedvertex)
                  GfxCommand vertex     OutV=vertex
                  GfxCommand vertex − 1     OutV=vertex − 1
                else
                  GfxCommand vertex − 1     OutV=vertex − 1
                  GfxCommand vertex     OutV=vertex
```

Figure 6 Algorithm of the mini-geometry kernel

algorithm in order to maximize frame rate at the expense of picture realism [1]. The multiple light source configuration increases the computational demands, thereby reducing the relative impact of bus utilization as the processor spends more time processing between each data element request.

The second kernel represents a very general streaming data algorithm which processes large data sets. This kernel continually walks though the cache with a new set of data, experiencing frequent cache misses as well as dirty writebacks.

r **tr**

The D geometry processing kernel, shown in Figure 6, is representative of a very frequently used algorithm in most triangle-based rendering engines. Geometry processing, consisting of intensive oating-point operations on a large quantity of data from memory, is mainly performed by the processor. It is one of the two key portions of a three-dimensional graphics rendering pipeline (the other portion being rasterization, which is typically performed by a dedicated graphics accelerator).

This kernel consists of three nested loops wrapped by two outer loops which iterate through frames and obD ects in the world space. The first innermost loop processes vertices for each obD ect assuming the entire ob ect is modeled by a single triangle strip. The basic functionality performed inside this loop are *trans or* , *ation and ighting co out and ut*

[1] This would be preferred in the DOOM application when processor performance is lacking.

```
-      ()      E
    array        array,        AX
for (m = 0  m <      m  oo          )
    for ( array   i    each set of L2 cache )
        write array   i      ay 0 to w
        write array   i     * 8 * 1 set_size      ay 1 to w
        write array   i     * 8 * 2 set_size      ay 2 to w
        write array   i     * 8 * 3 set_size      ay 3 to w
    for ( array        each cache line in L2 cache )
        read array
        compute array
        write array   m
    for ( array        each set of L2 cache )
        read array        to way in0
        read array        * 8 * 1 set_size    to way in1
        read array        * 8 * 2 set_size    to way in2
        read array        * 8 * 3 set_size    to way in3
        write array   a m
```

Figure Algorithm of the Cache Walk ernel

The *trans* *or* function *tion* pro the new location of each vertex on screen through a 4x4 matrix multiplication and a viewport transformation. The *ighting* function calculates the interaction of each vertex with light sources and generates the color intensity for each vertex. This calculation involves a dot product between the light direction vector and the vertex normal vector using a Phong illumination model [10]. A single parallel light source with diffuse only components is assumed in the lighting model. For a parallel light source, per-vertex normal transformations can be replaced by an inverse transformation of the light source location on a per-scene basis, thus eliminating a large number of computations. A color packing conversion then packs four single-precision oating-poin G A color is ensities into a packed 4-byte integer. (We assume the machine ISA of interest supports four wide SIMD computation).

After finishing with all the vertices in one ob ect, a loop imitating the functional y of a device driver is invoked (the *co* *out and* *out* function). This driver loop breaks one triangle strip into individual triangles and copies these transformed and lit vertices to the uncacheable graphics memory.

ac a r

The Cache-walk kernel is presented in Figure , and consists of three inner loops that exercise the L cache. The first loop walks through the L he cache dirtying lines. The second loop reads each cache line in $array_A[]$, performs some oating-poin computation and passes the results to inner loop invariant array elements. Finally the third loop reads another *array* [] that evicts $array_A[]$ from the caches.

ulation e and nal i

The simulation results are presented and analyzed in this section. For each kernel studied, we present two different data sets, one with no memory contention from other potential bus agents, and one with artificially in ected memory traffic.

.1 e e r

Table ws the simulation results for the SPEC95 benchmark suite using configurations - aseline, Eager and Free Writeback. The aseline case uses a single- ey writeback buffer, while Free Writeback

s	*ase*		*agere*	*e*	*ree*	*ar*	*e*
benchmark	cycles	cycles	speedup		cycles	speedup	
go	4106	4189 8316586	1.000		4105050891	1.000	
gcc	1425690611	14235	8223 1.001		1419981686	1.004	
li	401639628	401635232	1.000		401481	52 1.000	
i peg	212552148	21233220	0 1.001		211	908634 1.004	
perl	3 055	3 946 5106 5936	1.001		3683430936	1.006	
tomcatv	5436594306	54366	0500 1.000		5436456381	1.000	
su2cor	462520	5425 4 248569	1.000		462511	24 1.000	
mgrid	213883252	2132120132	1.003		2061823555	1.03	
fpppp	8404	0 5 8 4 1 2 0	60399 0.999		840404	239 1.000	
wave5	2221	4 2208 518	02430 1.006		21	92253 1.020	2

Table Performance of SPEC95 hmarks. (W buffer 1, E 4)

	ase	*ager*		*e*	*ager*		*ager*		*ree*	*ar*	
write bu er size	cycles	cycles	speedup		cycles	speedup	cycles	speedup	cycles	speedup	
to, ligh B Buf=1	2536463	238	6911 1.062		21838002	1.162	2183	952 1.162	21	98206 1.164	
to, ligh B Buf=4	25320139	2182062	1.160		21820566	1.160	21820566	1.160	21	98206 1.162	
to, ligh B Buf=256	25320139	21820566	1.160		21820566	1.160	21820566	1.160	21	98206 1.162	
3 di ts, ligh B Buf=1	30643341	29200004	1.049		2 1	1.128 6616	2 1	1.128 6833	2	13414 1.129	
3 di ts, ligh B Buf=4	30643153	2	158044 1.128		2	158049 1.128	2	158044 1.128	2	13414 1.129	
3 di ts, ligh B Buf=256	30643153	2	158044 1.128		2	158044 1.128	2	158044 1.128	2	13414 1.129	

Table 4 Simulated cycles of Mini-Geometry ernel.

models a system in which dirty writebacks do not generate any memory traffic on the bus (thus it serves as an upper bound on performance.)

