Analysis of Hardware Prefetching Across Virtual Page Boundaries

Ronald G. Dreslinski, Ali G. Saidi, Trevor Mudge, and Steven K. Reinhardt[*]
{rdreslin,saidi,tnm,stever}@eecs.umich.edu
Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
2260 Hayward Ave
Ann Arbor, MI 48109-2121

ABSTRACT

Data cache prefetching in the L2 is at the forefront of prefetching research. In this paper we analyze the impact of virtual page boundaries on these prefetchers. Conservative measurements on real hardware show that 30-50% of consecutive virtual pages are mapped to pages which are not consecutive in physical memory. Advanced hardware prefetching techniques that detect access patterns which span virtual page boundaries often end up prefetching data that is from the wrong physical page. Meanwhile, current simulation techniques for evaluating prefetching algorithms assume that all virtual pages are mapped consecutively. We show that not accounting for virtual page boundaries in simulation can lead to overestimates of as much as 29% (9% on average). We also show that a simple prefetch filter can improve performance up to 32% (7% on average) and recover the overestimated performance. This leads to the conclusion that although previous simulations may not have accounted for virtual page boundaries, the results they demonstrate are still attainable and that it is not necessary to simulate virtual page boundaries to get accurate results. However, actual hardware designers should take care to implement a simple filter or else their hardware may not show the same gains in performance as they did in simulation.

Categories and Subject Descriptors:
B.3.2 [Design Styles]: Cache memories, Virtual memory

General Terms:
Design, Performance, Measurement

Keywords:
Prefetching, Virtual Memory

[*]Also with Reservoir Labs, Inc.

1. INTRODUCTION

The speed gap between processors and memory has long been a focus of computer architects. Throughout the years many researchers have explored ways to mitigate the long latency of cache misses to improve performance. Due to the excessive cost of fabricating a chip in order to evaluate a design, researchers are left to simulate their ideas. Although simulation speeds the design process, it also creates potential for researchers to design hardware that is unrealizable in actual gates, and worse some shortcuts used in the simulator infrastructure can produce incorrect results. Perez showed in his MicroLib paper [12] that the design of SimpleScalar [2], an often used simulation environment in the academic community, assumes the presence of an infinite number of MSHRs within the cache. This neglecting of the MSHRs in simulation resulted in performance differences of greater than 15%.

Furthermore, many simulation tools also neglect to simulate the entire system including an OS and for simplicity just emulate the syscalls that applications make. Neglecting these aspects can lead to results showing performance gains that may not be realizable in a real system. In this paper we focus on one such full system aspect, address translation, and its effects on the results of hardware prefetching.

Recently, proposed advanced hardware prefetchers [10, 11] have been introduced to handle data prefetching in the L2 cache. As these new frontiers are explored in memory system architecture it is important that we have accurate simulation techniques to evaluate the actual performance gains that these prefetchers can provide. These recent prefetching papers use a simulator to evaluate their design that simply maps the virtual address space of the process to a flat consecutive physical address space. The simulator does this for simplicity; as there is no operating system and only one application is running there is no need to support most aspects of a real virtual memory system. Additionally, in a simulator it is very easy to something that isn't done in a real system such as giving the second level cache access to the TLB to do address translation or sending the virtual and physical address to the L2 cache. By doing so in a simulator an L2 could be using virtual addresses to calculate prefetches when in a real system the L2 cache is forced to use physical addresses. The flat memory mapping coupled with caches not having translation capabilities leads to the simulation environment producing incorrect results.

It is possible that a prefetch address outside the current physical page can be calculated, and with the naive approach of SimpleScalar memory mapping, it will result in these page crossing prefetches to get the intended data. Under ideal circumstances the kernel would like to map consecutive virtual pages to consecutive physical pages, however that option is not always available. We measured a real system and found that the frequency of consecutively mapped virtual pages ranged on average from 50-70%. In addition to the pages that are mapped non-consecutively other virtual addresses may not be mapped at all, and prefetches to these pages would also fail. These prefetches that correspond to incorrect addresses can degrade the performance of the machine in three ways:

- They occupy space in the prefetch queue, causing evictions of potentially useful prefetches.
- They consume bus bandwidth that could have been used for successful prefetches.
- They pollute the cache with useless data.

In order to measure the performance impact of these three effects we implemented several prefetching techniques in the M5 [1] simulator. In our study we found that with a stride prefetcher [4] performance can be degraded as much as 5% (2.2% on average) when assuming that no pages are mapped consecutively over the case where we assume that all pages are mapped consecutively. More aggressive prefetchers such as the Global History Buffer [10, 11] are much more sensitive to the number of consecutively mapped pages where performance can be degraded by as much as 29% (9% on average). Filtering out prefetches that cross page boundaries before they are inserted into the prefetch queue can improve performance by as much as 32% (7% on average) compared to a system that assumes no pages are mapped consecutively. Thus, although previous simulations neglect virtual page boundaries, the results are still attainable in real hardware if hardware designers add in a simple prefetch filter.

Creating a simulation environment in a full system simulator – one that runs real operating systems codes – that encompasses the steady state behavior of the virtual to physical page mappings is non-trivial. This is mainly due to full-system simulators booting and then immediately running the benchmark in question. Since the system has not been running for a while the memory map is clean and organized with large contiguous regions of free space. Simulating the system long enough to get the memory map into steady state would require enormous amounts of time. This work demonstrates that it is not necessary to model the full system to get accurate results for L2 data cache prefetchers as long as a simple prefetch filter is used.

