SIMULATION OF ADDRESS TRANSLATION TECHNIQUES

A Thesis

by

JAMES COMAN

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2021

Major Subject: Computer Engineering

ABSTRACT

As the memory footprints of modern compute workloads continue to grow[1], pressure on the memory hierarchy increases and address translations play an increasingly important role in system performance. Translation Lookaside Buffers (TLB) are a vital structure to the performance of modern virtual memory systems. They reduce the need for slow and expensive page walks by caching the most recent virtual-to-physical address translations. We analyze how well the cost of the page walk can be approximated in a five level memory hierarchy, and how simple and hypothetical optimizations are able to affect the memory system performance.

Initially we compare the performance of a realistic page walker to a fixed page walk penalty. This allows for future work to presume a demonstrably reasonable constant value in experimentation, not relying on intuition and saving on the additional time and energy of a simulated page walk. A suggested fixed value is put forward as well as an analysis of the variability across workloads and any limitations.

Making use of this fixed page walk penalty, we also look at the effect of a simple TLB optimization - doubling the available resources. allows us to asses the affect of the TLB on the memory system performance and discuss both what a future optimization may look like and what performance can be both reasonably expected and hoped for.

We analyze one potential in-TLB optimization, CHiRP[2], which seeks a replacement policy for the TLB more appropriate and optimized for the structure than least-recently-used (LRU). We analyze the structure of the policy and also the results of the CHiRP work against our hypothetical performance improvements. A strategy related to prefetching is also analyzed. ASAP[3] which prefetches *inside of and relevant only to a particular page walk* is examined.

DEDICATION

To my parents

ACKNOWLEDGMENTS

I would like to think the Texas A&M University office of High Performance Research Computing for providing the computational resources that made this work possible.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Daniel A. Jiménez (advisor) and Professor Dilma Da Silva of the Department of Computer Science & Engineering Department and Professor Paul V. Gratz of the Department of Electrical & Computer Engineering.

The data analyzed was provided by Professor Jiménez. The page walker implementation was provided by Georgios Vavouliotis of the Barcelona Supercomputing Center.

All other work conducted for the thesis was completed by the student independently.

This work has been funded in parts by NSF grants CCF-1912617 and CNS-1938064, SRC GRC task 2936.001, as well as generous gifts from Intel Labs.

NOMENCLATURE

CPU	Central Processing Unit
TLB	Translation Lookaside Buffer
MMU	Memory Management Unit
CHiRP	Control-Flow History Reuse Prediction
ASAP	Address Translation with Prefetching

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS vii

LIST OF FIGURES ix

LIST OF TABLES x

1. INTRODUCTION 1

2. BACKGROUND 2

 2.1 Memory Wall 2
 2.2 Virtual Memory 3
 2.3 Page Tables 5
 2.4 Caches 6
 2.5 Translation Lookaside Buffers 7
 2.6 Huge Pages 8
 2.7 Motivation 8
 2.8 The prolific growth of the modern dataset 9

3. METHODOLOGY 10

 3.1 Methodology 10
  3.1.1 ChampSim 11
  3.1.2 Workloads 12

4. RESULTS 13

5. DISCUSSION 17

 5.1 CHiRP 17

5.2 ASAP 18
5.3 Discussion 18

REFERENCES 20

LIST OF FIGURES

FIGURE Page

2.1 Illustration of the memory wall. 3

2.2 Illustration of Memory Fragmentation. If process C requires 16 kB it cannot be scheduled until either A or B finish as there is not a region of memory both available and of the required size. 4

4.1 Affect of Doubled TLB on MPKI 14

4.2 SPEC IPC Improvements 15

4.3 CVP IPC Improvements 16

LIST OF TABLES

TABLE Page

2.1 Common Latency and Memory Values for Server Systems, 2019 6

3.1 Simulation Parameters 11

1. INTRODUCTION

Virtual Memory plays an important role in the performance of modern computer systems by allowing for greater security, easier memory management without relative addressing, and hides fragmentation by abstracting the specifics of the memory storage system. While it provides a great boon to programmers in this way, the translation from idealized virtual addresses to physical addresses incurs significant overhead [4][5]. Among these overheads is the time, space, and, power required to translate virtual addresses to physical addresses through the page table structure[6]. Walking this table, which in modern systems consists of a five level radix tree, must be accomplished somewhat sequentially in order to determine the physical address. This process is also time and power consuming on the memory, requiring, in the worst case, a separate memory access for each level[7].

Caching into the TLB is applied in order to hasten this process for commonly referenced pages. Values in the cache do not need to be recalculated and can be accessed quickly. This is of particular importance as the TLB and page table are on the critical path for memory accesses [7]. As with any critical path cache structure, latency, size, power, and area are all important concerns[6]. As a mitigation, TLBs are arranged in an inclusive hierarchy in the same way as more general caches.

As a result, the last-level TLB (which we refer to as the TLB for simplicity) is what is considered here as it is the largest, is shared between cores, and is the most important [7]. Being present in any on-die cache is more important than which particular one as they all avoid a memory access and this TLB is the last level before total cache eviction. TLBs specific to particular cores are not considered here.

