Near-Memory Address Translation

THÈSE N^{O} 7875 (2017)

PRÉSENTÉE LE 15 SEPTEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DE SYSTÈMES PARALLÈLES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Javier PICOREL OBANDO

acceptée sur proposition du jury:

Prof. D. Atienza Alonso, président du jury
Prof. B. Falsafi, directeur de thèse
Prof. D. Wood, rapporteur
Prof. A. Bhattacharjee, rapporteur
Prof. E. Bugnion, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

Acknowledgements

This thesis has been a significantly long journey, which I undoubtedly would not have been able to complete alone. I would like to express my gratitude to some of the people who helped me make this thesis happen. Part of this thesis belongs to you.

First and foremost, I would like to thank my academic advisor Babak Falsafi, for providing me with the opportunity to pursue a PhD under his supervision. His financial support throughout my studies allowed me to focus on doing research. I also greatly benefited from the honest research and competitive environment he created at EPFL. I am without a doubt three orders of magnitude better than when I started. From Babak I learned to see the big picture at all times, a skill very valuable that I will use for the rest of my life.

I would like to thank Abhishek Bhattacharjee, Edouard Bugnion, and David Wood for serving on my thesis committee. Their insightful comments, questions, and feedback have truly improved the quality of this document. I would also like to thank David Atienza who accepted to be my thesis jury president.

A very special thanks goes again to Abhishek Bhattacharjee, this time as a collaborator (and not as a committee member). His vast wisdom and positive attitude helped me clear my thoughts in times when I felt completely lost.

I am also grateful to Eric Sedlar and Peter Hsu to give me the opportunity to intern at Oracle Labs for a seven-month period in California. Eric Sedlar has always surprised me with his vision and leadership skills. I always pay close attention to what he has to say. Peter Hsu was a great mentor and an immense source of technical, practical information, and know-how. Peter and I had great and long discussions about research and the future of computer architecture which I enjoyed every single bit. Thank you both.

I would not have gone to graduate school if it was not for my undergrad advisor, Jose Luis Briz. In fact, Jose Luis Briz is the person who made me interested in computer architecture, as he was the professor teaching the first computer architecture course in my undergrad. Thank you Jose Luis for all your help and guidance!

Special mention goes to a group of people who started the PhD with me: Nadia Christakou, Iraklis Psaroudakis, and Vasileios Trigonakis. Besides colleagues, I can call without a doubt these people my friends, without them the PhD would have been just unbearable. We have all shared our research and personal problems, spent great moments having lunches and dinners, traveling, going out, and many other great things I do not even remember. I thank you all and I am sure we will keep our friendship for the rest of our lives.

I was lucky enough to work in a great and amazingly fun research group, Parallel Systems Architecture Lab (PARSA), surrounded by brilliant people whose assistance, feedback, and support helped me become a better researcher, while their friendship made my PhD journey incredibly fun:

- Alexandros Daglis has been an excellent academic sibling and, much more importantly, friend for the past five years. We enrolled on the PhD program with only a year difference and therefore our PhD endeavours overlapped almost entirely. Furthermore, we have shared the same office for more than three years. We have experienced together both the tough and the fun times of the PhD life. We have spent countless hours together working on common projects, brainstorming, and giving each other insightful, valuable, and honest-to-goodness feedback. We never hesitated a single second to spend time on each other's projects in case we needed help, advise, or feedback, while our many fruitful discussions enriched my knowledge significantly. More importantly, we have always had each other's back even when the easy approach was to just look away. He has also been an immense source of knowledge in Latex to the point I usually call him "the little wizard of Latex" during conference deadlines. We also had a lot of fun in the office making jokes, laughing, and procrastinating together, which I have to admit helped me to remain mentally healthy during the tough times of the PhD. Not to mention the fun we had at restaurants, bars, travels, and conferences (e.g., ISCA'16 in Korea). I am sure we will keep our friendship for the rest of our lives. Thank you Alex!

- Djordje Jevdjic, Cansu Kaynak, Onur Kocberber, and Stavros Volos have also been great academic siblings and, as they were more senior than me, mentors in some way. I definitively consider these four people my friends. I have had the privilege of not only learning a lot from them but also collaborating with most of them. Every single one of them taught me something different. From Djordje I learned that if there is time to leave things for tomorrow, leave them for tomorrow. Djordje has always given me great and high quality feedback whenever I needed it. I owe him many beers for listening to my crazy ideas and always replying with sharp and accurate comments. Cansu taught me about the importance of perfectionism and careful and detailed planning. From Onur I learned that a bit of chaos is necessary for good research. Onur was always up for brainstorming on crazy ideas, either his or mine. I not only learned a lot from Onur but his positive attitude towards work and life in general had a big influence on me. Stavros Volos was an amazing friend and labmate during all the time our PhD journeys overlapped. Stavros Volos taught

me to always take a careful look at the tradeoffs, and use back-of-the-envelope calculations to have always a first-order model in my mind of what I am doing. Stavros was also a vast source of knowledge in Flexus. Thank you for that. We all five spent a lot of time together working and having a lot of fun. I have lost count of how many beers, dinners, and travels we had. Big thanks for everything. This journey would have not been as successful and fun without you!

- Mario Drumond, Nooshin Mirzadeh, Stanko Novakovic, Dmitrii Ustiugov, Arash Pourhabibi, and Mark Sutherland have been great friends and labmates. We have spent a lot of time together in the lab, probably more time than we should have. I think I have spent more time working with some of these people on their theses than on my own. However, it was totally worth if I was able to help them. I will always remember some characteristics from each of these people. Mario has the ability to sound like he knows what he is talking about even when he has no clue. Nooshin is always in a good mood and has regularly a smile on her face. Stanko was constantly obsessed with his ASPLOS'14 paper, soNUMA. Dmitrii had a really good touch with all the interns. Arash wanted to do everything in Latex, if we could browse the web with Latex he would do it. Mark shared my obsession with the amazing movie Glengarry Glen Ross. Coffee is for authors! We have also had a lot fun together during lunches, dinners, travels, and conferences. I will definitively miss you. Good luck, I am sure you will do well in the future!

