A Two Level Virtual Cache with Deferred Physical Address Translation

Based on the MIPS R4400

Revised

Marc Mosko

Cupertino

Dr. Anujan Varma

CMPE 202, Spring 1998

Preface to Revised Version

This revised version contains corrected data for the 022.Lisp series. Accidentally, we did not count L2 invalidations in our original trails. The changes only affect the material presented in Section 6. We have reprinted the entire document for completeness.

We did not have time to reprint the graphs in color. The black and white graphs are a little difficult to read, since the data series overlap so much. There are essentially two trends. The Dinero trials overlap and our invalidation trials overlap, similar to the 026.Compress graphs.

Table of Contents

List of Figures

Abstract

This paper investigates virtually indexed / virtually tagged L1 Instruction and Data caches with a unified virtually indexed, physically tagged L2 cache. The TLB is moved to the L2 level. Based on the MIPS R4400MC, we construct an architecture which enforces Multilevel Inclusion and a Write-Invalidate/Write-Update cache coherency protocol. Using trace simulations derived from SimOS SPECint92 workloads, we analyze the behavior of the system. We find that this architecture suffers from a large number of forced L1 and L2 invalidations, which are required to maintain consistency with the TLB and between cache levls. We explore a variety of invalidation schemes to reduce these negative effects.

1. Overview

The paper will investigate virtually indexed, virtually tagged (V/V) primary caches. We use the MIPS R4400MC architecture as our point of departure. Using simulation and analytical models, we will explore how a V/V primary cache affects system performance. Our design will model separate Instruction and Data V/V primary caches supported by a Virtually indexed, Physically tagged (V/P) second level cache. We will look at issues of Multi-Level Inclusion [Baer88, Wang89, Wheeler92] and virtual address synonyms [*ibid.*]. The analysis will focus on a uniprocessor design, though we have considered multi-processor coherence issues. We use simulations to measure various cache events. Our models use Cycles Per Instruction (CPI) to compare the relative benefit of our proposed architectures. We also compare our architectures MIPS rating with a physically indexed/physically tagged version.

In the following section, we present some of the motivation behind this research. Section 3 gives an overview of our architecture and illustrates the cache line structures we used. It also discusses some of the design issues we faced in creating our proposed architecture. Section 4 presents a more detailed description of the interactions between cache levels. This section also discusses data coherency and virtual address issues, such as Inclusion. In Section 5, we describe the simulations used to analyze our design and the statistics we collected. We also specify the computed statistics, such as Transfer Ratios

and Cycles per Instruction. Section 6 analyzes our results. References appear in Appendix A and data graphs appear in Appendix B.

2. Motivation

The principle advantage of a V/V cache is a reduced processor workload, not an increased cache hit rate. If we can sufficiently reduce the workload per pipeline stage, then we could reduce the PClock period. The MIPS R4400 has an eight stage pipe. Each primary cache access takes three stages (IF,IS,RF for I-cache and EX,DF,DS for D-cache). By using a V/V primary cache design, we would remove all TLB functionality from the processor pipeline. In our model, we would use a joint TLB at the secondary cache interface. We will analyze the R4400's secondary cache bus protocol and timing. The simulation will not include the pipeline, but only the cache.

3. Architecture

The proposed architecture is shown in Figure 1. Our goal was to design a Virtually Index/Virtually Tagged primary cache with a Virtually Indexed/Physically Tagged secondary cache. Since we moved the TLB to the L2 level, we must include some TLB functionality regarding coherency in to the primary cache lines. We also moved the TLB lookup out to the secondary cache interface. The External Agent interface to the System Bus needed a Reverse TLB since the L2 cache is virtually indexed.

Using a virtually indexed secondary cache caused problems when we attempted to connect the System Interface logic. Since the System Interface uses physical addresses, we needed a way to associate physical addresses to the L2 cache's virtually indexed table. There were several choices available at this stage. The most straight forward design would have been to use an in-line TLB for the secondary cache, thus making the cache physically indexed and physically tagged. This would be similar to the R6000. We decided against this design, since it did not meet the original goals. We could have also worked with the L2 cache, making it searchable by the page frame number. Another approach might have made the TLB associative on the two PFNs. Both of these approaches would require very fast logic, since the L2