2. BACKGROUND

2.1 Memory Wall

While from the perspective of the layman computers have certainly become more performant over the past several decades, this growth has not been proportional to all parts of modern computer systems. Of particular relevance here is the "Memory Wall" the phenomenon of the diverging performance of processor speed and memory speed. While both of these have certainly been improving and continue to do so, processor speed for many years saw much more improvement than the memory speed did[8].

There are many problems in computing that have large working sets of memory[1][9], and among these workloads, it may certainly be the case that there is much more potential for time saving performance improvements in the memory space than the compute space. Such workloads where the majority of the computing time is spent in memory as opposed to in processing are referred to as "memory-bound". The reasons for this performance characteristic are various. Dynamic Random Access memories (DRAM) have been the standard in memory technology for some time due to the balance offered of speed, capacity, energy, and physical size. Faster Static Random Access Memories (SRAM) offer advantages but are physically larger. As storage gets faster it must move physically closer to the processor as the information can only be transferred at the speed of light. So larger storages of faster memory cannot physically fit close enough to reduce latency and meet speed guarantees. For this reason the memory slots on a motherboard are close to the processor socket, and memory chips on consumer Graphics Processing Units (GPU) are frequently arranged around the GPU itself.

As the size of modern workloads grow, this performance bottleneck of memory is increasingly a factor in the time required to process large datasets. Fig 1. demonstrates the gap between the performances of processors and memory relative to 1980 values[10]. While since approximately 2006 processor speed has ceased to increase as rapidly due to the power wall and the limits of

Dennard scaling[11], the gap remains significant and the practical impact is exacerbated by an increasing number of workloads with vast working sets of data.

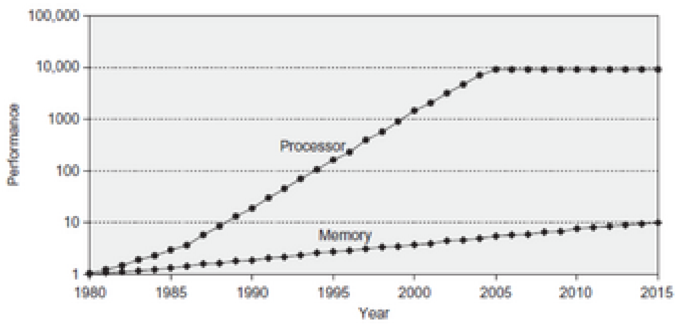

Figure 2.1: Illustration of the memory wall.
Reprinted from Computer Architecture: A Quantitative Approach, Sixth Edition

2.2 Virtual Memory

Virtual Memory functions as a layer of abstraction that offers some advantages and disadvantages over direct memory management[12]. Virtual memory breaks memory into pages of a fixed size (typically 4kB) and handles the job of laying out the pages into physical memory so a programmer does not have to[7]. As a result, virtual memory solves the problem of memory fragmentation.

Memory fragmentation is illustrated in Fig 2. Processes A and B are each consuming 8kB of a 32kB memory resource. If process C requires 16 kB it cannot be run until either A or B finishes because there is no contiguous region of memory available of 16kB in size. With Virtual Memory, C would be able to run as there is enough room available *total*, and virtual memory handles the complexities of fragmenting our memory across two distinct regions for us. How the memory will be specifically laid out and how much fragmentation occurs as a result will be determined by the memory allocator[13].

Pages may also be protected from each other, so that independent processes do not alter each other's memory. The lack of this protection without virtual memory makes programming haz-

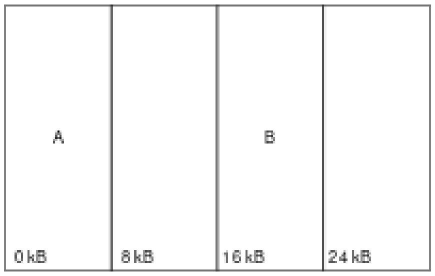

Figure 2.2: Illustration of Memory Fragmentation. If process C requires 16 kB it cannot be scheduled until either A or B finish as there is not a region of memory both available and of the required size.

ardous and insecure, there is no system to prevent writes into another program's memory. As a result of this ability to add permissions to different memory pages, pages can even be shared by multiple processes, a feature often used by libraries that may be used by multiple other programs on the system. Writing to shared pages is even allowed, as a new copy is only made when necessary - when the writes begin. Prior to that memory can be saved by allowing multiple processes to read the same memory page. This protection feature is essential to modern operating system security.

Virtual memory has some subtler effects as well; some pages may not even be pulled into the memory system if they have not yet been accessed - allowing them to remain on disk and reduce both power, and prevent a waste of space in memory. They will only be pulled into memory when needed, in a process know as "paging to disk". Since the Virtual Memory system handles allocating pages in physical memory, programmers can write their programs as though they have access to the entire memory pool, and do not need to write memory-handling code in a relative-addressing style that uses the address of the program in memory as the basis for memory calculations, a valuable simplification. Virtual Memory also allows pages to have certain properties - such as marking a page read only. For all of the powerful features, virtual memory does increase memory access time.