- Mike Ferdman and Alisa Yurovsky have always been great people to have around. Mike was the most senior person when I arrived at the lab. He has always surprised me with his technical skills and vast knowledge. I hope one day I will be able to say that I know as much as Mike. I remember he always gave me the opportunity to speak and cared of what I was thinking, even though I was the most junior person in the group. Thank you for this Mike, I really appreciate it. Alisa is an awesome person who is always positive and with a big smile on her face. It is always nice to talk to her. I remember her great organization skills when she was in Lausanne with barbecues by the lake and Halloween parties. Thank you both Mike and Alisa!

- Thanks to Adileh Almutaz, Pejman Lofti Kamran, and Boris Grot for the fruitful collaborations in the early stages of my PhD journey. Pejman offered me the chance to work on my first project in the lab. Almutaz and I spent a lot of time together distributing the first images of CloudSuite during my first ISCA conference. Last but not least, Boris was a great source of inspiration and positiveness, I enjoyed my time with him in all the collaborations we had. Thank you all three!

- Sotiria Fytraki, Damien Hilloulin, Aris Ioannou, Georgios Psaropoulos, and Evangelos Vlachos have been great labmates for the short period we overlapped. We had great discussions and time together. Special mention goes to Sotiria Fytraki for sharing the office with me for a few years. Hope to see you all in the future!

- Warm thanks to Effi Georgala for the fun breaks in the lab. She is always positive and

open for discussion, specially when it comes to accents in English. She also gives amazing advice on work and life. Thank you Effi!

- I would like to thank Rodolphe Buret for all the technical support he has done during my PhD. Stéphanie Baillargues and Valérie Locca were always there for providing administrative assistance. Ousmane Diallo was always an excellent source of advice due to his vast experience in industry. I would also like to thank Ousmane for being kind enough to translate my thesis abstract into French.

Though outside of the lab, there were other people at EPFL that made my PhD life much simpler and enjoyable and of course deserve credit:

- I would like to thank Stella Giannakopoulou, Christos Kalaitzis, Manos Karpathiotakis, Katerina Liapi, Anastasia Mavridou, Pavlos Nikolopoulos, Matt Olma, Eleni Smyrnaiou, and Stefanos Skalistis (a.k.a, the Greek crowd). I hope not to miss anyone or misspell any last name. As I could not find a Spanish crowd at EPFL, I have to admit you all were a very nice substitute. I barely noticed the difference. Thank you all for the time spent in lunches at EPFL, dinners in Lausanne, and Greek parties.

- A very warm thanks to Karolos Antoniadis, Georgios Chatzopoulos, Tudor David, David Kozkaya, Majet Pavlovic, and Dragos-Adrian Seredinschi. I have always enjoyed my time with you and felt part of LDP. Thank you all!

- A big thanks to Arwa Mrad for many things. She is a very energetic and positive person who is always trying to have fun and laugh at herself and others. She always puts everyone before herself and tries to make people feel well. Her cooking skills are beyond measure. Thank you Arwa!

- I would like to thank Grace Zgheib, Ana Petkovska, and Sahand Kashani-Akhavan for all the time spent answering my questions, either related to Quartus or Jenkins. Thank you all three!

- Thanks to Lana Josipovic for always being around since we first met. Although we did not have much of a choice but to meet at the beginning, afterwards we were doing it by our own free will. Lana has taught me many things but the most important is to appreciate good beer. The countless working hours at EPFL, specially during late hours and weekends, were definitively much more fun with you around. My last year would definitively not have been the same without you and I enjoyed your company every single bit.

I would like to thank the Spanish community in Lausanne for making me feel close to Spain at all times. Thanks to Xexu Ayala, Javier Bello, Pedro Canovas, Lina Hernandez, Chari López, Marta López de Asiaín, Diego Martin, Alberto Molina, Nicolás Mora, Andrea Navares, María Jesus Salvatierra, and Javier Villasevil. Though we do not meet as much as I would like, I enjoyed all time we spent together in Lausanne.

I would like to thank Sushant Malhotra, Sonali Parthasarathy, and Stephan Wolf for making my life during my internship in the bay area much nicer. I appreciate all the time we spent together. I feel we will become life-long friends and I hope we see each other more often in the future.

Special thanks to my life-long friends back in Spain whom I do not see as often as I would want. Nevertheless, I know you are there and you will always be. Thank you Alberto Samper Puertolas, Carlos Munilla Burillo, Jaime Rubio Atienza, and Alexander Guede García.

Last but not least, I would like to thank my family for all their love and support in all these years and the years to come. My parents, Rafael Picorel Castaño and María del Pilar Obando Soto, and my sister, Ivana Picorel Obando. Special mention goes to my awesome grandmother Melba Soto Saavedra for always taking care of me when I was little. I would also like to thank my ant and uncles, Myriam Obando Soto and her husband Richard Lee, Luis Alfredo Obando Soto, Aymer Obando Soto, and Julian Obando Soto. I will forever remain indebted to all of you.