The following sections give some simple background in prefetching techniques and address translation, discuss our simulation methodology, present our results, and summarize our conclusions.

2. PREFETCHING BACKGROUND

In order to demonstrate the effects of address translation on prefetching techniques, several different prefetching techniques were implemented and evaluated. The following section gives some background on these prefetching techniques.

2.1 Hardware vs. Software Prefetching

There are two distinct types of prefetching, hardware and software. With software prefetching the compiler or programmer explicitly uses an instruction to try to prefetch the data before it is necessary. In the case of hardware prefetching the cache(s) use cache miss information to try to predict what additional blocks will be needed and then try to prefetch them. This paper is focused entirely on hardware prefetching because the issue surrounding address translation has a large impact in that domain. Due to a number of recent research papers focused on second level cache (L2) data prefetching [10, 11, 4, 8], this is the type of prefetching we have focused on for this study.

2.2 One Block Ahead Prefetching

One block ahead prefetching is one of the earliest forms of prefetching [7]. On a miss in the cache a prefetch is issued for the next block. Variations of this technique have been used such as n-block ahead prefetching, which fetches the next n blocks after a miss. The prefetcher exists simply as a queue of prefetch addresses that are issued when there are no demand misses. With one block ahead prefetching the miss rate can be reduced by up to 50% for sequential accesses.

2.3 Tagged Prefetching

Tagged prefetching is a simple extension of the one block ahead prefetching technique. With tagged prefetching, when a block is placed in the cache that was the result of a prefetch it is tagged. If a demand access hits in the cache on a block that is tagged it signals the prefetcher that a prefetched block was accessed and the tag on the block is cleared. By doing this, when a prefetched block is hit in the cache it will generate a prefetch for the next block. This technique improves on the one block ahead prefetching scheme by allowing up to a 100% reduction in miss rate after the first miss in a sequential access pattern. This 100% reduction does not account for partial prefetches. A partial prefetch is when a demand miss for a prefetched block reaches the cache while the prefetch is still in the prefetch queue or has not returned the response from memory yet.

2.4 Stride Prefetching

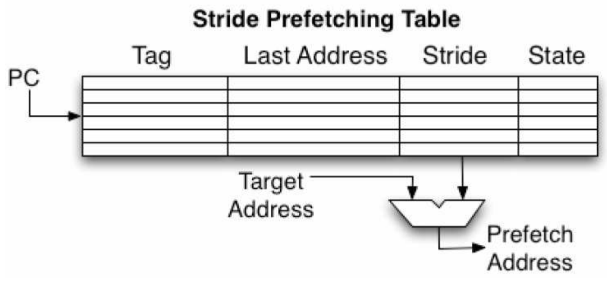

Figure 1: Stride Prefetcher[11]

Stride prefetching was originally proposed by Chen and Baer [3]. It was later expanded on by others [4, 8]. This technique detects the access stride of load instructions and prefetches accordingly. It also uses the tagged technique to improve performance. It is implemented as a table of entries. Each entry stores an address (PC) of the instruction it is prefetching for, the last memory address accessed, the last stride, and a confidence indicator. See Figure 1 for a diagram. On a miss (or a tagged hit) the entries are checked for an instruction address match. If a match occurs, the stored value for the last memory address is compared with the current address to calculate the current stride. A prefetch is then issued by calculating the current address plus that stride. The entry is then updated with the current address and stride. The entry also adjusts its confidence by comparing the last stride and the current stride. If the strides were the same it increases its confidence, otherwise it decreases it. This confidence can then be used to identify stride buffer entries to evict. If instead the address does not match any entry, a new entry is allocated for the access by choosing an empty entry or one with a low confidence. Variations of this prefetcher have been proposed that order requests within the prefetch queue based on confidence, and other replacement algorithms have been investigated. For this paper we have recreated the prefetcher described above so that it could be verified with the work done by Perez [12].

2.5 Global History Buffer Prefetching

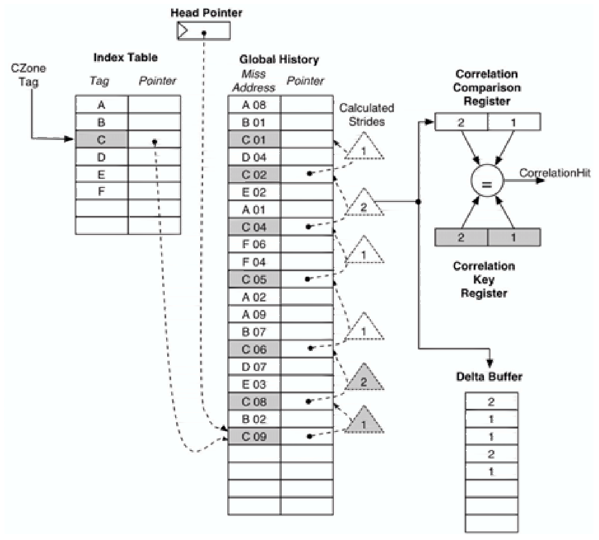

Figure 2: Delta Correlating GHB Prefetcher[10]

Global history buffer prefetching was introduced by Nesbit and Smith [11]. The mechanisms behind GHB involve two separate structures. The first is an index table (IT), and the second is the global history buffer (GHB). The IT uses a index to supply a pointer into the GHB. The GHB is a structure that contains some information about past misses with each node containing a link to the previous node with the same IT entry. In other words the GHB can be viewed as a set of linked lists, with the IT entry pointing to the heads of the linked lists. The IT/GHB can be configured in a number of ways to perform many different prefetching algorithms. For example stride prefetching is implemented by using the instruction address (PC) as the index into the IT, and the previous miss address is stored in each GHB entry. When indexing the IT with an instruction address, it returns a pointer to the first entry in the GHB that matches that address. The previous stride can be calculated by walking the linked list back one node. Nesbit [11] shows ways to create stride prefetching, and markov prefetching [6] using the GHB.

