Going the Distance for TLB Prefetching: An Application-driven Study*

Gokul B. Kandiraju Anand Sivasubramaniam
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802.
{kandiraj,anand}@cse.psu.edu

Abstract

The importance of the Translation Lookaside Buffer (TLB) on system performance is well known. There have been numerous prior efforts addressing TLB design issues for cutting down access times and lowering miss rates. However, it was only recently that the first exploration [26] on prefetching TLB entries ahead of their need was undertaken and a mechanism called Recency Prefetching was proposed. There is a large body of literature on prefetching for caches, and it is not clear how they can be adapted (or if the issues are different) for TLBs, how well suited they are for TLB prefetching, and how they compare with the recency prefetching mechanism.

This paper presents the first detailed comparison of different prefetching mechanisms (previously proposed for caches) - arbitrary stride prefetching, and markov prefetching - for TLB entries, and evaluates their pros and cons. In addition, this paper proposes a novel prefetching mechanism, called Distance Prefetching, that attempts to capture patterns in the reference behavior in a smaller space than earlier proposals. Using detailed simulations of a wide variety of applications (56 in all) from different benchmark suites and all the SPEC CPU2000 applications, this paper demonstrates the benefits of distance prefetching.

Keywords: Prefetching, Memory Hierarchy, Translation Lookaside Buffer, Simulation, Application-driven Study.

1. Introduction

Address translation using the Translation Lookaside Buffer (TLB) is one of the most critical operations in determining the delivered performance of most high performance CPUs. Several studies have quantified the importance of TLB performance on system execution and the necessity of speeding up the miss handling process [15, 9, 22, 3, 14, 25]. Anderson et al. [3] show that TLB miss handling is the most frequently executed kernel service and has an important consequence on performance. TLB miss handling has been shown to constitute as much as 40% of execution time [14] and upto 90% of a kernel's computation [25]. Studies with specific applications [26] have also shown that the TLB miss rate can account for over 10% of their execution time even with an optimistic 30-50 cycle miss overhead.

There are several approaches to improve the delivered performance of TLBs. On the software side - at the application, compiler or operating system level - optimizations for improving locality can help lower the number of TLB entries needed to cover the working set of the execution at any instant. On the hardware side, TLB structure in terms of its size and associativity, as well as multilevel hierarchies, can have a significant impact on both the miss rates as well as on the access times [28, 7]. Another solution to boost TLB coverage is by the use of superpaging [28, 27]. Finally, on the miss handling side, a considerable amount of effort has been expended on tuning software miss handlers [25] or for performing the necessary actions in hardware.

However, it is only recently [26, 24, 4] that the issue of prefetching/preloading TLB entries to hide all or some of the miss costs has started drawing interest. Some of these [4, 24] consider prefetching TLB entries only for the cold starts, which in many long running programs (such as the SPEC 2000 suite) constitute a much smaller fraction of the misses. The first work on prefetching TLB entries for capacity related misses has been undertaken in [26]. Despite the voluminous literature on prefetching techniques available for other levels of the memory hierarchy, prefetching TLB entries has not gained much attention. This is, perhaps, due to the fear of slowing down the critical path of TLB accesses (which is usually much more important than the other levels of the memory hierarchy) and the possible cost/space of the additional real-estate (one could make a less strong argument about this with the ability to pack in millions of transistors on chip, though there is still the issue of power consumption and distribution that needs to be considered) that may need to be provisioned on-chip. However, we need to understand the benefits and ramifications of prefetching TLB entries inorder to be able to make these

*This research has been supported in part by several NSF grants including Career Award 9701475, 0103583, 0097998, 9988164, 9900701, 9818327 and 0130143.

trade-offs. In this paper, we specifically focus on the data TLB (d-TLB), which is usually much more of a problem than instruction TLB (i-TLB) in terms of miss rates [18].

Addressing the critical path issue, Saulsbury et al. [26] propose a new mechanism, called Recency-based Prefetching (RP), that maintains an LRU stack of page references and prefetches the pages adjacent to the one currently referenced (on either side of the stack). The associated logic is placed after the TLB, i.e. it has the privilege of examining only the misses from the TLB (and does not look at the actual reference stream). However, this mechanism can possibly increase memory traffic due to the need for manipulating LRU stack pointers kept in the page table.

A number of prefetching mechanisms have been proposed in the context of caches [29, 8, 16, 12, 17, 20] and I/O. To our knowledge, no prior study has investigated the suitability of these earlier proposals for TLB prefetching. It would be very interesting to see how these earlier proposals would work with the miss stream coming out of the TLB. While many of these schemes may require a little more logic/real-estate on-chip than RP, they usually do not impose as much storage and bandwidth requirements as RP.

It is well beyond the scope of this paper to cover a detailed survey/classification of prefetching mechanisms or to evaluate all of them (if one is interested, a survey of these can be found in [29]). Rather, we want to cover some representative points of the spectrum of mechanisms in the context of TLB prefetching. In a broad sense, prefetching mechanisms can be viewed in two classes: ones that capture strided reference patterns (using less history, such as sequential prefetching or arbitrary stride prefetching (ASP) [12, 8]), and those that base their decisions on a much longer history (such as markov prefetching (MP)[16] or even the recency based mechanism (RP) discussed above).

Reference behavior can also be viewed as following broadly one of these categories: (a) showing regular/strided accesses to several data items that are touched only once; (b) showing regular/strided accesses to several data items that are touched several times; (c) showing regular/strided accesses to several data items, but the stride itself can change over time for the same data item; (d) not having constant strided accesses (either keeps changing constantly or there is no regularity in the stride at all), but repeating the same irregularity from one access to another for the same data item over time; and (e) not having any regularity either in strides and not obeying previous history either. Usually stride based schemes are a better alternative than history based schemes for (a) (there is no history established here), while both categories can do well for (b). Some of the more intelligent/adaptive stride based schemes such as ASP can track (c) also fairly well, but the history based schemes are not as good for such behavior. On the other hand, history based schemes can do a much better job than stride based schemes for (d). In (e), it is very difficult for any prefetching scheme to be able to do a good job.

As we can observe, neither of the classes can do well across all of (a) through (d). Instead, we propose a new prefetching mechanism called *Distance Prefetching (DP)*[1] in this paper that tries to get the better of both approaches. The idea is to approximate the behavior of stride based mechanisms whenever there are very regular strided accesses (and capture first time references as well which are not possible in a history based mechanism), and track the history of strides (that is indexed by the stride itself). The hope is that whenever the stride changes, the changes themselves form a historical pattern and we can refer to this history to make better predictions. We find that DP can do fairly well (approximating the better of the two classes) for all of (a) through (d). DP is a general prefetching technique, that can be used in several situations (for caches, I/O etc.). In this paper, apart from proposing this new general purpose technique, we specifically illustrate its design and use for tracking TLB misses (placed after the TLB) to make good predictions. It takes space that is comparable to that of some of the earlier history based mechanisms such as Markov (usually a 256 entry direct mapped table suffices), while making much more accurate predictions. It also incurs much less memory traffic compared to RP which is the only other prefetching technique proposed and evaluated specifically for TLBs. The benefits of DP are demonstrated using a wide range of diverse applications spanning several benchmark suites (26 applications from SPEC CPU2000 [11], 20 applications from MediaBench [21], 5 applications from the Etch traces [1], and 5 applications from the Pointer Intensive Benchmark suite [2]).