Abstract

Virtual memory (VM) is a crucial abstraction in modern computer systems at any scale, from handheld devices to datacenters. VM provides programmers the illusion of an always sufficiently large and linear memory, making programming easier. Although the core components of VM have remained largely unchanged since early VM designs, the design constraints and usage patterns of VM have radically shifted from when it was invented. Today, computer systems integrate hundreds of gigabytes to a few terabytes of memory, while tightly integrated heterogeneous computing platforms (e.g., CPUs, GPUs, FPGAs) are becoming increasingly ubiquitous. As there is a clear trend towards extending the CPU's VM to all computing elements in the system for an efficient and easy to use programming model, the continuous demand for faster memory accesses calls for fast translations to terabytes of memory for any computing element in the system. Unfortunately, conventional translation mechanisms fall short of providing fast translations as contemporary memories exceed the reach of today's translation caches, such as TLBs.

In this thesis, we provide fundamental insights into the reason why address translation sits on the critical path of accessing memory. We observe that the traditional fully associative flexibility to map any virtual page to any page frame precludes accessing memory before translating. We study the associativity in VM across a variety of scenarios by classifying page faults using the 3C model developed for caches. Our study demonstrates that the full associativity of VM is unnecessary, and only modest associativity is required. We conclude that capacity and compulsory misses—which are unaffected by associativity—dominate, while conflict misses rapidly disappear as the associativity of VM increases. Building on the modest associativity requirements, we propose a distributed memory management unit close to where the data resides to reduce or eliminate the TLB miss penalty.

Keywords: Virtual memory, address translation, die-stacked memory, near-memory processing, MMU, TLB, page table, DRAM, servers

Résumé

La mémoire virtuelle (VM) est une abstraction cruciale dans les systèmes informatiques modernes à n'importe quelle échelle, des appareils portatifs aux centres de données. VM fournit aux programmeurs l'illusion d'une mémoire toujours suffisamment grande et linéaire, facilitant ainsi la programmation. Bien que les composants principaux de VM restent largement inchangés depuis les premières versions de VM, les contraintes de conception et les modes d'utilisation de VM ont radicalement changé depuis leur invention. Aujourd'hui, les systèmes informatiques intègrent des centaines de gigaoctets à quelques téraoctets de mémoire, tandis que les plateformes informatiques hétérogènes étroitement intégrées (ex., CPU, GPU, FPGA) sont de plus en plus omniprésentes. Comme il existe une tendance évidente à étendre la VM de la CPU à tous les éléments informatiques du système pour un modèle de programmation efficace et facile à utiliser, la demande continue d'accès à la mémoire plus rapide nécessite des traductions rapides de téraoctets de mémoire pour tout élément informatique du système. Malheureusement, les mécanismes de traduction conventionnels ne permettent pas de traduire rapidement car les mémoires contemporaines dépassent la portée des mémoires caches de traduction d'aujourd'hui comme les TLB.

Dans cette thèse, nous fournissons des notions substantielles sur la raison pour laquelle la traduction d'adresse se situe sur la voie critique d'accès à la mémoire. Nous observons que la flexibilité traditionnelle entièrement associative pour mapper n'importe quelle page virtuelle sur n'importe quel cadre de page empêche l'accès à la mémoire avant de traduire. Nous étudions l'associativité des VM dans différents scénarios en classifiant les défauts de page à travers le modèle 3C développé pour les mémoires caches. Notre étude démontre que l'associativité complète de VM est inutile, et seule une associativité partielle est requise. Nous concluons que la capacité et les manquements obligatoires—qui ne sont pas affectés par l'associativité—dominent, alors que les manquements des conflits disparaissent rapidement au fur et à mesure que l'associativité de VM augmente. En s'appuyant sur les exigences partielles d'associativité, nous proposons une unité de gestion de la mémoire distribuée proche de l'endroit où résident les données pour réduire ou éliminer la pénalité des manquements du TLB.

Mots clefts : Mémoire virtuelle, traduction d'adresse, couches empilées de mémoire, traitement en mémoire-proche, MMU, TLB, table des pages, DRAM, serveurs

Contents

List of Figures

List of Tables

1 Introduction

Virtual memory is a crucial abstraction in any modern computer system, from handheld devices to datacenters. Virtual memory provides programmers the illusion of an always sufficiently large and linear memory, making programming easier. Virtual memory abstracts the programmer from the system's available memory capacity and explicit managing the allocation between storage and memory. Furthermore, virtual memory enforces isolation among applications, allowing them to access their own memory only, and thus preventing applications to overwrite one another's content. In summary, virtual memory is a fundamental abstraction to programmability, code portability, and memory protection, and a key part of any modern computer system.

The hardware and the operating system (OS) manage the relation between the virtual addresses employed by the software and their actual location in the physical memory. With virtual memory, every memory operation in the software has to be translated from a virtual to a physical address before accessing memory. Remarkably, the core components of the address translation mechanism have remained largely unchanged since early virtual memory designs of the 1960s. The OS manages the virtual-to-physical map in a data structure while a dedicated hardware cache, the translation look-aside buffer (TLB), caches frequently used translations. The virtual memory subsystem achieves high performance when most of the translations of the memory accesses are served by the TLB.

However, the design constraints and usage of virtual memory are nowadays greatly different from its original invention. One of the first machines to support virtual memory, the General Electric GE-645, integrated 2MB of physical memory. In contrast, modern systems integrate hundreds of gigabytes to a few terabytes of memory, as the cost of physical memory has been decreasing at a rate similar to Moore's law [69]. For instance, HP DragonHawk [84] integrates up to 6TB of physical memory and Huawei KunLun [133] supports physical memories of up to 32TB. Furthermore, emerging non-volatile memory (NVM) technologies such as Intel-Micron 3D XPoint [1] will further increase the pressure on the virtual memory subsystem. The continuous demand for fast memory accesses calls for a virtual memory subsystem that can provide fast translations to terabytes of memory.