Nesbit [11] also presents a new form of prefetching called global delta correlation. In this technique they use the instruction address (pc) as the index to the IT and the miss address is stored in the GHB. When a prefetch is calculated, the GHB linked list is traversed backwards and the delta pattern is calculated until it matches the current delta pattern, at which point prefetches are issued according to the calculated pattern. Figure 2 shows a digram of how global delta correlation works. This is useful when accessing 2-D arrays when the stride will usually follow a distinct pattern. The other new form of prefetching Nesbit [11] proposes is a hybrid address correlation predictor that uses the instruction address of the miss (PC) as the index to the IT and again stores the misses target addresses in the GHB. Instead of calculating strides on a miss they walk the linked list backwards and issue prefetches for the target addresses that missed immediately following previous accesses from this instruction address (pc). The global address correlation is presented in Figure 3. In Figure 3 a miss on address A would generate prefetches for both B and C.

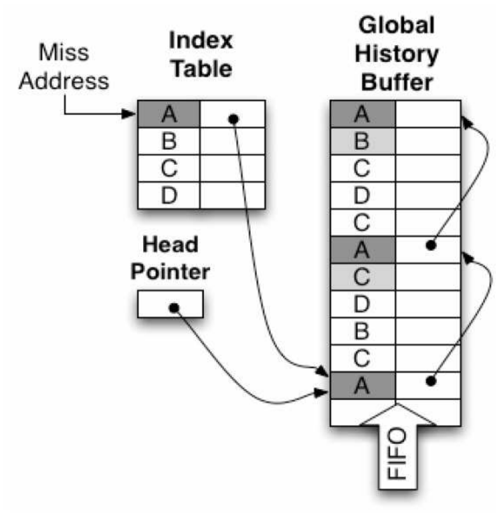

Figure 3: Address Correlating GHB Prefetcher[11]

3. EFFECTS OF PHYSICAL ADDRESSING ON PREFETCHING

This section briefly introduces virtual to physical translation, and then describes how improper simulation this can effect prefetching simulations.

3.1 Virtual to Physical Translation

In contrast to simulation systems such as SimpleScalar [2], real hardware runs multiple processes each with their own independent address space, but they share the same hardware. In order for each process to maintain its own address space and to provide a global address space, machines rely on address translation. Each process is allowed its own virtual address space, but it is possible that multiple processes have identical virtual addresses but they are in different address spaces. In order to maintain this, a mapping is kept of all virtual pages to their corresponding physical page. Before a load or store is issued to the memory system a translation step is performed to determine the physical address associated with the processes particular virtual address. Figure 4 shows a possible mapping of two processes, A and B, of virtual to physical addresses. Note that both processes have virtual addresses in the range 0x1000-0x4000, but each is mapped to a unique physical page. If process A were to access location 0x1100, the translation would return a physical address of 0x1100 since A's virtual page 0x1000 is mapped to physical page 0x1000. If on the other hand process B were to translate the address 0x1100, it would return 0x6100 because its virtual page 0x1000 is mapped to physical page 0x6000.

It is important to realize that as the kernel maps virtual pages to physical pages for a particular process it will sometimes map the pages to disjoint sections. See for example in Figure 4 how process A has virtual page 0x2000 mapped to physical page 0x2000, but virtual page 0x3000 is mapped to physical page 0x4000. This means that the two pages which were at consecutive virtual locations are not in consecutive physical locations. This will become important in Section 3.2.

The kernel also allows for virtual pages to reside in places other than physical memory, i.e. swap. This means that it is possible for some virtual pages not to have a physical page allocated (unmapped), and therefore present another case where a prefetch would fail to prefetch the correct data.

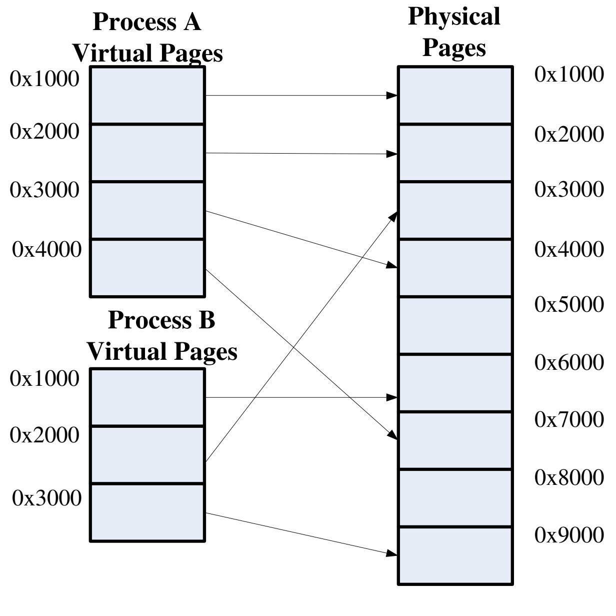

Figure 4: Virtual to Physical Mapping

3.2 Physical Addresses and Prefetching

As was discussed in Section 3.1 each process on a machine has it's own unique virtual address space. When this address space becomes mapped sparsely in the physical address domain, prefetching algorithms begin to break down at the page boundaries. Consider a stride prefetcher that calculates the next prefetch address to be the second line in the next virtual page. If that virtual page is mapped to a physical page disjoint from the current virtual page's physical page, then the prefetch will be grabbing possibly useless data. The same problem occurs if the next virtual page is unmapped.

