Recency-Based TLB Preloading

Ashley Saulsbury
Sun Microsystems Laboratories
901 San Antonio Road,
Palo Alto, CA, USA
ashley.saulsbury@sun.com

Fredrik Dahlgren
Ericsson Mobile Communications AB
Mobile Phones and Terminals
SE-221 83, Lund, Sweden
fredrik.dahlgren@ecs.ericsson.se

Per Stenström
Dept. of Computer Engineering
Chalmers Univ. of Technology
SE-412 96 Gothenburg, Sweden
pers@ce.chalmers.se

ABSTRACT

Caching and other latency tolerating techniques have been quite successful in maintaining high memory system performance for general purpose processors. However, TLB misses have become a serious bottleneck as working sets are growing beyond the capacity of TLBs.

This work presents one of the first attempts to hide TLB miss latency by using preloading techniques. We present results for traditional next-page TLB miss preloading - an approach shown to cut some of the misses. However, a key contribution of this work is a novel TLB miss prediction algorithm based on the concept of "recency", and we show that it can predict over 55% of the TLB misses for the five commercial applications considered.

1 INTRODUCTION

Contemporary processing systems supporting paged virtual memory employ an MMU to validate memory references and translate them to physical addresses. To efficiently perform this task, the MMU caches translations in its TLB. There are several options for the placement of the TLB [16], however, most systems today favor placing it in parallel with the first-level cache to avoid the synonym problem [2,20,10]. While previously considered acceptable [5,7], trends toward greater instruction level parallelism and higher clock frequencies have made TLB performance critical [1], especially when the TLB is software loaded [11].

To meet the demands of larger working sets, one can use the same methods for TLBs as for traditional cache hierarchies. Simply increasing the TLB capacity unfortunately leads to higher TLB access time which may directly affect the overall performance. This can be mitigated by changing organization to say a set associative design. Naturally, attempts toward multi-level TLB hierarchies have also been made. Micro-TLBs typically containing one or two translation entries, filter memory accesses from the main TLB, which can now afford a slightly longer latency, and hence larger capacity, to service those accesses which miss the micro-TLBs. The logical extension is a two-level TLB hierarchy, (for example the AMD K-7 [24], or HAL SPARC64-III [25]), to provide even greater capacity. Increasing TLB span, by using larger page sizes also helps if supported by the operating system. These methods for increasing TLB capacity can work well for applications with high spatial and temporal locality, but are only a partial solution to poor TLB performance.

In this paper, we take an orthogonal approach of hiding TLB miss latency by predicting and *preloading* (or *prefetching*) translations through hardware mechanisms. There has been much research in the area of instruction and data cache prefetching, but to the best of our knowledge little has been related to TLBs. The only work we are aware of particularly addresses certain cold start effects of TLB misses[13,15]. By contrast, the prefetch technique we propose and evaluate aims at addressing capacity-related TLB misses. We consider two prefetching algorithms; a conventional linear predictor, (as suggested for data cache prefetching [8]), and a novel technique that exploits history information about the relative recent usage of pages to predict future TLB misses.

Our first observation in Section 2 is that the TLB is already a performance bottleneck in some commercial codes by showing the impact of TLB misses on performance. Fortunately, we have observed that TLB misses exhibit a fairly predictable behavior, and the model we use to detect this behavior, which is based on stack algorithms [14], is presented in Section 3. Our main contribution to exploit this behavior is a novel prefetching scheme presented in Section 4. Section 5 describes the simulation methods used to test our algorithm, giving the results and analysis in Section 6. In Section 7 we cast our work in a broader context. After giving our conclusions in Section 8 we consider options for future work.

2 TLB THRASHING IN REAL CODES

In this section we first demonstrate through a simple example how TLB thrashing arises and then show its impact on the execution times in some commercial applications.

2.1 A TLB thrashing example

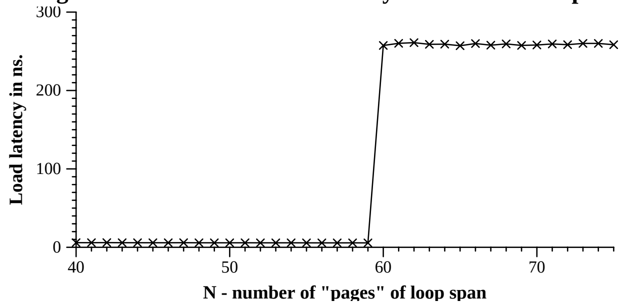
Consider the example code below which sums a column in a matrix of bytes. It is well known that by making the matrix row width a non-power of two (e.g. 8192 + 32 as in the example) cache thrashing effects (due to limited associativity) can be avoided.

```
for (i=0; i<N; i++) {
    sum += data[ i * (8192+32) ];
}
```

Indeed the entire loop may be iterated several times for quite large values of N without risk of thrashing even the first level data cache. (In this case with a 16KB data cache, N may be as large as 512). Clearly such a code sequence is a classic TLB killer. Figure 1 below illustrates the effect of running this code on a 200 MHz UltraSPARC-II system for increasing values of N.

The UltraSPARC-II has a 64-entry fully-associative TLB, with a pseudo-LRU replacement policy, and Solaris permanently locks down 4 TLB entries for mapping the kernel nucleus. In Figure 1 we can clearly see the effect (at 60 pages) when the TLB starts to thrash. The impact in this case is over a factor of 50 in effective memory latency, and is only this low because rapid repetitive use quickly brings the software TLB reload handler and all the required Translation Table Entries (TTEs) into the system caches.

Figure 1 Per-iteration load latency in TLB killer loop.



2.2 TLB thrashing in commercial codes

If we turn our attention to real application codes we can see that TLB thrashing is not restricted to contrived benchmarks. Figure 2 shows the measured TLB miss rate (on the same machine) for a highly used commercial Computational Fluid Dynamics code from a company called Fluent, described in Section 3.2. This shows an average of around 500,000 TLB misses per second during the whole of the execution. Each miss takes a minimum of 50 machine cycles (as measured above), although often significantly more as the reload handler itself incurs cache misses. Thus in any second, our 200 MHz machine is spending a minimum of 50*500,000 / 200,000,000 = 12.5% of its cycles simply executing the TLB reload handler. Actual measurements reveal this closer to 20% because of cache misses - precise values vary a couple of percent according to L2 cache size, and OS version.

Another application which exhibits very similar behavior is the Vortex benchmark from the SPEC '95 suite. Vortex's TLB miss rate is shown in Figure 3. Again, this application spends some 15% of its execution time in the TLB reload handler. Yet, both Fluent UNS and Vortex are relatively small applications.

This brings us to two important points: First, software-reloaded TLBs should be avoided. Our observations conform with those of previous researchers [18,19,11,17] concerning the performance overhead of software-loaded TLBs. Second, hardware-reloaded TLBs alone are not a remedy for the problem of larger problem/working-set sizes leading to increased TLB miss ratios. Our approach is to extend hardware reloaded TLBs with a prefetching[1] scheme based on a novel heuristic. However, before describing the prefetching technique, we provide, in the next section, a background into the kind of history information our hardware heuristic algorithms exploit.

3 TLB Entry Re-use in Codes

In this section we use the LRU stack algorithm [14], briefly introduced in Section 3.1, to model capacity-related TLB misses. Based on a case study containing five commercial applications, described in Section 3.2, we find in Section 3.3 that there is a high likelihood of TLB entries to be re-used at certain temporal distances. We use this observation to discuss the potential for a recency-based preloading scheme in Section 3.4.

[1]We use the terms prefetching and preloading interchangeably throughout this paper.