Unfortunately, while integrated memory capacity per computer system has been growing exponentially, the reach of the TLB has grown at a much modest rate; the fraction of physical memory covered by the TLB has decreased over time. This problem is commonly known as the limited TLB reach. Two decades ago, an Intel Pentium Pro server included a TLB of 64 entries and up to 2GB of physical memory [24, 49]. Today, a commodity server using the Intel Haswell architecture integrates a TLB hierarchy of 1124 entries and 256GB of physical memory [131]. In two decades, the size of the TLB has only increased by a factor of 18, whereas the size of the physical memory has scaled 128 times. The scaling gap between TLBs and physical memory is even larger for large-memory servers (e.g., Huawei KunLun) or with the integration of emerging non-volatile memory as part of the physical memory. The end result is that the primary component used to achieve high performance address translations is no longer as effective.

Furthermore, to accommodate for larger memory capacities, the architecture of memory systems has become more complex over time. The building block of commodity memory is the dual in-line memory module (DIMM), which encompasses a set of DRAM chips integrated on a PCB. Each DIMM connects to a DDRx bus through a physical slot in the motherboard. A few DIMMs share the same DDRx bus which usually connects to a CPU chip through a memory channel. Unfortunately, as DDRx buses operate at higher speeds to keep up with faster logic chips like CPUs, other issues arise such as poor signal integrity due to signal reflexion and cross-talk. This issue is exacerbated as more DIMMs are placed in the same DDRx bus. As an example, the original DDR SDRAM specification allowed four DIMMs in a single channel, while modern DDR4 specifications only allow two DIMMs. The end result is that as the frequency of DDRx buses increase, system architects have to reduce the number of DIMMs allowed per memory channel [45], reducing its memory capacity.

At the same time, limited pin counts of logic chips has limited the number of memory channels that can be supported on a single package. Consequently, system designers are scaling memory capacity by either integrating more CPU chips or placing logic between the CPUs and DRAM (e.g., Buffer-on-board [45]). The logic chips are interconnected with narrow and high-speed serial interfaces (e.g., Intel QuickPath Interconnect or AMD HyperTransport) which involve costly latencies of tens of nanoseconds per hop [87, 106, 165]. For instance, HP DragonHawk contains 16 CPU chips and Huawei KunLun integrates 32 CPU chips. In such systems, a memory access to a remote location can incur in a round-trip time of hundreds of nanoseconds. In summary, though memory clocks have increased to keep up with the requirements of modern CPUs, the distributed architecture of modern memory systems has considerably increased the memory access latency.

Long memory latencies affect the performance of the virtual memory subsystem. A TLB miss requires walking the data structure used by the OS to store the virtual-to-physical map. This data structure is called the page table. This overhead is known as the TLB miss penalty and the walking operation is called the page walk. In current systems, the page walk can take several memory accesses, for instance, up 4 memory accesses in x86_64 [47], while up to 5 memory accesses will be required in the near future [70]. Therefore, a page walk can incur a latency overhead of several hundreds of nanoseconds in large memory systems. As a result, TLB misses

are a severe virtual memory performance bottleneck, and consequently, a system performance bottleneck today.

Computing architecture has also incurred profound changes from when virtual memory was invented. However, not only the memory architecture has had profound changes from when virtual memory was invented, the computing architecture has also dramatically shifted away. The contradictory trends of the increase in computing demands and the end of Dennard and slowdown of silicon density scaling has pointed system architects toward specialized architectures to maximize computing density and efficiency. Modern computer systems integrate not only several CPUs, but also GPUs for tasks such as deep learning, analytics, and high-performance computing. ASICs are now ubiquitous for accelerating cryptography, video and signal processing, and compression/decompression, and FPGAs are becoming more and more popular to accelerate applications amenable to spatial computation. The end result is that contemporary computer systems are comprised of a distributed network of heterogeneous computing elements and not a single computing building block.

For efficiency and programmability, system architects are advocating for tight integration of all the computing elements in the system [64, 85, 132, 142], which means these elements must actively interact with each other and operate on the vast datasets available in the system. Hence, all of the distributed and heterogeneous computing elements in the system require fast address translations to share access to terabytes of memory. Any approach for an efficient address translation mechanism appears to be challenging as one can see the computation elements and the translation information as a fully connected bipartite matching, making any approach to localize translations challenging. Furthermore, a translation mechanism has to be robust across largely different computing elements each of which with very different idiosyncrasies and characteristics.

To make matters worse, many modern applications exhibit a lack of data locality due to their abundant use of dynamic data structures (e.g., hash tables, skip lists). A few examples are graph processing, key value stores, online transaction processing (OLTP), online analytical processing (OLAP), and Web search. Processing on a graph involves chasing pointers over a large and irregular data structure [10]. Modern key value stores manage the data in either a hash table or skip list to avoid linear search times [65]. OLTP database management systems employ indexes such as B+ trees and skip lists for fast accesses to the database [180]. OLAP database management systems use hash tables for the ubiquitous join relational operator [114]. Last, web search engines provide a mapping between terms and inverted lists through a hash table or B+ tree [34]. All these pervasive applications exhibit frequent pointer chasing over hundreds of gigabytes to a few terabytes of data, stressing not only the hardware data caches but also the hardware translation caches (i.e., the TLB hierarchy). In such scenarios, TLB misses can incur up to 50% of the total execution time of an application [22, 107].

In summary, both design constraints and usage patterns of virtual memory have largely changed from when it was invented. Profound changes in the computing and memory architectures, along with the lack of data locality of modern and ubiquitous applications are stressing the traditional

virtual memory architecture. In this thesis, we identify TLB misses as a critical performance bottleneck and propose mechanisms to either reduce or eliminate its overhead.