2.3 Page Tables

Virtual memory must have a way to translate the virtual addresses used by programs themselves to the physical addresses where the data is stored, possibly in a fragmented state. This is done with a page table, a table that tracks the translations from virtual to physical page addresses. Not every memory address needs to be stored, just the address of the 4kB pages, and from there an offset can be used to find the requested value.

Such a table would be impractically large for an entire 64-bit memory space - and would in fact exceed the memory capacity of modern systems. As a result, the page table is arranged in a tree structure to save space. This savings, however, has a trade-off. The levels of the table, of which there are frequently five on modern systems, must historically be accessed in a sequential order as one cannot determine which of the lower level tables to access without first accessing the higher level's table. For a given virtual address, each level of the page table implies a new memory access, so the translation process for each address is itself vulnerable to the inconsistencies of the memory hierarchy. This calculation process is known as page table walk and is part of the reason virtual memory does impose a slowdown to the memory system. This slowdown and complex process is considered worthwhile in context of the powerful feature set of virtual memory.

The name given to these page tables with multiple levels is a "radix-tree" page table. Each level uses a different set of bits from the virtual address. The outer most table may only have a few entries, as the capacity of one entry may exceed the address space of all the memory that can physically be installed in the system, which does afford some optimization. Only the leftmost bits are used for this first indexing, and on a 64 bit machine, even if the lower 12 are kept the same, the remaining 48 bits can effectively address 282 TB - far more than any system can currently accommodate. As a result, the outer layers of the tree may be sparser, and the very left-most bits may be ignored entirely. Moving right across the virtual address, lower and lower bits are used to access lower levels of the table, until the last bits are used to calculate the offset within the page itself, not determine the address.

More levels allow for greater flexibility in the size of the page table while also increasing access times. While the entire address space would never be mapped, the tree allows for an efficient storage of the needed parts at any one time. Parts of a large program or dataset may not even have translations that exist until the page in which requested data resides is accessed.

2.4 Caches

The Memory Hierarchy of modern computer systems allows for a series of capacity-speed trade-offs. Disks hold the most information but are painfully slow to access. While disks may even be attached over the network and in another physical locations, memory, the next level, is almost always on located on the same board as the processor socket due to the timing constraints discussed above. Above this, are the caches. It is common to see a three level cache structure. Caches exist on the same physical silicon die as the processor, allowing for very fast access times due to their proximity and the speed of on-die connections. Die space is a prized resource and so once again, as the speed increases the capacity must decrease. The largest L3 caches that are frequently shared between cores, and are the most featureful, with smarter replacement policies, and more sophisticated management. L2 caches are faster and smaller still, but do not represent the nearest and dearest storage, that of the L1. At this level, the capacity is quite small, but with latencies small enough to be hidden by reordering instructions. Table 1[15] shows common Latency and Memory values for server systems.

Device	Latency	Capacity
Disk Storage	16-64 TB	5-10 ms
Memory	32-256 GB	50-100 ns
L3 Cache	16-64 MB	10-20 ns
L2 Cache	256 kB	3-10 ns
L1 Cache	64 kB	1 ns

Table 2.1: Common Latency and Memory Values for Server Systems, 2019

The specific latencies and capacities in question are not particularly relatable in human terms. For illustration, allow us to upscale these numbers. If we model a L1 access as taking one second instead of 1ns, we can begin to get a feel for the discrepancies in access times and why optimizations to the memory hierarchy are worthwhile. At this scale, the time it takes to access main memory would be a minute to 90 seconds - long enough to refill your coffee, and an access to disk would be nearly 4 months.

The capacities are similarly unapproachable. If we model the L1 cache, 64 kB as a single piece of paper. 105 microns thick, then the 64TB disk would be a stack of paper 105 km tall, reaching from sea level to the outer edge of Earth's atmosphere.

At this scale it becomes clear that an efficient memory hierarchy and use of caches is imperative for performance. Many orders of magnitude of performance rely on it. To facilitate this, only the most. frequently used data is put into caches and the replacement policy of caches is designed to reduce the number of cache misses - instances where a needed value is not in the cache and must be fetched from a lower level of the memory hierarchy.

In addition to making informed decisions about what value to evict when the cache is at capacity and another value is fetched, modern systems attempt to predict which values will be read into the cache in an effort to have these values ready when needed and to prevent the long access time that would normally be compulsory. This process is known as "prefetching"

2.5 Translation Lookaside Buffers

In order to prevent the punitive latency of the multiple memory accesses occurring from a single translation the TLB exists to cache the translations. In the common cases where memory accesses have their translation in this cache, the process is much faster as the physical address is immediately available. Only in the event of a TLB miss does the expensive page walk process have to be undertaken to determine the value[7].

TLBs are the primary object of our study. Unlike general caches, data caches, and instruction caches, there is not a large body of prior work on the topic of TLB replacement policy or TLB prefetching. These structures do differ in some ways from more general caches. They need not be concerned with a cache bypass, a value that is determined to be not worth storing as the use is predicted to be one-off. This is not a realistic expectation for an entire page of memory.