3.1 TLB miss prediction using an LRU stack

An LRU stack algorithm [14] can be used to predict the miss rate of a fully-associative TLB with an LRU replacement policy. To see how, recall that an LRU stack maintains a list of address references ordered according to how recently they were accessed with the most recent at the top. Suppose that the LRU stack is infinitely large and that we want to model an N-entry fully-associative TLB having LRU replacement. We examine each address reference in the order presented to the TLB. If the address has been used before, it resides at level R in the stack. We call this the *recency* of the reference. If $R<N$ the reference results in a TLB hit. This entry is removed and pushed onto the top of the stack. If there is no match, the address reference is pushed onto the top of the stack and the entry at level N becomes the victim for replacement.

Figure 2 TLB miss rate for Fluent UNS

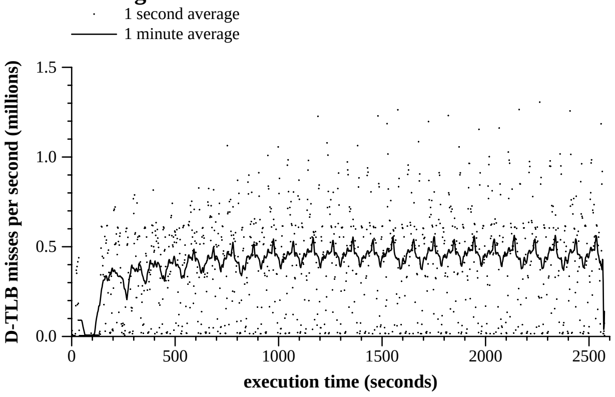
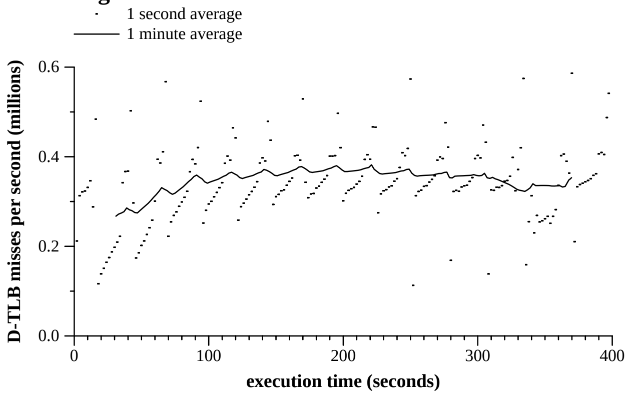


Figure 3 TLB miss rate for SPEC '95's Vortex



Using the stack algorithm we can determine the miss rates of all possible TLB sizes in one pass. To do this we build a histogram table to count the number of times an address is found at a particular recency, i.e. depth, in the stack. From this table we can deduce that entry 0 in the histogram table indicates the number of times an address was found with recency 0, i.e. at the top of the stack, and so would hit in a single entry TLB. In general, we see that by summing elements 0 through $N-1$ of the histogram table, we can determine the number of hits in a TLB of N entries. We now apply this methodology to analyze the TLB miss rates for a suite of applications.

3.2 Applications

This study investigates the benefit of hardware prefetching support in cases where conventional TLB performance is poor. Therefore, for this study, we have attempted to choose workloads of significant commercial interest that have TLB performance problems. This choice was not trivial; virtually all of the Spec '95 suite, for example, has benign TLB behavior. In other cases license agreements limit or prohibit publication of results.

The following programs were selected from a broad range of application areas solely because they exhibit bad TLB behavior. By accident rather than design, all of the codes have small instruction footprints, and so instruction referenced TLB misses were practically insignificant. For this reason the impact of instruction TLB misses is not considered further in this paper.

Fluent UNS is a highly used commercial Computational Fluid Dynamics code from Fluent International [21]. The chosen problem set simulates the exhaust flow through an engine valve. This is an iterative application with a run-time of over 45 minutes on a 300MHz UltraSPARC-II system (180 billion instructions).

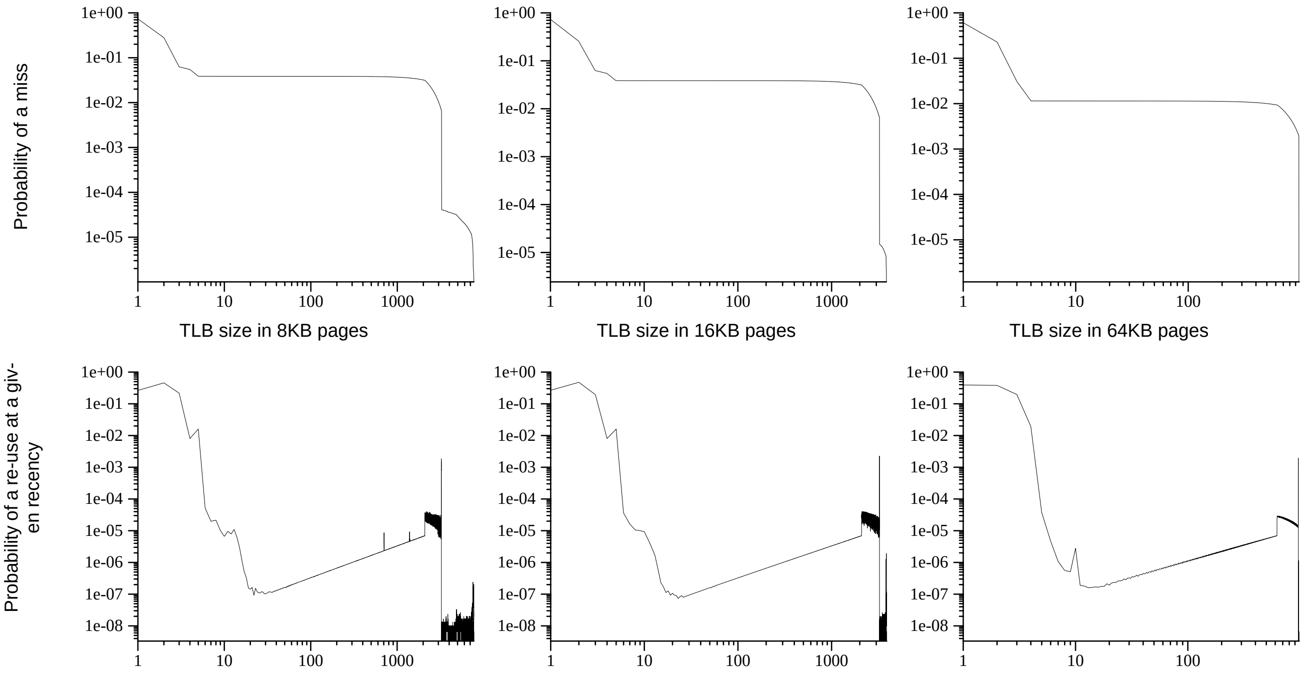
LightWave is a commercial rayshader/renderer (used for TV shows and movies) from NewTek [22]. The chosen problem was the rendering of a typical (in complexity) frame from an animated TV short.

Sphinx is a natural speech recognition system developed by CMU. Typical of speech recognition systems, it uses a hidden Markov model database for recognition decisions. For this application the problem set is around 1 minute of spoken audio from National Public Radio.

PNMRotate is a trivial image manipulation operation taken from the netpbm toolkit [23]. It uses the 3-shear method to rotate an uncompressed raw 24-bit RGB image (2Kx1K pixels) clockwise by 90 degrees. This application was chosen because it is representative of the genre of image manipulation algorithms, and therefore easily attackable via software or hardware stride prediction.

Vortex is a database manipulation code from the SPEC '95 benchmark suite. It is practically the only application in this suite that stresses TLB design. Its commercial significance is not so much as a database code, but as part of the SPEC'95 suite, and therefore used as a metric for microprocessor design. In accordance with SPEC'95 rules, it was compiled for peak UltraSparc-I performance.

Figure 4 .The TLB miss rates and recency probability distributions for *PNMRotate*



3.3 Application behavior

We now apply the stack-algorithm methodology described in Section 3.1. to the applications described in Section 3.2.