Looking at the table it is apparent that there is little performance gain possible for the programs in this suite, since the difference in the cycle count between the baseline case and the upper bound is negligible. This is not surprising, since it is well-known that the SPEC95 benchmark suite does not exercise the memory system aggressively. The SPEC95 suite is not a good candidate for memory system performance studies primarily due to its small working set size. For the rest of this study we will focus on the benchmarks that more aggressively exercise the memory system, and are arguably more representative of future workloads.

. i i i er e e r

t tract c r

Table 4 contains the number of mini-geometry kernel execution cycles for a variety of memory configurations. In this table, each row represents a different combination of writeback uffer size and lighting conditions, while the columns contain different writeback strategy cycle counts. The first column, *ase*, contains the cycle count using a conventional writeback policy. The next 6 columns contain the results for different variations of the *Eager Writeback* scheme and the speed-up of each scheme over the baseline case, with each scheme identified by the size of its Eager ueue (E). The simplest design choice is E 0, in which Eager Writebacks are dismissed if the writeback buffer is full. The other two cases can queue up attempted eager writebacks within Eager ueues of specified sizes. The rightmost column contains the Free Writeback case, which as stated earlier is the upper bound to available performance.

There are several things of interest to note in this table. Perhaps most significantly, it can be seen that increasing the depth of the writeback buffer has virtually no impact on the performance of the

aseline case. In fact, going from 1 to 56 entries in the writeback buffer only improves performance by 0.1 This is because a large number of dirty writebacks are competing for bandwidth with the demand fetches, and the bus congestion can not be alleviated by a deeper writeback buffer.

On the other hand, adding Eager Writeback increases the performance of the system by 4.9 to 16. (depending on the light sources and the depth of the Eager Queue). For the simplest case of no Eager Queue and a single entry writeback buffer, the speedup ranges from 6. (for no light source) to 4.9 (with light sources). This speedup is smaller than for the other cases, because many eager writebacks are dropped due to the lack of space in the writeback buffer. When the number of writeback buffer entries is increased (or the Eager Queue size is increased), the speedup achieved approaches the upper bound.

The bandwidth shifting effect is quite apparent in Figure 8 and Figure 9. These two figures present the utilization profile of memory bandwidth requested by the processor using the aseline (Figure 8) and Eager Writeback (Figure 9) configurations, running the mini-geometry kernel. The y-axis plots the instantaneous bandwidth[2] versus the execution timeline on the x-axis.

The 1 broadest that saturate the peak DRAM bandwidth in Figure 9 occur within the driver loop, where command output is being written into the write-combining graphics memory while eager writebacks of dirty lines are concurrently taking place. Since within the driver loop there is still some computation occurring, the bandwidth is not fully utilized, and eager writeback writes can use the available idle slots and maximize bandwidth. Conversely, in the baseline case, the same writebacks occur within the geometry computation loop. This means these requests compete for the bus with the return of the data requested by vertex loads, and thus slow down the processing. This maximization of the utilization of the bandwidth during the driver loop leads to a lower and sparser average memory bandwidth in Eager Writeback than in the aseline case outside the driver loop.[3]

The overall performance improvement is obviously gained from the shifting of dirty writeback traffic to where this traffic does not impede the return of any critical data. This can be seen in Figure 10, which presents an execution profile of the benchmark. In this figure the sequence of vertex data load requests appears on the y-axis, and the cycle upon which the corresponding data item returns is plotted on the x-axis. As execution begins, the profiles of aseline and Eager Writeback are completely overlapped, because data is returning at the same time for both schemes. Beginning at around .6 million cycles, these two curves start to deviate from one other, and continue to diverge as execution time increases. This implies that the speed-up due to Eager Writeback will continue to climb as the loop frame continues to execute, and would be greater than the measured 16.1 if larger simulations are run.

By looking carefully at this figure it is possible to distinguish the geometry computation loop from the device driver loop. The segments with shorter but steeper slopes are where the driver loop is executing. The steepness of the slope occurs because the requested data, [], was returned faster (since the loop read the output vertices generated in the transformation and lighting stages from the L he cac directly, rather than from memory).

Table 5 shows how Eager Writeback affects the performance bottleneck in the Register Update Unit (RUU) of the processor. The layout of this table is similar to Table 4, and contains the number of cycles the processor is stalled due to the RUU being full.