General Architecture

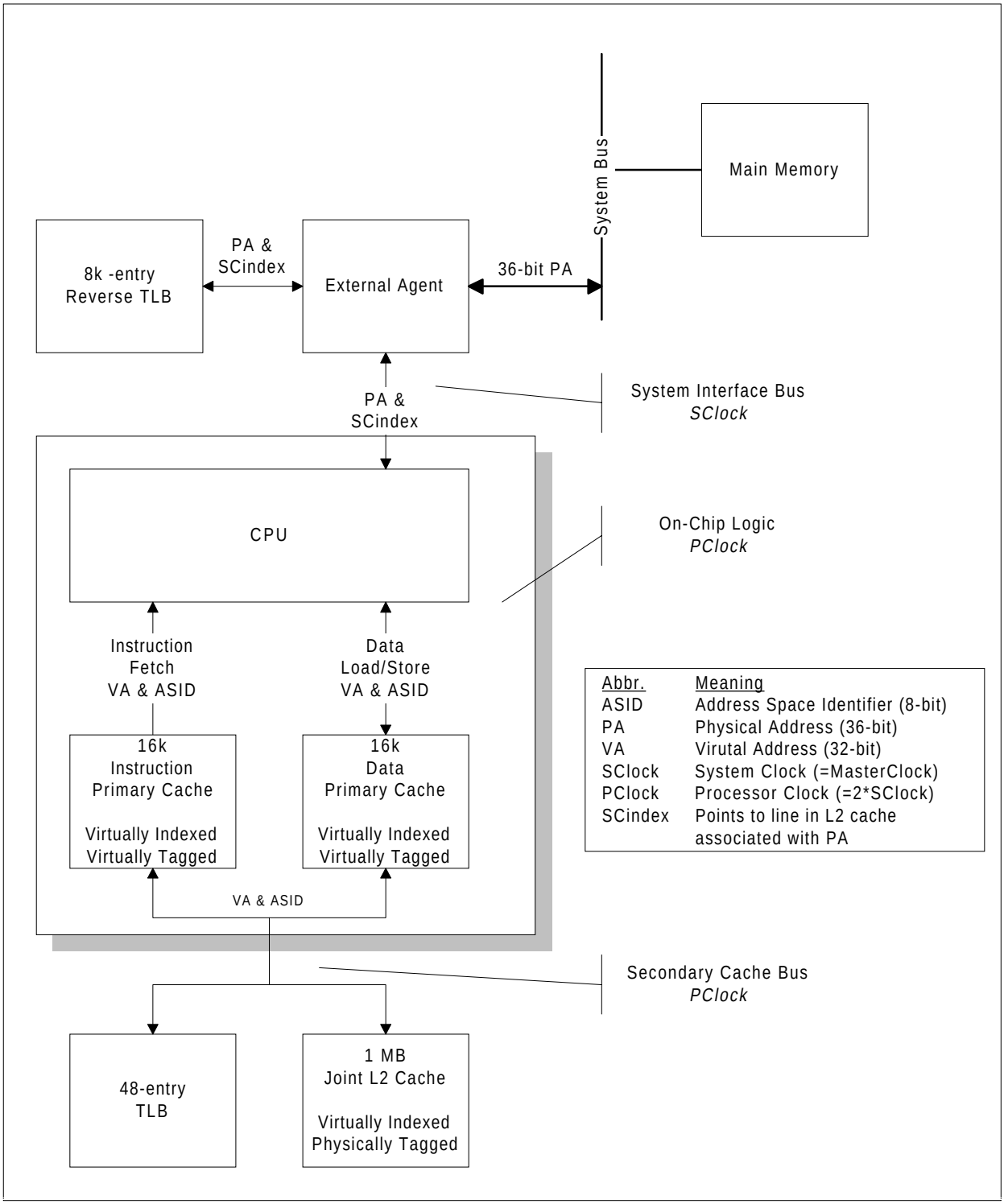

Figure 1

cache runs at *PClock*. Some systems solved this problem by using globally unique virtual addresses (ether software or hardware generated) and moving the TLB to the main memory level [Wood90].

We chose to solve the virtual index problem by using a reverse TLB at the External Agent. There are several benefits to this. The External Agent is only used for main memory requests, I/O requests, or processor coherence functions. We believe these would be significantly less frequent than L2 cache requests, so the logic is used less than if we placed it on the L2 side of the processor (though this assertion lacks empirical proof). The External Agent interface runs at *SClock*, which is ½ the rate of *PClock*, so the logic does not need to be as fast as if it were on the L2 side of the processor. Addresses on the L2 side of the processor do not need this mapping. Only addresses that enter from the System Bus require a reverse physical to virtual mapping. Another benefit is that we only operate on those addresses that need a physical to virtual mapping. It can also be used to detect synonyms.

We have left the exact implementation of the Reverse TLB undefined. The RTLB, we expect, would be fully associative, but with non-uniform access time. The RTLB would probably be an expensive piece of hardware and introduce noticeable delay. We have kept it in the design for the sake of completing our proposed L1 V/V and L2 V/P cache scheme.

Adding a reverse TLB to the External Agent logic does introduce another point of possibly stale data. We must ensure that all entries in the reverse TLB point to the correct L2 cache entries. The RTLB can lose consistency whenever an L2 cache line is invalidated. We can easily ensure consistency when an L2 line is invalidated because of an L2 cache miss, since a new Main Memory update is performed. All these updates must pass through the External Agent, and we may “snoop” the contents of the L2 cache. The only other time[1] an L2 line is invalidated is on a TLB update (see below), for which we must add more cache coherency logic.

[1] There is an instance of a CACHE instruction in the multiprocessor coherence protocol when this is not true.

4. Implementation

Figure 2 shows the various data structures we use to implement our L1 virtually indexed, virtually tagged Instruction and Data caches, along with an L2 virtually indexed, physically tagged cache and TLB. Figure 2 shows the L1 structures for a 16k cache with a 16 byte line size. We will vary these sizes later, but the basic diagrams still hold. We have left the exact implementation of the Reverse TLB undefined. We have not investigated the effects of DMA I/O that must pass through cache memory.

Like the R4000, we use direct mapped caches. The main reason is simplicity. Using an associative cache design would make the Multi-Level Inclusion and TLB-Cache consistency problems much more complicated. Our model uses the same multiprocessor coherency protocol as the R4400MC. We believe that our mapping of cache coherency information from the TLB, which used to be at the L1 cache level in the R4400MC, to our D-Cache tag line is sufficient to maintain the multiprocessor coherence. We have not performed an analysis of cache state transitions under multiprocessor write-invalidate/write-update.