3.3.1 PNMRotate: An illustration

To aid understanding we start with a trivial application, *PNMRotate*. If we look at the memory reference trace generated by running the application in a simulator, we can use the stack algorithm to ascertain the performance of all possible TLB sizes. These results are given in Figure 4 for page sizes of 8KB, 16KB and 64KB respectively

Each of these graphs depicts the probability of a TLB miss as a function of the TLB size for a fully-associative LRU based TLB. As the number of TLB entries increases the probability of a TLB miss decreases, however the probability function is far from linear. There are three distinct areas in each of the graphs. Starting with small values and increasing the size of TLB, we can see the probability of a TLB miss rapidly drop off to a few percent. Thus a tiny TLB of less than 10 entries or so is sufficient to cover the page translation requirements of the most active part of the working set.

From between 10 to 1000 TLB entries we hardly see the probability of a TLB miss drop at all. This demonstrates that this application has very bad TLB behavior. Indeed the miss rate does not begin to drop off again until the TLB can map practically the entire application working set. At the end of the graph we see another tiny plateau generated by references to data made both at the start and end of execution, but not used at all during the execution. Thus, only the largest TLBs would be big enough to hold these page translations from start to finish of the application.

From these probability graphs we can quickly read off the miss rate for any given TLB. An UltraSPARC-II system has a 64 entry TLB (with 4 reserved entries). From the graphs we see a miss probability of between 5% to 1% as the page size is varied from 8KB pages to 64KByte pages.

More interesting than the miss probability graphs are the stack histograms used to create them. These are depicted underneath the probability graphs for the 8KB, 16KB and 64KB page sizes. The histograms depict the probability density function of an address reference as a function of its recency.

We can quickly see that the most frequent memory references have very low recency (typically less than 10), which tells us that a small TLB will capture the most active working set. Unfortunately between recencies from 20 to around 1000 there is a very low probability of a memory reference (remember these are log-log graphs). This indicates that a TLB much larger than around 10-20 entries will not do very much for us.

Surprisingly, an interesting feature appears at a recency of about 4000, for 8KB pages, 2000 for 16KB pages and 1000 for 64KB pages. Here we see a very sharp "spike", which indicates that there are a very large number of accesses to pages which have essentially the same recency value. It is this feature which we make use of for our pre-load algorithm.

Of all the memory references made, those in the "spike" with recency 4000 (for 8KB pages) constitute around 0.5% - which we can read directly from the graph. However, if we consider only those accesses with a recency value greater than a modest TLB size (say 20 entries), we see that the "spike" at recency 4000 dominates the other accesses with a probability almost two orders of magnitude higher.

If, having filtered away the low recency re-use accesses (which we do with a TLB), we were to guess at what the next memory reference is to be, the obvious choice is the one with the recency value of the "spike" - the one with the highest probability.

This strongly indicates that some form of recency-based prefetching would be beneficial.

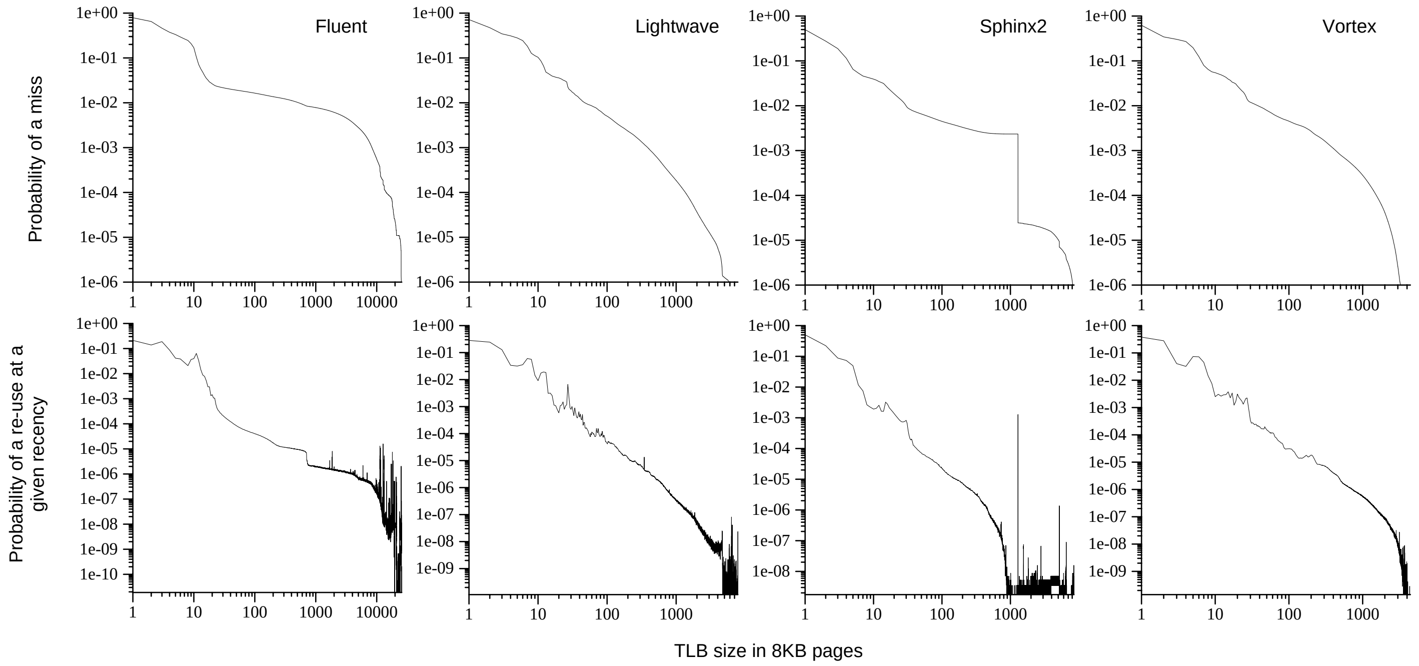
3.3.2 Other applications

Figure 5 shows the TLB miss probability graphs, and recency probability density functions for the other real workloads we use during the course of this paper.

We quantitatively see that all of these applications exhibit a high TLB miss-rate for conventional sized TLBs (10s of entries). Typically the miss-rates do not fall below 1% of all memory accesses until the TLB is of the order of 100 entries or more. In the case of *Fluent UNS*, we need over 1000 TLB entries. Therefore, substantial increases over conventional TLB sizes are needed to impact these miss-rates.

However, this figure also shows that our applications' probability distributions share the characteristic spikes, seen with PNMrotate, that indicate prefetching benefit.

Figure 5 TLB Performance of real applications



3.4 Discussion

All prefetching schemes rely on some predictable access behavior. We therefore consider our recency-based prefetching scheme in the context of various application access behaviors. We group application behaviors into three main categories: **Linearly Predictable**, **Non-Deterministic** and **Deterministic Iterative**:

Linearly Predictable access behavior generally results from large dense regular data structures which are walked through in a linear fashion. Examples are scientific or image processing.

Several cache-based prefetching techniques that exploit such reference patterns have been proposed in the past. The most notable ones are sequential prefetching and stride prefetching (see [8] and the references contained therein). Linear (or sequential) prefetching is a simple heuristic that predicts that the next page to miss in the TLB is the next virtual page. This simple heuristic has been used to prefetch code blocks into cache in many processors. It has also been evaluated in context of data caches for shared-memory multiprocessors [8]. Linear prefetching essentially exploits spatial locality. As we will show, while it works well in some cases it is restricted to applications that access all elements of huge data structures. Linearly predictable applications are expected to be easily attackable via simple prefetching schemes in either hardware or software. As a comparative metric, we consider linear prefetching in our evaluations.

A Non-Deterministic access behavior has almost no discernible behavior pattern. Typically, both working set, and order of working set access are arbitrarily determined by input data. These characteristics make prefetching and prediction extremely difficult if at all possible.

A Deterministic Iterative access behavior results from a large class of applications which typically iterate over a more or less constant data set. In such applications the data itself need not be regularly ordered, the only ordering is temporal through re-use. It is this kind of application which is expected to lend itself best to a recency-based prefetching scheme.

