Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy

Wen-Hann Wang, Jean-Loup Baer and Henry M. Levy

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

Abstract

We propose and analyze a two-level cache organization that provides high memory bandwidth. The first-level cache is accessed directly by virtual addresses. It is small, fast, and, without the burden of address translation, can easily be optimized to match the processor speed. The virtually-addressed cache is backed up by a large physically-addressed cache; this second-level cache provides a high hit ratio and greatly reduces memory traffic. We show how the second-level cache can be easily extended to solve the synonym problem resulting from the use of a virtually-addressed cache at the first level. Moreover, the second-level cache can be used to shield the virtually-addressed first-level cache from irrelevant cache coherence interference. Finally, simulation results show that this organization has a performance advantage over a hierarchy of physically-addressed caches in a multiprocessor environment.

Keywords: Caches, Virtual Memory, Multiprocessors, Memory Hierarchy, Cache Coherence.

1 Introduction

Virtually-addressed caches are becoming commonplace in high-performance multiprocessors due to the need for rapid cache access [11, 3, 17]. A virtually-addressed cache can be accessed more quickly than a physically-addressed cache because it does not require a preceding virtual-to-physical address translation. However, virtually-addressed caches have several problems as well. For example:

1. They must be capable of handling synonyms, that is, multiple virtual addresses that map to the same physical address.
2. While address translation is not required before a virtual cache lookup, address translation is still needed following a miss.
3. In a multiprocessor system, the use of a virtually-addressed cache may complicate cache coherence because bus addresses are physical, therefore a reverse translation may be required.
4. I/O devices use physical addresses as well, also requiring reverse translation.
5. A virtual cache may need to be invalidated on a context switch because virtual addresses are unique to a single process.

None of these problems is insolvable by itself, and several schemes have been proposed for managing virtual caches. For example, dual tag sets, one virtual and one physical, can be used for each cache entry [7, 6]. As another example, the SPUR system restricts the use of address space, prohibits caching of I/O buffers, and requires bus transmission of both virtual and physical addresses [11]. However, these schemes tend to have performance shortcomings or unpleasant implications for system software. Virtually-addressed caches are fundamentally complicated, and this time or space complexity reduces the ability of the cache to match the ever-increasing needs of modern processors.

To attack this problem, we propose a two-level cache organization involving a virtually-addressed first-level cache and a physically-addressed second-level cache (recent studies of two-level uniprocessor and multiprocessor caches can be found in [4, 5, 12, 13]). The small first-level cache can be fast to meet the requirements of high-speed processors; it is virtually addressed to avoid the need for address translation. The large second-level cache will reduce miss ratios and memory traffic; it is physically addressed to simplify the I/O and multiprocessor coherence problems. Furthermore, we show how the second-level cache can be utilized to solve the synonym problem and to shield the first-level cache from irrelevant cache coherence traffic. Overall, we believe that this two-level virtual-real organization simplifies the design of the first-level, where performance is crucial, while solving some of the difficult problems at the second level, where time and space are more easily available.

Our organization involves the use of pointers in the two caches to keep track of the mappings between virtual cache and physical cache entries [7]. We also provide a translation buffer at the second level which operates in parallel with first-level cache lookups in case a miss requires reverse translation. Trace-driven simulations are used to demonstrate the advantages of a two-level V-R (virtual-real) cache over a hierarchy of real-addressed caches in a multiprocessor environment.

The rest of this paper is organized as follows. Section 2 describes the approaches taken in solving various problems related to virtual address caches and presents some design choices for high performance multiprocessor caches. Section 3 gives the specific organization of a V-R two-level cache hierarchy and its detailed operational description. Section 4 presents performance results from simulations, and conclusions are drawn in section 5.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0884-7495/89/0000/0140$01.50

2 Design issues of two-level V-R caches for high performance multiprocessors

This section addresses some important issues in the design of two level V-R caches and motivates our design choices. A more detailed operational description of our approach is given in the following section. The proposed architecture for this evaluation is a shared-bus multiprocessor where each processor has a private, two-level, V-cache–R-cache hierarchy as shown in Figure 1.

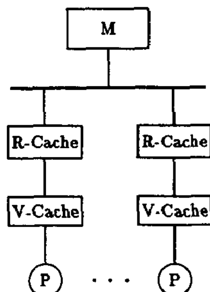


Figure 1: Shared-bus organization

Write policies

For a two-level cache, the write policy can be selected independently at each level. In the literature, write-through has been proposed as the most reasonable write policy for the first-level cache in a two-level hierarchy, while write-back is advocated for the second level [10, 8, 13]. A major motivation for the choice of write-through at the first level is that cache coherence control is simplified. In this case, the first- and second-level caches will always contain identical values.

There are several problems, however, with using a first-level write-through cache. First, assuming no write-allocate, write-through caches will have smaller hit ratios than write-back caches. Second, a write takes longer under write-through because the second-level cache must be updated as well; primary memory may also need to be updated depending on the write policy for the second level.

The reduced write latency with write-through can be greatly hidden by the use of write buffers between the first and second levels, but several write buffers may be needed. Table 1, for example, shows that in the execution of the VAX program *pops* (cf. section 4), 30% of writes are due to procedure calls, each of which typically generates six or more successive writes. Table 2 shows the inter-write interval distribution for a snapshot (411,237 references) of the same trace using a 16K direct-mapped cache with a 16-byte block size. As can be seen, the high percentage of short inter-write intervals confirms the need for several buffers.