Cache Line Formats

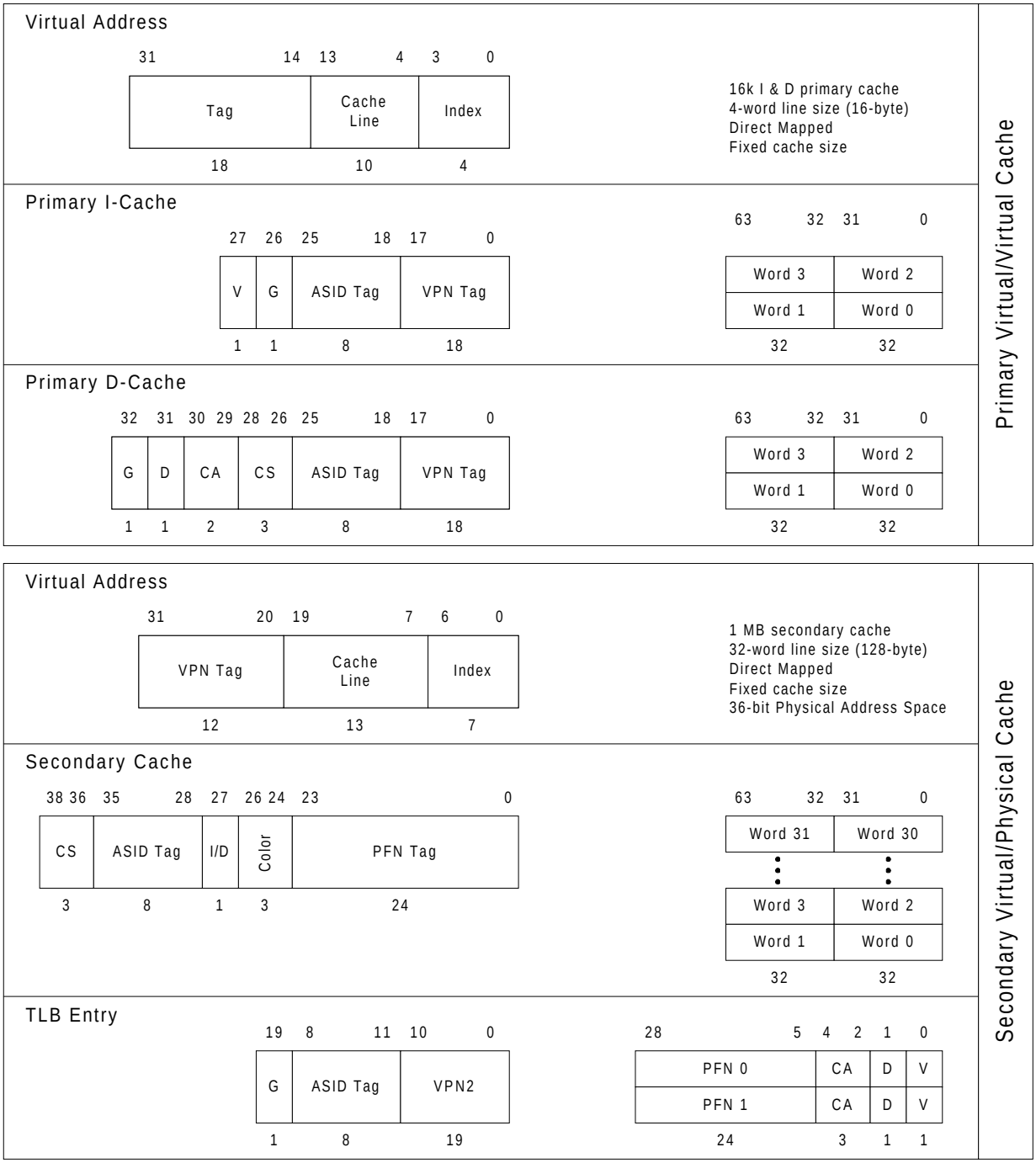

Figure 2

4.1 Primary Cache Overview

We have separate Instruction (I-Cache) and Data (D-Cache) caches. We do not use a TLB for the primary caches. They entirely rely on matching the Virtual Page Number (VPN) tag and the process ASID. We had to move some information from the TLB in to the primary cache tag lines. In our simulations, we will vary the L1 cache sizes between 16k and 64k, with line sizes between 16 bytes and 128 bytes.

In the R4400, the TLB contained a Global bit and a Valid bit. The Global (G) bit determined if the ASID should be compared for a cache hit. The TLB also contained a Valid (V) bit for each Page Frame Number (PFN). We include the G-bit in both the I-cache and D-cache lines. The V-bit is used in the I-Cache. The D-Cache does not use a V-bit, but uses the Cache State (CS) bits, as defined for the R4400 [MIPS96].

The R4400 uses a Write-Invalidate/Write-Update cache coherence protocol. Since the I-Cache is read-only, we do not need to maintain any coherency information for it, apart from valid/invalid. The D-Cache must maintain coherence information. In the R4400, the TLB contained the Cache Algorithm (CA) used per page (4k in our case) and the D-Cache contained the Cache State (CS) information. Since we moved the TLB to the secondary cache level, we need to include both the CA and CS information in the D-Cache tag line. When a line is written to L1 cache, it inherits these coherence characteristics from the TLB and L2 cache.

The D-Cache uses a write-back protocol. The need for write-back is controlled by the Dirty (D) bit. The D-bit determines if a line must be written to L2 cache before being replaced or invalidated. This is different than the model of the R4400. The R4400 does not require write-back before invalidating a line. The R4000 model allows a primary cache line to write back directly to memory, even if there is no corresponding L2 line (in some cases) [MIPS96]. Since we have no physical address information at the L1 level, we must rely on the L2 cache for this information. If an L2 line is replaced or invalidated, all dirty

dependent L1 lines must be written-back to L2 cache before we lose the physical memory address information. As described below, this could cause a large number of L1 lines to be invalidated and written back immediately on an L2 miss.

4.2 Secondary Cache Overview

The secondary cache contains both instructions and data. We chose a 1 MB L2 cache size, with 32-word (128-byte) line size. It is direct mapped. Like the R4000, we use a 48-entry, fully associative TLB, which maps two physical pages per TLB entry. We call these PFN0 and PFN1, for the even and odd page, respectively. The L2 cache is virtually indexed and physically tagged. We use a parallel L2 lookup and TLB probe, with a PFN tag comparison. The V, D, CA, and CS bits are as described above for the primary cache. On a TLB miss, we throw a TLB Refill exception. On an L2 miss, we load the L2 line from memory using the PFN from the TLB. Both of these operations have implications for the data in the L1 cache, as described in the next section.

To maintain L1 inclusion, we must selectively invalidate L1 cache entries on an L2 invalidate or update. This turns out to be the major performance issue with our cache design. We would also need to write back to main memory, via the L2 cache, all dirty D-Cache entries we are invalidating. We must do this to ensure inclusion and because we will lose the physical memory address when the L2 line is updated. One simple method would be to invalidate all 16 L1 cache lines associated with a single L2 line. This paper will investigate several schemes to manage these invalidations.

Our base-line L1 invalidation scheme, called *Color Invalidation*, invalidates an L1 line when a difference L1 line references the same L2 line. This changes the “color” of the L2 line, and thus the name. In an L2 line, we color it to index which L1 line is currently holding the L2 entry. This is similar to the Pinx of the R4400’s L2 cache. The 3-bit color (this size may vary with L1 cache line size) determines which of the 8 possible L1 lines last referenced the L2 line. We further track whether it was a reference from the I-Cache or D-Cache. On each L1 miss and L2 hit, we first invalidate the L1 entry pointed to by the L2 line, if the ASID and VPN match, before giving ownership to the new L1 line. If the old L1 line was dirty, we

write-back to L2 and clean the lines. This enforces inclusion, coherency, and minimizes the work that must be done when an L2 line is invalidated or updated. It turns out that this protocol is overly conservative and leads to terrible performance.