4 RECENCY-BASED TLB PREFETCHING

In this section we present our TLB prefetching technique by first providing a rationale for it in Section 4.1 after which we propose an implementation in Section 4.2.

4.1 Rationale

Quite simply, our algorithm speculates that after a given TLB miss the required translation for the next TLB miss will most likely have the same (or similar) recency.

One can see how this might make sense. If an application iteratively ploughs through a set of data structures, then there is a temporal ordering between the structures even though there may be no ordering in either the virtual or physical address space.

For example, consider three data structures A, B and C. A deterministic iterative application might loop through these structures in the same order several times during a computation - for example if they were connected on a linked list. After the first iteration, it is clear that A preceeded B which preceeded C, which has now preceeded A again. As we return to A we can determine that the program last accessed A two structures previously (notional recency 2), and as the computation progresses to A we predict that the computation will eventually move to the structure which is now two structures temporally behind A - i.e. correctly predicting the move to B, and so forth to C and round again.

Often the temporal ordering will remain the same, even though the actual content of data structures is altered as part of the computation. In order to maintain recency ordering we can use an LRU stack as described in Section 3.1. For a simulator there is no problem to maintain a full LRU stack for temporal address ordering, recency values of translations can be determined, and the reverse of a translation for a recency value can also be determined. Thus the effectiveness of a recency-based predictor is easy to evaluate, but how might this be practically implemented?

4.2 A hardware implementation

Speculative TLB preloading algorithms may be implemented in either a software or a hardware TLB reload handler. In the case of software, while flexible, the cost of implementation is a direct increase in the latency of handling a TLB reload.

However, in the case of a hardware implementation, the fundamental TLB miss can be serviced, the CPU restarted, and *then* any speculative operations can proceed concurrently with continued CPU execution. With this in mind, we propose and evaluate a *hardware* prefetching implementation for a *hardware* TLB reloader[1].

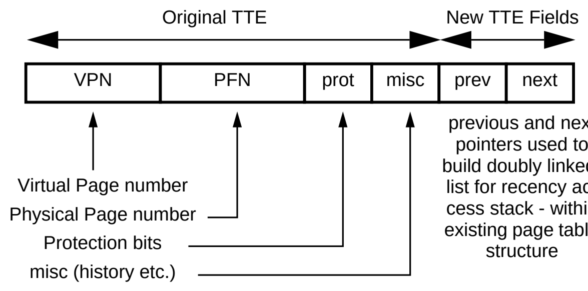
Such a system consists of the conventional MMU components; the TLB and reload hardware, augmented with a small buffer for pre-fetched TLB entries and the prefetching engine itself. With the prefetcher in operation, a load or store is likely to be handled by the MMU in one of three ways:

i) A match in the TLB results in the conventional translation of addresses, and update of the TLB's entry replacement state.

ii) A miss in the TLB may match in the prefetch buffer. Address translation occurs as in i), the CPU is restarted, the matching entry is transferred to the TLB, and the prefetcher started to prefetch another entry.

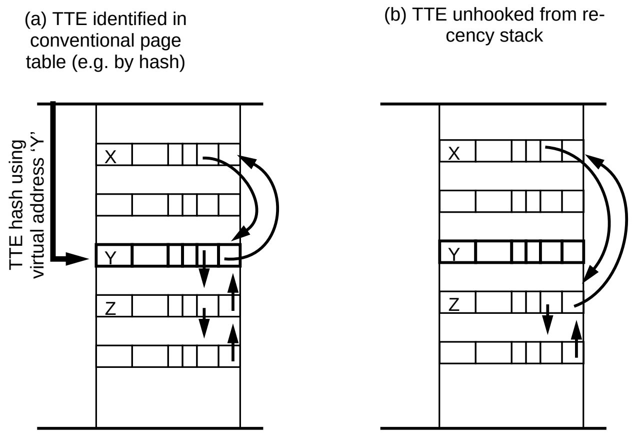
iii) A miss in the TLB does not match in the prefetch buffer either. The correct TLB entry has to be fetched, whereupon the cpu is restarted. The prefetcher can use this miss information to predict new translation entries for the prefetch buffer.

Figure 6 Augmented Translation Table Entry



Original TTE
New TTE Fields
VPN
PFN
prot
misc
prev
next
Virtual Page number
Physical Page number
Protection bits
misc (history etc.)
previous and next pointers used to build doubly linked-list for recency access stack - within existing page table structure

Figure 7 Recency stack implemented within existing page table



(a) TTE identified in conventional page table (e.g. by hash)
(b) TTE unhooked from recency stack
TTE hash using virtual address 'Y'
X
Y
Z
X
Y
Z

4.2.1 Implementation of a recency stack

To start with, the TLB reload mechanism remains the same - a page table walk, or hash table, for example, is used in the conventional way to find translation entries for the TLB.

A **key** observation enables a recency based predictor to be built on top of this mechanism without the need for complex hardware support: For recency-based prediction, unlike the stack-based miss-rate calculation, we are not interested in the absolute recency position of a translation entry in the LRU stack which maintains temporal ordering. Instead, we are only interested in an entry's nearest neighbors in the stack. We shall see how this works shortly, but first it is important to see how to build and manage the LRU stack used for recency prediction.

[1],but without the loss of generality which would preclude a software implementation.

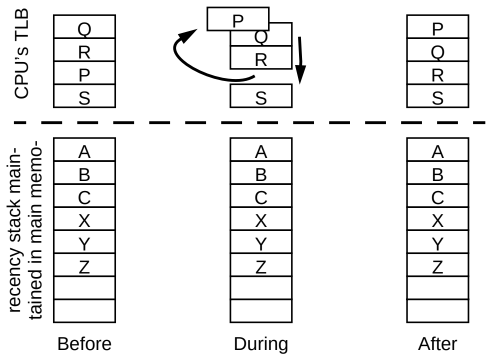
To build the LRU stack, on top of the conventional reload mechanism, we use a doubly-linked list, each element in the list is a page translation table entry as used by the reload mechanism.

Thus we have the conventional page tables for a normal MMU, except that each Translation Table Entry (TTE) (or virtual to physical translation) is augmented with *previous* and *next* pointers, (Figure 6), so it may be also threaded into a doubly-linked list which forms the recency stack.

Next we note that the only manipulation of the recency stack that occurs is the removal of an element from an arbitrary position in the stack to place on the top of the stack - all elements with lower recency values being propagated down the stack. All elements with higher recency values maintain their position. The simple doubly-linked list structure enables the extraction of individual elements with O(1) time to maintain the integrity of the list. Similarly O(1) time is required to insert elements to the top of the stack. There is no movement of data going on here, merely simple pointer manipulation.

Thus, upon a TLB miss the required TTE is selected by hardware in the usual manner, for example by table walk, or hash table - as depicted in Figure 7a. Once found, the translation is supplied to the TLB and the CPU restarted. Then, the previous and next pointers in the TTE, and those of the respective previous and next TTEs are manipulated to remove the TTE from its current position on the stack (Figure 7b), ready for insertion as the new head of the stack.

Figure 8 Hardware TLB maintains temporal ordering for us



To maintain temporal ordering of virtual page references in this way for every load and store instruction is clearly not a practical solution. Even though no work has to be done for top of stack (recency=0) references, the job is still impractical.

Key to resolving this is the observation that the TLB itself implements the upper N elements of our recency stack. This can be seen in Figure 8 where entry P is already in the TLB when accessed, so only the TLB's LRU ordering is updated leaving the remainder of the recency stack untouched.