The rest of this paper is organized as follows. Section 2 discusses the prefetching mechanisms, together with some of the hardware that is required, Section 3 gives performance results with actual applications and Section 4 concludes with a summary of contributions.

2. Prefetching Mechanisms

Since we extensively refer and compare against previously proposed prefetching mechanisms (including those used for caches), we briefly go over these to refresh the reader and to point out the exact implementation that is used later on in the evaluations. We also present our new prefetching mechanism - DP - in this section. It is to be noted that for uniformity in this adaptation, all these mechanisms initiate prefetches only by looking at the miss stream from the TLB, that is done in the earlier proposed RP mechanism [26] for TLB prefetching. All these mechanisms bring the prefetched entry into a "prefetch buffer" that is concurrently looked up with the TLB, and the entry is moved over to the TLB only on an actual reference to that entry from the application. Prefetching can thus not increase the miss rates of the original TLB. There is, however, the issue of additional memory traffic that is induced

[1] Distance Prefetching also tracks strides to make predictions. In the interest of distinguishing this mechanism clearly from the earlier stride based mechanisms, we give it a different name using the term "distance". "Distance" and "stride" mean the same thing and refer to the spatial separation (could be positive or negative) between any two successive references.

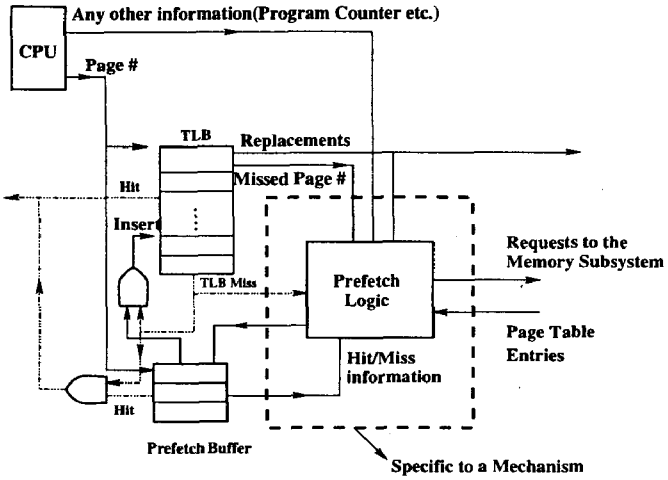

Figure 1. Schematic of Hardware for Prefetching in all the Considered Mechanisms

by prefetching, which can have a bearing on the execution time. In ASP, MP and DP, the prefetching engine uses a prediction table that has a given number of rows (r). MP and DP allow aggressive predictions, and each row of the table can have s slots. In ASP, each row contains only one slot as defined in [8] since this mechanism makes at most one prediction on a given reference. The indexing of the rows and what goes into each slot is specific to a scheme. The slots essentially determine what entries to prefetch, and thus s puts a bound on number of entries that can be prefetched on a given miss. The prefetch buffer size b is the same across all the mechanisms. A schematic of the overall prefetching hardware implementation is given in Figure 1.

2.1. Sequential Prefetching (SP)

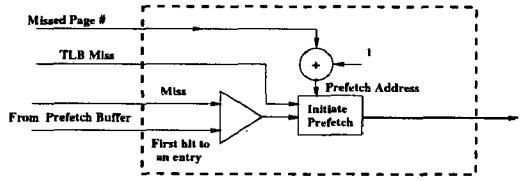

Figure 2. Schematic of Hardware for SP

This mechanism tries to exploit the sequentiality of references, and prefetches the next sequential unit (page table entry) based on the current reference. Several variations have been proposed, that are discussed by Vanderwiel and Lilja [29]. They point out that of all the schemes, *tagged* sequential prefetching - where a prefetch is initiated on every demand fetch and on every first hit to a prefetched unit, is very effective. Another variation proposed by Dahlgren and Stenstrom [12] dynamically varies the number of units to prefetch based on the success rate. However, simulations have shown only slight differences between these schemes [29, 12]. Consequently, we limit ourselves to the tagged version of SP in this paper. On a TLB miss, if the translation also misses in the prefetch buffer, it is demand fetched and a prefetch is initiated for the next virtual page translation (stride = 1) from the page table. The CPU resumes as soon as the demand page translation arrives. In case of a prefetch buffer hit, CPU is given back the translation (and resumes), the entry is moved to the TLB, and a prefetch is initiated for the next translation in the background. A simplified hardware block diagram implementing SP is given in Figure 2.

2.2. Arbitrary Stride Prefetching (ASP)

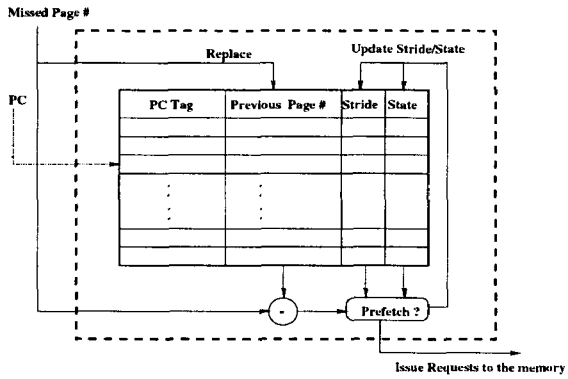

Figure 3. Schematic of Hardware for ASP

SP captures only spatial proximity, but there are several applications that have regular strided reference patterns. Prefetching mechanisms to address this have been proposed by Chen and Baer [8], Patel and Fu [13] and several others. It has been pointed out [29] that the scheme proposed by Chen and Baer is the most aggressive of these. We use this scheme, referred to as Arbitrary Stride Prefetching (ASP) in this paper, for comparisons. ASP uses the program counter (PC) to index a table (referred to in [8] as Reference Prediction Table (RPT)). Each row has one slot which stores a tuple containing (i) the address that was referenced the last time the PC came to this instruction, (ii) the corresponding stride, and (iii) a state (PC tag may also need to be maintained for indexing). The address field needs to be updated each time the PC comes to this instruction, and the prefetch is initiated only when there is no change in the stride for more than two references by that instruction (the state is used to keep track of this information). Such a safeguard tries to avoid spurious changes in strides. This is the mechanism that is evaluated in this paper, though there are several variations proposed [8]. A simplified hardware block diagram implementing ASP is given in Figure 3.