As the table shows, Eager Writeback is able to remove a substantial number of stall cycles due to a full RUU and keep the execution pipeline running smoother. These stalls are reduced because in conventional writeback schemes dirty writebacks are competing with demand fetches for available bandwidth, causing delays in data arrival and the filling of the reservation stations in the RUU. The

---

This was calculated by sampling the data phase on the memory bus every ks if e.g. cores are seen by on the bus in contemporaneous bandwidth is its instant M sec for a t should be emphasized that the total bandwidth required by a system using writeback is not reduced rather it is redistributed by the early eviction of dirty

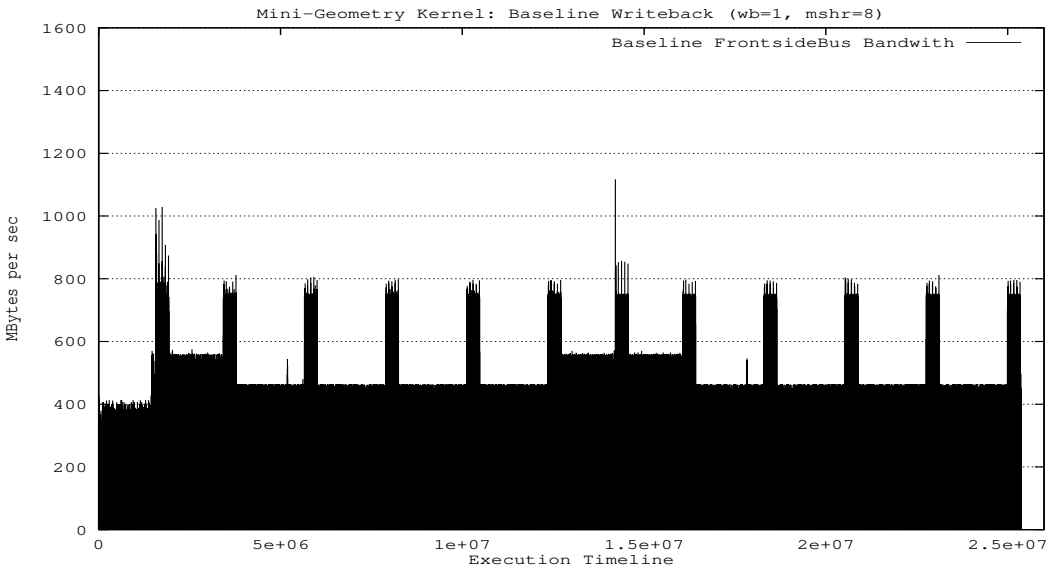

Figure 8 Memory andwidth Profile b *aseWriteback* for Mini-Geometry ernel (t) o ligh

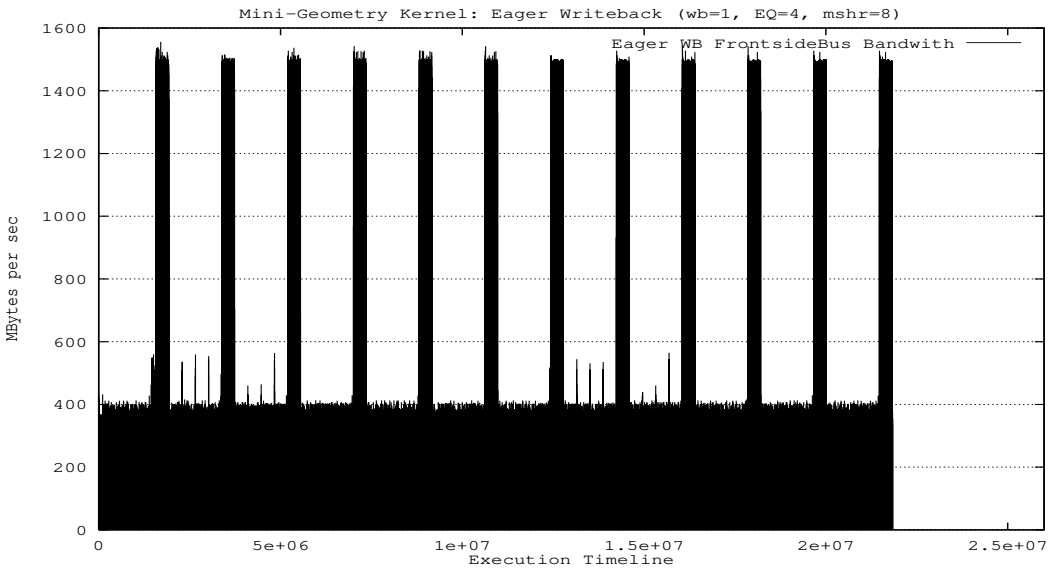

Figure 9 Memory andwidth Profile b *Eager Writeback* for Mini-Geometry ernel (t) o ligh

	ase	*ager e*		*ager*		*ager*		*ree*	*ar*	
RUU ull cycles	cycles	cycles	improved	cycles	improved	cycles	improved	cycles	improved	
to ligh B Buf = 1	8404023	66	8659 20.5	4452469	4	40152265	4	40109553	4	.5
to ligh B Buf = 4	83	5643939	9 4	.0139226	4	.0139226	4	.0109553	4	.35
3 di use ligh B Buf = 8	1045	6541028	18.	4361651	45.8	4361259	45.8	4313	1046.4	
3 di use ligh B Buf = 8	033850	4344	945.92	43446	45.92	4344653	45.92	4313	1046.31	

Table 5 esource Hazard Improvement of Mini-Geometry ernel.

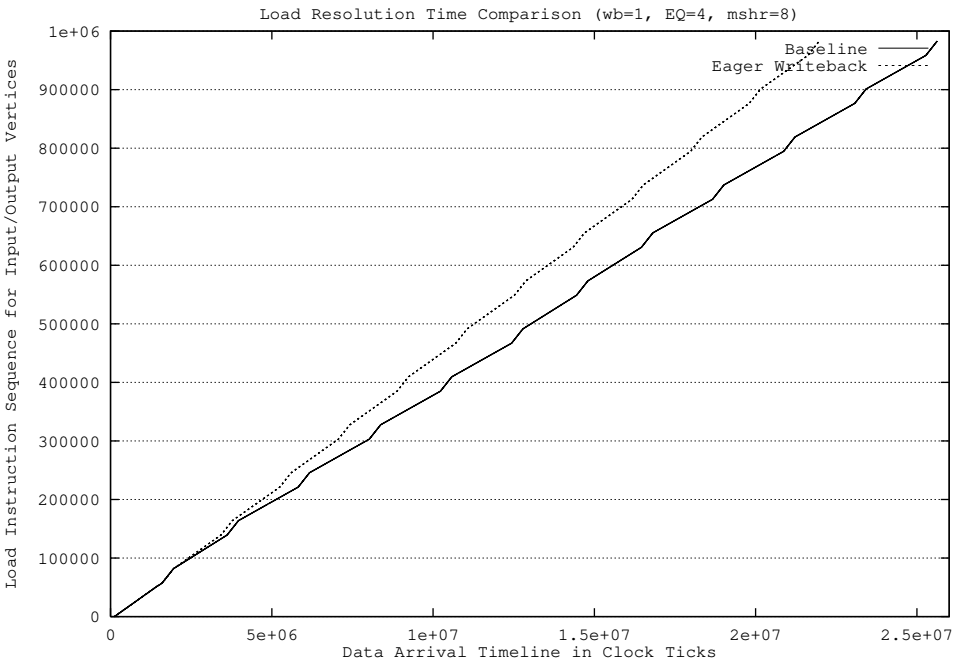

Figure 10 Load esponse Time for Input ertex in Mini-Geometry ernel

a e	s h Baseline		aspeed-up	es Baseline	gh	aimproved
0 GB/sec	2536463	21838002	1.16	8404023	4452469	4 .0
0.4GB/sec (160B/400clks)	2 323	25434535	1 1.0	1052981	8448695	19. 6
0.8GB/sec (320B/400clks)	3356	33 530	0.58 5	16 601	998 024045	-1.6
1.2GB/sec (480B/400clks)	606995	591 362	1.03 3	44206642	42864369	3.0
0.4GB/sec (1280B/3200clks)	32539684	286360	2 1.14	15604083	113646	9 2 .2
0.8GB/sec (2560B/3200clks)	4 365	9 2 5 59653	1.11	30356564	25269290	16.8
1.2GB/sec (3840B/3200clks)	8 400	8 3 4 26435	1.05	0248 2 2 0 15191		6.0

Table 6 Memory Traffic In ection to Mini-geometry (Eager ernel. ueue 4)

eager writebacks shift the dirty writes to an earlier time, freeing up the bandwidth to handle ust data reads and reducing the pressure on the UU.