2.6 Huge Pages

A proposed solution for the growth of datasets in modern computing is to have larger pages, thereby reducing the amount of pages, and the pressure on the page table. Huge pages of various sizes, most commonly 2MB but up to 2GB are supported by the Linux kernel[16]. The system by which these are implemented has required specific hardware support for each page size and so different architectures present different options for huge pages. Huge pages of different sizes can be used by different programs at the same time, and huge pages may be used in conjunction with regular pages. The Linux memory allocation system even has the ability to decide when to make a page huge, if explicitly asked to do so by the programmer.

Not all systems support swapping huge pages - just as the capacity of a 2MB huge page is a factor of 512 larger than a regular page, so is the work on the I/O system and memory controller to swap these pages in and out. This problem becomes exacerbated by random access to different huge pages. If all pages are accessed with some degree of sequentialness, huge pages are more likely to provide a benefit, but this is not likely to be the case if the accesses are random.

Unless the system is quite application specific, and that application would benefit from the use of huge pages, it is very unlikely that turning on huge pages globally would be of benefit - both due to additional stress on memory and disk I/O but also because the capacity for waste is increased as well. A page is the smallest amount of memory that can be allocated and so a small allocation has a much larger effect on the memory system if that page is not fully populated with useful data and most of the space is unused.

2.7 Motivation

The goal of this work is to demonstrate the potential for performance improvements in the TLB by preventing misses and incurring costly page walks. It is well established that memory system performance is crucial to modern systems, but we look to determine how much performance can be specifically gained from TLB improvements.

As part of our work we analyze the practice of modeling the page walk processor with a fixed penalty. This is of great relevance to software simulators. Page walk simulation is expensive both in terms of simulation time, programmer time, and power which are very real constraints for computing research. As a result we will determine the accuracy of modeling this process with a simple fixed penalty, and how this affects simulation results.

In an effort to not merely evaluate the potential for performance improvement in the TLB, we will look at several related proposals and discuss their results and effectiveness at achieving this hypothetical performance increase. These proposals vary in their approaches. Huge pages, due to the limited scope of their usefulness, are not determined to be an adequate solution to increases in TLB and memory pressure.

A study on the Intel i7-6700 Skylake processor found that translations that missed in the TLB took from 61 to 1158 cycles. Such steep penalties of many hundreds of cycles exist in spite of this processor model utilizing caches of the page table structure in order to reduce the page-walk time[17].

2.8 The prolific growth of the modern dataset

Working sets of data for programs are growing. The reasons for this are numerous, machine learning applications require large amounts of data to train, - and frequently use low precision values that reduce the amount of processing required per datum- social media applications amass incredible amounts of data in servers, computer vision algorithms must process large high quality images or even videos[1][9]. Big data projects, unsurprisingly from the name, put strain on these systems that have limited physical ability to keep up and where designed decades ago, or must maintain compatibility with such systems, designed when workloads were magnitudes smaller.

These workloads place tremendous pressure on caches and TLBs alike. The memory system has not grown at the rate that processing has and improvements to these systems will have to be realized in order to maintain performance as workloads continue to grow and grow. This challenge grows by the day and certainly motivates study into more efficient memory system design.

3. METHODOLOGY

We use ChampSim to implement a realistic page walk simulator, and to compare this to the existing ChampSim practice of a fixed page walk penalty. This enables us to discuss the accuracy of both the specific value used in the simulator, suggest a more accurate one, and discuss the variability and specific drawbacks of using a fixed penalty. This is tested by running the simulator with and without the page walker on collections of memory traces from various realistic workload benchmarks from SPEC [18].

From here, a trivial example TLB optimization is made, by doubling the TLB in size. This can be done in one of two ways, and both are included if they perform sufficiently differently. The purpose of these optimizations is to get a reasonable expected value for what high-performance TLB looks like so that other more serious work can be compared to it.

From here the surveyed work can be compared, contrasted, discussed, and evaluated against our hypothetical numbers. Some strokes are necessarily broadened as it is not viable to test all the surveyed systems and our own under the same conditions with the same workloads. This problem has been solved in the past with competition formats such as IEEE's HPCA CVP2 [19] which allow different proposals to run on the same physical system, with the same operating system configuration, and workloads.

3.1 Methodology

A recent version of ChampSim is used, and the page walker was implemented as a feature that can be enabled at run time as a replacement for the fixed value, which is also variable at run time. The simulator is ran with a selection of memory intensive traces from both SPEC and GAP in order to compare performance of the fixed value and page walker determined latencies. Traces are analyzed at various lengths, and shorter traces are repeated per ChampSim's behavior. Various lengths are used to determine the shortest amount of accesses to read from the traces to obtain a reliable result. The simulations are run in parallel on Texas A&M's ADA supercomputer cluster, an

LSF x86 machine (2.6.32-754.35.1.el6.x86_64). It is compiled with g++ (6.3.0) and glibc (2.12)

Three sets of tests are run, a control with a fixed page walk value, a set with the realistic page walker enabled and a set with a doubled TLB.