Unfortunately, while write buffers can reduce the write latency of the first-level cache, they re-introduce a complexity that write-through was intended to avoid, namely cache coherence. Write buffers can hold modified data for which other processors might encounter a miss. Thus, cache coherency control must be provided for the write buffers on every cache coherence transaction.

These difficulties lead us to favor the write-back policy for our virtually-addressed cache at the first level.

no. of wr. per call	count	total writes
1	3	3
2	2	4
3	0	0
4	2	8
5	2	10
6	4123	24738
7	1266	8862
8	1246	9968
9	2634	23706
10	797	7970
11	539	5929
12	441	5292
13	0	0
14	0	0
15	0	0
16	43	688
no. of wr. due to proc. call		87178
total no. of writes		283057

Table 1: Number of writes due to procedure calls

interval	count
1	4589
2	1015
3	1270
4	786
5	1482
6	687
7	63
8	481
9	735
10 and larger	3245

Table 2: Inter-write intervals (snapshot of 411,237 references)

The synonym problem

As previously noted, a two-level V-R organization can be used to solve the synonym problem. The solution requires the use of a *reverse translation table* [15] for detecting synonyms, and a natural place to put that table is at the second level.

Our two-level organization permits and detects synonyms, but guarantees that at most one copy of a data element exists in the V-cache at any time. Each second-level cache block will have a pointer to its first-level child block, if one exists. If we guarantee an inclusion property, where the R-cache contains a superset of the tags in the V-cache, the reverse translation information can be stored in log(V-cache size/page size) superset bits in each R-cache block. For each entry in the R-cache with a child in the V-cache, these extra bits, together with the page offset, provide the V-cache location of its child.

When a miss occurs in the V-cache, the virtual address is translated (using a second-level translation buffer) and the R-cache is accessed. If an R-cache hit occurs, the R-cache checks whether the data is also in the V-cache under another virtual address (a synonym). If so, it simply invalidates that V-cache copy and moves the data to the new virtual address in the V-cache. Thus, while a data element can have synonyms, it is always stored in the V-cache using the last virtual address with which it was accessed.[1]

[1] Note that our approach in dealing with the synonym problem has some similarities to Goodman's approach [7]. One can view our approach as moving Goodman's real directory from being just for snooping to being associated with the level two cache. This move provides two benefits. First, it hides the cost of Goodman's extra, real directory by making it the level two cache directory. Second, it reduces the misses caused by real-address collisions via making the real directory much bigger.

Context switching

In a multiprogramming environment, addresses are unique to each process and therefore the V-cache must be flushed whenever a context switch occurs. This might be costly for a large virtually-addressed cache. For small caches we believe the penalty on hit ratios will be negligible and this is confirmed by our simulation results (cf. Section 4). However, if a write-back policy is used for the V-cache, a substantial number of write-backs may occur at each context switch, which greatly increases context-switch latency.

Another solution to avoid the address mapping conflict is to attach a process identifier to each tag entry of the V-cache. This approach does not improve the hit ratio for a small V-cache [1], but can avoid the large number of write-backs at context switch time. Unfortunately, this approach increases the complexity of a two-level hierarchy because the V-cache needs to be purged or selectively flushed when a TLB entry of an inactive process is replaced by an entry of the active process, or a process-id is reassigned.

We wish to have the benefits of reduced context-switch latency without needing to flush the V-cache when a TLB entry changes. Our approach meets these goals by invalidating all V-cache blocks on a context switch but *not* writing them back at that time. Instead, each block is written back only when it is replaced, that is, when a new block is read into that cache slot. The writes are thus distributed in time where the latency can be hidden using write-back buffers.

To implement this scheme, we add two new fields to each V-cache block. First, we add a *swapped-valid bit*, which is set for each V-cache block on a context switch. Upon a replacement, if the V-cache finds a block with swapped-valid set, it checks whether that block is also marked both dirty and valid; if so, that block must be written back. Second, we add an *r-pointer*, which is the low-order bits of the page number, to each V-cache block. The r-pointer, together with the page offset, is sufficient to link a V-cache entry to its corresponding location in the R-cache. This linkage makes a write-back or a state check efficient, since there is no need for an address translation. This approach uses space comparable to that of the process identifier scheme, but without its disadvantages.

Table 3 shows the effect of the swapped-valid bit; here we see the inter-write interval from the same benchmark as Table 2 when the swapped-valid bit is used. Because swapped write-backs are typically far apart from other (swapped) write-backs, a single write-back buffer is sufficient to overlap swapped write-backs with processor execution. Our simulations show that with a single buffer the amount of stalling on a swapped write-back is indeed negligible. On the other hand, if the incremental write-back is not used we need to write back over a hundred blocks at context switching time for this specific benchmark. Notice that the number of write-backs needed due to context switching is a function of cache size, cache organization, the duration of the running state of a process, and the workload.