2.3. Markov Prefetching (MP)

The previous two are representative of schemes that attempt to detect regularity of accesses (by observing sequentiality or strides), and fail if there is no such regularity in the

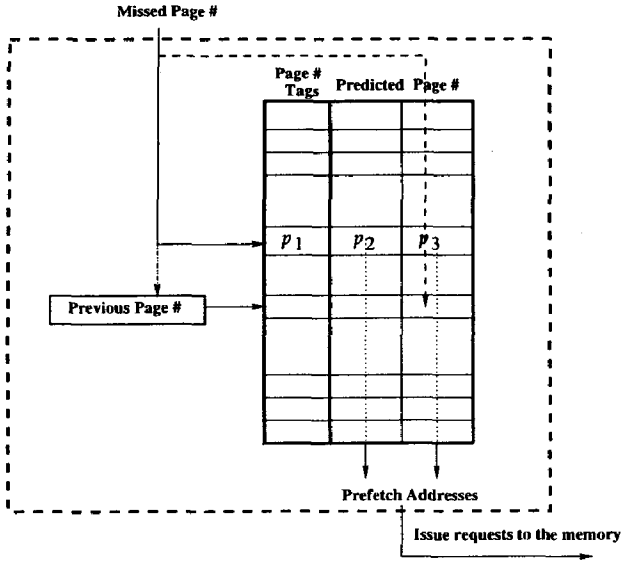

Figure 4. Schematic of Hardware for MP

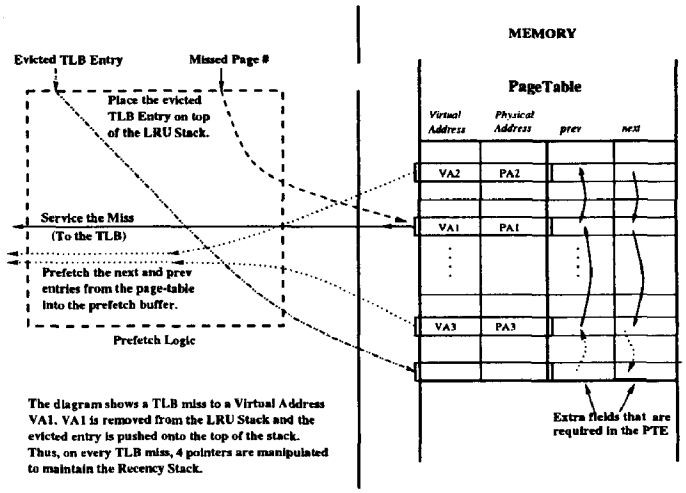

Figure 5. Schematic of Hardware for RP

differences between successive address references. However, it is possible that history repeats itself, even without any regularity in strides, and MP tries to address that angle. MP attempts to dynamically build a Markov state transition diagram with states denoting the referenced unit (pages in this context) and transition arcs denoting probability of needing the next page table entry when the current page is accessed. The probabilities are tracked from prior references to that unit, and a table is used to approximate this state diagram. This scheme was initially proposed for caches [16], and we have extended this to work with TLBs as discussed below.

The prediction table for MP is indexed by the virtual page address that misses. Each row of the table has s slots, with each slot containing a virtual page address that is initially empty (they correspond to entries to be prefetched when this address misses the next time). On a miss, this table is indexed based on the address that misses. If not found, then this entry is added, and the s slots for this entry are kept empty. In addition, we also go to the entry of the previous page that missed, and add the current miss address into one of its s slots (whichever is free). If all the slots are occupied, then we evict one based on LRU policy. As a result, the s slots for each entry correspond to different virtual pages that also missed immediately after this page. If a missed address hits in the table, then a prefetch is initiated for the corresponding s slots of this address. Since the table has limited entries, an entry (row) could itself be replaced because of conflicts. A simplified hardware block diagram of MP with $s = 2$ is given in Figure 4.

2.4. Recency Based Prefetching (RP)

While all the previous mechanisms have been proposed for caches, Recency Prefetching is the first mechanism, to our knowledge, that has been proposed solely for TLBs. This mechanism works on the principle that pages referenced at around the same time in the past will also be referenced at around the same time in the future. It builds an LRU stack of page table entries to achieve this. Specifically, when an entry is evicted from the TLB it is put on top of the stack, its next pointer is set to the previous entry that was evicted (whose previous pointer is set to this entry). As a result, each entry has two pointers, which are actually stored in the page table (this scheme requires considerably more space than the other schemes, and increases the page table size). When an entry is loaded on a miss, the prefetch mechanism fetches the entries corresponding to the next and previous pointers into the prefetch buffer in the hope that they will be needed as well (this is the mechanism that is implemented and evaluated here, though there is a variation in [26] with regard to prefetching some more entries). RP, thus, keeps its prediction information in the page table itself and does not have additional storage costs on-chip. This comes at the cost of an increase in page table size. Further details can be found in [26] and a hardware schematic of this mechanism is given in Figure 5.

2.5. Distance Prefetching (DP)

The advantage with SP and ASP is that they take very little space to detect patterns and initiate actions accordingly, while MP and RP can take considerably more space because they can detect more patterns than the restricted patterns that SP and ASP can detect. They also take a while to learn a pattern, since only repetitions in addresses can effect a prefetch for RP and MP (not first time references). Our DP mechanism can be viewed as trying to detect many of the patterns that RP or MP can accommodate (and maybe some that even they cannot), while benefiting from the regularity/strided behavior of an execution. In fact, if there is so much regularity that SP and ASP can do very well in a reference pattern, then DP should automatically take only as much space as these two. Remember that MP and RP need considerable space even to capture sequential scans while

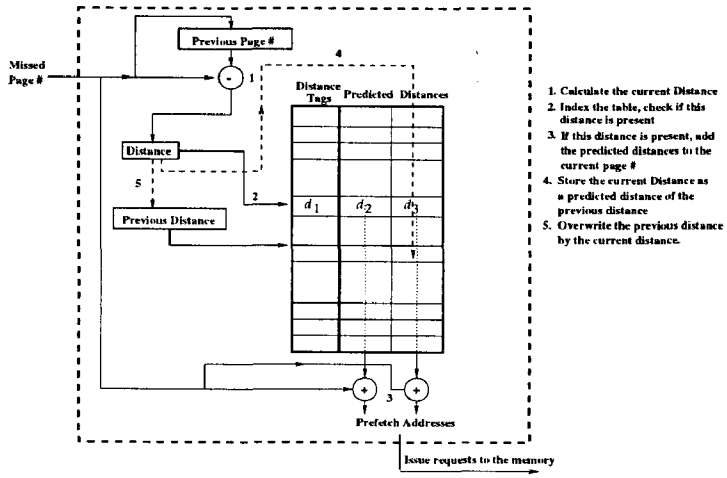

Figure 6. Schematic of Hardware for DP

SP and ASP can do this in little space.