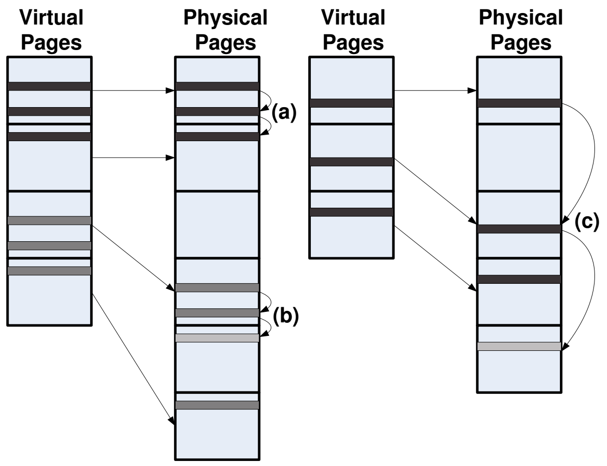

Figure 5: Prefetching Across Virtual Pages. (a) Sequential Pages, successful prefetch. (b) Non-Sequential Pages, failed prefetch. (c) Sequential Pages, failed prefetch.

Figure 5 shows an example of a possible stride prefetcher as it performs prefetches for a given process. In example (a) the miss pattern in black generates a prefetch for the third block, and the prefetch is successful because the pages were mapped consecutively. In (b) the miss pattern generates a prefetch for the light gray box, but because the virtual pages were not mapped consecutively the prefetch is for data not being used by the program. And in example (c) a prefetch stride is calculated from two pages mapped non-consecutively leading to a prefetch for the wrong block even though the current miss and the next miss are on consecutive pages. The problem in part (c) can be solved by passing the virtual address to the L2 along with the physical address and then using the virtual address to calculate the stride and add that to the physical address (since translation is unavailable at the L2). This technique however does not solve the case in example (b).

It is important to note that the translation mechanism to translate from virtual to physical addresses is located in the CPU core and is usually tightly coupled with the load/store queue; therefore the L2 cache has no ability to do a translation. As could be seen from the examples in Figure 5 it is possible to issue prefetches for blocks that aren't relevant. Simulators such as SimpleScalar [2] assume that all virtual pages are mapped to consecutive physical pages and therefore neglect simulating these misses properly.

When prefetches are issued for irrelevant blocks there are three things which occur that degrade performance.

- First if the prefetch queue is full the irrelevant prefetch will remove a possibly useful prefetch.
- Second if the irrelevant prefetch reaches the head of the prefetch queue and gets issued it will occupy the bus

for several cycles as the data is transfered. This bus time could have been utilized by a relevant prefetch instead.

- Third the prefetched data will be placed into the cache causing cache pollution by evicting a potentially useful block for one that is irrelevant.

We will show in Section 5 that in an actual system the assumption that all virtual pages are mapped consecutively is inappropriate. We will also do an analysis of the extremes by assuming that all access are like example (a) in Figure 5, or in other words assume 100% of pages are mapped to consecutive addresses. We also simulate a system where all accesses are like example (b) of Figure 5, or 0% of pages are mapped to consecutive pages.

We also implement a simple prefetch filter that removes prefetches that span pages before they are ever entered into the prefetch queue. This is done by simply checking the high order bits of the miss address and the prefetch to determine if they are from different pages. Results of this filter technique are presented in Section 5.

However, this page crossing behavior does not impact certain types of prefetchers. In particular, prefetchers that only fetch previously accessed addresses, i.e. GHB Address Correlating [11]. In these types of prefetchers miss sequences are replayed and unless the virtual page has been remapped to a new physical page the address will still be valid.

4. METHODOLOGY

We evaluated the effects of virtual page boundaries on L2 data cache prefetching using the M5 simulator [1]. We also used a kernel module [9] in the Linux 2.4 kernel to evaluate the frequency of non-consecutive virtual page mappings. The kernel module, simulator, and benchmarks will be discussed in the following sections.

4.1 Kernel Module

A kernel module that was proposed for embedded systems by Movall [9] was adapted for use to calculate the frequency of consecutive virtual pages being mapped to consecutive physical pages. This analysis was critical in showing that in many cases assuming the pages are mapped sequentially is wrong. The kernel module works by walking the Linux 2.4 process list and extracting all the processes memory information into a device mapped in /dev. Then a user can use dd to capture the collected data. For our purposes a simple parser was written to calculate how many adjacent virtual pages were mapped to adjacent physical pages.

4.2 Simulation Environment

The M5 simulator was adapted to incorporate the prefetchers we wished to study. In order to validate the design, parameters were set to match those found in studies done by Perez [12]. After running simulations the prefetchers were found to be within 2% of the recorded values for all the prefetchers.