Cache coherence

While two-level caches are attractive, cache coherence control is complicated by a two-level scheme. Without special attention to the coherence problem, the first-level cache will be disturbed by every coherency request on the bus. A solution to this problem is to use the second-level cache as a filter to shield the first-level cache from irrelevant interference. In order to achieve this, we need to impose an inclusion property where the tags of the second-level cache are a superset of the tags of its child cache. We say that a multilevel cache hierarchy has the inclusion property if this superset relation holds. Imposing inclusion is also essential for solving the synonym problem as stated above.

interval	count
1	2
2	3
3	0
4	2
5	5
6	0
7	1
8	2
9	1
10 and larger	119

Table 3: Write interval with write-back and swapped write-back (snapshot of 411,237 references)

In a multiprocessor environment, the inclusion property cannot be held even with a global LRU replacement [4]. In [5] the following replacement algorithm was proposed as one of the conditions to impose the inclusion.

- First level: Any replacement algorithm will do (e.g., LRU). Notify the second level cache of the block being replaced.
- Second level: Replace a block which does not exist in the first level (this is done by checking an inclusion bit; there is one inclusion bit per block to indicate whether the block is present in the first level).

The general problem with inclusion is its implications for a large set size in the second level (i.e., high associativity). By following the same approach as in [5], and letting S_i be the number of sets, B_i be the block size, and $size(i)$ be the cache size of a level i cache, we can show that in order to impose inclusion under the above replacement algorithm, the set-associativity of the second-level cache A_2 must be:

$$A_2 \geq \frac{size(1)}{pagesize} \times \frac{B_2}{B_1}$$

under the usual practical situations where $S_2 > S_1$, $B_2 \geq B_1$, $size(2) > size(1)$ and $B_1 S_1 \geq pagesize$[2].

In practical cases, this constraint can be too strict to be feasible. For example, if the V-cache is 16K bytes, the page size is 4K bytes, and B_2 is 4 times as large as B_1, even with a direct-mapped V-cache we need a 16-way R-cache to achieve the inclusion.

To relax the strict constraint on the set-associativity of the R-cache, we change the replacement rule of the R-cache to operate as follows: *replace a block with the inclusion bit clear if there is one; otherwise replace a block according to some predefined replacement algorithm and invalidate the corresponding V-cache block.* Note that the latter won't happen very often since the R-cache is much larger than the V-cache. For example, the analysis of the multiprocessor trace, pops (over 3 million memory references), shows that only 21 inclusion invalidations are needed if the V cache is 16K bytes, 2-way set-associative with a 16 byte block size and the R cache is 256K bytes with same set size and block size.

[2] if $B_1 S_1 < pagesize$ the results of [5] apply.

3 Organization of a V-R two-level cache

A simplified organizational block diagram of a V-R two-level cache is given in Figure 2. The V-cache is accessed via virtual addresses, which are also forwarded to the TLB at the second level so that address translation can proceed concurrently with the access to the V-cache. This translation and the access to the R-cache are aborted if there is a valid hit in the V-cache. A number of tag and control bits that we call tag entry are associated with data blocks in both caches as shown in Figure 3. Each tag entry in the V-cache contains a tag, an r-pointer, a dirty bit, a valid bit and a swapped-valid bit. The r-pointer contains the lower log(R-cache-size/page-size) bits of the real address page number. Together with the page offset, it can be used to address the related entry in the R-cache. The swapped-valid bit is used to indicate whether the entry belongs to a swapped process. This is needed in order to avoid a large context switch overhead, as previously described.

Each tag entry in the R-cache tag store contains a tag and a number of subentries, one subentry per V-cache block since we allow larger block sizes in the R-cache. A subentry contains an inclusion bit that indicates whether a copy of the data is in the V-cache or not, a buffer bit that indicates if a copy of the data is in a write buffer of the V-cache, a few state bits for sharing status and cache coherence control (with other R-caches), two dirty bits, one for V-cache dirty and for R-cache dirty, and a v-pointer which contains the lower log(V-cache-size/page-size) bits of the virtual page number. Together with the page offset, the v-pointer can be used to address the entry in the V-cache.

In order to properly provide the data and manage cache coherence and synonyms, we list in Table 4 the communication buses between the V-cache and the R-cache. The following is a detailed operational description of a two-level V-R cache.

For simplicity, let us assume that an invalidation protocol is used at the R-cache level although our scheme will also work for other protocols as well. An invalidation protocol invalidates all other cache copies before updating shared data in the local cache. Write-backs to memory are performed when a dirty block moves from one cache hierarchy to another. A number of existing protocols belong to this category [16].

V-R hierarchy algorithm

1. Read hit in V-cache. Give the data to the processor. The hit signal is sent to the R-cache to abort the R-cache and TLB accesses.

2. Read miss in V-cache. Raise the replacement signal if a V-cache block needs to be replaced to give room to the incoming new data. Give the R-cache both the v-pointer, which is the V-cache location for the new data, and the r-pointer, which is the R-cache location of the block being replaced. If the replaced block in V-cache is clean, the