DP works on the hypothesis that if we could keep track of differences between successive addresses (spatial separation or stride, which we call as *distance* for this mechanism) then we could make more predictions in a smaller space. For instance, let us say that the reference string is 1, 2, 4, 5, 7, and 8. Then, if we just keep track of the fact that a distance of "1" is followed by a (predicted) distance of "2" and vice versa, then we would need only a 2 entry table to make a prediction as opposed to the markov mechanism where an entry is needed for each page (6 entries in this example). This is exactly what our distance prefetching mechanism does. A reference string touching all pages sequentially (that SP optimizes) can be captured by DP using an entry saying distance of "1" is followed by a (predicted) distance of "1". The reader is referred to [19] for several such reference string examples that show how DP can provide better predictions than the other schemes.

The hardware implementation (Figure 6 shows the schematic with $s = 2$) for DP requires that the table be indexed by the current distance (difference of current address and previous address). Each entry has a certain number of slots (maintained in LRU order) corresponding to the next few distances that are likely to miss when the current distance is encountered (similar to how MP keeps the next few addresses based on the current address). Pages corresponding to the distances in these slots are prefetched when this virtual address misses. One could, perhaps, envision indexing this table using the PC value together with the distance, or using a set of consecutive distances. These are issues that could be investigated in future research, and are not discussed in this paper.

2.6. Review of Hardware Requirements

Table 1 gives a quick review of the above description by comparing the schemes in terms of the hardware requirements and functionality. ASP usually subsumes SP, and we do not show SP separately here or in the experimental results. For the ASP, MP and DP mechanisms, we uniformly use a parameter r to study its effect on the resulting performance as mentioned earlier. The previously proposed RP mechanism, keeps information (2 pointers) in each entry of the page table. Since the number of virtual pages is usually quite large, the space taken by RP considerably dominates over the much smaller r (32 to 1024 rows) that we consider for ASP, MP and DP. The only benefit for RP in this regard is that the storage is in main memory, while the other three require on-chip real-estate. These two pointers for RP refer to the previous and next pointers of the LRU stack. Both MP and RP, index the information based on the page number that misses in the TLB, and DP indexes using the current distance (stride). ASP, on the other hand, indexes using the PC value. In ASP, MP and DP, the corresponding tag information (of the indexing field) needs to be maintained to ensure the corresponding match since more than one entry can map on to a single row. There is, thus, not a significant difference in storage requirements across the schemes for a single row.

ASP, MP and DP, have all the necessary information to initiate a prefetch on-chip, and thus need not incur any additional memory references. On the other hand, removing the page table entry that is currently required and pushing the evicted entry on top of LRU stack requires manipulating four pointers in RP. This can become an issue in increasing memory traffic, thus interfering not only with other prefetch actions but also with normal data traffic.

The maximum number of prefetches that can be initiated on a miss for MP and DP depend on the chosen s values. This is, typically, quite small (around 2-4) that is not only shown to be a good operating point later in this paper, but has also been pointed out by [16] for MP. ASP, as defined in [8], prefetches the address incremented by the corresponding stride. RP prefetches entries on either side of the LRU stack upon a miss, and there is also a version discussed in [26] that prefetches three entries. It should be noted, that the number of prefetches that are initiated is not necessarily indicative of the performance of the scheme. Eventually, the prefetches are put in the (small) prefetch buffer, and a more aggressive scheme can end up evicting entries before they are used.

3 Performance Evaluation

3.1. Experimental Setup

We have conducted an extensive evaluation of the prefetching mechanisms for a wide variety of applications spanning several benchmark suites. Our evaluations use all 26 applications from SPEC CPU2000 [11], 20 applications from MediaBench [21], 5 applications (`bcc`, `mpegply`, `msvc`, `perl4`, and `winword`) from the Etch traces [1] and 5 applications (`anagram`, `bc`, `ft`, `ks` and `yacr2`) from the Pointer Intensive Benchmark suite [2]. In all, we have considered 56 applications that we hope are representative enough of realistic scenarios. The MediaBench ap-

	ASP	MP	RP	DP
How many rows?	r	r	No. of PTEs	r
What are the contents of a row ?	PC Tag, Page #, Stride and State	Page # Tag 2 Prediction Page #s	*next, prev* pointers	Distance Tag, 2 Prediction Distances
Where is the table?	On-Chip	On-Chip	In Memory	On-Chip
How is the table indexed?	PC	Page #	Page #	Distance
How many memory system operations per miss? (*excluding prefetching*)	0	0	4	0
How many prefetches can be initiated?	1	2	1-3	2

Table 1. Comparing the Hardware Issues of the Schemes at a glance. s is assumed to be 2 for MP and DP. PC Tag, Page # Tag, and Distance Tag for ASP, MP and DP respectively are needed for tag comparison when indexing/looking up the table.

plications are characteristic of those in embedded and media processing systems, and the Etch applications are characteristic of desktop/PC applications. The Pointer Intensive suite helps us evaluate the mechanisms for non-array based reference behavior, which can be more irregular. The SPEC 2000 applications are really long running codes and it is extremely difficult to simulate all of them completely, as has been pointed out by others [6, 23]. In this paper, we fast forward (skip) the first two billion instructions of their execution, and present results for the subsequent one billion instructions. The simulations have been conducted using SimpleScalar [5], using the default configuration parameters. Most of the simulations are conducted using sim-cache since we are mainly interested in the memory system references, and the prediction accuracies of the schemes. We also present one set of execution cycle results for one billion instructions with five of the applications with high TLB miss rates to compare DP and RP using sim-outorder (as can be imagined, these experiments take an excessively long time). The MediaBench, Etch and Pointer Intensive suite were simulated using Shade [10]. Though it is also important to consider the effect of the OS, the evaluations are only for application behavior in these results as in the earlier study [26].

We consider different TLB configurations - 64, 128 and 256 entries that are 2-way, 4-way and fully associative, and different values for prefetch buffer size (16, 32 and 64 entries). We have also varied the s and r values for the prediction table configurations of the mechanisms. We present representative results using 128 entry fully associative TLB and 16 entry prefetch buffer, with a page size of 4096 bytes. The reader is referred to [19] for a more detailed sensitivity analysis of different TLB configurations, prefetching parameters and page sizes.

3.2. Comparing the Schemes

In our first set of evaluations, we compare RP, MP, DP and ASP (compared qualitatively until now) with the 56 workloads in Figures 7 and 8. Since MP, DP and ASP predictions depend largely on the size of the prediction table that is allowed, we have varied r (the number of entries) as 32, 64, 128, 256, 512 and 1024. Further, we have allowed the corresponding tables to be indexed as direct-mapped (D), set associative (2 and 4 way) and fully associative (F). Since the graph becomes very difficult to read, we show results for DP and ASP only with direct-mapped (D) configurations. We show F, 2 and 4 way associativity influence only for MP. We would like to point out that the indexing mechanism for the prediction table (F, 2 or 4 way) has very little influence on the prediction accuracy in most cases (as one would infer from the bars for MP, and in the bars for DP later in section 3.3). In these graphs, the left-most bar for each application is for RP, following which is a gap and then the bars for MP, a gap, bars for DP, a gap and finally the bars for ASP. In some cases, the bars are either completely or partially absent because the prediction accuracy is close to 0.