3.1.1 ChampSim

ChampSim was chosen as the simulator for this work for the respect and familiarity the architecture community has with this simulator and because it provides a reasonable API for analyzing LLC replacement by allowing a painless change of policies. While there is no such convenient API for our proposed structure of study, the TLB, the principles can be applied to allow for two modes, one with a variable fixed penalty, and one with a realistic page walker. The drawbacks and caveats of this simulator are (at least relatively) understood by the larger community. The project is partially maintained by Intel, which makes use of the page table structure under study and lends some credibility to this choice.

The latencies, structures, and capacities used for our simulation are found in Table 2. For the study of the doubled TLB, the number of ways per set is doubled to 16 (as opposed to 8) for a capacity of 2048 (as opposed to 1024). In both cases, the page-walk latency is fixed at 100 cycles.

L1 i-Cache	64KB, 8 way, 4 cycles
L1 d-Cache	64KB, 8 way, 4 cycles
L2 Unified cache	256KB, 16 way, 12 cycles
L3 Unified Cache	8MB, 16 way, 42 cycles
DRAM	Simulated access time
Branch Predictor	Hashed perceptron, 4K entry BTB, 20 cycle miss penalty
L1 i-TLB	64 entry, 8 way, 1 cycle
L1 d-TLB	64 entry, 8 way, 1 cycle
L2 Unified TLB	1024 entries, 8 way, 8 cycle hit latency, 100 cycle miss penalty

Table 3.1: Simulation Parameters

3.1.2 Workloads

A wide array of benchmarks are used. Both SPEC 2006 and 2017 are included, as well as a set of server workloads from the first Championship Value Prediction contest. This collection consists of various application server and scientific workloads. Some of which characterize large modern database applications and others originate from scientific applications. While some of these workloads could conceivably be relevant to the workstation space, none of these workloads come from mobile applications. The memory pressure central to our study is not present in mobile devices. A total of 1035 benchmarks are tested. For our testing, structures are warmed for the first half of the trace and results are measured during the second half. No more than 100 million total instructions are simulated. This was found to be less than 1% different than running all traces to completion.

4. RESULTS

Our baseline analysis of the SPEC and CVP workloads yields an IPC of 1.13 and MPKI for SPEC and 1.59, 0.29 for CVP respectively. This serves to demonstrate the differences in the workloads, and also the figures to which we will compare. The realistic page walker gives results within 0.5% of the IPC values for each benchmark suite (the page walker has no effect on MPKI). This serves to support the common assumption that using a fixed TLB miss penalty is acceptable, and does not significantly vary performance numbers. Such a close figure is likely also the object of the length of study, at 100 million instructions and also indicates a lack of overall effect on performance resulting from TLB miss latency.

Long studies should be taken on regardless, and so this result points to affirm this practice which saves programmer time, simulation time, and bug surface area by reducing potentially hundreds of lines of complex page walk simulation code with a fixed value.

Doubling the TLB also yields a small result in the IPC. For SPEC, IPC improved 1.3% and 2.3% for CVP. MPKI was much more affected here, being drastically reduced by the larger capacity to store entries and non later re-calculate them. SPEC saw a 50% drop in MPKI and CVP saw a 70% drop. These values are shown in Fig 3. The general lack of IPC improvement, however, indicates that even though TLB performance may be greatly improved, this does not have a huge effect on performance generally.

While these average numbers are telling, it does not paint a full picture. This certainly isn't to say there aren't workloads that do benefit from increased TLB performance. Figures 4 and 5 address this by looking at the frequency of improvements across the two suites. The SPEC suite sees only a small minority of benchmarks benefiting from increased TLB performance. CVP, however, gives a more interesting result. 119 of the traces see a performance improvement of over 5%, and 29% see an improvement over 15%. This would suggest that there certainly are modern workloads that stress the TLB to the point of performance degradation and validates more study as this problem is expected to increase. 14% of these benchmarks, despite being several years old as,