t ract c r

In order to evaluate the effectiveness of Eager Writeback in a real system, we implemented a memory traffic in ector which we used to model other bus agents requesting the memory bus and consuming memory bandwidth. For this benchmark study, we in ected three different external bandwidths using two different in ection frequencies on the bus during the simulations. The external bandwidths chosen were 400M 800Msec, and 1.sec G For each bandwidth configuration, data was in ected at a high frequency (every 400 processor clock cycles) and a low frequency (every processor 00 clock cycles). Data was in ected to the bus in blocks - for example, in the 800M high frequency case, every 400 cycles the in ector took the bus and held it until it had completed transferring 0 yt of data. The in ections are uniformally distributed throughout the simulation.

The results for simulations of the mini-geometry kernel using no light sources are shown in Table 6. The top line of the table is the base case with no in ected memory traffic, while the other entries are for the different in ected bandwidths at the different frequencies. In this table we can see that (as

s	ase		ager e	e	ager	ager		ree	ar e
write bu er	size cycles	cycles	speedup	cycles	speedup	cycles	speedup	cycles	speedup
B buf = 1	10230328	9054559	1.130	9053851	1.130	9053851	1.130	9045154	1.131
B buf = 4	1006	905295	1.112	905295	1.112	905295	1.112	9045154	1.113

Table Simulated cycles of Cache_walk ernel.

expected) memory traffic in ections causes extra stall cycles in the UU. In addition, as the amount of in ected bus traffic increases, the opportunity to do Eager Writeback decreases and the UU stalls increase dramatically.

The table also shows that Eager Writeback provides virtually no speedup when a bandwidth of 0.8G sec is in ected at the higher frequency, while the same bandwidth in ected at a lower frequency allows a speedup of 11 . y examining the dirty writeback bandwidth utilization profile of this scenario (Figure 11 and Figure 1 one), can see that many eager writebacks (i.e. the spikes) are prevented from occurring by the higher frequency in ection. The advantages of Eager Writeback are lost and it performs almost on par with the baseline scenario, due to more frequent bus contention.

.3 _we er e

The mini-geometry kernel highlighted the problem of implicit dirty writebacks causing loss of performance due to delays in receiving data. Finite memory peak bandwidth is another serious performance issue, which is exposed by the Cache_walk kernel.

t tract c r

Table tains the results of simulation runs of the Cache_walk kernel, presented in the same format used in Table 4. For this benchmark, an eager queue of length 0 (E 0) is enough to approximate the optimal case of no dirty writeback traffic at all. Further size increases of the E vide only marginal performance gains.

Looking at the memory bandwidth utilization profiles for this kernel (Figure 1 and Figure 14), we see three spikes that appear repeatedly in both writeback schemes (because the outer loop contains three iterations). The spikes are much wider in the ase case, however, indicating the program is spending more execution cycles in these phases. Examining the algorithm, it is clear these spikes are related to the time during the third inner loop where incoming *array* [] data items collide and share memory bandwidth with the induced dirty writebacks of $array_A$[]. ecause the finite memory bandwidth (1.6 G in this study) must be shared between both memory accesses[4], the rate of demand fetches for *array* [] in the third inner loop is (theoretically) cut in half and thus the overall performance degrades.

Figure 1 also shows three bandwidth grooves where memory bus bandwidth has dropped to zero. This corresponds to the second inner loops, where all data references hit in the cache. To take the advantage of this available resource, Eager Writeback fills these bus idle states with early evictions of dirty data cache lines as shown in Figure 14. y shifting these bandwidth requests to idle cycles, the memory bandwidth during the course of the third inner loop can be fully dedicated to the demand fetches of *array* [], speeding up the cache fill requests.

As was done for the mini-geometry kernel, we examined how Eager Writeback interacted with internal processor resources when running this benchmark. Table 8 shows that the Load Store ueue is used heavily by this benchmark, and that Eager Writeback can remove more than half of the stalls

ead and write turnarounds between demand fetch and dirt writeback streams also pre ent peak memor bandwidth from being achie ed.