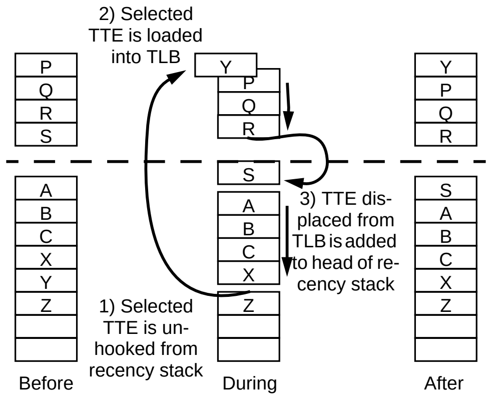
The memory-based LRU stack is only manipulated on a TLB miss, when a translation from outside the TLB is moved to the top of the "stack"; as a consequence, the replaced TLB entry falls out of the TLB down on to the top of the in-memory LRU stack. This is depicted in Figure 9. Thus, the only in-memory recency stack manipulations required are those related to TLB misses.

4.2.2 Predicting the next TTE

Although our algorithm bases the prediction of the next miss on the same (or similar) recency to the preceeding miss, the actual recency value is not required.

According to the stack algorithm, a required TTE is plucked from the LRU stack and moved to its top, the TTEs above the one selected move down the stack making room for the selected one at the top. From Figure 9 it can be seen that any arbitrary TTE 'Y' with recency *R*, once selected and moved to the top of the stack gains recency 0. Meanwhile the TTE 'X' immediately prior to 'Y', previously with recency *R-1*, moves down the stack taking over the position with recency *R*. Also, the TTE ('Z') with recency *R+1* remains the same.

Figure 9 Actions taken on actual TLB miss


2) Selected TTE is loaded into TLB
3) TTE displaced from TLB is added to head of recency stack
1) Selected TTE is unhooked from recency stack
Before
During
After

Trivially then, once a TTE 'Y' is selected to fulfil a TLB miss, the predicted TTE with the same recency is the TTE 'X' immediately above in the recency stack. The TTE with recency+1 is the TTE ('Z') below. In fact predicted TTEs with recencies +/- a small amount are easily found by simply following the previous and next pointers from the selected TTE.

4.2.3 Handling the predicted TTE

A predicted TTE is only a guess, and cannot be moved from its position in the recency stack until it has been confirmed as correct. If it were moved to the top of the recency stack regardless, the correct temporal ordering of TTEs would be perturbed and no longer viable for prediction.

For the same reason (as the TLB models the upper part of the recency stack on our behalf) a predicted TTE cannot be directly loaded into the TLB.

Instead the only additional hardware we recommend is a side buffer to hold prefetched TTEs. This buffer is checked in parallel with the main TLB. If a TLB match occurs no action is taken. If a TLB miss occurs and nothing in the predictor buffer matches, then a conventional TLB reload is required. If however on a TLB miss there is a match in the prefetch buffer, the matching contents of the buffer can be used to allow the cpu's load or store to complete.

Once a prefetched TTE match occurs, the matched entry can be inserted into the TLB, and the unhooking from the LRU stack is taken care of in the background by the hardware TLB loader, at the same time as inserting the evicted TLB entry on to the top of the LRU stack.

4.2.4 Non LRU TLBs

For speed and complexity reasons most contemporary CPUs implement only an approximation to a Least Recently Used algorithm. Some approximations are better than others, and the deviation from a true LRU algorithm will affect the results. However this may be positive for some applications - forcing some

TLB replacements to have greater temporal regularity for given translations; for other applications the effect will be worse.

It is the responsibility of a CPU architect to determine the impact of the actual replacement scheme when related to recency-based prefetching. On the positive side, most early ejections from the TLB as a result of a sub-optimal replacement policy are likely to be quickly fetched back into the TLB without progressing too far down the recency stack to the point where most recency prefetches will occur.

5 EXPERIMENTAL APPARATUS

The research in this paper is based on the use of execution-driven simulation to obtain measured results of the prediction accuracy for the applications introduced in Section 3.2. Compared to trace-driven simulation this provides a greater versatility for the non-intrusive measurement of a range of systems, some of which may not be actually realizable.

Moreover, there is no dependence on captured trace files which, for size reasons, often only represent a snap shot of complete application behavior. Further, although TLB misses are fairly frequent events, the kind of data re-use we were interested in often would only occur after many billions of instructions, making traces huge and impractical. We next consider the prefetch schemes we have evaluated and then the experimental setup used.

5.1 Evaluated prefetch schemes

For the results presented, the following prefetching algorithms are used.

5.1.1 Linear prefetching

Predicts a future miss based on the virtual page address of a current miss. Upon each TLB miss, the **data** virtual address causing the exception is shifted to extract the virtual page number concerned. A prediction of the next TLB miss to occur is then made using the extracted virtual page number plus or minus one or two page frame offsets. The virtual-to-physical translation is also loaded at the same time into a TLB prefetch buffer, from where it is inserted into the TLB automatically (according to the TLB replacement algorithm) if the prediction proves to be correct.

If the prediction is incorrect, a TLB exception occurs as usual. If the prediction is correct, then the predicted (correct) virtual page number is then used to predict again with the same constant.

5.1.2 Recency-based prefetching

This prefetcher would maintain an LRU stack in memory, as described earlier. The stack enables the measure of recency for virtual page addresses (and their translations).

Upon a TLB miss, the required virtual page is found in the LRU stack, and its recency noted. The stack is adjusted with the selected entry migrated to the top as per the algorithm described earlier. Then, a prediction of the translation required for the next miss is made based on the recency of the miss currently being handled. The page translation with the same recency value is selected and held in a TLB prefetch buffer until the next TLB miss occurs.

Just as the linear prefetcher above, if the prediction is correct it is moved to the TLB and also used for the next prediction. If incorrect, then a full TLB reload occurs, and the next miss prediction is based on the correctly loaded translation.

5.2 Simulation environment

The Shade-V [4] execution-driven simulator enables the dynamic re-compilation of complete applications, enabling the insertion of custom monitoring code. This code can be used either to garner trace information, or in our case to simulate the target architecture as the program executes.

Simulation measurements were taken for a range of simulated TLB sizes, and TLB page sizes ranging from 8KB to 64KB in size.

Very big pages (or superpages) were not simulated; primarily because, although large, the benchmarks are small compared to such pages, and the results would be un-representative.

Superpages essentially rely on an extremely high degree of spatial locality for efficient use, which works for well behaved applications with linear data structures, but not in the general case. Intuitively we can see that for real applications, where even a super-page is a small fraction of the data set size, it is of little benefit to have a page much larger than the program's intrinsic data structure size since the probability of reuse before eviction is small. This is analogous to the diminishing benefits of increasing cache line sizes. So, still today, the use of such superpages is often restricted to special purpose operations, such as frame buffers and kernel mappings.

6 RESULTS

6.1 Prediction Accuracy

Each of our five target applications was executed using Shade for each of our prefetching scheme simulators, to track the performance of the prefetchers themselves. The results from these experiments are presented in Figure 10 through Figure 14. Each figure represents a single application, but using different TLB and page-size configurations as well as different predictors.

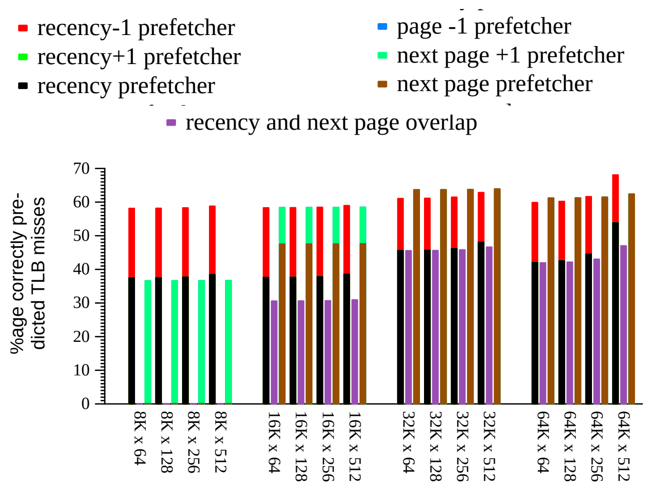
Along the X-axis of each graph results are grouped at the coarsest level into four different simulated page sizes from 8KB to 64KB pages. For each page size we have simulated four increasing TLB sizes, from 64 to 512 fully-associative entries each with a least-recently-used replacement algorithm. Finally, for each page size and TLB-size combination we show three prediction effectiveness bars.