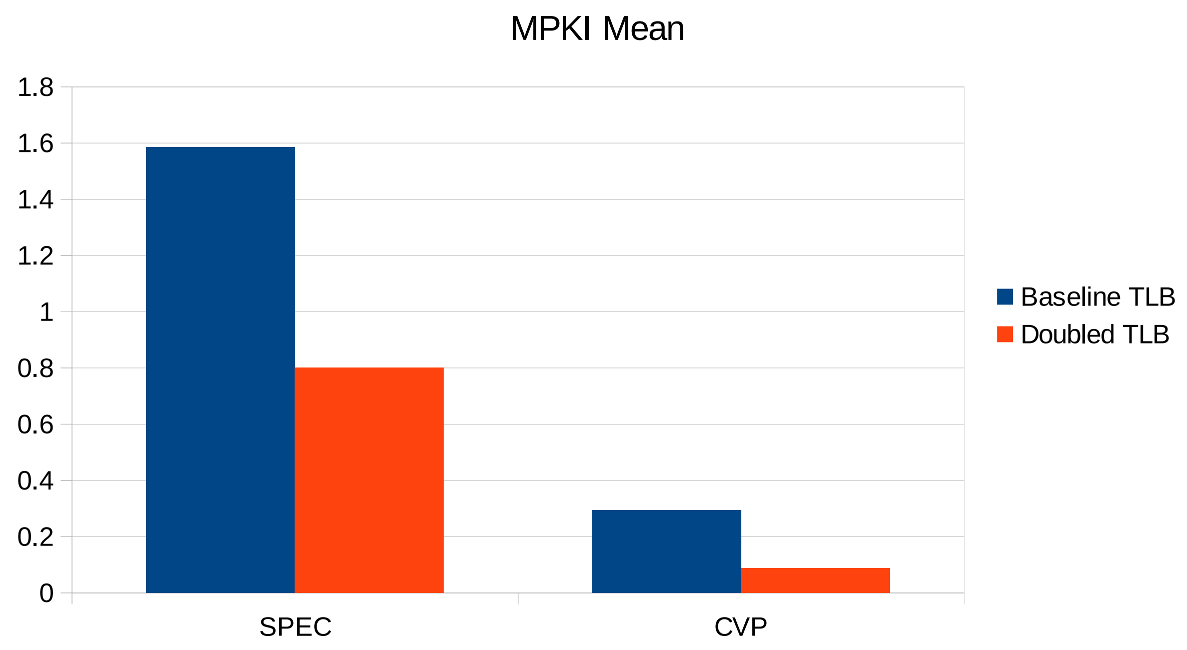

Figure 4.1: Affect of Doubled TLB on MPKI

still see notable (more than 5%) improvement from increased TLB performance.