1.1 Computer System Trends

Large IT services deployed by technology giants such as Microsoft or Amazon are often hosted in large datacenters, comprised of hundreds of thousands of servers and referred to as warehouse-scale computers (WSCs) [21]. With both data volumes and user request rates growing at a pace comparable to Moore's Law [3, 64], server computing platforms are under constant pressure to keep up with the increase in service demands.

While the demand for server performance continues to grow, the end of Dennard scaling (i.e., the ability to increase performance per area unit at the same power density) has made the task of increasing performance with each technology node difficult. Hence, continued transistor density increases results in the inability to power up the whole chip at the maximum performance point. As a result, the growing demand for IT services has led to an era of ever-larger and power-hungry datacenters. The power draw of top-of-the-line datacenters has increased 10-fold over the last decade, from around 5MW per datacenter in 2005 to 50MW in 2015 [129].

At the same time, scaling transistors to smaller technology nodes is becoming increasingly complex [163]. This complexity translates into higher costs for fabrication and lower device yields, both increasing the cost per chip across transistor generations. For example, Intel has already ended the "tick-tock" development model, extending the life of each process technology [14]. Intel Skylake, the first microarchitecture in the 14nm technology, was succeeded by Intel Kaby Lake in the same technology node (instead of the expected 10nm). Additionally, feature sizes below a few nanometers are reaching physical limitations. For instance, premium fabs such as Intel and TSMC have not yet disclosed a road map below 5nm. As a result, the compute density of future chips across technology nodes is likely to stagnate.

The confluence of the slowdown of Dennard and transistor density scaling and the increase in computing demands points system designers towards specialized architectures to maximize processing efficiency and density. In fact, specialized computing elements have already been deployed in datacenters at reasonable large scales [13, 36, 61, 101]. Facebook deploys GPUs for machine learning algorithms [61], Microsoft integrates FPGAs in their Bing servers [36], Google deploys ASICs for neural networks [101], and Amazon AWS has started offering server instances with FPGAs and GPUs [13]. The conflicting trends of technology scaling and computing demands will drive the shift from traditional general-purpose computing towards more specialized architectures in the datacenter.

Memory is not oblivious to the problems associated with smaller technology nodes. The rate at which the capacity of a DIMM increases has slowed down in recent years due to the difficulty in decreasing the size of a cell's capacitors [45]. As a result, memory manufacturers have

turned to novel technologies that will allow them to scale memory capacity beyond traditional two-dimensional DRAM chips. Recent advances in three-dimensional integrated circuits have enabled several memory vendors such as Hynix and Micron to stack multiple DRAM dies on top of a logic chip within a single package [95, 127, 162], scaling capacity by utilizing the third dimension. The ample benefits in bandwidth and proximity to data has triggered a research wave of custom memory-side processing units (MPUs) implemented in the logic layer [10, 71, 111, 130, 145, 146, 176]. Along with the inclusion of die-stacked memories in industry products [68, 149], there is substantial evidence suggesting that future server systems may integrate MPU-capable memory chips.

Overall, computer systems are steadily moving towards an architecture of a distributed network of heterogeneous computing and memory chips interconnected with narrow and high-speed serial interfaces [153].

1.2 Unified Virtual Memory

Computer systems with heterogeneous computing and memory chips must offer an efficient and simple programming model to ensure widespread adoption. Initiatives in industry such as the Heterogeneous System Architecture (HSA) Foundation, a consortium founded by AMD, ARM, Qualcomm and Samsung among others, are proposing a shift towards unified virtual memory (VM) between CPUs and any other computation element in the system [85]. Unified VM has many benefits: pointers are equally valid in any computation element, simplifying data sharing, eliminating redundant data copies, and manually maintaining data consistency. Additionally, VM enables efficient fine-grained memory accesses and transparent memory allocation and protection. Essentially, VM provides a familiar and powerful abstraction to programmers and operating systems alike. All these benefits have led commercial GPUs to adopt an HSA-compliant unified virtual memory, beginning with AMD's Carrizo chip [174]. Given the benefits of HSA-style unified virtual memory and its early adoption into commercial products, we expect future computation units to also follow this path.

The programmability benefits provided by virtual memory are further amplified at the scale of large IT services. Specifically, Google has reported that shifting the burden of managing complex interactions away from the programmer to the operating system makes the code base notably simpler [20]. Simplifying the codebase is of great importance in warehouse-scale environments where source code is touched by thousands of developers, with significant software releases on a daily basis.

Unfortunately, the benefits of VM currently come at a significant performance overhead. A simple approach consists of offloading address translation to the CPU cores on behalf of the execution elements (e.g., GPU, ASIC, MPUs) [71, 153]. Though this only registers simple hardware, the execution elements and CPUs reside on different chips, resulting in frequent and time-consuming cross-chip traffic and valuable CPU time lost. Similarly, the execution elements can offload

address translation to I/O memory management units (IOMMUs), introduced in commercial CPUs to allow devices to translate virtual addresses [166]. As IOMMUs also reside in the CPU, they require frequent and costly cross-chip communication which compromises performance. A more sound approach is to integrate a memory management unit (MMU) per execution element, just like the CPU cores, each having a TLB hierarchy and page walker. However, such MMUs suffer from the same performance overheads that the CPUs do, limited TLB reach and high TLB miss penalty. Although translations provided by the TLBs achieve low overhead, page walks are frequent as contemporary memory sizes exceed the reach of modern TLBs [22, 139, 140]. Furthermore, due to the arbitrary distribution of page table entries across all the memory chips in the system, page walks may involve several cross-chip transfers of tens of nanoseconds per hop [87, 106, 165] for each level of the page table [70], resulting in unacceptable overheads which diminish the gains of specialization.