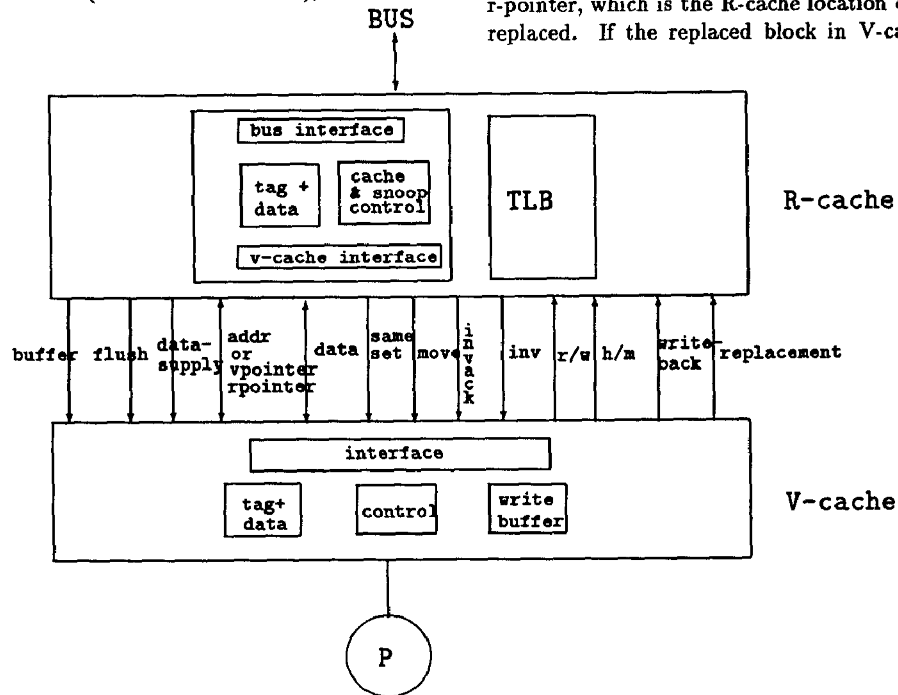


Figure 2: V-R cache organization

V-cache tag entry

tag	rpointer	dirty	v	sv
20	6	1	1	1

R-cache tag entry

tag	I	B	state	vpointer	H1	H2	I	B	state	vpointer	H1	H2
16	1	1	2	2	1	1	1	1	2	2	1	1

Figure 3: Contents of tag stores (assume page size is 4K, C_1 is 16K and C_2 is 256K, $B_2 = 2 \times B_1$.)

From V to R:	
read/write	tells the R-cache whether the current request is a read or a write.
replacement	tells the R-cache that a V-cache block needs to be replaced.
hit/miss(v-pointer, r-pointer)	tells the R-cache whether the current access results in a hit or a miss in the V-cache. If it is a miss, the target v-pointer gives the V-cache slot for the new data and the r-pointer gives the R-cache entry where the inclusion bit is to be erased if the block to be replaced is clean, or where the buffer bit is to be set if the block is dirty. If it is a hit the R-cache access is aborted.
write back(r-pointer)	tells the R-cache that the data in the write buffer will be written back to the place pointed to by the r-pointer.
From R to V:	
sameset(v-pointer)	tells the V-cache that there is a synonym copy in the same set; no need to write back; and the data is available under v-pointer.
move(v-pointer)	tells the V-cache that there is a synonym copy in a different set; the data is available under v-pointer.
data supply(r-pointer)	tells the V-cache that the data is ready to be loaded and gives its location in the R-cache to be stored as part of the tag entry.
invalidation(v-pointer)	tells the V-cache to invalidate the data under v-pointer.
flush(v-pointer)	tells the V-cache to flush the data under v-pointer.
invalidation(buffer)	tells the V-cache to invalidate the data in the buffer.
flush(buffer)	tells the V-cache to flush the data in the buffer.
invack	tells the V-cache that the coherency has been cleared and that it can update the data.

Table 4: V-R interface

R-cache resets the inclusion bit. If the block is dirty, the V-cache copies the block into the write buffer and the R-cache sets the buffer bit to indicate that the block is still in the write buffer of the V-cache. This bit gets reset when the write-back occurs or when the write-back is canceled (see below).

(a) Hit in R-cache.

i. The data is in the V-cache under another virtual address. The R-cache tests whether the two locations are in the same set. If so, a *sameset* signal is sent to the V-cache so that the write-back can be canceled if the replaced block is dirty; the R-cache will reset the buffer bit if the replaced block is dirty, or it will set the inclusion bit if the replaced block is clean[3]. If the blocks are in different sets, the R-cache sends a move(v-pointer) to the V-cache so that the data can be stored at the new location. Valid bits are set to valid. The v-pointer tag entry of the R-cache is modified accordingly. Notice that in both cases the v-tag is updated to reflect the new virtual address.

ii. No other copy in V-cache. R-cache raises the data supply signal and sends the block to the V-cache. The R-cache also supplies the r-pointer to the V-cache to set up the link information. R-cache sets the inclusion bit and the v-pointer and the V-cache stores the r-pointer, sets the valid bits, and resets the dirty bit.

(b) Miss in R-cache. Proceeds as described in the cache coherence subsection. Gets a clean copy and then back to (a)ii.

3. Write hit on clean block in V-cache. Wait till the R-cache raises the *invack* signal (cf. the cache coherence subsection); then update the data and set the dirty bit in the V-cache.

4. Write miss in V-cache. The replacement proceeds as in the case of a read miss.

[3]the inclusion bit was reset earlier to reflect the replacement.

(a) Hit in R-cache. Resolve the cache coherency (cf. below); resolve the synonyms as in the case of a read miss; load the block into V-cache; update the data and sets the dirty bit in the V-cache.

(b) Miss in R-cache. Proceed as described in the cache coherence subsection; get a clean copy, load the block into the V-cache and the R-cache and set appropriate pointers and inclusion as in the case for a read; update the data, and set the dirty bit in the V-cache.

It is worth noticing that the cost of handling a synonym is approximately the same as a first-level miss and second-level hit. This observation will be used in our performance evaluations.