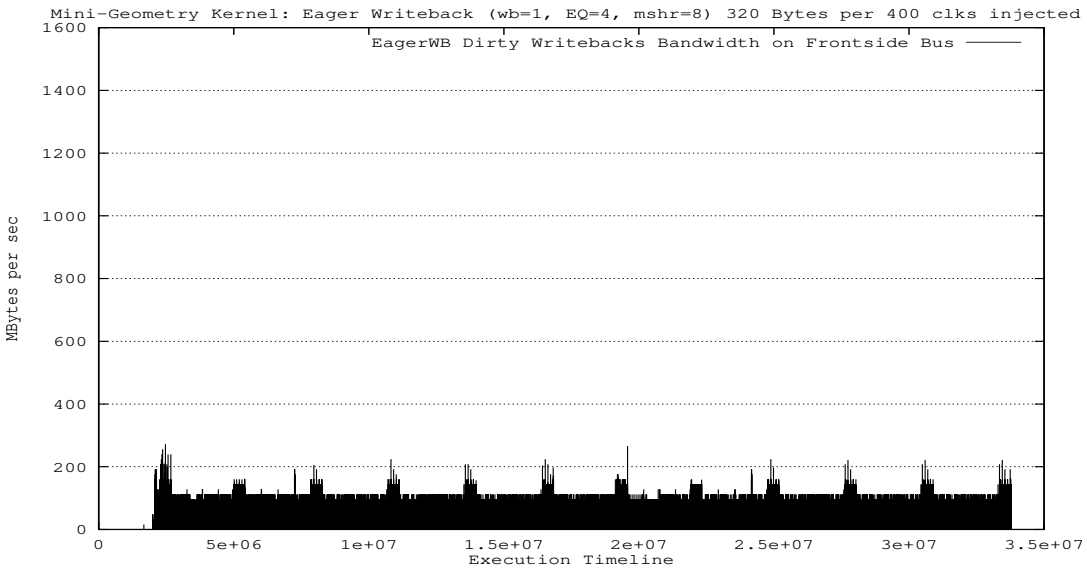

Figure 11 Dirty W L -to-Mem andwidth with In 0 ecting *Eager*( ) 400 clks for Geometry

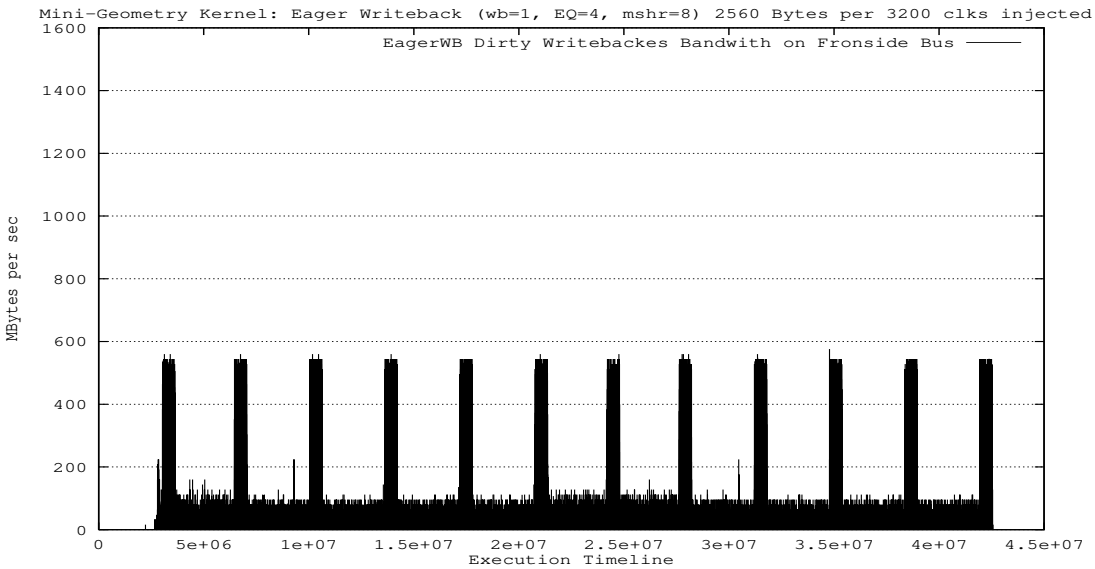

Figure 1 Dirty W L -to-Mem andwidth with In 560 ecting *Eager*( ) for Geometry 00clks

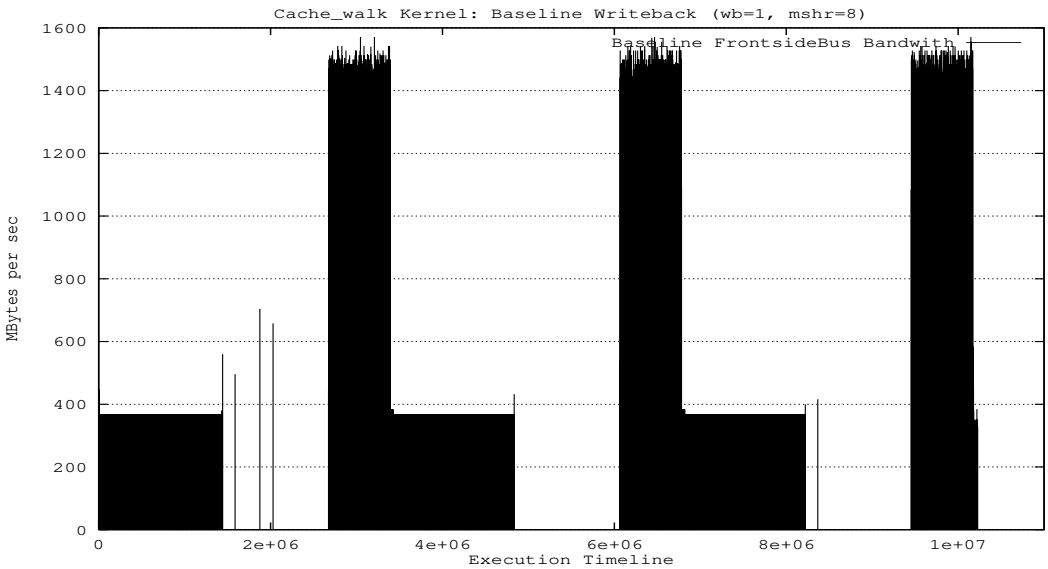

Figure 1 Memory andwidth Distributiyon b *aseWriteback* for Cache_walk ernel

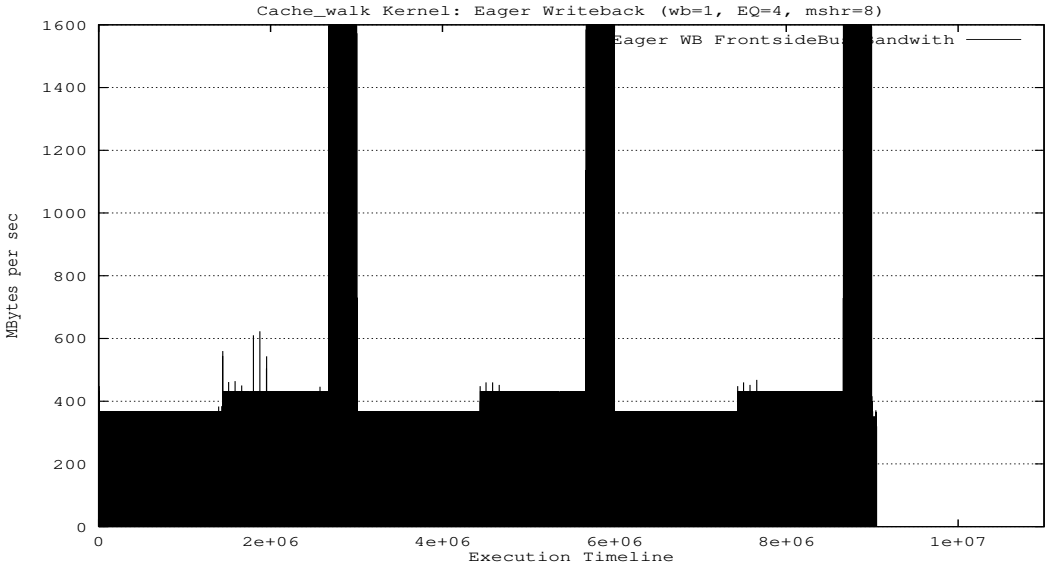

Figure 14 Memory andwidth Distributiyon *Eager Writeback* for Cache_walk ernel

Bottlenecks	ase cycles	ager e cycles	ager e improved	ager cycles	ager improved	ager cycles	ager improved	ree ar cycles	ree ar improved
ull cycles	5	4594401	20.3	4594631	20.3	4594631	20.3	458	638 20.49
RUU ull cycles	42	4826	84 0.33	4260	03 0.33	4260	03 0.33	4258811	0.38
L ull cycles	19	8596486	56.29	866341	56.21	866341	56.21	862880	56.39

Table 8 esource Constraint Improvement of Cache_walk ernel (Writeback buffer 1)

a e	s h Baseline	s h	ager ed-up	a e es Baseline	a e es	ager improved	es Baseline	es	ager improved
0 MB/sec	10230328	9053851	1.13	5	4594631	20.4	19	8596 866341	56.2
0.4GB/sec (160B/400clks)	1180	1408 39848	1.18	3406	15	6536 24.0	2903145	1205358	58.5
0.8GB/sec (320B/400clks)	1502595	12389159	1.21	105408	9080	25.0	44284	188 8258	5 .5
1.2GB/sec (480B/400clks)	24250335	21412 35	1.13	19	1 16880309	46 1.4	8309036	5480188	34.05
0.4GB/sec (1280B/3200clks)	123	9209 91058	1.13	9085	3521201	1	2 5030932	141	595 30.2
0.8GB/sec (2560B/3200clks)	16593	15 1815348	1.10	12101456	10622058	12.2	4264295	2818313	33.9
1.2GB/sec (3840B/3200clks)	29048835	2 135235	1.0	24495295	22585042		8 9003039	00	21.3451

Table 9 Memory Traffic In ection the Cache_walk ernel (Eager ueue 4)