These mechanisms are compared in terms of their prediction accuracy, which has been the metric used in earlier research [26] to argue the capabilities and potential of TLB prefetching. Prediction accuracy is defined as the percentage of TLB misses that hit in the prefetch buffer at the time of the reference. Accuracy is an important concern since it has a direct bearing on the amount of stall time incurred by the CPU during a TLB miss. Uniformly, a prefetch buffer of size $b = 16$ entries is used in all these experiments. Remember, that *a mechanism which fetches more aggressively can evict entries from this buffer before they are actually used for the translation* (and will consequently have an effect on the prefetch accuracy). One can in fact observe this effect with ASP, when the prediction accuracy decreases for a more aggressive $r = 1024$ entry table (compared to smaller prediction table sizes) in some applications like `apsi`, `ft` and `wupwise`.

There are applications such as `facerec`, `galgel`, `art`, `gap`, and `mesa` where nearly all mechanisms give quite good prediction accuracies. In these applications, there are regular strided accesses that repeatedly go over the items already accessed in the same regular fashion. Consequently, both stride-based predictions (ASP) and history-based predictions (RP and MP) do a fairly good job of pre-

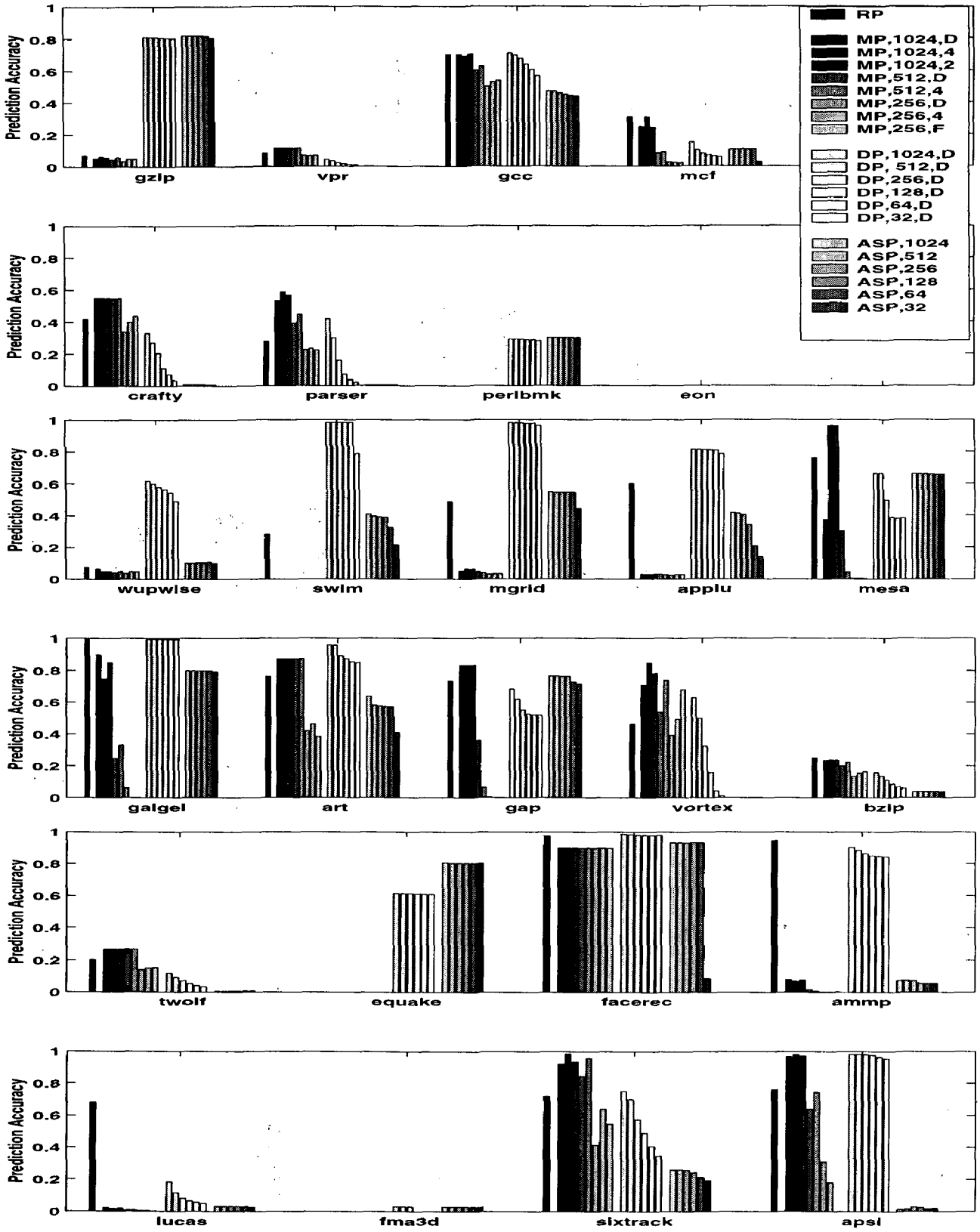

Figure 7. Prediction Accuracy of different Prefetching mechanisms for all the SPEC CPU2000 Applications