Cache coherence

Processor induced:

1. Read miss in the V-cache and in the R-cache. Initiate a read-miss bus transaction and get the block. Set the state of the block as shared if another cache acknowledges having this block; otherwise set the state as private.

2. Write hit on a clean block in the V-cache. Check the state in the R-cache. If private, raises the *invack* to let the V-cache proceed; sets the vdirty bit in the R-cache. Otherwise, the R-cache initiates an invalidation bus transaction and when it is completed, raise the *invack* signal and set the vdirty bit in the R-cache.

3. Write miss in the V-cache.

   (a) Hit in the R-cache. Check the state in the R-cache. If shared, initiate an invalidation bus transaction. Supply the block to the V-cache when the transaction is completed and set the vdirty bit in the R-cache.

   (b) Miss in the R-cache. Initiate a read-modified-write bus transaction; get the block; reset the rdirty bit in the R-cache and set the vdirty bit in the R-cache.

Bus induced:

1. Read-miss. Acknowledge the sharing status if in possession of the requested block and:

(a) If the block is modified in the V-cache, the R-cache sends a flush(v-pointer) to V-cache and gets the block, updates itself, changes its state to shared, resets the vdirty bit, resets the rdirty bit, supplies the block to the requesting cache and updates the memory.

(b) If the block is modified in the write buffer of V-cache, R-cache sends a flush(buffer) to V-cache and gets the block, updates itself, changes its state to shared, resets vdirty bit, resets the rdirty bit, resets the inclusion bit, supplies the block to the requesting cache and updates the memory.

(c) If the block is dirty in the R-cache, the R-cache supplies the block to the requesting cache, updates the memory, changes its state to shared, and resets its dirty bit.

(d) Otherwise, memory supplies the block.

2. Invalidation. The R-cache invalidates its own entry if present and checks the inclusion bit. If it is set, the corresponding entry in the V-cache is invalidated. This is done by issuing invalidate(v-pointer) to the V-cache.

3. Read-modified-write. Treated as a read-miss followed by an invalidation.

Replacement

(a) V-cache: Any replacement algorithm will do (e.g., LRU).

(b) R-cache: Replace a block with all inclusion bits (i.e., for each subentry) reset. If there is none (this might happen if we follow the strategy of the end of section 2), randomly choose one block and invalidate the copy (or copies if $B_2 > B_1$) in the V-cache.

4 Performance

In this section, we compare the relative performance of virtual-real (V-R) and real-real (R-R) two-level caches. We also examine the merits of splitting the first-level virtually-addressed cache into I and D caches. Finally, we measure the effect of the R-cache in shielding the V-cache from irrelevant cache coherence interference.

To gather the performance figures, we use trace-driven simulations and three parallel program traces: pops, thor and abaqus [2, 14]. In pops and thor, context switches occur rarely while they are frequent in abaqus. Table 5 gives a summary of some characteristics of these traces.

Relative performance of V-R and R-R two-level caches

To compare the performance of V-R and R-R two-level caches, we gather the hit ratios at different levels; the hit ratios are then used in generic memory access time equations to predict relative performances. We assume that the inclusion property defined previously also holds for the R-R two-level cache. For simplicity, we consider only direct-mapped caches at both levels.

The generic access time equation of a two-level cache hierarchy is as follows:

T_{acc} = Prob(hit at level 1) × access time at level 1
+ Prob(hit at level 2 & miss at level 1) × access time at level 2
+ Prob(miss at level 1 and 2) × memory access time

that is:

$$T_{acc} = h_1 t_1 + (1 - h_1) h_2 t_2 + (1 - h_1 - (1 - h_1) h_2) t_m$$

where h_1, h_2 are hit ratios at levels 1 and 2, t_1 and t_2 are access times at the two levels, and t_m is the memory access time including the bus overhead.

Because the second-level caches are the same for both V-R and R-R organizations, and because inclusion holds, the number of misses and the traffic from the second-level cache are the same in both organizations. Therefore the third term in the above equation is the same for both V-R and R-R organizations. Assuming that handling a synonym has a cost equivalent of handling a miss in the first-level cache that hits in the second-level cache, the relative performance where there is a hit in the hierarchy can be estimated solely on the first two terms of the above equation.

Table 6 shows the hit ratios at both levels of V-R and R-R organizations for the three traces under three different pairs of first and second-level cache sizes. Figures 4, 5 and 6 depict the relative performance of the two organizations under different degrees of assumed R-cache degradation due to address translation overhead. These figures plot the relative performance of the two hierarchies with $t_2 = 4t_1$ vs. the percentage of slow down due to address translation for various first-level/second-level cache sizes. The points on the y-axis correspond to no slow down at all. From these figures we can draw the following conclusions.

Let us assume that there is no time penalty involved in performing a virtual-real address translation in conjunction with the access to the first level cache. When context switches occur rarely, as is the case for the first two traces (Figures 4 and 5), the performances of the V-R and R-R hierarchies are almost indistinguishable (the points on the y-axis are the same). When context switches are frequent, as in the third trace (Figure 6), the V-R hierarchy is slower by 2 to 6% depending on the size of the V-cache (a larger V-cache seems to imply a larger relative degradation).