4.3 L1/L2 Coherency

One advantage to a multilevel cache design is that the L2 (or higher) cache can insulate the L1 (or lower level) cache from the cache coherency protocol. As noted in other papers, such as [Baer89] and [Wang89], this is true only if the L1 cache is a proper subset of the L2 cache. This is true in the MIPS R-series, for the data cache. Our architecture, however, adds complications to the R-series models. Not only must we enforce Multi-Level Inclusion (MLI), but we must also ensure that the L1 ASID's are valid in the L2 TLB.

To enforce MLI and minimize the burden of L2 line invalidation, we only allow one L1 line to reference an L2 line, as described in the previous section. On every L1 miss/L2 hit, we invalidate the last L1 line to reference the L2 line and perform any needed L1 write-back. We then give ownership to the new L1 line by recording its color and type in the L2 line. The color of an L1 line is VA(6:4)[2]. These three bits select one of eight possible L1 lines that could reference a given L2 line. We further record whether the request came from the I-Cache or D-Cache in the I/D-bit of the L2 line. Before we invalidate the old L1 line, we must match the VPN tag and ASID (if not Global) tags. If they do not match, we can proceed without invalidating or writing-back the L1 cache line, since it belongs to a different L2 line.

In addition to the L1/L2 coherency, we must enforce a TLB/L2 coherency. We solve the TLB consistency problem similar to [Wood90]. In normal operation, the operating system will selectively invalidate ASIDs by invalidating lines in the TLB. In the R4400 model, these changes are effective instantly, since the TLB is referenced for every instruction fetch and load/store. In our model, we must invalidate L1 lines affected

[2] In this document, we will refer to Virtual Addresses as VA and Physical Addresses as PA. We show the inclusive bit range in parentheses. Thus VA(6:4) means bits 4,5, and 6 of the Virtual Address.

by destructive TLB changes[3]. Since the TLB does not contain sufficient information to directly invalidate a primary cache, we invalidate the corresponding L2 cache entries and let that filter down to the L1 caches by the means described above, and those methods in Section 5.2.

A pair of 4k PFN entries in the TLB corresponds to a possible 64 L2 lines. Each PFN from the TLB matches 32 L2 lines. For each valid TLB PFN that we are updating, we must scan the corresponding 32 L2 lines and if the PFNs match, we must invalidate that L2 line. L2 line invalidation follows the rules outlined above, and we must possibly invalidate and write-back an L1 line for each L2 line. This could possibly lead to 64 L1 write-backs in the worst case[4]. We only need to update main memory when an L2 line is replaced, which would not necessarily correspond to a TLB update. Under this scheme, we can never have an L2 hit if there were a TLB miss. This procedure demands further optimization, as described below, in Section 5.2.

4.4 Virtual Address Synonyms

Like the R4400 and other systems, the operating system should avoid virtual address synonyms by forcing all shared data to reside in aligned cache lines. Whereas in the R4400 this meant the same L1 cache line, in our implementation it means the same L2 cache line, which is substantially larger. We may operate at the L2 level, since we only allow one valid L1 line per L2 line. This would change under other L1 invalidation schemes.

In those instances where the operating system does not obey this rule, we detect the synonyms in the RTLB of the External Agent. On boot-up we invalidate all lines of the RTLB. On L2 line invalidation, we invalidate the corresponding line of the RTLB (see below). Since the RTLB is physically indexed, we can

---

[3] A destructive TLB change would modify the ASID, VPN2, PFNs, or V-bits. Modifications of the D- or CA- bits would be considered non-destructive.

[4] If we did not require L1 write-back on change of L2 ownership, it could lead to 512 L1 write-backs.

tell if two virtual addresses correspond to the same physical address. When such a situation arises, we can throw a coherency exception.

4.5 Reverse TLB

The Reverse TLB is necessary, since our L2 cache is virtually indexed. Without the RTLB, on receipt of a physical address from the External Agent, we would have no idea which L2 line it referenced – if any – without an exhaustive search of the 8192 L2 lines. Our solution is to use a fully-associative RTLB at the External Agent interface. It is indexed by the 24-bit PFN plus PA(11:7) to extend the index to a specific L2 line. There must also be some sort of tag comparison. The table stores a Valid (V) bit and an 8-bit L2 line index (SCindex). SCindex is VA(19:12), which when extended with PA(11:7), forms the 13-bit L2 cache index. We maintain a one-to-one mapping between RTLB entries and L2 cache lines. We have modified the System Interface to accommodate sending the SCindex. We add entries to the RTLB when the processor issues a main memory request (i.e. load/store). We do not add entries for in-bound activity. In-bound commands are matched against the RTLB. If a match is found, the request is forwarded to the processor, since we know the physical address is held in some state by the L2 cache. If the RTLB does not match the PFN, we can ignore the system bus activity, since the processor does not hold the physical address in any L2 lines.

We must remove RLTB entries for certain L2 cache events. When an L2 line is invalidated and clean, we must remove the RTLB entry at that time. If an L2 line is invalidated but dirty, we do not remove the RTLB entry until the L2 line is written back. Note that we do not store coherency information in the RTLB. We only store the fact that a physical address page is in use in the L2 cache.

5. Simulation Models

Since our architecture is dependent on virtual and physical addresses, we chose to use the Stanford SimOS [Herrod98] environment for our initial modeling. We implemented the architecture described above in the Mipsy cache code. We were able to extract virtual address/physical address traces from

SimOS to run through an external simulator, which ran much faster. Our custom simulators were also platform independent[5], unlike SimOS. We were also able to use Dinero to generate statistics from a physical address trace, for comparison. The simulation used two SPECint92 benchmarks, 022.Lisp and 026.Compress. The choice of benchmarks was determined by which we could get to run, easily, under SimOS. The remainder of this section will describe the traces we extracted, the different simulation runs, and the formula we used to calculate performance statistics.