In summary, as computer systems are becoming more heterogeneous, there exists a call for an efficient and easy to use programming model. Extending the CPU's virtual memory to all computing elements in the system is a crucial step in this direction. Unfortunately, the implications are that any computing element in the system has to efficiently translate addresses spanning a distributed physical memory of hundreds of gigabytes to a few terabytes. This thesis identifies TLB misses—page walks—as a critical performance bottleneck in virtually addressed execution units such as CPUs, GPUs, or MPUs and proposes mechanisms to eliminate the page walk overhead based on the novel observation that the conventional full associativity of VM is unnecessary.

1.3 A Case Study with Memory-side Processing Units

In this thesis, we illustrate and evaluate the benefits of our novel virtual memory subsystem in the context of memory-side processing units (MPUs). Though we can evaluate our VM subsystem in any context (e.g., CPUs), we believe MPUs stress the address translation mechanism the most and hence are the most challenging scenario for the following three reasons. First, MPUs are scattered across a large number of memory chips, making cross-chip memory traffic common. For instance, modern proposals consider memory networks of 4 to 16 memory chips [68, 71, 146, 149]. This memory traffic behavior is exactly the same as in multi-socket CPUs (e.g., Huawei's KunLun 32 CPU machine). As page table entries are arbitrarily distributed across the memory chips, page walks require expensive cross-chip traffic. Second, unlike modern CPUs, MPUs leverage the reduced physical distance to data stored in memory and integrate either shallow cache hierarchies or no cache at all [64]. In contrast, CPUs usually integrate deep cache hierarchies of multiple megabytes. These caches may store parts of the page table [108], avoiding accessing the memory for page walks that hit in the cache hierarchy. Third, tight area and power constraints in the logic layer of the memory chip [154] preclude aggressive computation units such as OoO cores with large instruction windows and speculative execution. These large instruction windows and speculative execution can help latency by overlapping page walks with useful work. In contrast, because MPU cores provide from little to no overlapping, they expose all of the page walk time.

As a result, we focus on memory-side processing units (MPUs) for the rest of this thesis, although our insights and translation mechanisms are widely applicable to any other context.

1.4 Thesis and Dissertation Goals

Given the large programmability benefits of unified virtual memory between all execution units in the system, the next logical step towards a flexible and efficient implementation is to tackle the performance bottleneck of page walks. Therefore, the goal of this dissertation is to provide fundamental insight into the reason why address translation in general and page walks in particular sit on the critical path of accessing physical memory. Then, we build on the novel observation that the conventional full associativity of virtual memory is a largely unnecessary feature, to build an efficient VM system with no compromises.

The **thesis statement** is as follows:

Reducing the associativity of virtual memory nearly eliminates the translation overhead enabling a scalable unified virtual address space.

1.5 Associativity in Virtual Memory

In this chapter, Chapter 3, we perform an associativity study of virtual memory—the number of distinct page frames that a virtual page can map to—and conclude that the conventional full associativity of VM is unnecessary, and only modest associativity is required. For our study, we classify across a wide range of scenarios page faults by employing the classic 3C model originally developed for caches [83]. Our study demonstrates that the full associativity of VM is unnecessary, and only modest associativity is required. We conclude that capacity and compulsory misses (which are unaffected by associativity) dominate, while modest associativity is sufficient to eradicate conflict misses. More specifically, for working sets that are memory resident, compulsory misses dominate when the VM associativity equals the number of processes executing in the system. For working sets that exceed the memory size, capacity misses dominate and grow as a function of the working set size. In contrast, conflict misses become scarce once the associativity of virtual memory matches the number of processes in the system, and virtually disappear after the first few additional ways.

Essentially, we make the observation that one can think of memory as simply a fully associative software-managed virtually addressed cache, where the tags are the page table entries and the data are the page frames. In fact, our associativity trends for VM match seminal work on set-associative caches [35, 83]. To provide nearly all the flexibility benefits of full associativity with much faster translation times architects could exploit the modest associativity requirements. Though in a completely different context, our work is in spirit similar to that of three decades old work on caches [81]; showing that set-associative or direct-mapped caches can provide nearly all

the flexibility benefits of full associativity with much faster cache access times.

1.6 Set-Associative Virtual Memory

This chapter, Chapter 4, proposes SAVAgE (Set-Associative VirtuAl mEmory), an efficient translation mechanism for MPUs that eliminates most of the overhead of page walks. Our translation mechanism builds on the modest associativity requirements and characteristics of our network of memory chips. SAVAgE restricts the associativity of VM so that a virtual address uniquely identifies a memory chip and memory partition. This design allows the MPUs to access memory as soon as the virtual address is known. Each memory partition integrates an MMU, which includes a TLB hierarchy and page table, that translates the virtual address and fetches the data, both of which are always located in the MMU's local memory partition. Translation is fast due to four reasons: First, the translation is completely localized in the partition where the page frame resides, which allows the data fetch request and its translation to always target the same partition and completely overlap. Second, a memory access within a memory partition is faster than accessing remote memory partitions, therefore achieving lower latency page walks. Third, as the page frame resides in the same partition as data, the data fetch can immediately start after translation finishes. Last, as a memory partition offers associativity on the order of a hundred thousand ways, the number of page faults due to conflicts is virtually identical to full associative virtual memory. SAVAgE achieves low-overhead page walks as the page walk and data fetch operations overlap almost entirely.

Essentially, SAVAgE turns memory into a virtually addressed software-managed cache, just that the memory is set-associative as opposed to fully associative. Again, one can think of memory as a cache, where the data array is the set of page frames, and the tag array is the page table.