Now, let us assume a time penalty for the translation. There are two possible reasons for this penalty. The first is that TLB access and cache access cannot be completely overlapped as soon as the cache size is larger than the page size multiplied by the set associativity. Second, even if there were total overlap, there would still be an extra comparison necessary to check the validity of a cache hit. From the observations of the previous paragraph, it is clear that the V-R hierarchy will perform better in the case of rare context-switches. The relative improvement is approximately equal to the overhead of address translation. What is interesting is to see the cross-over point for the case of frequent context-switches. From Figure 6, we see that the V-R hierarchy will have a better performance when the address translation slows down the first level R-cache access by 6% or more.

Since 6% is a conservative figure for the penalty due to the insertion of a TLB at the first level, it appears that the V-R hierarchy is a better solution. Its performance is as good as that of an R-R hierarchy and its cost is less since the TLB does not have to be

trace	num. of cpus	total refs	instr count	data read	data write	context switch count
thor	4	3283k	1517k	1390k	376k	21
pops	4	3286k	1718k	1285k	283k	7
abaqus	2	1196k	514k	600k	82k	292

Table 5: Characteristics of traces

trace	thor			pops			abaqus		
sizes	4K/64K	8K/128K	16K/256K	4K/64K	8K/128K	16K/256K	4K/64K	8K/128K	16K/256K
h1VR	.925	.957	.968	.928	.943	.954	.852	.873	.888
h1RR	.925	.958	.969	.928	.943	.954	.857	.889	.908
h2VR	.692	.531	.463	.609	.608	.567	.551	.559	.585
h2RR	.691	.526	.449	.608	.608	.563	.536	.493	.498

Table 6: hit ratios

trace	thor			pops			abaqus		
sizes	.5K/64K	1K/128K	2K/256K	.5K/64K	1K/128K	2K/256K	.5K/64K	1K/128K	2K/256K
h1VR	.755	.828	.872	.727	.882	.909	.766	.793	.822
h1RR	.755	.828	.872	.727	.882	.909	.767	.797	.827
h2VR	.905	.883	.867	.897	.810	.781	.716	.728	.739
h2RR	.905	.883	.867	.897	.810	.781	.715	.723	.732

Table 7: Hit ratios for small first-level caches

Figure 4: Average access time vs. slow-down of R-cache (thor)

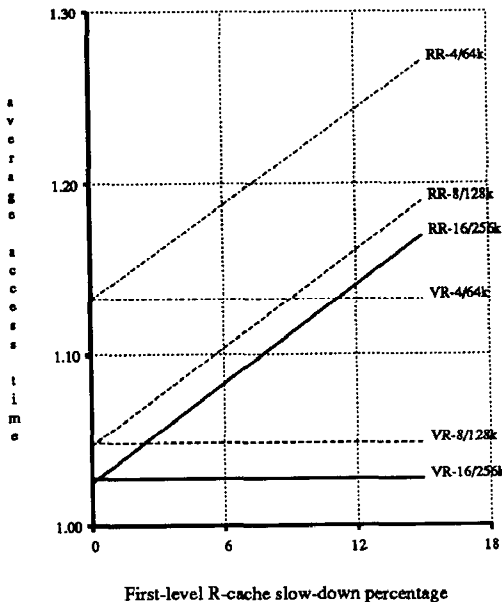


Figure 5: Average access time vs. slow-down of R-cache (pops)

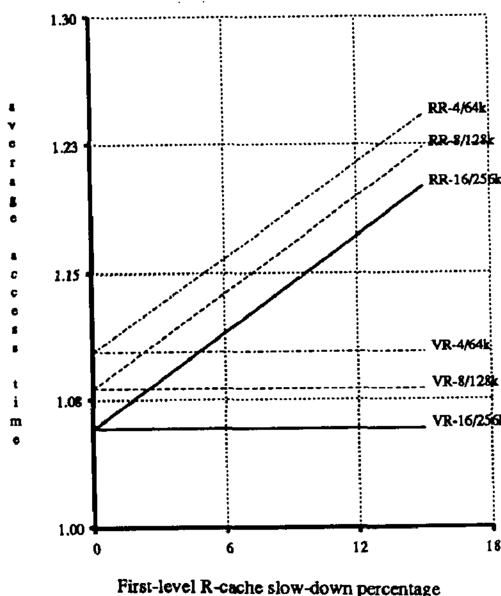
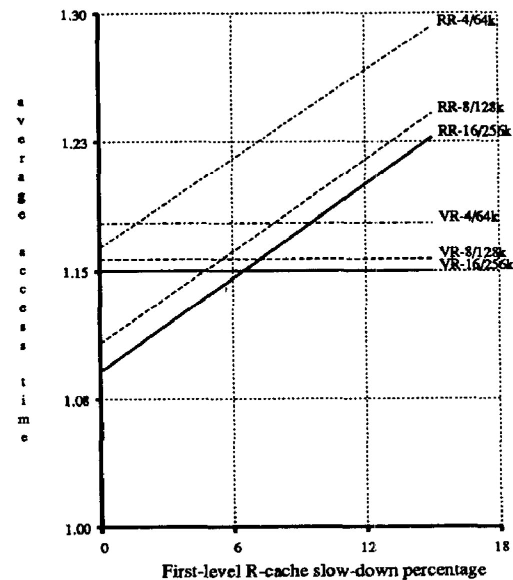


Figure 6: Average access time vs. slow-down of R-cache (abaqus)



implemented in fast logic. Another advantage is that problems such as TLB coherence can also be handled at the second level.