5.1 SimOS Traces

Our traces included several types of information. We tracked, for each access, virtual address, physical address, and ASID. We also recorded the type of operation, namely Instruction Fetch or Data Load/Store. We also recorded TLB write updates as L2 cache invalidation requests. We recorded the VPN and PFNs stored in the TLB with an operation type of L2 Invalidate. Each TLB operation generated two trace entries, since there are two PFNs per TLB entry. We did not include information on Global memory references. We drew our traces from the first 200 million cycles of the benchmarks. Using methods in SimOS, we were able to start dumping our traces on the first cycle of the first command (xlisp or compress) of each benchmark, and then run for a specific number of cycles. For each benchmark, we modeled twelve cache configurations. We used cache sizes of 16k, 32k, and 64k for the L1 cache. We varied the line size between 16, 32, 64, and 128 bytes. The L2 cache size was held constant at 1 MB with a 128 byte line size.

Table 1 summarizes the characteristics of the 022.Lisp and 026.Compress traces. One will notice that the Lisp trace spends its time in only two processes, the operating system and **xlisp**. Compress, on the other hand, runs several processes. The benchmark alternatively compresses and uncompresses files. There is substantially more task switching than in the Lisp trace. These differences will also appear in the

[5] We ran simulations on IRIX 5.3, Solaris 5.4, and FreeBSD 2.2.6. Each simulation took between 20 and 40 minutes, depending on the machine (90k – 200k references/second).

relative performance measures. We measure the percentage of time spent in the kernel as the number of memory references outside KUSEG divided by the total number of memory references.

Benchmark	Total Refs	Total Insts	% IF	% RD	% WR	% Kernel	ASID	% Refs
022.Lisp	286,696,356	199,998,859	69.8%	18.8%	11.4%	10.0%	0x0000	0.009%
							0x0024	99.991%
026.Compress	266,762,027	198,394,496	75.0%	18.9%	6.2%	10.6%	0x0000	1.898%
							0x000D	0.026%
							0x000E	34.710%
							0x0010	61.456%
							0x0011	1.856%
							0x0012	0.049%
							0x0013	0.006%

Table 1. Trace Characteristics

When extracting the traces from SimOS, we used a Perfect memory model and had the caches always hit. Using this scheme, we could be sure to get the most compact traces from SimOS, without any dependency on the SimOS memory system. This method accounts for internal CPU stalls (e.g. data hazards) without delays caused by the cache/memory system. We did not include memory access size (e.g. half word). We recorded all references as full word (4-byte) references. We did not have enough time to model non-word references. Dinero will correctly process and note a multi-block word access. Our routines will truncate the access to the lower word boundary.

We also extracted a physical and virtual address trace in DineroIV binary format. We ran a reference set so we would have something by which to compare our results.

5.2 Simulation Runs

We used seven different variations on the architecture described above. We describe, briefly, each of these configurations. There are three L1 invalidation protocols, *Color, Fill,* and *Owner*. We also experimented with three L2 invalidation protocols, *Blind, PFN Matching,* and *PFN Matching with L1 Bit Map.* The L2 protocols take effect on a change to the TLB, caused by Write Random or Write Indexed TLB operations. These require that we invalidate the corresponding L2 lines.

L1 Color Invalidation

When an L2 line changes color (L1 referee) or Ownership (Instruction or Data cache), we invalidate the previous reference. On every L2 reference, we match the L1 line requesting the data. If the line number (color) is different than that stored in the L2 line, we invalidate the previous L1 line. Using this technique, only one L1 cache line may be outstanding which references a given L2 line. On an L2 invalidation or refill, we only need to invalidate one L1 line.

The performance of this technique was so bad (except when L1 line size equaled L2 line size), we have not included it in any of the graphs in Appendix B. It throws off the scale too much. As an example, the average L1 miss rate for this technique is 15.2%. The other techniques range between 1.1% and 1.6%.

L1 Fill Invalidation

In this scenario, we do not track the color or ownership of an L2 line. On an L2 invalidation or fill, we invalidate (and possibly write back) all corresponding L1 lines. This may range from 2 to 16 L1 lines, depending on the L1 line size.

L1 Ownership Invalidation

As long as the ownership (Instruction cache or Data cache) remains the same for an L2 line, we do not invalidate or write back. On an ownership change, L2 invalidation, or L2 fill, we invalidate all L1 lines from the old owner (I- or D- cache). Depending on the line size of the L1 cache, this may be anywhere from 1 to 8 L1 lines.

L2 Blind Invalidation

We do no checking and blindly invalidate all L2 lines that could possibly match a given TLB entry, based on the VPN stored in the TLB. Unless otherwise specified, this is the L2 scheme used throughout the paper.

L2 PFN Matching (PFNM)

We match the PFNs stored in the TLB with the physical address tag stored in the L2 lines. If a PFN is valid and matches the L2 tag, we invalidate the L2 line. We ran this scheme only with the L1 Ownership invalidation method, since that was the optimal L1 invalidation scheme.

L2 PFNM with L1 Bit Map (BMAP)

This is similar to PFNM with L1 Ownership invalidation. Rather than invalidate all possible corresponding L1 lines, we keep a bitmap in the L2 line of which L1 lines have referenced the L2 line. The bitmap is reset on fills, invalidations, or ownership changes. This scheme causes even fewer L1/L2 invalidations than the others.

5.3 Dinero Runs

We ran a physical address trace through Dinero IV, using cache sizes that corresponded to our models. We use these numbers as a gauge to the performance of our designs. We call this run “Din PA”. We also ran a Virtual Address trace through Dinero to measure the locality of the virtual addresses. It turns out that the invalidation rate in our models due to mismatched ASID is very small (under 1%), so we thought this would be a god measure of locality in lieu or Compulsory/Capacity/Conflict measurements. We use the virtual address trace to see if our virtual address traces showed similar miss rates.

Our virtual address cache management routines, of course, are not directly comparable to a physical address system. The physical trace system would require an in-line TLB. In a later section, we will compare the MIPS ratings of the two systems. The MIPS rating would estimate the performance penalty we could tolerate for an in-line TLB to achieve comparable performance.

It would have been more appropriate to use a cache simulator that runs virtual indexes and physical tags, but we did not have one. SimOS actually uses physically indexed, physically tagged caches.