In order to measure the effects of virtual page boundaries on the prefetchers we adapted the simulator. The M5 simulator supports full system simulation, which would allow for the testing of the prefetchers and have actual address translation. The drawback of simulating in the full system version of the simulator is that upon boot-up of the machine

Simulation Parameters	
Processor Core	
Frequency	2 GHz
Fetch, Decode, Issue BW	8 instructions per cycle
Rob Size	196 entries
LSQ Size	64 entries
SB Size	64 entries
IQ Size	128 entries
Commit width	up to 8 instructions per cycle
Functional Units	8 IntALU, 3 IntMult/Div, 6 FPALU, 2 FPMult/Div, 4 Load/Store Units
L1 Data Cache	
Size	32KB/direct-mapped
Block Size	32 Bytes
MSHRs	8
Targets per MSHR	4
Hit Latency	1 cycle
L1 Instruction Cache	
Size	32 KB/4-way Associative/LRU
Latency	1 cycle
Unified L2 Cache	
Size	1 MB/4-way associative/LRU
Block Size	64 Bytes
MSHRs	8
Targets per MSHR	4
Latency	12 cycles
L1 to L2 Bus	
Frequency	2 GHz
Width	32 Bytes
L2 to Mem Bus	
Frequency	400 MHz
Width	64 Bytes
Main Memory	
Latency	100 cycles
Tagged Prefetcher	
Queue Size	16 entries
Stride Prefetcher	
PC Entries	512
Queue Size	1 entry

Table 1: Simulated System Parameters

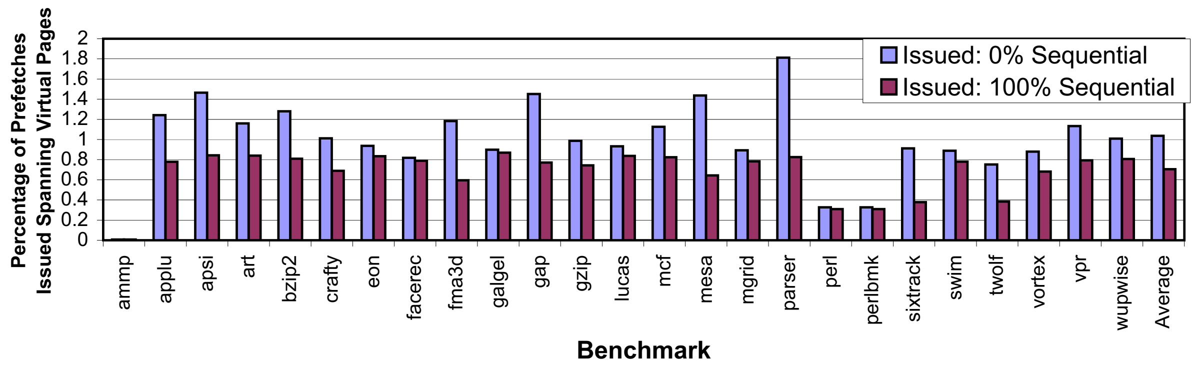

Figure 6: Analysis of tagged prefetches that span pages. *Note Small Scale

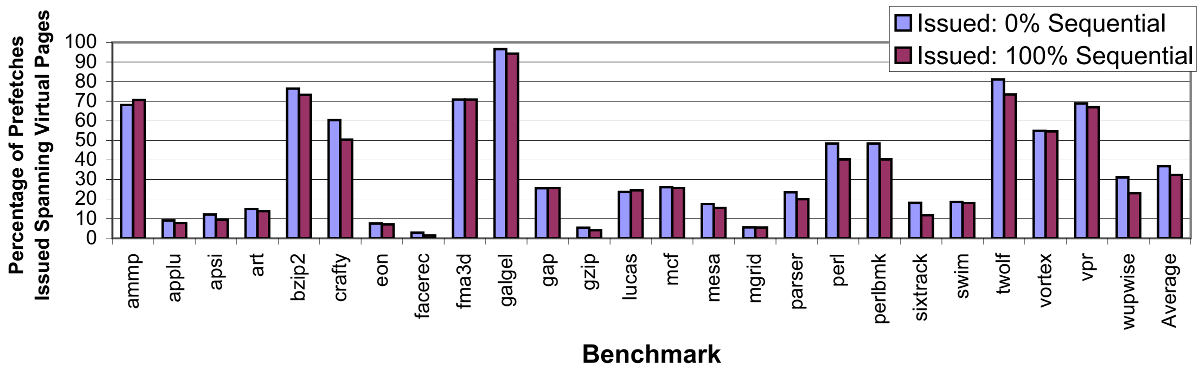

Figure 7: Analysis of stride prefetches that span pages.

the physical address space is mostly free and allows for most applications to keep their virtual mappings in large contiguous blocks. In order to simulate what a system in steady state was, we would need to generate a checkpoint after the simulated system had run for a long period of time. In the essence of time, we adapted the non-full system simulator to perform at both extremes. As was mentioned before the simulator is already designed so that all virtual pages are mapped to consecutive physical pages. We then expanded the prefetcher to grab a block at a large distance away, 100,000 blocks ahead, when a prefetch was calculated that crossed a page boundary. In this way we simulate the system using prefetch queue space and bus bandwidth as well as polluting the cache for prefetches that cross page boundaries. This in effect simulates a system in which no virtual pages are mapped to consecutive physical pages. Therefore these two points can be viewed as the extremes in terms of what percentage of pages are mapped consecutively. It is expected that a real system would operate somewhere between these two extremes.

The system was also adapted with a new filtered version of a prefetcher that filters out prefetches that cross page boundaries before they are even placed in the prefetch queue.

For all the runs the configuration parameters for the system were derived from the work done by Perez [12] and can be found in Table 1.

4.3 Benchmarks

The benchmarks used for our simulation were the SPEC CPU2000 benchmark suite [5]. For each benchmark we used a 500-million instruction trace after the early SimPoint [14]. SimPoints were previously shown to give simulation results within 18% of the full benchmark simulation [14].