due to a full Load Store ueue. As the LS is kept less full, instructions are able to leave the IF faster and as a result cycles lost due to a full IF are reduced substantially.

t ract c r

We also repeated the experiments involving in ecting memory traffic on the bus for this benchmark program. The results are shown in Table 9, and indicate that higher frequency in ection seems to ha a greater impact on the aseline case than on the Eager Writeback case. The number of simulated cycles for the aseline case using high frequency in ection increases faster than for the Eager Writeback case, while the increase stays roughly the same for both schemes while in ecting for frequency traffic.

The reason the cycle count climbs faster for the aseline case than for the Eager Writeback case can be understood by analyzing Figure 15. This figure contains an execution profile of the Cache_walk benchmark, plotting the arrival time for each load instruction. From left to right, the four curves represent Eager Writeback with no extra bus in ection, aseline with no extra bus in ection, Eager Writeback with higher frequency in ection, aseline with higher frequency in ection. Each curve can be divided into repeated patterns, which bear the following three piecewise line segments at (zero increment), steep rise, and slowdown knee. These line segments correspond to the three inner loops in the benchmark.

The first loop contains only data stores, so the load instruction count stays as at execution time continues. The steep vertical climb corresponds to the second inner loop, which has a high number of cache hits (a large number of loads completing in a short period of time). Finally, the third segment represents the behavior of the third loop, which loads another array that misses in both the L1 and L caches.