The results presented above assumed 4K to 16K first-level caches, which may be impractical for some advanced technologies, such as GaAs. However, we believe that the V-R organization is even more attractive for hierarchies with smaller first-level caches. Our results in Table 7 show that for smaller first-level caches (e.g., .5K to 2K), the first-level hit ratios of V-R and R-R organizations are nearly identical. Therefore, performance of a V-R hierarchy will be superior given any penalty for a TLB lookup. In addition, for technologies in which space is at a premium, we can trade the first-level TLB of an R-R hierarchy for a larger first-level cache in a V-R hierarchy. This in turn provides larger hit ratios and hence smaller average access time.

Splitting the first-level virtually-addressed cache

There are a number of reasons why it is advantageous to split the first-level cache into separate I and D caches. First, the bandwidth can almost be doubled for pipelined processors where an instruction fetch can occur at the same time as a data fetch of a previous instruction (e.g., the IBM801 and Motorola 88000). Second, each I and D cache is smaller and has the potential to be optimized for its speed. Third, and this pertains mostly to V-caches, the I cache is simpler than the D cache since it does

not need to handle the synonym and the cache coherence problems provided that self-modifying programs are not permitted. A disadvantage, however, is that we need more wirings or pins for the processor and cache module. It is important to assess, however, if splitting the cache into I & D components will improve performance.

Our results in Table 8, 9 and 10 show that the hit ratios of split I&D caches are very close to that of a unified I&D cache and are not necessarily worse. In these tables, the I and D separate caches are of equal sizes (i.e., in the 4K example the I-cache and the D-cache are each 2K). Similar results have been found in [9, 13]. Thus, we would advocate such a split for a V-R hierarchy.

thor	4K/64K	8K/128K	16K/256K
data read split	0.924	0.937	0.945
unified	0.913	0.938	0.950
data write split	0.952	0.962	0.969
unified	0.946	0.966	0.972
instruction split	0.957	0.963	0.989
unified	0.930	0.973	0.984
overall split	0.942	0.952	0.968
unified	0.925	0.957	0.968

Table 8: Hit ratios of level 1 caches for the thor trace

pops	4K/64K	8K/128K	16K/256K
dataread split	0.902	0.912	0.923
unified	0.900	0.915	0.926
data write split	0.936	0.946	0.955
unified	0.937	0.948	0.958
instruction split	0.947	0.966	0.978
unified	0.948	0.963	0.974
overall split	0.928	0.944	0.955
unified	0.928	0.943	0.954

Table 9: Hit ratios of level 1 caches for the pops trace

abaqus	4K/64K	8K/128K	16K/256K
data read split	0.795	0.818	0.837
unified	0.806	0.829	0.845
data write split	0.841	0.861	0.875
unified	0.847	0.857	0.895
instruction split	0.920	0.947	0.949
unified	0.907	0.926	0.938
overall split	0.852	0.876	0.888
unified	0.852	0.873	0.888

Table 10: Hit ratios of level 1 caches for the abaqus trace

Shielding cache coherence interference

An important advantage of the two-level approach is that the R-cache can shield the V-cache from irrelevant cache coherence interference. For example, on a read miss bus request, the R-cache needs to send a flush request to its V-cache only when the V-cache contains a modified copy of the data; otherwise the V-cache will not be disrupted. Note that this shielding effect is achieved because the inclusion property holds in our V-R two-level cache. Imposing inclusion might not seem to be essential for an R-R two-level hierarchy because the synonym problem is not present. However, the results in Tables 11, 12 and 13, which give the number of coherence messages being percolated to each first-level cache, show that a V-R two-level cache has much less coherence interference at the first level than that of an R-R two-level cache without inclusion. The results also show that inclusion is important in an R-R two-level cache since it results in approximately the same savings in coherence messages to the first-level cache.[4]

We believe that the shielding effect on cache coherence will be more prominent as the number of processors increases. This is due to the fact that more bus coherence requests will be generated from a larger number of processors, and without the shielding, a first-level cache will be disrupted more often. Our results in Tables 11, 12 (4 cpus) and 13 (2 cpus) reflect this effect. For example, on the average, the first-level cache of a V-R hierarchy encounters about half the coherence messages than that of the R-R hierarchy without inclusion for the two processor trace (cf. Table 13), whereas for four processor traces the first-level cache of the V-R hierarchy encounters from three to six times fewer coherence messages. We plan to further confirm this observation when we are in possession of larger-scale traces.

5 Conclusions

One of the most challenging issues in computer design is the support of high memory bandwidth. In this paper, we have proposed

[4] We notice that RR with inclusion has over 10% fewer coherence messages than that of VR for the abaqus trace. This discrepancy is due to a large amount of inclusion invalidations incurred in this specific trace due to a large number of context switchings.

pops	4K/64K			8K/128K			16K/256K		
cpu	VR	RR(incl)	RR(no incl)	VR	RR(incl)	RR(no incl)	VR	RR(incl)	RR(no incl)
0	6717	7113	23804	7237	7707	20783	8309	8854	19468
1	10015	10351	30523	10606	11027	26128	11771	12357	24258
2	9518	9861	30063	10143	11027	26407	11344	11906	24817
3	9368	9963	31311	10001	10650	27528	11144	12061	25932