5. RESULTS

5.1 Consecutive Page Analysis

Consecutive	Non-Consecutive	Total Mapped	Percentage Consecutive
259	168	427	60.65
310	303	613	50.57
628	422	1050	59.81
485	358	843	57.53
466	358	824	56.55

Table 2: Measured Amount of Page Mappings at several different times

We installed and ran the kernel module from section4.1 to extract the information of all virtual pages that are mapped in the system. We then parsed those results to find out, per process, how many mapped virtual pages were consecutive and mapped to consecutive physical pages. The results are presented in Table 2. The system was booted up and then several ray tracing applications were run. The measurements were taken at various times throughout the day to see what the current status of the machine was. These numbers do not accurately represent what we were trying

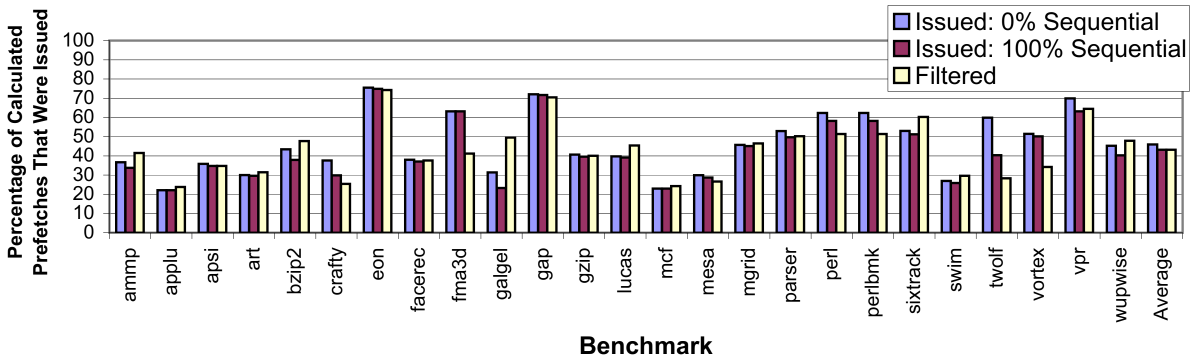

Figure 8: Analysis of stride prefetches that get issued.

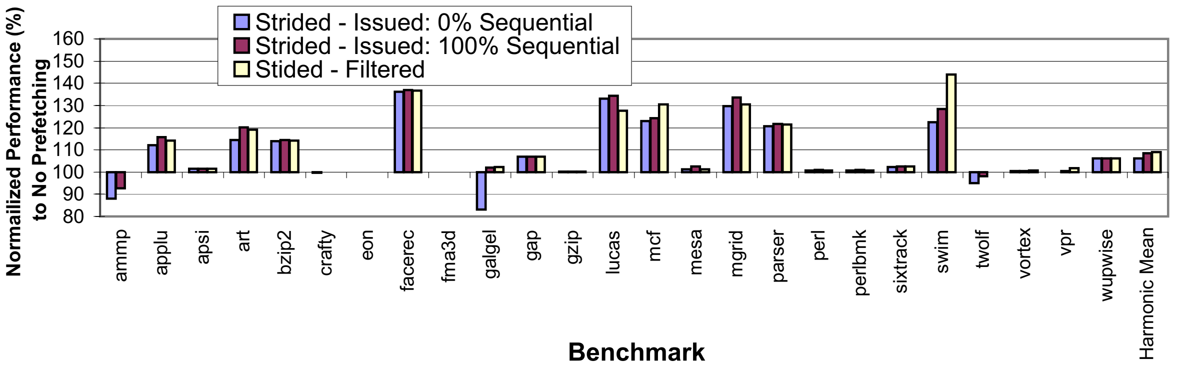

Figure 9: Performance Improvement for Stride Prefetcher Over No Prefetching

to measure because they reflect not only data, but also instruction pages as well. They are also inaccurate due to the fact that we do not measure the number of unmapped virtual pages. Even with these conservative overestimates on the number of pages mapped consecutively we still see percentages ranging from only 50-61%.

5.2 Tagged Prefetcher Analysis

The tagged prefetcher was implemented and then verified against the results presented by Perez [12]. We then ran the tagged prefetcher to determine how often the prefetches crossed page boundaries. The results are presented in Figure 6. Note carefully that the axis is in the range of 0-2%. The first and second bars represent the percentage of prefetches spanning a virtual page if no (0%) consecutive virtual pages were mapped to consecutive physical pages, and all (100%) consecutive virtual pages being mapped to consecutive physical pages respectively.

These results seem reasonable because tagged prefetching is just a variation of one block ahead prefetching. In this case we would expect to see on average 1 miss per number of blocks in a page. Since the simulated system we are using is the Alpha 21264, the page size is 8kB. Our block size was 64 B, so we would expect to see about 1 page crossing every 128 blocks, which is approximately what we measured.

Due to the fact that the tagged prefetcher rarely crosses boundaries we saw very little performance change, in terms of IPC, assuming 0% or 100% of the pages were mapped consecutively. Therefore we have left the graphs relating to IPC performance out of this paper.

5.3 Stride Prefetcher Analysis

Unlike the tagged prefetcher, the stride and GHB prefetchers show many more prefetches that span page boundaries. Figure 7 shows how many of the issued prefetches of a strided prefetcher spanned a page boundary. Note that some applications such as bzip2, galgel, and twolf see prefetches that span page boundaries more than 70% of the time. It is also interesting to note that depending on the benchmark we may issue more or less prefetches that span pages when we assume that no virtual pages are mapped consecutively. This is due to the fact that by prefetching the wrong block we may miss a tagged hit and that can alter the order of prefetches in the prefetch queue. This change in order can alter the miss pattern because something that would have hit before has become evicted by the irrelevant prefetch block.