This third segment, shown as a knee in the curve, is the key to the performance deviation between aseline and Eager Writeback. Figure 16 shows a close-up view of part of Figure 15, focusing on the knees of the curve. The slopes (tan) of these knees are the key - the atter the slope, the longer it will take to complete. Comparing the slope changes between aseline and Eager Writeback, it is obvious that the slope of the aseline segment is much shallower than that of the Eager Writeback segment. This means that for the same number of loads in the third loop, the execution time of the aseline case was more sensitive to and severely delayed by other transactions, which in this case are composed of

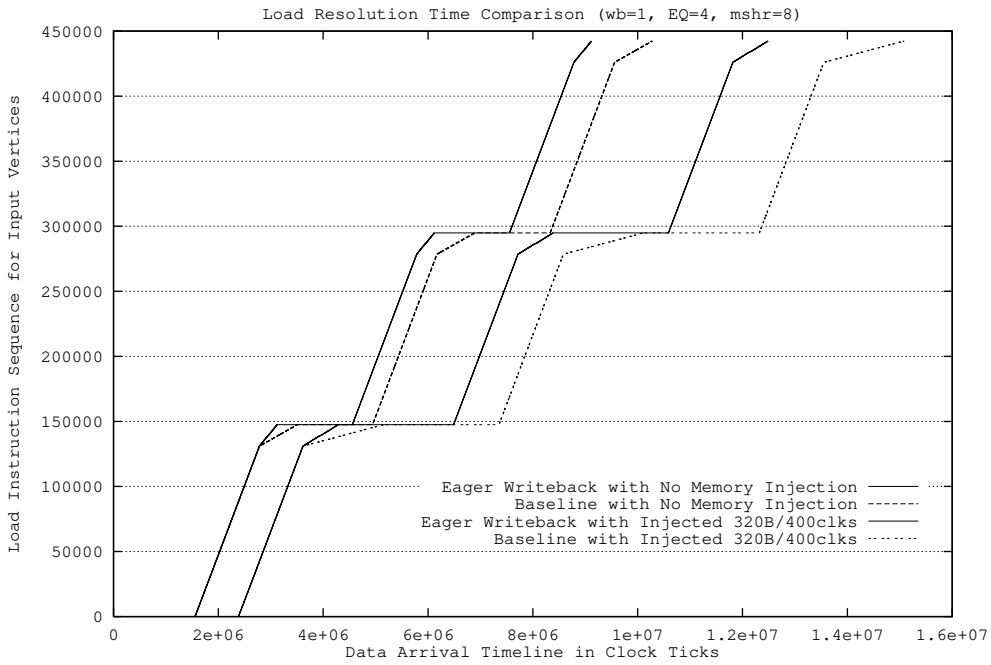

Figure 15 Load esponse Time for Data eads in a Clac ernel (Higher requency In ection)

the dirty writebacks induced by the loads and the periodic in ection of memory traffic. In the Eager Writeback case, the dirty writebacks were mostly completed in the second loop, so the slope of the knee is steeper and the third loop can be completed more swiftly than its aseline counterpart.

epeating the same experiment using lower frequency in ection (as plotted in Figure 1 and Figure 18) reveals that the slope of the knees of the curve are much more similar to one another. As a result, roughly the same number of penalty cycles were added to both aseline and Eager Writeback, and the speedups due to Eager Writebacks are smaller in Table 9. These results suggest higher frequency memory interference can deteriorate performance in the baseline case more in a bandwidth-limited code.

onclu ion

Systems employing write-back caches have to contend with the following two issues (1) Dirty write-backs contend with demand fetches for bandwidth and can impede the delivery of data, and ( ) Finite memory bandwidth shared between demand fetches and implicit dirty writebacks limit the performance of memory bound programs. These performance issues are important to a large and growing class of programs – those that consume large amounts of memory bandwidth and generate many data stores.

In this paper we have presented a new technique for dealing with these issues, called Eager Writeback, which can effectively improve the overall system performance by shifting the writing of dirty cache lines from on-demand to times when the memory bus is idle. We have shown that applying this technique can alleviate bandwidth constraints and improve performance for two kernels that are representative of these classes of applications. We have shown that when conventional writebacks compete with memory loads and defer the delivery of data, the Eager Writeback technique was able to remove the competition by evicting dirty data earlier. We have also shown that when finite memory bandwidth limits overall performance, eager writeback can alleviate this situation by utilizing earlier idle bus cycles.