The left-most bar depicts the percentage of correctly predicted TLB misses for recency-based prefetching. This bar is itself subdivided to show the prediction rate of preloading using pages with the same recency ("recency prefetcher") as the previous miss, pages with recency-1 and pages at recency+1. The total height of the bar is then the effect of prefetching all three pages.

The right-most of each group of three bars shows the equivalent statistics for a linear predictor. The Predictors are next page, next page+1, and previous page.

Finally the (always) smaller bar between the two predictor bars indicates the correctly predicted TLB misses which were predicted by *both* the same-recency predictor and the next-page predictor. Using this measure, we can see whether or not the recency predictor chooses pages also predicted using a linear stride predictor.

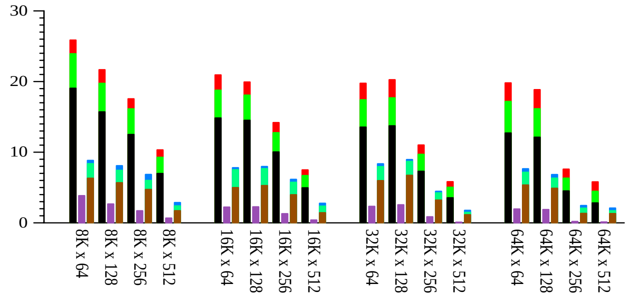
Figure 10 *PNMRotate*



The simplest of our target applications is PNMRotate. It has a very regular stride data access, which is easy to predict. Note in Figure 10 there are no correctly predicted "next page" prefetches for 8KB pages. We observe therefore, that the dominant miss rate has a stride larger than a single 8KB page, since for 8KB pages, the next page prefetcher has no success, but yet the next page +1 (next next page) prefetcher has a success rate. Yet, when we move to the larger page sizes, the next page prefetcher starts working. All of these are results we would expect for both predictors - the linear array access by the application means a big potential win for the linear next page predictor. Similarly, the regular temporal reuse of data makes the recency based scheme consistently successful.

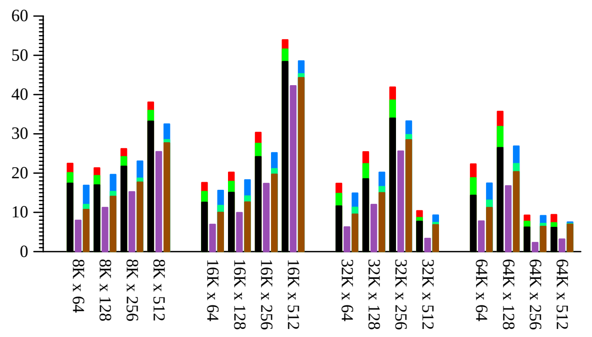
It is interesting to note however that while one could use a stride predictor to tackle such applications, the recency predictor essentially performs as well, but without the need for significant extra hardware to compute and retain predicted stride patterns. This indicates that a recency-based predictor could do well for regular array computational codes - when compiler inserted software prefetching is not an alternative.

Figure 11 Vortex



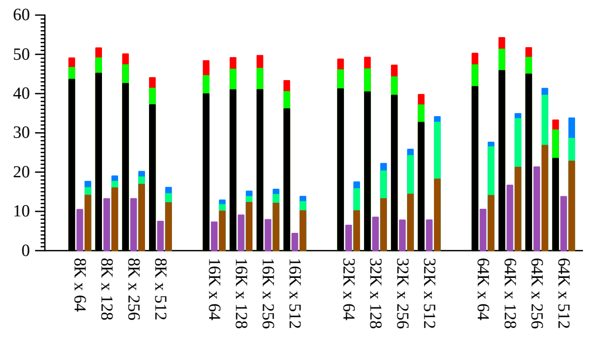
Vortex has high (almost 20%) correct prediction of TLB misses for small TLB and pages sizes. However, as the TLB reach (page size and capacity) increases the benefit of recency prefetching decreases. This is also true for the next-page (stride) predictor. Looking back to the probability density function of access recency for Vortex (shown in Figure 5) we see that Vortex has a relatively small application working set, of only a few thousand 8KB pages, and with the majority of accesses concentrated in the most recent few hundred pages. There are no real characteristic "spikes" in this density function which implies that outside of the core few hundred pages of working set, the remainder of data is accessed in no particular temporal order. Also we can see from the performance of the next page predictor, data access appears not to be spatially linear either. Therefore neither predictor will work well once the TLB has a large enough span to cover the core working set of this application. For example, the 64K x 128 TLB has the same span as 1024, 8K pages, which is 1/4 of the entire data set.

Figure 12 Sphinx2



For Sphinx, both prefetchers perform well, with the recency based scheme slightly better in each instance. For this application not only is there data reuse which the recency prefetcher can take advantage of, but the data appears to have good spatial locality across pages so that the linear prefetcher also does well. We can see that for the most part, both the linear and the recency prefetchers are predicting the same pages - the middle bar of each triplet of results.

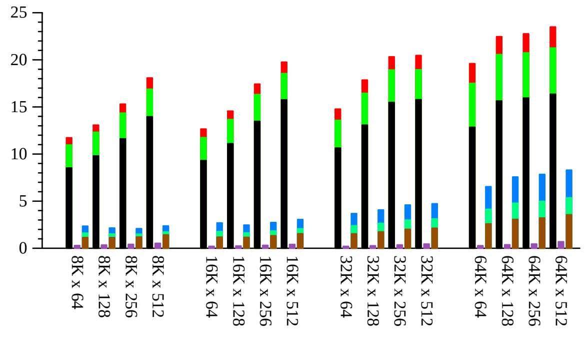
Figure 13 LightWave



For LightWave (shown in Figure 13), the recency predictor does extremely well. This makes sense, because as a ray-shader the core of the application will traverse the same internal data structures in more or less the same order to compute each pixel in the final rendered scene. However, the linear (next page) predictor does poorly in comparison because there is apparently little spatial locality between the allocated objects in the scene database. It is not until the page sizes become significant and the TLB starts to cover most of the application working set that the performance of both predictors once again begins to roll off.

Finally Fluent UNS (shown in Figure 14) shows a significant prediction rate for the recency based TLB prediction.