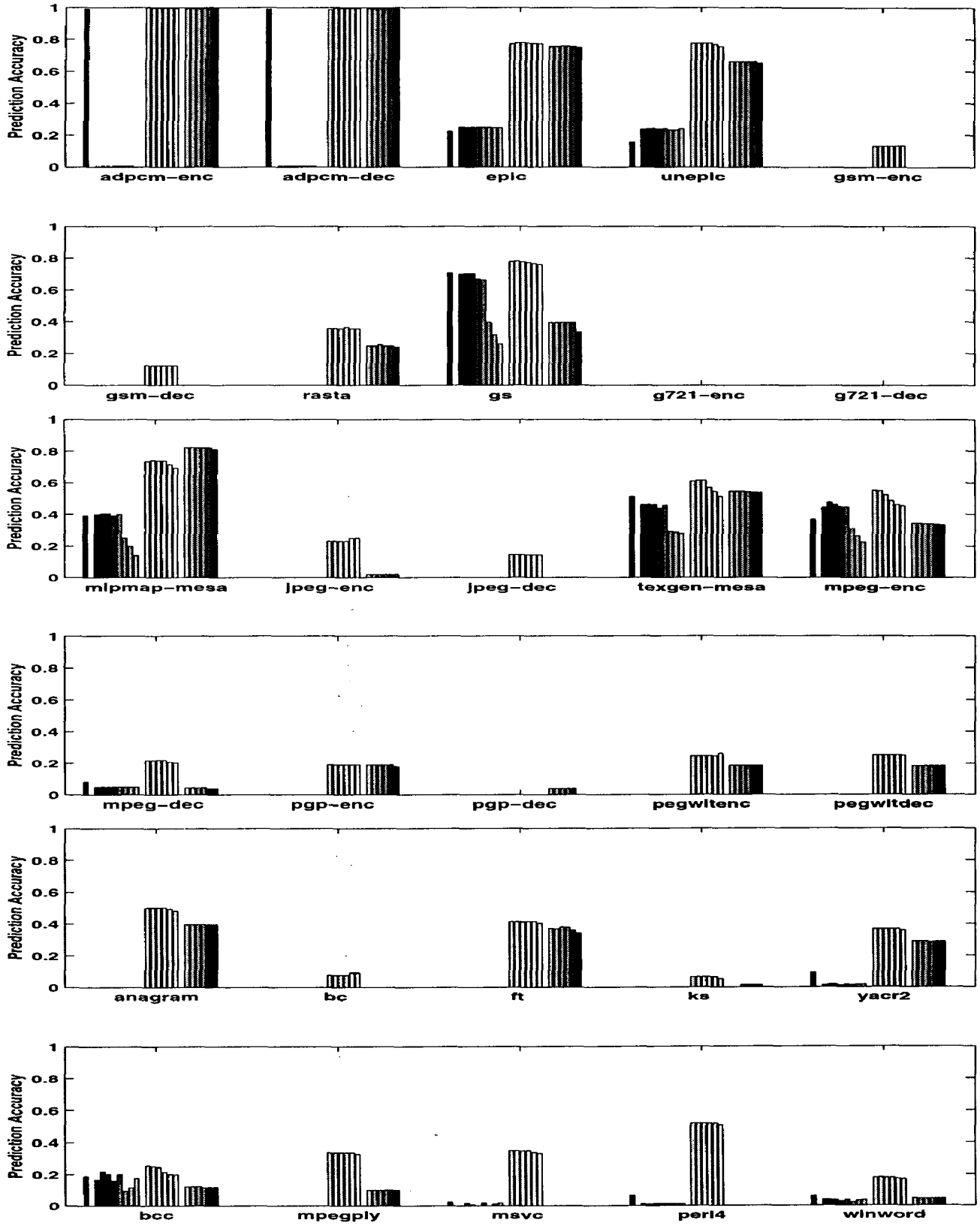

Figure 8. Prediction Accuracy of different Prefetching mechanisms for Mediabench, Etch and Pointer Intensive benchmark Suites. Legends are same as in Figure 7.

dicting the future. The only exception is that in some cases (such as galgel, art, mesa) MP performs poorly with small r. Since these are quite large datasets, keeping the history for all the references needs considerably more space, and small tables are not adequate for this purpose. RP, on the other hand, builds the history in memory and is not limited by on-chip storage as in MP. We find that our DP mechanism gives good prediction accuracies, being able to capture the strided patterns, without requiring the higher space requirements of MP to maintain history. Even a $r = 32$ predictor table for DP, gives very good predictions. In the following discussion, we go over each mechanism pointing out where it does the best and when it does not do as well.

Apart from the above five where all mechanisms give good performance, we find RP giving the best, or close to the best performance for applications such as gcc, crafty, ammp, lucas, sixtrack, apsi, adpcm-enc/dec, gs, and texgen. These applications have good repetition of history, i.e. the next reference after a given address is very likely to remain the same the next time we come to this address again.

MP gives the best or close to best performance for many of the applications that RP does very well. However, as was pointed out a little earlier, sometimes the history information that needs to be maintained can get quite long, and this can lead to poor predictions for small tables (such as $s = 32$). In some applications, where past history is a good indication of the future (i.e. RP does very well) such as in adpcm-enc/dec, MP performs very poorly for this very reason. RP is able to track history for all addresses since it keeps the information in memory, but MP does not have that luxury and may have to keep evicting its table entries from the on-chip storage. There are some applications such as parser and vortex where MP does better than even RP despite this downside. The possible reason is that RP can look at only what happened at this address the previous time the program came to it, while MP can possibly keep track of what happened the last few times (depending on the s value of its table). In these applications, it is possible that there is alternation (i.e. a sequence such as 1, 2, 3, 4, 1, 5, 2, 6, 3, 7, 4, 8, 1, 2, 3, 4, ... would do better with MP than RP for $s = 2$) in history that is leading to this behavior (this is also the reason ASP does not do well for these applications).

ASP does very well in many of the applications that are suited to RP and MP such as facerec, galgel, art, gap and mesa, and also in some where RP does better than MP (adpcm-enc/dec and texgen). The regularity in strides in these applications help this mechanism provide good accuracy. This regularity also helps ASP capture many of the first time reference predictions that history based mechanisms are not very well suited to, as in gzip, perlbmk, equake, epic/unepic, mipmap, pgp-enc/dec, anagram, and yacr2. The working sets are much smaller in some of the non-SPEC 2000 applications, and cold misses do become prominent for these. On the other hand, there are applications such as crafty and parser where the accesses are not strided enough for ASP to perform well, but historical indications can give a much better perspective of future behavior for RP and MP.

Moving on to DP, we find that it gives very good prediction accuracies in several cases. DP comes very close to RP or MP in several applications where history-based predictions do the best such as gcc, mesa, galgel, gap, parser, and ammp. On the other hand, if history is not a good indication (or has not established) but strides are more determining (as in gzip, adpcm-enc/dec, mipmap, and perlbmk where ASP does very well), DP is able to deliver as good accuracies as ASP. Beyond coming close to the better of history or stride based schemes, there are several applications such as wupwise, swim, mgrid, applu, mpeg-dec, bc, mpegply, msvc, and perl4 where DP does much better than the others. In fact, for gsm-enc/dec, jpeg-enc/dec, ks, msvc and bc, DP is the only mechanism which makes any noticeable predictions (even if the accuracy does not exceed 20%).

We would like to point out, that there are a few applications such as eon, fma3d, g721-enc/dec and pgp-dec where none of the mechanisms are able to make any significant predictions. Many of these applications (eon, g721-enc/dec, pgp-dec, bc, ks) have so few TLB misses that a significant history does not build up nor does a strided pattern (and *TLB prefetching is not as important for them anyway*). In fma3d, the irregularity makes it very difficult for any mechanism to do well, and this motivates the need for further work on prefetching mechanism.