1.7 Eliminating Associativity in Virtual Memory

Large-scale IT services are increasingly migrating from storage to memory because of massive data consumption rates [16, 53, 179]. Online services such as web search, social connectivity, and media streaming exhibit stringent response time requirements which mandate memory-resident data processing. Similarly, business intelligence over analytical pipelines is increasingly memory resident to minimize query response times. Furthermore, first-party workloads deployed by IT giants such as Microsoft and Google often run on dedicated servers, taking up all the system resources to ensure performance isolation and predictability [22, 79]. The end result is that memory has become pivotal to server design, which aims to maximize the throughput per server and therefore minimize the total cost of ownership of datacenters.

This chapter proposes DIPTA (Distributed Inverted Page Table), an address translation mechanism that completely eliminates the overhead of page walks. Our translation mechanism builds on the observation that the associativity of VM can be virtually eliminated for in-memory workloads.

DIPTA restricts the associativity so that a page can only reside in a few number of physical locations which are physically adjacent—i.e., in the same memory chip and DRAM row. Hence, all but a few bits remain invariant across the virtual-to-physical mapping, and with a highly-accurate way predictor, the unknown bits are predicted so that address translation and data fetch are completely independent. Furthermore, to ensure that the data fetch and translation are completely overlapped, we place the page table entries next to the data in the form of an inverted page table, either in SRAM or embedded in DRAM. Hence, DIPTA achieves zero-overhead page walks for in-memory workloads.

Essentially, DIPTA turns memory into a virtually addressed hardware-managed cache. This cache's tag array is implemented as a distributed inverted page table and is accessed in parallel with the data array [96, 147].

1.8 Contributions

In this thesis, we explore and propose techniques to eliminate the overhead of page walks. We begin by studying the associativity of virtual memory and concluding that the conventional full associativity of virtual memory is unnecessary. Based on its modest associativity requirements, we propose two techniques to eliminate the overhead of page walks, attaining almost all the flexibility of a fully associative VM with much faster translation.

Through a combination of trace-driven functional and cycle-accurate simulation, we demonstrate:

- **Address translation forms a significant fraction of the execution time when using MPUs.** We demonstrate that because of the limited reach of conventional TLBs, page walks occur frequently. Due to the particularly high TLB miss penalty in systems with MPU-capable memory chips, the execution time can increase by more than 3× due to translation.

- **Associativity study of VM on modern server workloads.** We study VM associativity using the 3C model to classify misses (i.e., page faults). We show that full VM associativity is a largely unnecessary feature, as the majority of misses are either classified as compulsory or capacity, hence insensitive to associativity. Consequently, only modest associativity is required. Our study covers all following scenarios: memory-resident and larger-than memory working sets, single-process, and multi-programmed workloads.

- **Set-Associative Virtual Memory (SAVAgE).** We propose SAVAgE, a translation mechanism for MPUs that eliminates most of the overhead of page walks. SAVAgE builds on modest associativity requirements of VM. SAVAgE restricts the associativity so that a virtual address identifies a memory chip and partition uniquely. An MMU in the memory partition, translates the virtual address and fetches the data. SAVAgE achieves low-overhead page walks as the page walk and data fetch operation overlap almost entirely.

- **Distributed Inverted Page Table (DIPTA).** We propose DIPTA, a translation mechanism that completely eliminates the overhead of page walks for in-memory workloads. This mechanism builds on the observation that the associativity of VM can be virtually eliminated for memory-resident workloads. DIPTA restricts the associativity so that a page can only reside in a few number of physically adjacent locations—i.e., same memory chip and DRAM row. Hence, all but a few bits remain invariant in the virtual and physical addresses, and we speculatively predict the rest using a highly-accurate way predictor, decoupling address translation with data fetch. To ensure that data fetch and address translation fully overlap, we place the page table entries next to the data. DIPTA completely eliminates the overhead of page walks for in-memory workloads.

1.9 Takeaway Message

One of the main takeaway messages of this thesis is that one can think of memory as a software-managed fully associative cache, where the tags are the page table entries and the data are the page frames. Though in a different context, the results of our work consistently corroborate seminal work on caches [35, 83]. Our results demonstrate that the conventional full associativity of virtual memory is largely unnecessary. The trends clearly indicate that compulsory and capacity misses dominate, and conflict misses, which associativity alleviates, drop rapidly as the associativity increases.

Another important takeaway message of this thesis is that just as set-associative or direct-mapped caches provide nearly all the benefits of full associativity with much faster access times [81], a set-associativity or direct-mapped virtual memory can provide nearly all the benefits of full associativity with much faster translations. Again, though in a different context, our work is in spirit similar to that of three decades old work.

1.10 Organization

The rest of this thesis is organized as follows. Chapter 2 introduces background on virtual memory and memory-side processing units. In Chapter 3, we perform an associativity study of virtual memory across a variety of scenarios. In Chapter 4, we propose SAVAgE, an efficient translation mechanism that builds on the modest associativity requirements of VM. In Chapter 5, we propose DIPTA, an efficient translation mechanism for in-memory workloads that builds on the observation that the associativity can be virtually eliminated for memory-resident workloads. We discuss related and future work in Chapter 6 and Chapter 7, respectively. We conclude the thesis in Chapter 8.

2 Virtual Memory 101

Virtual memory is one of the most crucial abstractions in the history of computing. Back in 1960s, virtual memory became a standard feature of nearly every mainframe. In the 1990s, it also became a standard feature in personal computers (PCs). Interestingly, while virtual memory generated a huge controversy after its introduction in the late 1950s, today, nearly all computer systems at any scale, ranging from handheld devices to datacenters rely on virtual memory and it became such an ordinary feature that few people think about it.