5.4 Measurements

We measured the following statistics from each run. We used Dinero to verify our counting methods. The goal of these measurements was to give us L1 and L2 miss rates, L1 and L2 transfer rates, misses caused by invalidations (Invalidation Miss Rate), and Cycles per Instruction (CPI). In section 5.5, we present the equations used to compute these functions.

We tested our counting methods with small, hand-made traces run through both Dinero and our routines. We wrote a program the read an ASCII file and output either Dinero Binary or our propriety trace format. Our format includes virtual address and ASID information beyond the Dinero format. We verified that both Dinero and our routines counted the same number of Instruction Fetch, Data Load, and Data Store at the L1 level. We also verified that the number of Dinero misses equaled our number of misses less the number of misses caused by ASID mismatch. We avoided address sequences we knew would produce differences because of L1/L2 inclusion.

We used a select set of counters from Dinero and SimOS. From SimOS, we measured the number of instructions executed for both 022.Lisp and 026.Compress. We use this number, along with the number of cycles from our routines, to compute the CPI. From Dinero, we use the number of I-Cache and D-Cache references, I-Cache and D-Cache misses, L2-Cache references, L2-Cache misses, and the number of bytes transferred between memory levels.

From our routines, we measured the following counters. All miss counters were classified by cause: invalid cache line, mismatched ASID, or mismatched tag.

- I-Cache references
- I-Cache misses.
- D-Cache Read and Write references
- D-Cache Read and Write misses.
- D-Cache write backs.
- The number of I-Cache and D-Cache references to each memory segment (kuseg, kseg0, kseg1, ksseg, kseg3).
- I-Cache and D-Cache lines invalidated.
- L2-Cache instruction fetches
- L2-Cache instruction fetch misses.
- L2-Cache data reads and writes references.
- L2-Cache data read/write misses.
- The number of L2-Cache references that changed the color of an L2 line, broken out by Instruction or Data references.
- L2-Cache invalidations.
- L2-Cache write-backs.

5.5 Computed Statistics

This section presents the equations and constants we used to compute the following statistics. Table 1 presents the constants and their abbreviations. The MIPS R4400 has back-to-back reads and writes, so the times to read 32 bytes is shown in addition to the time for a 16 byte read. We use the appropriate sizes in our calculations (i.e. 16-byte lines use 16 byte reads, all others use 32 byte reads). The R4400 has a 32-bit main memory data bus. We did not model any advanced piggy-backing that it can do with certain types of operations. We assume an unlimited number of write buffers between memory levels, so our write times are zero.

Abbr.	*Cycles*	*ns*	*Notes*
Pclock	200	5	200 MHz
L1 Hit	1		per word
L2 16 Rd	6		per 16 bytes
$Cycles_{L2Rd}$	9		per 32 bytes
L2 16 Wr	0		per 16 bytes
$Cycles_{L2Wr}$	0		per 32 bytes
$Cycles_{Mem}$	16	80	per 8 bytes

Table 2. Performance Constants

In the following equations, we use the following shorthand. IF is “Instruction Fetch”, “Rd” is Data Load (read), and “Wr”. $REFS_{L1}$ is the total number of L1 references (instruction fetch, data load/store). REFSmem, similarly, means the total number of main memory references (i.e. L2 IF, RD, and writeback). We only show equations for L1 line sizes of 32 bytes or larger.

Total Cycles

$$Cycles = (REFS_{L1} \bullet L1Hit) + Cycles_{L2} \quad \text{(Eq. 1)}$$

L2 Time

$$Cycles_{L2} = \begin{array}{l} (IF_{L2} + Rd_{L2}) \bullet Cycles_{L2\,Read} \bullet \frac{LineSize_{L1}}{32} \\ + \; Wr_{L2} \bullet Cycles_{L2Write} \bullet \frac{LineSize_{L1}}{32} \\ + \; Cycles_{Mem} \end{array} \quad \text{(Eq. 2)}$$

Memory Time

$$Cycles_{Mem} = REFS_{Mem} \bullet Cycles_{MemRdWr} \bullet \frac{LineSize_{L2}}{8} \quad \text{(Eq. 3)}$$

Transfer Ratio

For cache level N,

$$Ratio = \frac{Total\ Bytes\ To/From\ Level\ N+1}{Total\ References\ to\ Level\ N} \quad \text{(Eq. 4)}$$

Thus, for L1 cache, the L1 Transfer Ratio measure the amount of traffic to the L2 Cache. Similarly, the L2 Transfer Ratio measure traffic between L2 and Main Memory.

5.6 Run Summary

Benchmark	Cache Size	Line Size	Invalidation Methods	References
026.Compress	16k	16	Color, Fill, Owner, PFNM, BMAP	Dinero
		32	Color, Fill, Owner, PFNM, BMAP	Dinero
		64	Color, Fill, Owner, PFNM, BMAP	Dinero
		128	Color, Fill, Owner, PFNM, BMAP	Dinero
	32k	16	Color, Fill, Owner, PFNM, BMAP	Dinero
		32	Color, Fill, Owner, PFNM, BMAP	Dinero
		64	Color, Fill, Owner, PFNM, BMAP	Dinero
		128	Color, Fill, Owner, PFNM, BMAP	Dinero
	64k	16	Color, Fill, Owner, PFNM, BMAP	Dinero
		32	Color, Fill, Owner, PFNM, BMAP	Dinero
		64	Color, Fill, Owner, PFNM, BMAP	Dinero
		128	Color, Fill, Owner, PFNM, BMAP	Dinero
022.Lisp	16k	16	Color, Fill, Owner, PFNM, BMAP	Dinero
		32	Color, Fill, Owner, PFNM, BMAP	Dinero
		64	Color, Fill, Owner, PFNM, BMAP	Dinero
		128	Color, Fill, Owner, PFNM, BMAP	Dinero
	32k	16	Color, Fill, Owner, PFNM, BMAP	Dinero
		32	Color, Fill, Owner, PFNM, BMAP	Dinero
		64	Color, Fill, Owner, PFNM, BMAP	Dinero
		128	Color, Fill, Owner, PFNM, BMAP	Dinero
	64k	16	Color, Fill, Owner, PFNM, BMAP	Dinero
		32	Color, Fill, Owner, PFNM, BMAP	Dinero
		64	Color, Fill, Owner, PFNM, BMAP	Dinero
		128	Color, Fill, Owner, PFNM, BMAP	Dinero

6. Data Analysis

Reviewing Figures 15 and 16, Cycles Per Instruction for 022.Lisp and 026.Compress, we see that in all instances our virtual cache model performed worse than Dinero with either a physical or virtual address trace. As we see from Table 3, the CPI s between our different invalidations schemes do not vary over a wide range and they are significantly worse than those computed from Dinero. The remainder of this section will analyze why our CPI is so high compared with Dinero.