In Figure 8 we introduce a new bar, filtered, which represents the prefetcher which filters page spanning prefetches before they are inserted into the prefetch queue. In this graph we are analyzing the percentage of prefetches that we calculated that actually get issued. This plays a role in the performance of the prefetcher because the lower the percentage the more often we have to evict a prefetch before it reaches the head of the prefetch queue. This can happen in two ways. First the queue could be come full and it could get evicted. Second, a demand miss may reach it before it was issued. This second type is bad because had the prefetch gotten through the queue faster the miss would never have occurred. When the percentage of issued prefetches is low it is expected that filtering out prefetches that are poten-

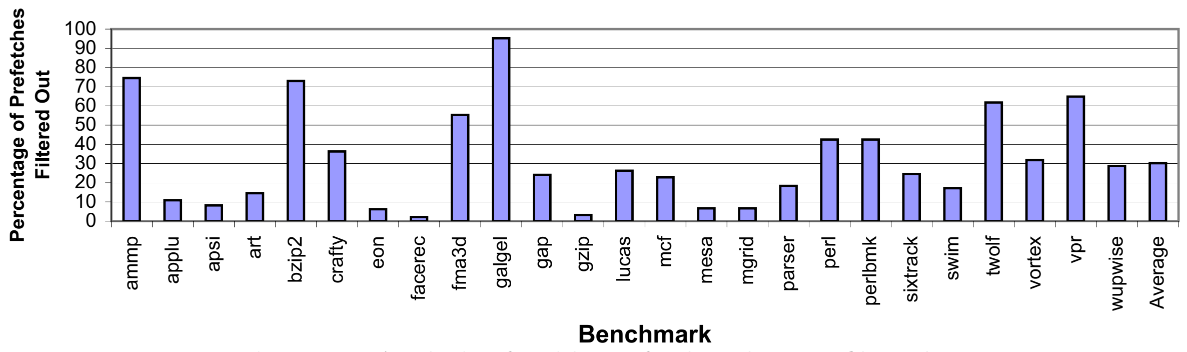

Figure 10: Analysis of stride prefetches that get filtered.

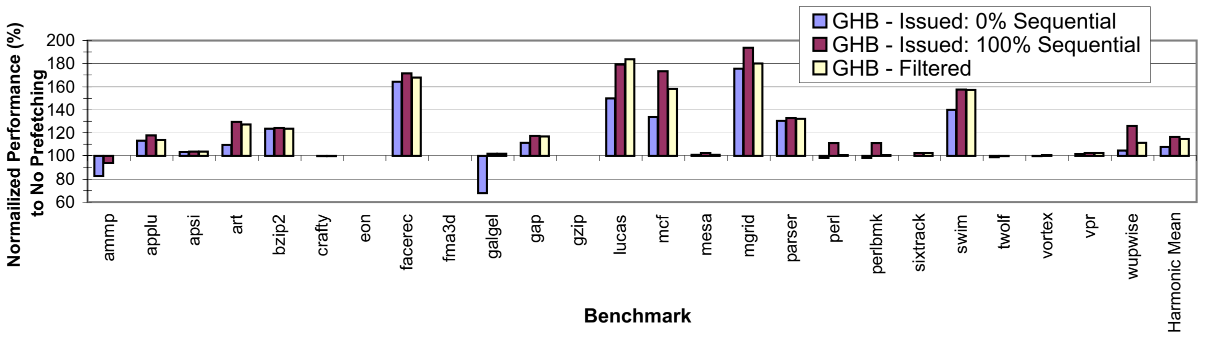

Figure 11: Performance Improvement for GHB Delta Correlating Prefetcher Over No Prefetching

tially bad could help performance because there are plenty of other prefetches to chose from.

Next we present the performance, in terms of IPC, of a strided prefetcher in Figure 9. In this figure we are looking at the relative speedup over a non-prefetching system for each of the three cases. It is important to note that in every case the 0% consecutive performs equal to or worse than the 100% case. The 100% case corresponds to prefetching results produced by SimpleScalar [2].

In cases such as art, the filtered prefetcher performs somewhere in between the other two. A real system would be expected, based on its percentage of consecutive pages, to operate somewhere between the 0% and 100% case. In some cases, such as mcf and swim, the filter is able to outperform both metrics. This is due to the fact that by removing the prefetches that crossed page boundaries the filtered prefetcher essentially gave priority to other prefetches that were more useful which may have been evicted by the filtered prefetch, or delayed too long to have an impact. Looking back at Figure 8 it can be seen that these benchmarks had a low number of prefetches issued, which means filtering some of them improved the performance. In Figure 10 we present the percentages of prefetches that were filtered out with the prefetch filter. The low number of prefetches that are issued in combination with the fact that a large percentage of prefetches 15-22% were filtered out, as seen in Figure 10, leads to the performance gain for mcf.

Other applications like mesa, mgrid, and applu also exhibited low numbers of prefetches actually being issued in Figure 8, but they did not see the filtered version outperform both due to the low number of prefetches that were actually filtered in Figure 10.

Some benchmarks saw very little performance improvement at all such as eon and fma3d. This is due to the fact that most of the prefetches calculated for these benchmarks were issued, see Figure 8.