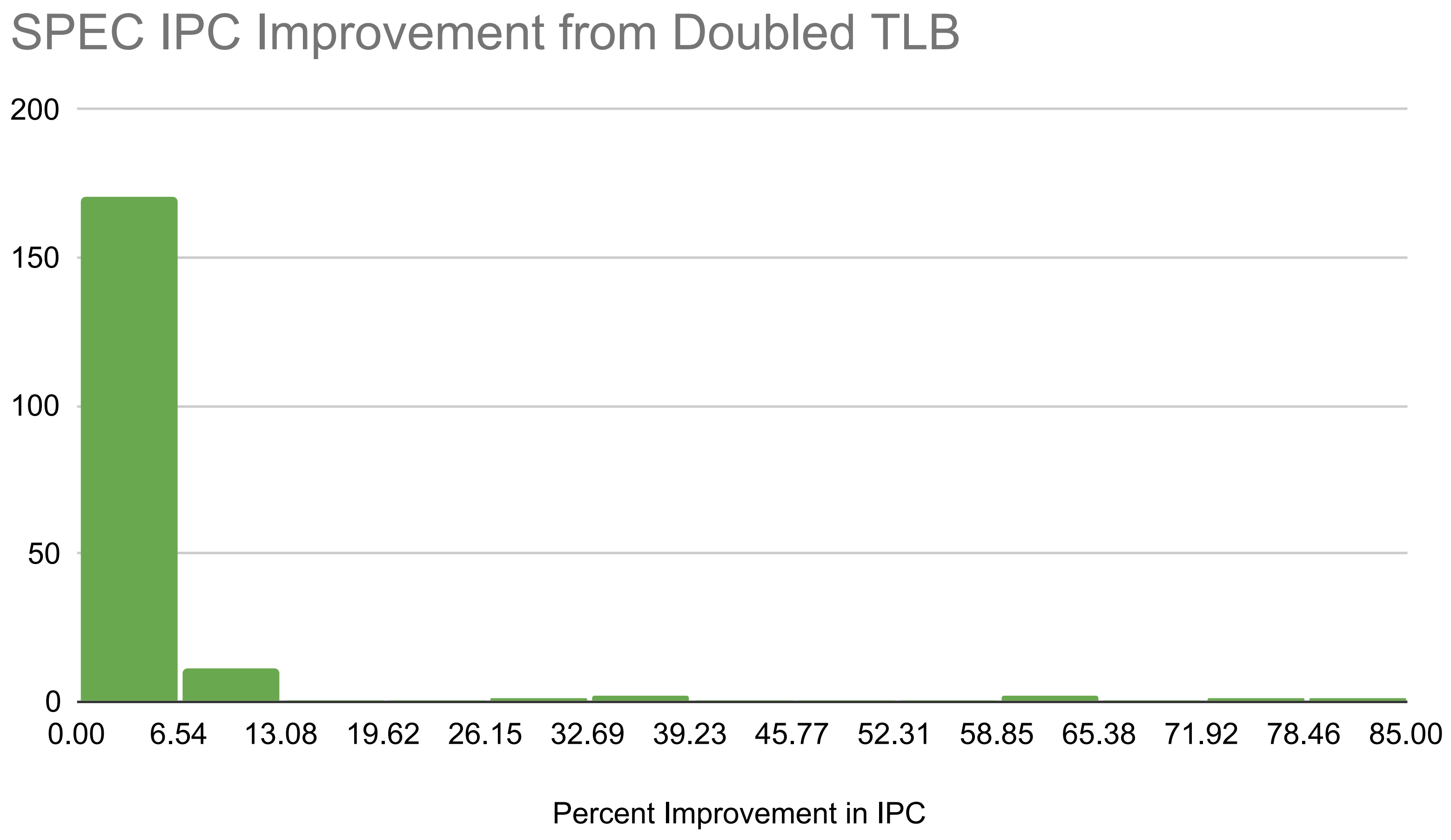

Figure 4.2: SPEC IPC Improvements

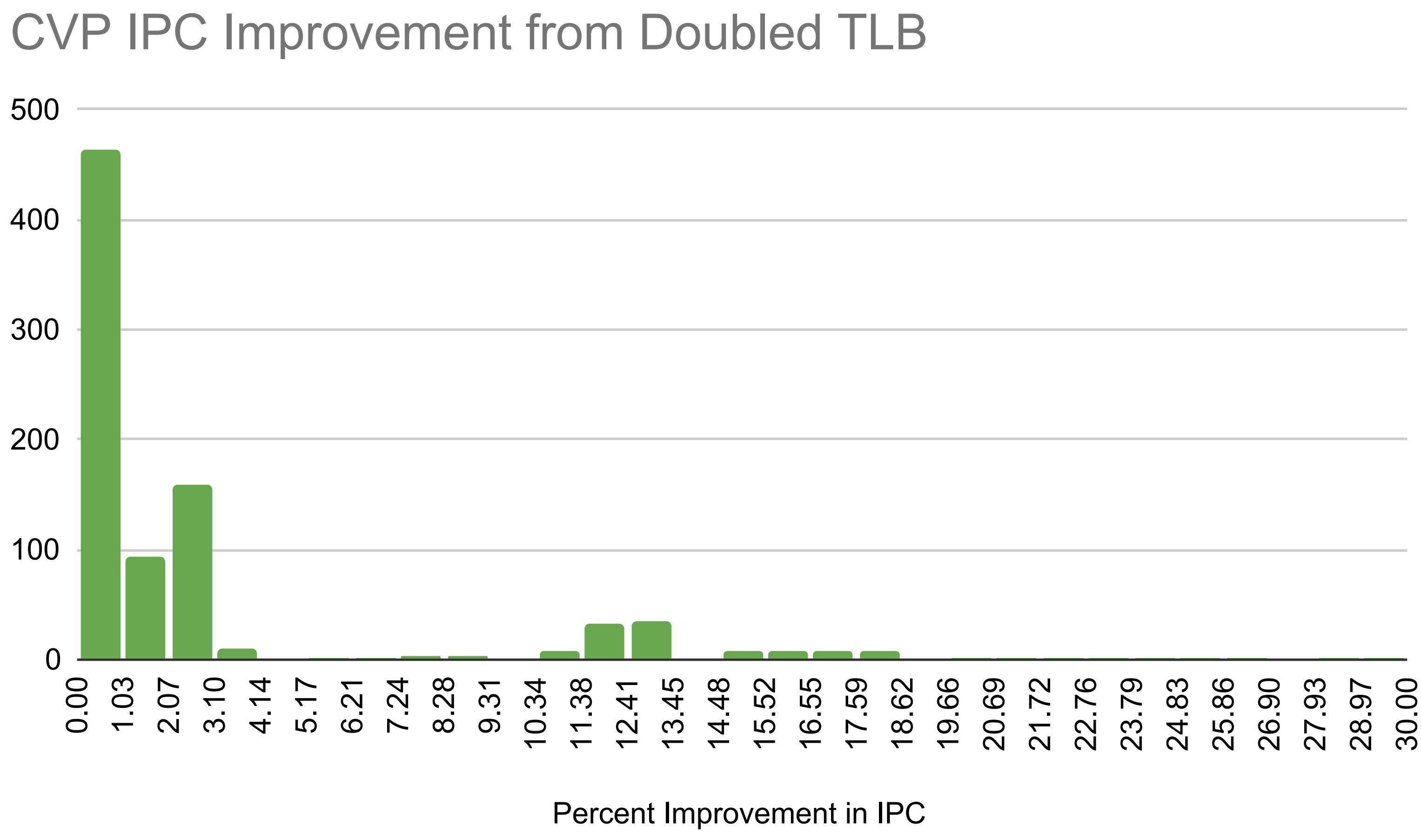

Figure 4.3: CVP IPC Improvements

5. DISCUSSION

Various approaches have been explored in realistic TLB optimization. Here we consider replacement policy, and prefetching during a page walk to reduce latency. Since the problems of TLB are fundamentally linked to the design and implementation choices of virtual memory, many potential approaches to this problem are limited by the ability to maintain backwards compatibility with older virtual memory systems and optimizations either in kernels or user programs.

5.1 CHiRP

Control-Flow History Reuse Prediction (CHiRP) seeks to improve performance by optimizing the replacement policy of the TLB structure. LRU is commonly used, but not in any way particularly suited for the job of TLB replacement. The replacement policy relies upon a few features: PC value, global path history, conditional branch history, and unconditional branch history. These features are used to create a signature that is used to index a table. This table contains saturating values that are compared to a threshold to make replacement decisions.

CHiRP was able to achieve a 28.21% reduction in MPKI compared to LRU. This was able to outperform the policies that it was compared to that were designed for general cache replacement. A more significant improvement of 4.8% was achieved over LRU in IPC. The workloads used for CHiRP's study were have significant overlap with our own, and between the inclusion of more traces and the difference in simulation architecture, the difference between our 2.3% improvement in IPC and 4.8% seems reasonable.