Figure 14 Fluent UNS

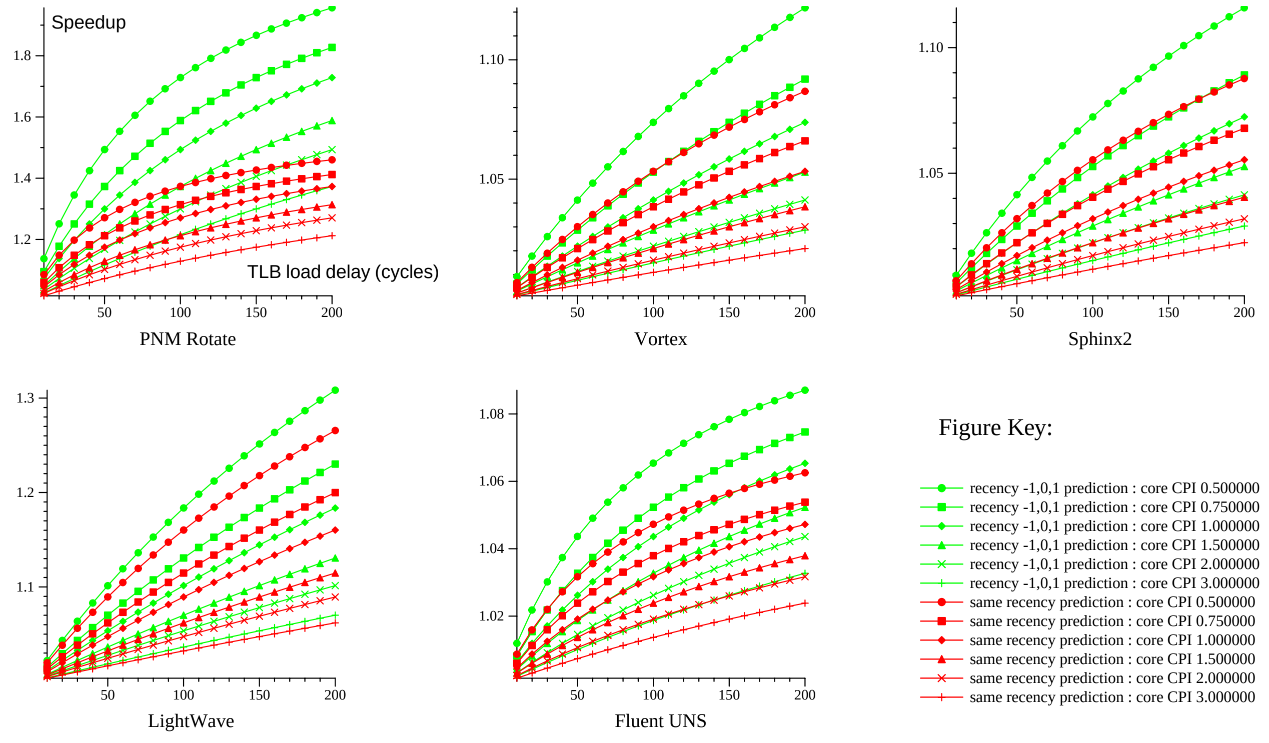


Importantly, as this is the largest of all the applications with a working set of some 200 MBytes, we do not see the roll off in prediction with increased TLB and page size. This is because the span of the TLB never becomes a significant fraction of the application working set size. Moreover, the prediction rate improves with greater TLB span, presumably because as the TLB grows we see fewer misses from the inner working set of the application which interfere with the more regular misses at higher recency values - note that in Figure 5 we see a step down in the miss probability density function at around 500 (8KB) pages suggesting at least one inner and outer working set.

6.2 Performance Impact

From these results we can use a relatively simple analytical model to determine the impact on performance of our TLB pre-loading scheme. There are so many factors determining CPU performance, that it is often most useful to consider the impact of a single component in terms of its Cycles Per Instruction (CPI) impact. Therefore, for this model, we compute the CPI impact of TLB misses both with and without the benefit of TLB pre-fetching. As CPI components are additive, we can then treat the remaining components of the CPU as a whole, and abstractly describe their performance as a single additional CPI value - this value is the performance of the CPU core, caches, branch predictors and all other components aside from the MMU.

Figure 15 Speedup ratios for recency based prefetching



Based on this reasoning, we can plot the following ratio:

$$\frac{CPI_{core} + CPI_{WithoutTLBPrefetch}}{CPI_{core} + CPI_{WithTLBPrefetch}}$$

as a function of the number of cycles delay caused by a TLB reload, and as a function of the additional processor CPI component CPI_{core}. This ratio gives us an upper bound for the speedup potential of our TLB prefetching technique.

Figure 15 shows the results of this computation for each of our applications, for a 64 entry fully associative TLB with 8KB pages. The two sets of curves are for; prefetching a predicted page translation with the same recency as the preceeding TLB miss, and prefetching three translations with the same recency, recency-1, and recency+1 as described earlier. Curves for the other TLB combinations simulated are omitted for brevity, but are easily drawn using the miss rates given in Figure 10 through Figure 14.

Even conservatively, from Figure 15, if we choose a TLB reload delay of around 100 cycles - as representative of a contemporary CPU - then the simple same-recency prefetcher has the potential to improve performance by a factor of between 1.01 to 1.35 depending on the application (TLB miss rate) and core CPI performance of the CPU.

These figures are encouraging, however more interesting is the trend for improving performance potential as both the TLB load delay increases, and as the CPU's core CPI decreases.

For future CPU designs, with main memory latency lagging the increases in core clock frequency the number of cycles taken to satisfy a TLB miss is likely to increase, therefore increasing the performance potential for TLB prefetching.

Also, as core CPU designs improve, with greater instruction level parallelism, larger caches with fewer misses, and better speculative prediction, then the remaining CPI components of performance begin to diminish with respect to the TLB re-loading impact. We can see in Figure 15 the dramatic effect such improvements are likely to have on the relevance of TLB prefetching as the core CPI component decreases.

7 Discussion and Related Work

We have described our preloading algorithm as a mechanism for a hardware reloaded TLB. Yet, it is also applicable to software reloaded TLBs. However, observations in recent studies [18,19,11,17], suggest that software TLB reloading is an expensive operation, and increasing this software burden may outweigh the benefits of prefetching.

A study by Jacob and Mudge [11] has shown that for modern microprocessors software-based reload handling is expensive for a range of different reload strategies (or Operating System page-table layouts), and that a simple hardware-reload mechanism can have better performance because its operation can sometimes be performed concurrently with CPU execution. In another study [17], Qiu and Dubois have shown that for contemporary CPU designs, software reload routines have significant execution expense beyond executing the reload code itself. Their study details a method of hiding some of the cost of entering the reload code by handling the exception as late as possible.

Instead of increasing the TLB size with its associated latency penalty, other approaches include (i) using larger pages, (ii) multi-level TLBs, and (iii) using virtually-addressed caches. Larger pages, or super-pages [9,18,19], like larger cache line sizes exploit spatial locality in an application. At page frame granularities larger pages are highly dependent on application locality, especially while the page size is still only a tiny fraction of the program data set size. As for multi-level TLBs, there are several limitations. Like increased TLB size, they consume die area. Indeed the inclusion property essentially wastes die area. Moreover, they do not eliminate the TLB miss penalty for the lowest level.

Another alternative is to consider virtually-addressed caches [2,3,10,16,20] to minimize the number of required references to the MMU. Only cache misses require translations which potentially enables larger, slower TLBs. This can work well for certain kinds of applications, however the synonym problem (of one physical page frame being simultaneously mapped to two different virtual addresses) arises for others. While there are techniques for resolving the synonym problem, they are sometimes expensive (e.g. forcing cache flushes), and moreover synonyms are being found useful for certain optimizations such as zero-copy TCP/IP implementations [6].

As with data caches the TLB miss rate determines the efficacy. Thus, the widely used 3C-model (Compulsory, Capacity, and Conflict misses) is useful to contrast different techniques.

Compulsory TLB misses have been attacked using prefetching techniques [13,15]. In [13] the Kernel has prior knowledge of likely future page faults and can preemptively create additional mappings and avoid the expense of each page being faulted. This work also shows how the fine-grain structure of micro-kernel based Operating Systems leads to excessive page re-mapping and sharing between the many communicating virtual address spaces. To mitigate the cost of such (re)mapping they "prefetch" by speculatively calculating the new page translations for the recipient virtual address space upon each trap into the micro-kernel for communication. The results of this speculation are then cached in a kernel software table and used by the TLB reload handler to avoid a full blown page fault. Interestingly they also note that the translations might instead be placed directly into the TLB to avoid the reload trap.

Unfortunately this technique really only applies to potential manipulations of processes' virtual address spaces particularly in relation to the excessive message communications of micro-kernel operating systems. In addition, this technique may still be applicable to zero-copy TCP/IP [6] and other communication operations between peer-applications in specialized situations such as web-servers.

[15] evaluates a prefetching technique for compulsory TLB misses implemented in a software reloaded TLB. For reasons that have been mentioned, the efficiency of such a scheme is impacted by the overhead of software-loaded TLBs. Our prefetch algorithm attacks the more difficult capacity miss class. Other approaches to attacking capacity misses target the selection and replacement algorithms. LRU approximating algorithms tend to perform sufficiently well for a wide range of applications, therefore we have not explored this option.