The ammp benchmark saw a slowdown with both forms of prefetching but the filtered version helped by filtering a high number of prefetches. It is clear from this that the ammp benchmark does not actually use many of the prefetches that cross the page boundary and by filtering them we were able to cut down on the cache pollution and improve the performance back to that of the non-prefetching version.

Galgel is an interesting benchmark because it exhibits behavior that shows that prefetches across pages really hurt performance if they don't succeed. They see about a 20% degredation when none of the pages are mapped consecutively. This is probably due to the fact that more than 90% of its prefetches in Figure 7 span page boundaries. By filtering those out we are able to keep the bad prefetches from polluting the cache and negatively impacting performance.

5.4 GHB Prefetcher Analysis

The last figure presented, Figure 11, is the performance, in terms of IPC, of a Delta Correlating GHB prefetcher. As was the case with the stride prefetcher some benchmarks exhibit performance gain over the 100% sequential when using the prefetch filter. However on average the filtered version performs slightly worse than if all the pages could be placed consecutively. In reality using a filter helps to bring the performance of a prefetcher closer to the case where all pages could have been mapped sequentially, and can sometimes improve performance beyond that by filtering some prefetches that were not used.

6. RELATED WORK

There are many research papers related to prefetching [7, 4, 8, 6, 11, 10]. They differ from this paper in that they proposed new ideas, whereas this paper is analyzing the impact of a design decision that was left out of the analysis.

Perez's[12] MicroLib paper did similar work in comparing different prefetchers to evaluate their performance in a neutral reproducible environment. They were careful to consider one of the faults of SimpleScalar [2] in which an infinite number of MSHRs is available. This bug does not exist within M5, but the work we have done here is similar in that it shows that designers need to be conscious of the virtual memory translation when exploring hardware design with simulators such as SimpleScalar where a unrealistic implementation could be neglecting something important.

Some hardware prefetchers already contain page filters but designers often ask what, if any, performance gains the filter is prohibiting them from exploiting. In this work we show that although sometimes a large number of prefetchs are filtered out, the negative impact of the non-consecutive pages prohibits most additional gains seen from removing the filter.

7. CONCLUSION AND FUTURE WORK

In this paper we analyzed the effects of L2 prefetches crossing virtual page boundaries. We found that in a real system only 50-70% of mapped virtual pages are mapped to consecutive physical pages, however the published results of recent prefetcher papers assume 100% of virtual pages are mapped to consecutive physical pages. This assumption can lead to a performance degradation which we mitigate with a prefetch filter.

With a stride prefetcher we measured that on average 30% of prefetches cross a page boundary. Assuming no pages are mapped consecutively, we measured a performance degradation of about 2.2% on average. And by filtering prefetches that cross page boundaries we see on average a 3% increase in performance. Which results in a minimal increase in average performance over naively assuming all pages are mapped sequentially.

The effect is more pronounced for a delta correlating global history buffer prefetcher. Again with no pages mapped consecutively a degradation of 9% on average is observed and by using a filter we were able to improve performance to within 2% of the baseline approach that assumes the OS can map all pages sequentially.

From these results it can be seen that although prefetching studies in the past did not account for page boundaries, the results are still within a couple percent of what can actually be achieved if a simple prefetch filter is introduced. The complexity of running a full system simulation to get accurate steady state page mappings to evaluate prefetching techniques is complex and requires long simulation times. However, this complexity can be avoided by simulating in the traditional manner as long as hardware designers are careful to put in a simple filter to prevent prefetches from spanning pages.

Although it was not explored, another consideration is that for some architectures the page size is variable. Unless the prefetch filter has access to the TLB to determine the page size, it is forced to assume the smallest page size. This means that if it happened to calculate a prefetch on a larger page that crossed the small page boundary, then the prefetch filter would end up rejecting a prefetch that could have been to useful data.

In the future this work could be expanded to make better measurements of how many virtual page boundaries really represent a problem for data prefetching. This involves calculating unmapped pages, as well as sorting data and instruction pages. Other future work includes investigating larger block sizes as they start approaching the page size. This becomes particularly relevant in the context of 3D stacked caches and memory structures[13] where very wide buses are possible. Additional research could also look at exposing the TLB to the L2 cache to allow for more flexible prefetching or even new techniques to try to eliminate TLB misses by walking the page table when page spanning prefetches are detected.

8. ACKNOWLEDGMENTS

This work was supported by gifts from Intel and Sun Microsystems.

9. REFERENCES

[1] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 36–43, Feb. 2003.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[3] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proc. Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 51–61, Oct. 1992.

[4] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *25th Ann. Int'l Symp. on Microarchitecture*, pages 102–110, 1992.

[5] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[6] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proc. 24th Ann. Int'l Symp. on Computer Architecture*, pages 252–263, June 1997.

[7] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Ann. Int'l Symp. on Computer Architecture*, pages 364–373, 1990.

[8] S. Kim and A. V. Veidenbaum. Stride-directed prefetching for secondary caches. In *International Conference on Parallel Processing*, pages 314–323, 1997.

[9] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proc. 2005 USENIX Technical Conference*, pages 23–32, 2005.

[10] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. Ac/dc: An adaptive data cache prefetcher. In *Proc. 13th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 135–145, 2004.

[11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proc. 10th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, page 96, 2004.

[12] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *37th Ann. Int'l Symp. on Microarchitecture*, pages 43–54, dec 2004.

[13] K. Puttaswamy and G. H. Loh. Implementing caches in a 3d technology for high performnace processors. In *IEEE International Conference on Computer Design(ICCD) 2006*, pages 525–532, 2005.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, Oct. 2002.