In summary, DP gives very good predictions for many of the applications. In fact, it provides the best or within 10% of the best prediction accuracy in 39 (and best in 36) of the 56 applications considered (the others are less than half this number). DP does well for regular and irregular applications, and applications that have strided and/or history-based access patterns. Another important point to note is that *DP can provide such good predictions with just a 32-256 entry prediction table*, compared to the others (MP and ASP) which may need many more entries, nor requiring the considerable storage and memory bandwidth taken by RP. Examining only the miss stream from the TLB, and not the actual reference stream (which to a certain extent can be viewed as a case in favor of RP because there is an implicit LRU tracking within the TLB) does not seem to penalize DP in any significant way.

DP also turns out to be the best in terms of the average prediction accuracy that was calculated over all the benchmarks($(\Sigma\ p_i)/n$) for each scheme. From the second column in Table 2, we can see that DP and RP take the first and second places respectively. One could argue, that *it is important to not just provide good accuracies for all applications, but to those where it really matters* (i.e. the higher TLB miss rate incurring applications). To capture this effect we present the weighted average ($(\Sigma\ (m_i \times p_i)/(\Sigma\ m_i))$ of the prediction accuracy (i.e. the accuracy p_i for each benchmark is weighted by the corresponding TLB miss rate m_i) for the schemes in the third column of Table 2. As we can see, RP comes out a little in front (around 5% better) of DP in this case because a long history helps a select set of ap-

Prefetching Scheme	Average $(\Sigma\ p_i)/n(=56)$	Weighted Average $\Sigma\ (m_i \times p_i)/(\Sigma\ m_i)$
DP	0.43	0.82
RP	0.29	0.86
ASP	0.28	0.73
MP	0.11	0.04

Table 2. Table showing the average and weighted average of prediction accuracy for the prefetching schemes which was calculated using the miss rates(m_i) and prediction accuracies(p_i) over all the 56 applications. $s = 2$ and $r = 256$ for DP, MP and ASP.

plications with very high miss rates (even though DP does better in a majority of applications). However, this comes at a higher storage cost in memory, as well as higher memory traffic. Consequently, the rest of this subsection gets into greater detail comparing DP with RP, in terms of performance implications of these prediction accuracies, particularly for the applications with higher TLB miss rates.

Comparing DP with RP in greater Detail: Having compared the prediction capability of the mechanisms using all the applications and all the different configurations, we specifically focus on 8 applications (`galgel`, `adpcm-encoder`, `ammp`, `mcf`, `vpr`, `twolf`, `lucas`, `apsi`) which have the highest TLB miss rates (0.228, 0.192, 0.0113, 0.090, 0.016, 0.013, 0.016, and 0.018 respectively) for a 128 entry fully associative TLB amongst all these applications [18]. Of these 8 chosen applications, RP provides better accuracy than DP for 5 applications - `vpr`, `mcf`, `twolf`, `ammp` and `lucas`. Further, RP is the only other prefetching mechanism explored for TLBs, and we would like to show some of the trade-offs that DP provides over RP despite slightly lower prediction accuracies in these 5 applications (which is what tilted the balance in favor of RP in Table 2).

RP requires as many as 6 possible memory system references upon a TLB miss. While the CPU resumes computation as soon as the miss is serviced, there are other references needed to maintain the LRU stack. If the item was in the middle of the stack, then it needs to be removed (taking 2 references), and the evicted item needs to be put on top (taking 2 references). After this, the actual prefetching can proceed (since it prefetches on either side of the removed item, this takes 2 more references). On the other hand, DP references memory only to bring in the s (which is 2 here) predicted entries, i.e. DP does not need to update any state information in memory. It is conceivable, that some or all of these references in both these schemes can be serviced from the cache. However, in the following discussion we model these as actual memory references.

To study the impact of the additional memory traffic imposed by RP and DP, we conduct a simple experiment using SimpleScalar, wherein we use its memory system model to account for the overheads associated with the prefetch operations. It should be noted that in this examination, the prefetch memory traffic does not contend with the normal data traffic, but only with other prefetch traffic (this in fact, *is a more biased model that favors RP over DP*). When the CPU incurs a TLB miss, and does not find the data in the prefetch buffer, but the prefetch for that entry has already been issued, it is made to stall until the entry arrives. Further, if a prefetch needs to be issued on a TLB miss, this memory loading operation will be impacted by any prior issued prefetch memory transactions (such as the pointer manipulations for RP, or the actual prefetching of entries for DP and RP). One other issue where we give the benefit of doubt for RP in its implementation is that, if there is a TLB miss soon after the previous one (and not for the same entry) and the prefetching initiated earlier is not complete, we only wait for the LRU stack to get updated and do not prefetch those items at that time (this is as though there was a wrong prediction, but we are not going to incur the corresponding memory traffic in fetching the nearby entries at that time). In this case, there would be only 4 memory transactions instead of 6. These applications are run using sim-outorder (with a 4 issue width) to account for actual CPU cycles. The prefetching and state maintenance (for RP) operations are treated as cache misses and need to be serviced from main memory with a cost of 50 cycles. A constant TLB miss penalty of 100 cycles is assumed.

We present the results from this experiment in terms of the cycles taken for execution of the programs (normalized with respect to no prefetching for the billion instructions considered) as shown in Table 3. The results are presented for the five benchmarks where RP has better accuracy over DP (as can be expected, for the rest, DP automatically provides better performance than RP).

	RP	DP
ammp	0.97	0.86
mcf	1.09	0.95
vpr	0.99	0.98
twolf	0.98	0.98
lucas	1.00	0.99

Table 3. Comparing DP with RP: Normalized execution cycles(w.r.t. no prefetching) for RP and DP for 1 billion instructions after the first 2 billion instructions. $s = 2$ and $r = 256$ for DP.

We find that despite the slightly higher prediction accuracy that RP provides for these applications, DP still comes out in front when considering execution cycles (one can also see the execution time savings, that is more significant in `ammp`, with prefetching compared to the execution without any prefetching in place). This is because RP generates much more memory traffic ranging from anywhere between 2-3 times that for DP [19]. As was pointed out, DP

gives fairly good predictions even with $r = 32$ which incurs even lower traffic. It should be remembered that in this simulation, we are in fact more biased towards RP, since the prefetch traffic does not interfere with the normal data traffic, and consequently a more realistic model would favor DP further.

3.3. Sensitivity Analysis of DP

The impact of several parameters such as table configuration (r), table associativity (D, 2, 4, F), number of prediction entries (s), prefetch buffer size (b), TLB configurations and page size on the effectiveness of DP has also been studied. Some of these results are shown in Figure 9 and the reader is referred to [19] for further results/details. In general, we found that DP is fairly insensitive to many of these parameters, and even a small direct-mapped 32-256 entry table suffices to give very good predictions.