Benchmark	**Owner**	**Fill**	**Pfnm**	**Bmap**	**Din PA**	**Din VA**	**% Diff**
022.Lisp	2.10	2.10	2.10	2.10	1.59	1.61	32.1%
026.Compress	3.92	3.92	3.82	3.82	1.92	1.96	98.8%

Table 3. Average CPI

6.1 Review of Data

We present our data in graph format in Appendix B. The graphs show standard performance measures, such as the Miss Rates (I-Cache, D-Cache, L1 combined, and L2), Transfer Ratios (L1 to L2 and L2 to Memory), and CPI. We also have two graphs illustrative of the main performance bottleneck in our design. Figures 17 and 18 show the percentage of L1 misses caused by invalid lines and the L1 miss rate without L1 invalidation.

Reviewing the L1 miss rates for I- and D- caches (Figures 3,4 for 022.Lisp and 5,6 for 026.Compress), we see that our miss rates are similar to Dinero, except for the 026.Compress I-Cache, which is significantly worse. As mentioned above, 026.Compress represents many more ASIDs than 022.Lisp and we believe this is the primary cause of the significantly worse performance. The average I- and D- cache miss rates are summarized in Table 4. We present these averages for an approximation of how well our invalidation schemes performed against the physical address Dinero trace. We see that apart from the 026.Compress I-Cache, we were between 16% to 29% worse than a pure physical address trace.

Benchmark	Owner	Fill	Pfnm	Bmap	Din PA	Din VA	% Diff
(I) 022.Lisp	0.57%	0.57%	0.57%	0.57%	0.40%	0.39%	41.4%
(I) 026.Compres	0.63%	0.68%	0.61%	0.61%	0.08%	0.08%	656.3%
(D) 022.Lisp	1.44%	1.45%	1.44%	1.42%	1.14%	1.15%	23.9%
(D) 026.Compre	7.71%	7.74%	7.64%	7.45%	5.78%	6.20%	28.9%

Table 4. Average I- and D-Cache Miss Rate

At the L2 level, the statistics are a little misleading. Figures 9 and 10 show the L2 Miss Rates. The increase in miss rate is really a function of a decrease in the number of references, not an increase in the number of misses. In the larger cache sizes, the L2 miss rate come closer to representing the Compulsory miss rate[6]. What is more important are the transfer ratios.

[6] Unfortunately, we did not have sufficient time to analyze the compulsory/capacity/conflict behaviors.

We see in Figures 11 and 12 that the transfer ratios oscillate depending on L1 line size. If one looks at a particular line size series, such as the 64 byte 022.Lisp series on Figure 11, we see that the transfer ratio steadily decreases with an increase in cache size. This is what one would expect, since there are fewer L1 misses.

The L2 to Memory transfer ratios in Figures 13 and 14 show a generally increasing rate. Within a cache size, such as 16k, the increase is expected. As we increase the L1 line size but hold the L2 size constant, there should be an increase since each L2 line holds fewer L1 lines. As we look across cache size series, such as 16k to 32k, we see the transfer ratio is higher for any given line size. As with the L2 Miss Rate, the amount of memory traffic is about constant, but the number of references is decreasing.

What is interesting to look at is the correlation between the L2 invalidation rate and the L2 miss rates. As Table 5 shows, even if we perform no L2 invalidations[7], there is still an enormous difference between our L2 miss rate and the Dinero's L2 miss rate. The cause of the performance problem must lie elsewhere.

Benchmark	Owner	Fill	Pfnm	Bmap	Din PA	Din VA	% Diff
Invalidations	23.27%	22.88%	22.51%	22.93%	3.43%	3.43%	555.6%
No Invals	18.35%	18.13%	18.35%	18.57%	3.43%	3.43%	428.0%
Reduction	4.92%	4.75%	4.16%	4.37%	0.00%	0.00%	

Table 5. Average L2 Miss Rates (026.Compress)

6.2 Inclusion

As shown in Figure 18, if we subtract out the number of L1 misses caused by L2 to L1 invalidation, we have an L1 miss rate comparable to the Dinero miss rate. We estimated this rate by subtracting the minimum of the number of misses caused by invalidation or the number of invalidations from the actual number of misses. It is a crude estimate, but leads us to believe that the main performance issue with our architecture is the multi-level inclusion. As noted above, the I-Cache statistics are particularly bad.

[7] This implies that we take no action on a TLB change. We simply "returned" from the L2 invalidation function and ignored those lines in the trace.

We could substantially improve the miss rate by removing the I-Cache from the inclusion protocol, as is done with real machines. In our architecture, unfortunately, we cannot do that while keeping the TLB at the L2 level.

With this in mind, we would have expected the L1 Ownership or L1 Bmap invalidation schemes to have had a more noticeable effect in reducing the number of L1 misses. The Bmap scheme, in particular, only invalidates exactly those lines in L1 cache that need to be invalidated. With L2 invalidation turned off (ignore TLB changes) we still have a substantial L1 invalidation rate. The only cause for this would be L2 refill, which forces us to invalidate the L1 level to maintain inclusion. The reasons for the number of misses still bears further investigation.

6.3 MIPS Rating

We compute the average MIPS (millions of instructions per second), in Table 5, which shows us approximately the same information as the average CPI shown above. We note that our architecture under performs a physical address cache by 25% to 52%. Given these great disparities, it seems unlikely that one could shorten the processor clock cycle enough by using a virtually indexed / virtually tagged cache to afford this performance penalty.