Further investigation of this Eager Writeback mechanism will include the effects this approach has

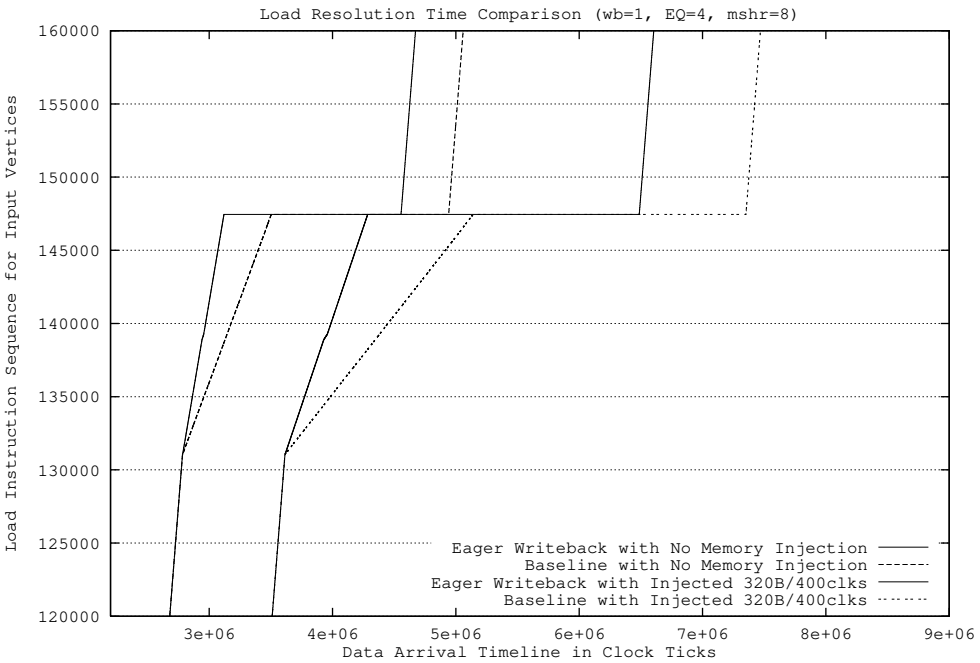

Figure 16 oom-In of the Load esponse Time for Data he_walk in Cac ernel (Higher Frequency In ection)

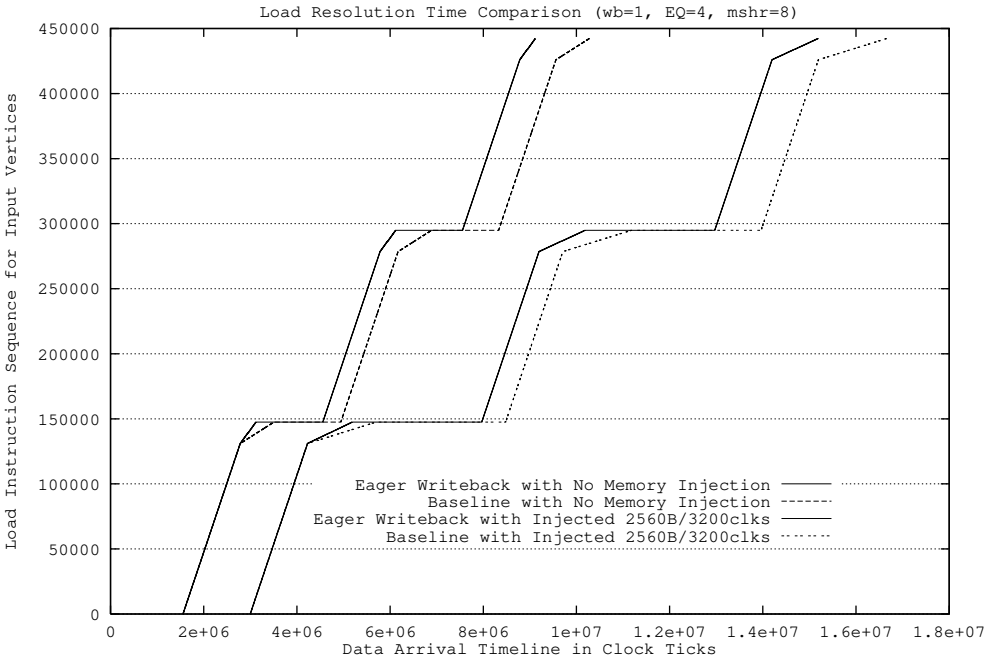

Figure 1 Load esponse Time for Data eads_walk in Cac ernel (Lo Frequency In ection)

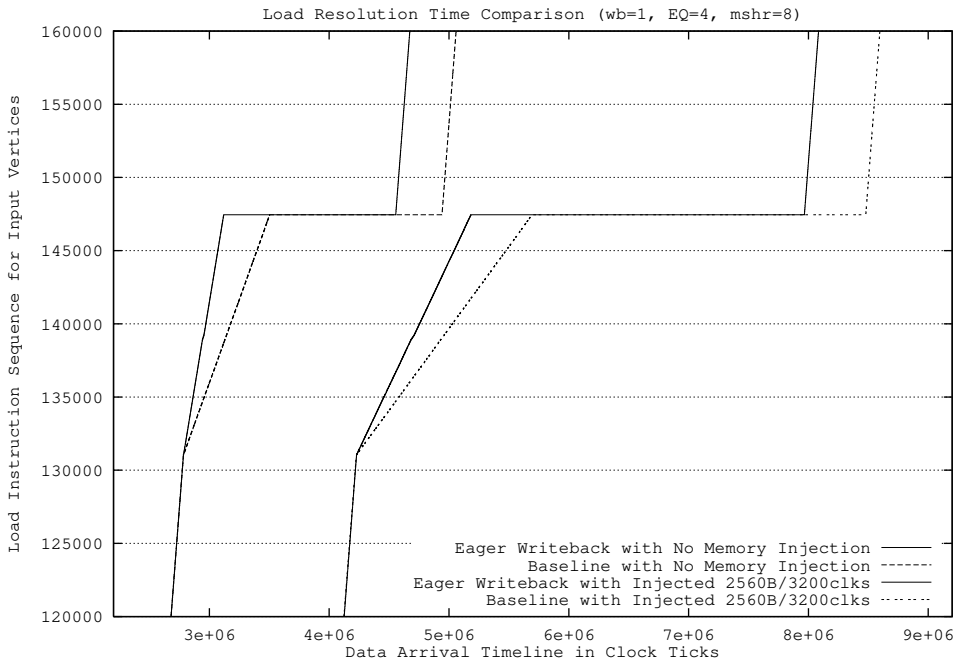

Figure 18 oom-In of the Load esponse Time for Data he_eads in Cac ernel (Lo Frequency In ection)

on other system performance issues. For example, Eager writeback can potentially reduce context switching time overhead by ushing lines in advance of the context switch. In addition, Eager Writeback can push modified data closer to the globally observable memory level earlier to reduce coherence miss latency, and as a result, respond to other processors requests faster. Similarly, the same analysis performed in this paper can be applied to write-update and write-invalidate protocols in a shared memory system to reduce coherence traffic.

eference

[1] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[2] Intel Corporation. Pentium pro family developer's manual, volume 3: Operating system writer's manual. Intel Literature Centers, 1996.

[3] Rambus Corporation. Direct rambus memory controller (rmc.dl) data sheet. http://www.rambus.com/docs/RMC.d1.0036.00.8.pdf, 1999.

[4] Standard Performance Evaluation Corporation. Spec cpu95 benchmarks. http://www.specbench.org/osg/cpu95/, 1995.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann Publishers, Inc., 1990.

[6] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of 8th Annual International Symposium on Computer Architecture*, 1981.

[7] Simplescalar Tool set. X benchmark suite. http://www.cs.wisc.edu/ austin/simple/xbenchmarks.tar.gz, 1998.

[8] Kevin Skadron and Douglas W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of 3th International Symposium on High Performance Computer Architecture*, 1997.

[9] Suhii and Sriram a anpey. Instruction issue logic for high-performance interruptable pipelined processors. *Proceedings of nnual Int A national Symposium on Computer Architecture*, 1987.

[10] Alan Watt. *3 Computer aphics*. Addison-Wesley Publishers, 1993.