4 Concluding Remarks

There is a plethora of related literature [29] on prefetching mechanisms that try to examine patterns in reference behavior to predict references for the near future. Nearly all of them have been proposed to alleviate latencies in the memory hierarchy by prefetching blocks into the cache or for prefetching data from I/O devices. However, there has been only one prior study [26] that has proposed, evaluated and demonstrated the benefits of prefetching entries for the TLB. However, the suitability and associated benefits of the previously proposed prefetching mechanisms for caches has not been examined for prefetching TLB entries until now.

Prefetching mechanisms usually try to detect strided behavior or history-based behavior to make their predictions. With the former, one can make good predictions with very little space, whenever there are regular/strided reference patterns. However, they may not do a good job when there is no such regularity. History-based predictions can do better than stride-based predictors, albeit at a higher storage cost, when previous references can give a good indication of what to expect next (even when references are not strided).

In this paper, we have presented a new mechanism called *Distance Prefetching* (DP), that can automatically provide strided predictions when there is such behavior, and becomes more history-based when there is not. It exploits the fact that even if there is variability in strides, there is probably a pattern to this variability itself and the past information on such variability can help make future predictions. While DP is a fairly generic mechanism, that can possibly be used in the context of caches, I/O etc., in this paper we have specifically evaluated it for TLBs.

Considering representative examples from stride-based (Arbitrary Stride Prefetching) and history-based (Markov Prefetching) predictors, we have presented a qualitative and quantitative comparison of our DP mechanism for TLB prefetching with these earlier proposals. In addition, we have also evaluated our mechanism with the only other proposal - RP - for TLB prefetching which has been shown to improve TLB performance significantly. We find that DP gives better prediction accuracy than the others in many of the applications, and in fact, DP gives the best or close to the best prediction in 39 of the 56 considered applications. Even in the applications where we found RP to be a little better in terms of prediction accuracy, we demonstrated that DP comes out in front in terms of execution cycles.

DP can operate fairly well with a small direct-mapped prediction table of 32-256 entries, with most of the results quite insensitive to a wide spectrum of table and indexing parameters. Further, the prediction accuracies are quite good even with a 16 entry prefetch buffer. DP is able to make good predictions across different TLB configurations and page sizes as well.

The contributions of this paper are in: (a) the novel mechanism - Distance Prefetching - that can be used to predict application reference behavior using a relatively small space (which can possibly be used in the context of different levels of the storage hierarchy - TLBs, caches, I/O), (b) adaptation and qualitative comparison of this mechanism and others previously proposed for caches to the domain of TLB prefetching, (c) a detailed application-driven evaluation of all these mechanisms using a wide spectrum of public-domain benchmarks to show the benefits of distance prefetching, and (d) identifying the parameters for a distance prefetcher implementation, based on a sensitivity analysis. Our ongoing work is examining issues about using other information (PC, several previous distances, etc.) within the context of this new mechanism, and evaluating its benefits for other levels of the storage hierarchy (caches and I/O). We are also investigating prefetching issues in a multiprogrammed environment (flushing/switching the prefetch tables), the effect of the OS and the effect of superpaging.

References

[1] Etch traces. http://memsys.cs.washington.edu/memsys/html/traces.html.

[2] Pointer-intensive benchmark suite. http://www.cs.wisc.edu/ austin/ptr-dist.html.

[3] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, California, April 1991.

[4] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation*, pages 243–253, 1994.

[5] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.org.

[6] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. October 2001. http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/.

[7] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th*

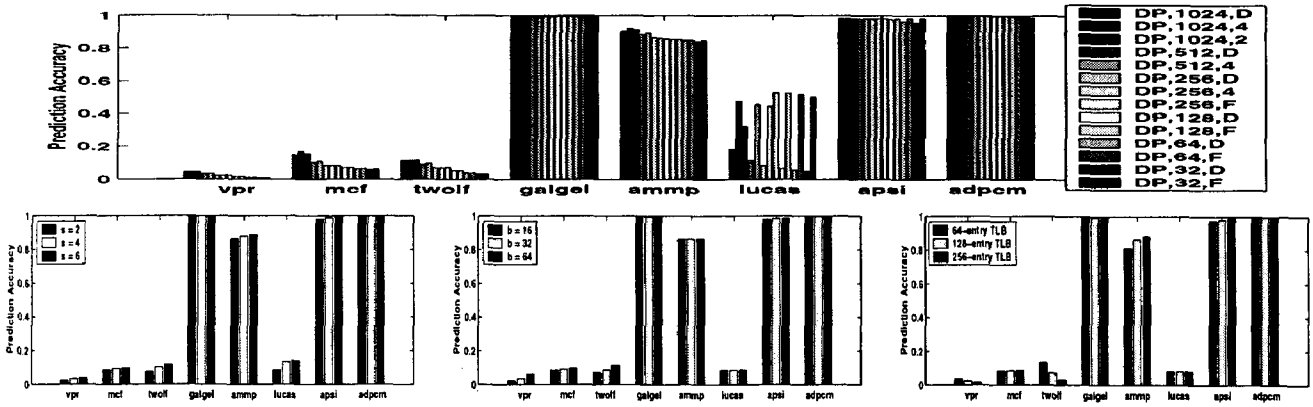

Figure 9. Sensitivity of DP to Hardware Parameters

Annual International Symposium on Computer Architecture, pages 114–123, 1992.

[8] T. Chen and J. Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[9] D. W. Clark and J. S. Emer. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1), 1985.

[10] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[11] S. P. E. Corporation. http://www.spec.org.

[12] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. *International Conference on Parallel Processing*, pages 56–63, August 1993.

[13] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th MICRO*, pages 102–110, 1992.

[14] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, May 1993.

[15] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating System*, pages 295–306, 1998.

[16] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. *IEEE Transactions on Computer Systems*, 48(2):121–133, 1999.

[17] N. Jouppi. Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, 1990.

[18] G. B. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2002.

[19] G. B. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. Technical Report CSE-01-032, Dept. of Comp. Sci. & Eng., Penn State Univ., November, 2001.

[20] D. Koppelman. Neighborhood prefetching on multiprocessors using instruction history. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[21] C. Lee, M. Potkonjak, and W. Magione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997. http://www.cs.ucla.edu/ leec/mediabench/.

[22] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design Tradeoffs for Software Managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 27–38, 1993.

[23] A. K. Osowski, J. Flynn, N. Meares, and D. J. Lilja. *Adapting the SPEC2000 Benchmark Suite for Simulation-based Computer Architecture Research*. Kluwer-Academic Publishers, 2000. (papers from Workshop on Workload Characterization).

[24] J. S. Park and G. S. Ahn. A Software-controlled Prefetching Mechanism for Software-managed TLBs. *Microprocessors and Microprogramming*, 41(2):121–136, May 1995.

[25] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[26] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 117–127, June 2000.

[27] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using Superpages backed by Shadow Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, 1998.

[28] M. Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. PhD thesis, Dept. of CS, Univ. of Wisconsin at Madison, 1995.

[29] S. VanderWiel and D. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 1999.