Benchmark	Owner	Fill	Pfnm	Bmap	Din PA	Din VA	% Diff
022.Lisp	95.84	95.81	95.89	95.93	127.05	125.66	-24.6%
026.Compress	51.34	51.24	52.64	52.73	105.90	103.70	-51.6%

Table 6. Average MIPS

6.4 Clock Cycle

In our introductory section, we stated that it was our intention to shorten the clock cycle of a normal virtually indexed, physically tagged cache by removing the need for a tag check pipeline stage. Using the available data, we would need to make a about a 38% clock cycle reduction to, on average, have a similar performance between our best cache architecture (Bmap) and a physical address cache.

As an example, the R4400 pipeline has 8 stages, two of which are used primarily for tag comparisons. Even if we could redistribute that clock time between the remaining 6 stages, we would only have a 25% cycle time reduction. We would conclude that this sort of architecture would not yield any performance benefits.

7. Conclusions

We have seen that because of L1 inclusion, the performance of our virtual cache architecture is substantially worse than similar sized physical caches. Similar results have been noted by others. [Inouye92], for instance, notes the negative performance effects of cache flushing in virtual caches from an operating system perspective. The [Wang92] paper is similar to our work, except the paper looks at standard virtually indexed/physically tagged L1 caches.

Unlike our results, Wang found almost no difference in hit rates between cache architectures in a low context-switch, zero TLB time configuration. As context switches increased, the virtual cache was worse by 2-6%. We found a substantial difference. From Table 4, we saw that even in a low-context switch environment of 022.Lisp, our cache design was 16-28% worse than the Dinero modeled version.

Even in a uniprocessor environment, where we could ignore cache coherence and inclusion, there would still be a performance penalty to this architecture. Because we lose the physical address of a block in the L1 caches, we must always enforce L1/L2 inclusion for the data cache. Otherwise, we would not be able to write-back data if the L2 line with the PFN tag goes away. Perhaps a write-though scheme would work better.

A. References

[Baer88] Baer, J.L., W.H. Wang, "On The Inclusion Properties for Multi-Level Cache Hierarchies," *Proc. 15th Ann. International Symp. on Computer Architecture*, May 1988, pp. 73-80.

[Baer89] Baer, J.L, W.H. Wang, "Multilevel Cache Hierarchies: Organizations, Protocols, and Performance," *Journal of Parallel and Distributed Computing*, Vol 6, No 3, June 1989, pp. 451-476.

[Cekleov97] Cekleov, Michel, Michel Dubios, "Virtual-Address Caches" (2 part series), *IEEE Micro*, Sept./Oct. 1997, pp. 64-71 and Nov./Dec. 1997, pp. 69-74

[Chase94] Chase, J.S., H.M. Levy, M.J. Feeley, E.D. Lazowska, "Sharing and Protection in a Single-Address-Space Operating System," *ACM Trans. On Computer Systems*, Vol. 12(**4**), Nov. 1994, pp. 271-307.

[Good87] Goodman, James R., "Coherency for Multiprocessor Virtual Address Caches," *Proc. 2nd International Conf. On Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Oct. 1987, pp. 72-81.

[Herrod98] Herrod, S.A., "Using Complete Machine Simulation to Understand Computer System Behaviour," *Doctoral Dissertation*, Feb. 1998.

[Hsu94] Hsu, Y., *Design of the R8000 Microprocessor*, MIPS Technology, 1994 (unpublished paper, submitted to IEEE Micro).

[Inouye92] Inouye, J., R. Konuru, J. Walpole, "The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance," *Operating Systems Review*, Oct. 1992, pp. 14 – 29.

[Kane92] Kane, G., J. Heinrich, *MIPS Risc Architecture*, Prentice Hall, 1992.

[Katz85] Katz, R.H., S.J Eggers, et. al., "Implementing a Cache Consistency Protocol," *The 12th Ann. International Symp. on Computer Architecutre*, June 1985, pp. 276-283.

[Kim97] Kim, D., J. Lee, and S. Park, "A Partitioned On-Chip Virtual Cache for Fast Processors," *Journal of Systems Architecture*, Vol 43(**8**), May 1997, pp. 519-531.

[Kohonen87] Kohonen, T., *Content-Addressable memories*, 2nd ed., Springer-Verlag, Berlin, 1987.

[Kold92] Koldinger, E.J., J.S. Chase, S.J. Eggers, “Architectural Support for Single Address Space Operating Systems,” *Proc. 5th International Conf. On Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Oct. 1992, pp. 175-186.

[Mips94] MIPS Technologies, *R8000 Microprocessor Chip Set Product Overview*, MIPS Tehcnologies, 1994 (unpublished).

[Mips96] MIPS Technologies, *R4400 Microprocessor User's Manual, 2nd ed.*, MIPS Technologies, 1996.

[Mips96a] MIPS Technologies, *MIPS R10000 Microprocessor User’s Manual, Version 2.0*, MIPS Technologies, 1996.

[Rosen97] Rosenblum, M., E. Bugnion, S. Devine, S.A. Herrod, “Using the SimOS Machine Simulator to Study Complex Computer Systems,” *ACM Trans. On Modeling and Computer Simulations*, Jan. 1997, pp. 78-103.

[Smith82] Smith, A.J., “Cache Memories,” *ACM Computing Surveys*, Vol 14(**3**), Sept. 1989, pp. 473-530.

[Wang89] Wang, W.H., J.L. Baer, H.M. Levy, “Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy," *Proc. 16th Ann. International Symp. on Computer Architecture*, 1989, pp. 140-149.

[Wheeler92] Wheeler, Bob, Brian N. Bershad, “Consistency Management for Virtually Indexed Caches,” *Proc. 5th International Conf. On Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Oct. 1992, pp. 124-136.

[Wilson97] Wilson, K.M., K. Olukotun, “Designing High Bandwidth On-Chip Caches,” *Proc. 24th Ann. International Symp. On Computer Architecture*, 1997.

[Witchel96] Witchel, E., M. Rosenblum, ”Embra: Fast and Flexible Machine Simulation,” *Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, 1996.

[Wood90] Wood, D.A. “The Design and Evaluation of In-Cache Address Translation,” *Doctoral Dissertation, U.C. Berkeley*, May 1990.