Finally, conflict misses are more difficult to handle for TLBs than in caches, since the virtual address space usage is almost entirely under the control of the application; whereas for a data cache the physical address space is under the control of the Operating System. Thus there is pressure for a high associativity in TLB design - thereby reducing the potential for conflicts. Techniques such as skewed accesses (as used for data caches [26]) may yet show promise here.

8 Conclusions

In a conventional processor each access to virtual memory an application makes must be handled via the Translation Lookaside Buffer. This means that the pressure on the TLB scales with main memory capacity, while the speed of handling a TLB miss scales only with the memory latency. TLB performance is becoming more and more critical as the capacity of system memories rapidly outpaces reductions in latency.

Unfortunately, the TLB is often on the critical path for cache access validation within a processor, which places extra constraints on its speed, and hence its capacity.

In this paper, we have focused on the possibility of pre-loading (or prefetching) TLB translations before they are required, thereby avoiding TLB misses and their impact on system performance. We have considered this principally in the context of a hardware TLB reload mechanism.

We have used a stack algorithm to study the performance of applications for all possible TLB sizes, and introduce the concept of recency. By plotting frequency of access as a function of recency, we see, rather than a smooth linear function, probability density functions for most applications which strongly suggest a component of data accesses which are temporally regular. This indicates that a large class of applications iterate around their working set several times with only small deviations in the order of access. This temporal regularity is the foundation of the new recency based prefetching algorithm we present and evaluate.

We show that a temporally ordered stack of page translations is easily implemented and maintained by hardware, and then used to simply “guess” the required translation for an upcoming TLB miss based on the assumption it has the same or similar recency to the preceeding miss.

Quantitatively we consider the effectiveness of TLB prefetching, using both a simple linear “next page” predictor and our recency based predictor. The next page prefetcher relies on spatial locality rather than temporal locality for its operation.

We have concentrated on a range of commercially significant applications for our study. Our stack based studies of these applications show that, despite their diverse nature, TLB performance is important for all. Our results show that a simple preloading scheme can reduce TLB misses by over 50%.

In those applications where a linear or stride predictor would yield significant benefit, we have seen that the recency-based predictor also predicts the same required translations. As a consequence the recency-based predictor frequently outperforms the linear predictor in terms of correctly predicted potential misses.

As the trend towards faster processors exploiting more instruction-level parallelism continues, the impact of TLB misses is likely to become worse. Following this trend, results from a simple analytical model illustrate an increasing performance potential for recency based prefetching. Therefore, TLB preloading techniques will likely be in future processor designs.

9 A NOTE ON FUTURE WORK

To date most prefetching techniques have been focused on reducing the impact of data and instruction cache misses during the execution of program code. However, there is a close relationship between the Memory Management Unit and the system caches. We feel that schemes, such as recency-based prefetching, which are based on the temporal behavior of applications, offer the opportunity to identify phase changes in application execution. If we can successfully predict such changes for the TLB, we are implicitly given information about which data is also expected from caches.

Using recency-based prefetching (in its presented form) directly for caches is likely to be awkward, because of the relatively higher miss rates, and much finer granularity of data caches. However, such a prefetcher, while benefiting TLB performance, might also be used to guide or filter data cache prefetches from a multiple of more conventional prefetching algorithms - thereby improving efficiency and lowering the prefetch bandwidth requirement.

10 ACKNOWLEDGMENTS

Grateful thanks to Sun Labs staff for their support and comments on this work. Per Stenström is supported by the Swedish Research Council for Engineering Science (TFR). Contract 221-97-593.

11 REFERENCES

[1] T. Austin and G. Sohi, “High-Bandwidth Address Translation for Multiple-Issue Processors,” in *Proceedings of the 22nd Ann. Int. Symp. on Computer Architecture,* pp. 158-167, 1995.

[2] M. Cekleov and M. Dubois, “Virtual-Address Caches, Part 1: Problems and Solutions in Uniprocessors” pp. 64-71, in *IEEE Micro,* Nov/Dec 1997.

[3] J. Chase, H. Levy, and M. Feeley, “Sharing and Protection in a Single-Address-Space Operating System,” in *ACM Trans. on Computer Systems*, pp. 271-307, Nov. 1994.

[4] B.Chemlik, “The SHADE simulator”, Sun Labs T.R. 1993.

[5] J. Chen and A. Borg, “A Simulation Based Study of TLB Performance,” in *Proceedings of the 19th Ann. Int. Symp. on Computer Architecture*, pages 114-123

[6] H.K.J. Chu, “Zero-Copy TCP in Solaris”, in *1996 USENIX Annual Technical Conference*, January 22-26, 1996, San Diego, California

[7] D. W. Clark and J.S. Emer, “Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement,” in *ACM Trans. on Computer Systems*, vol. 3, no. 1, 1985.

[8] F. Dahlgren and P. Stenström “Evaluation of Stride and Sequential Hardware-based Prefetching in Shared-Memory Multiprocessors,” in *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 4, pp. 385-398, April 1996.

[9] J. Huck and J. Hays, “Architecture Support for Translation Table Management in Large Address Space Machines,” in *Proceedings of the 20th Ann. Int. Symp. on Computer Architecture*, pp. 39-50, May 1993.

[10] B. Jacob and T. Mudge, “Software-Managed Address Translation,” in *Proceedings of the 3rd Int. Symp. on High-Performance Computer Architecture*, pp. 156-167, Feb 1997.

[11] B. Jacob and T. Mudge, “A Look at Several Memory Management Units and TLB-Refill Mechanisms and Page Table Organizations,” in ASPLOS-VIII, pp. 295-306. 1998.

[12] http://www.speech.cs.cmu.edu/speech/sphinx.html

[13] K. Bala, M.F. Kaashoek, W.E.Weihl, “Software Prefetching and Caching for Translation Lookaside Buffers”, in *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.

[14] R.L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger, “Evaluation Techniques for Storage Hierarchies”, in *IBM Systems Journal* 9 (2):pp.78-117, 1970

[15] J. S. Park and G. S. Ahn, “A Software-controlled Prefetching Mechanism for Software-managed TLBs,” in *Microprocessing and Microprogramming*, Vol .41, No 2. pp. 121-136, May, 1995.

[16] X. Qiu and M. Dubois, “Options for Dynamic Address Translation in COMAs,” in *Proceedings of the 25th Ann. Int. Symp. on Computer Architecture,* pp. 214-225, June 1998.

[17] X. Qiu and M. Dubois, “Tolerating Late Memory Traps in ILP Processors,” in *Proc. of 26th Ann. Int. Symp. on Computer Architecture,* pp. 76-87, 1999.

[18] M. Talluri and M. Hill, “Surpassing the TLB Performance of Superpages with Less Operating System Support,” in *Proceedings of the Sixth Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct 1994.

[19] M. Talluri, S. Kong, M. Hill, and D. Patterson, “Tradeoffs in Supporting Two Page Sizes,” in *Proceedings of the 19th Ann. Int. Symp. on Computer Architecture,* May 1992.

[20] B. Wheeler and B. N. Bershad, “Consistency Management for Virtually Indexed Caches,” in *Proceedings of the Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct 1992.

[21] http://www.fluent.com

[22] http://www.newtek.com

[23] pnmrotate, part of Net PBM distribution, version 7: ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM

[24] AMD K-7 Product announcement at microprocessor forum. http://www.amd.com/products/cpg/k7/micropforum.html

[25] HAL SPARC64-III, Microprocessor Report, Dec 8, 1997 http://www.hal.com/home/sparc64-3_mda.html

[26] A. Seznec, “A Case for Two-Way Skewed-Associative Caches”, *Proc. 20th Annual Symposium on Computer Architecture*, pp. 169-178, May 1993