Unlike with other cache structures, bypassing is not a relevant problem for the replacement policy to solve, slightly reducing the scope of the problem. Programs generally do not access a page only once or a small number of times total, and so the notion of bypassing the TLB, in the sense that we may do with a general cache, is not relevant.

5.2 ASAP

Address Translation with Prefetching (ASAP) prefetches values from the lower levels of the page table in the event of a miss in the TLB. This is made possible by restructuring the virtual memory system to lay the page table out in continuous memory, essentially, removing the features of virtual memory for this region. Lower level addresses can then be calculated easily from the reliable layout of *physical* memory. These prefetches are cached normally in the L1_D cache, enabling the page walk occurring at the same time to scoop them up much faster, and decreasing overall latency.

As a result of this strategy and the minor changes required to implement it, an average reduction in page-table latency of 25% was achieved natively, and this increased to 45% under virtualization. Latency is not a direct topic of study, so the degree to which our works can inform each other is limited. There is no discussion of IPC improvement; it would be reasonable to assume there is no significant increase present, which would align with our study. Since ASAP, there has been some development in TLB prefetching that sees a 37% reduction in memory references resulting from page walks, and measures IPC, seeing an average of 11.1% speedup[20].

5.3 Discussion

As ASAP and CHiRP optimize different parts of the TLB and page walk system, it is not inconceivable that they could be combined, to both reduce the rate of page walks and the penalty they create when they must happen. Such improvements in the virtual memory space will only continue to be relevant as the size of workloads grow, placing ever more stress on TLBs. While we have shown a limited relevance to better TLB performance in the form of our doubled TLB, the relevance certainly may increase as the minority of workloads that benefit from increases TLB performance grows.

The viability of such a combined approach is high. CHiRP is exclusively an architectural change requiring no other support and addresses only the problem of replacement. This isn't to say it would be simple, but would be significantly less complex than proposals that break backwards compatibility with radix-tree page tables and existing virtual memory features. Furthermore, ASAP, while requiring operating system work, is somewhat comparable to the effort undergone to support huge pages in Linux[16] and addresses only reducing latency. There is nothing preventing this from being an optional feature in the operating system, and no distinct downside to excluding it. Further the two proposals address different domains of the problem.

REFERENCES

[1] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, D. Brooks. 2014. Profiling a warehouse scale computer. In *ISCA 15 Proceedings of the 42nd Annual International Symposium on Computer Architecture* 158-169

[2] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. Jiménez. 2020. Chirp: control-flow history reuse prediction. In *53rd Annual IEEE/ACM International Symposium on Microarchiecture (MICRO)*

[3] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot. 2019. Prefetched address translation. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* 1023-1036

[4] T. Juan, T. Lang, and J.J. Navarro. 1997. Reducing tlb power requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* 196-201

[5] T. W. Barr, A. L. Cox, adn S. Rixner. 2010. Translation caching: Skip, don't walk (the page table). In *SIGARCH Comput. Archit. News* vol. 38, no. 3 48-59

[6] B. Jacob, S. Ng, and D. Wang, Memory Systems: Cache, DRAM, Disk. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[7] Intel. 2017. 5-Level Paging and 5-Level EPT. Intel white paper. https://software.intel.com/content/dam/develop/public/us/en/documents/5-level-paging-white-paper.pdf

[8] Sally A. McKee. 2004. Reflections on the memory wall. In Proceedings of the 1st conference on Computing frontiers (CF '04). Association for Computing Machinery, New York, NY, USA, 162.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, Clearing the clouds: A study of emerging scale-out

workloads on modern hardware, in Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 3748.

[10] Hennessy and D. A. Patterson. 2019. Computer Architecure a Quantitaive Approach. 79

[11] O. Villa et al., "Scaling the Power Wall: A Path to Exascale," SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 830-841

[12] "SYSTEM COMPONENTS: Dynamic Relocation" (PDF). System/360 Model 67 Time Sharing System Preliminary Technical Summary (PDF). IBM. 1966. p. 21.

[13] Bhattacharjee, Abhishek; Lustig, Daniel (2017). Architectural and Operating System Support for Virtual Memory. Morgan & Claypool Publishers. p. 1. ISBN 9781627056021. Retrieved March 19, 2021.

[14] J. L. Hennessy and D. A. Patterson. 2019. Computer Architecure a Quantitaive Approach. 80

[15] https://www.kernel.org/doc/gorman/html/understand/understand006.html

[16] https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html

[17] Intel Corporation, "Intel 64 and ia-32 architecures and optimization reference manual", Intel Corporation, Tech. Rep. Order Number: 248966033, 2016

[18] Müller M., Whitney B., Henschel R., Kumaran K. (2011) SPEC Benchmarks. In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA.

[19] https://ieeetcca.org/2020/11/16/championship-value-prediction-2-workshop-cvp2/

[20] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas, "Exploiting Page Table Locality For TLB Prefetching", Proceedings of the 47th International Symposium on Computer Architecture, June 2021