Table 11: Number of coherence messages to the first-level cache

thor	4K/64K			8K/128K			16K/256K		
cpu	VR	RR(incl)	RR(no incl)	VR	RR(incl)	RR(no incl)	VR	RR(incl)	RR(no incl)
0	3755	3743	23005	4342	4317	21773	5785	4473	18123
1	4144	4139	27056	4727	4722	23538	5473	5170	18304
2	4229	4229	27005	4810	4820	23915	5561	5229	18776
3	4135	4129	25210	4699	4692	21593	6797	5103	16231

Table 12: Number of coherence messages to the first-level cache

abaqus	4K/64K			8K/128K			16K/256K		
cpu	VR	RR(incl)	RR(no incl)	VR	RR(incl)	RR(no incl)	VR	RR(incl)	RR(no incl)
0	10961	8436	18855	11677	9379	21295	11067	9853	22603
1	10527	8029	20726	10547	9528	24202	10599	10028	26845

Table 13: Number of coherence messages to the first-level cache

a two-level cache hierarchy to address this issue. We have argued that the first level cache is best accessed directly by virtual addresses. We back up the small virtually-addressed cache by a large second-level cache. A virtually-addressed first-level cache does not require address translation and can be optimized to match the processor speed. Through the use of a swapped-valid bit, we avoid the clustering of write-backs at context switching time. The distribution of these write-backs is more evenly spread over time. The large second-level cache provides a high hit ratio and reduces a large amount of memory traffic. We have shown how the second-level cache can be easily extended to solve the synonym problem resulting from the use of a virtually-addressed cache at the first level. Furthermore, the second-level cache can be used effectively to shield the virtually-addressed first-level cache from irrelevant cache coherence interference.

Our simulation results show that when context switches are rare, the virtually-addressed cache option has comparable performance to its physically-addressed counterpart, even assuming no address translation overhead. When context switches occur frequently, the virtually-addressed cache option has a performance edge when a small address translation penalty is taken into account, and the smaller the virtually-addressed cache the larger the relative performance edge. We also advocate splitting the virtually-addressed cache into separated instruction and data caches. This approach has the potential of doubling the memory bandwidth since our results show that the hit ratios of split instruction and data caches are very close to that of a single I&D cache.

As a final remark, we note that cache performance is workload dependent. In this study we have confined ourselves to a limited VAX multiprocessor workload. We plan to enlarge our workload sample as soon as we are in possession of other multiprocessor traces.

Acknowledgment
This work was supported in part by National Science Foundation (Grants No. CCR-8702915 and CCR-8619663), Boeing Computer Services, Digital Equipment Corporation (the System Research Center and the External Research Program) and a GTE fellowship. The experimental part of this study could not have been possible without Dick Sites who made the traces available to us and Anant Agarwal who allowed us to share his postprocessing programs and who patiently answered our many questions. We also thank the members of the "Computer Architecture lunch", especially Tom Anderson, Jon Bertoni, Sanglyul Min and John Zahorjan for their excellent comments and suggestions.

References

[1] Agarwal, A., R. L. Sites and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proc. 13th Symposium on Computer Architecture*, pages 119–127, 1986.

[2] Agarwal, A., R. Simoni, J. Hennessy and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. 15th Symposium on Computer Architecture*, pages 280–289, 1988.

[3] Atkinson, R. R. and E. M. McCreight. The dragon processor. In *Proc. Architectural Support for Programming Languages and Operating Systems(ASPLOS-II)*, pages 65–69, 1987.

[4] Baer, J.-L. and W.-H. Wang. Architectural choices for multi-level cache hierarchies. In *Proc. 16th International Conference on Parallel Processing*, pages 258–261, 1987.

[5] Baer, J.-L. and W.-H. Wang. On the inclusion property for multi-level cache hierarchies. In *Proc. 15th Symposium on Computer Architecture*, pages 73–80, 1988.

[6] Cheriton, D.R., G. Slavenburg and P. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proc. 13th Symposium on Computer Architecture*, pages 367–374, 1986.

[7] Goodman, J. Coherency for multiprocessor virtual address caches. In *Proc. Architectural Support for Programming Languages and Operating Systems(ASPLOS-II)*, pages 72–81, 1987.

[8] Goodman, J. and P.J. Woest. The Wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proc. 15th Symposium on Computer Architecture*, pages 422–431, 1988.

[9] Haikala, I.J. and P.H. Kutvonen. Split cache organizations. In *Proc. Performance '84*, pages 459–472, 1984.

[10] Hattori,A., Koshino,M. and S.Kamimoto. Three-level hierarchical storage system for FACOM M-380/382. In *Proc. Information Processing IFIP*, pages 693–697, 1983.

[11] Hill,M. et al. Design decisions in SPUR. *Computer*, 19(11):8–22, November 1986.

[12] Przybylski, Steven A. *Performance-Directed Memory Hierarchy Design*. Ph.D Dissertation, Stanford University, 1988.

[13] Short R.T. and H.M. Levy. A simulation study of two-level caches. In *Proc. 15th Symposium on Computer Architecture*, pages 81–88, 1988.

[14] Sites, R.L. and A. Agarwal. Multiprocessor cache analysis using ATUM. In *Proc. 15th Symposium on Computer Architecture*, pages 186–195, 1988.

[15] Smith,A.J. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[16] Sweazey, P. and A.J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proc. 13th Symposium on Computer Architecture*, pages 414–423, 1986.

[17] Cheng, Ray. Virtual address cache in UNIX. In *Proc. USENIX Conference*, pages 217–224, June 1987.