In the early days of computing, programmers had to solve a memory *overlay* problem as part of their programming work. Computer systems were organized hierarchically, with a small amount of expensive random access memory (RAM) and a larger secondary storage, a drum. Back then, programmers not only had to slice their programs into blocks, called overlays, that fit in memory, but also decide which overlays to bring to RAM and which overlays to replace. The process of moving overlays back and forth from memory to secondary storage was called *overlaying*. It was commonly assumed that programmers spent more than half of the time planning the overlaying control flow.

Virtual memory was invented as a way to automate a solution to the error-prone and time-consuming overlay problem. The first machine to support virtual memory was the Atlas Computer [110], designed by a group of researchers from the University of Manchester in 1959. Their virtual memory subsystem was called a "one-level storage system". The Atlas Operating System simulated a memory sufficiently large to hold the whole program. Addresses from this *virtual* memory were translated to their physical location in RAM. If an address was not in RAM, the operating system moved a fixed-size page from the drum to the small RAM. From the programmer perspective, the Atlas Machine's memory appeared as a large and slower RAM.

Commercial vendors of the large mainframes of the 1960s (e.g., Burroughs, IBM) rapidly noticed the huge potential of the virtual memory abstraction. Interestingly, as these commercial systems integrated large RAMs which allowed many individual programs to be entirely loaded, avoiding the overlay problem was not virtual memory's main advantage. Instead, the virtual memory abstraction showed to be useful to support other features such as multiprogramming, time-sharing

computing, and fault tolerance. Virtual memory allowed user programs to be isolated from each other, dynamic relocation of program addresses, and read-write access control to the program's memory. In summary, the virtual memory abstraction provided larger benefits than just automatic storage allocation, it also provided memory protection, and program reuse and sharing, explaining its rapid adoption in early computer systems [55].

2.1 Early Virtual Memory Designs

Even in the early days of computing, system designers realized that fast access to a large amount of storage is hard and expensive, and therefore computer memories have to be organized as a hierarchy. Traditionally, the memory hierarchy comprises two levels, *main memory* and *secondary memory*.

- Main memory: The main memory, often called random access memory (RAM), is directly connected to the central processing unit (CPU). A program's content (e.g., code, data) can only be referenced if it resides in main memory. RAM is a relatively fast and expensive storage medium. Furthermore, RAM is volatile; a power loss of the machine erases all the RAM's content.
- Secondary memory: The secondary memory, originally a drum and today a hard- disk drive (HDD) or solid-state drive (SSD), is connected to RAM by a subsystem that moves blocks of data. The secondary memory is a much slower and cheaper than RAM. Furthermore, the secondary memory is persistent; data remains in it until is it explicitly deleted.

The unit of storage in RAM and disk is the *block*. A block is a group of spatially contiguous bytes. Furthermore, a block is also the granularity of transfers between the two levels of the memory hierarchy. In computer systems where all the blocks have a fixed size, the blocks are called *pages*. In computer systems that allow variable-sized blocks, the blocks are called *segments*.

As shown in Figure 2.1, the processor can only execute a program from the main memory. Consequently, although the entire program is stored in the secondary memory, blocks of the program's content have to be copied to the main memory. Obviously, the challenge is to partition the program into chucks of data, overlays, which are composed of blocks of data, and orchestrate the scheduling of moving overlays back and forth between the two levels of hierarchy. It was assumed that programming time doubles with overlaying with respect to a programming model where no overlays are needed.

2.1.1 Virtual Addressing

Virtual memory was originally invented the solve the overlaying problem by automating the migration of pages between the levels of the memory hierarchy. The first machine to support

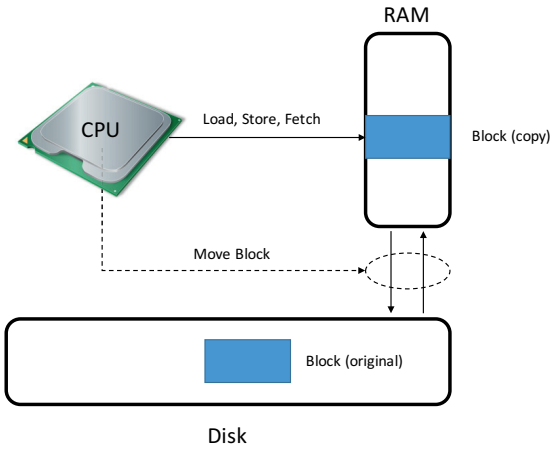


Figure 2.1 – Example of a two-level memory hierarchy with RAM and disk.

virtual memory was the Atlas Computer at the University of Manchester in 1959 [110]. The core of this memory subsystem is to create a level of abstraction in the addresses employed by a program. This abstraction created a distinction between an *address* in a program and its actual physical *location* in memory. The memory subsystem of the Atlas Computer provided programmers the illusion of a large and linear address space, even when the amount of RAM was scarce. Programmers could then write their applications as if all the code and data fit into a single and contiguous memory, avoiding managing overlays and increasing productivity.

The virtual memory abstraction pushed the designers of the Atlas Computer to architect a set of hardware and software mechanisms. First, the Atlas Computer had a hardware *address translation* mechanism to automatically translate each address in the program to its location in main memory. RAM was organized in pages and there was a register for each page storing the program address allocated in that page. In the Atlas Computer, RAM was scarce and a few tens of registers sufficed to cover main memory, all of which were associatively searched on every memory reference. Second, the Atlas Operating System featured *demand paging*, a mechanism that moved missing pages from secondary memory to main memory on demand. Upon a missing translation, the address translation mechanism triggered an interrupt to the CPU to run an OS handler to move the missing page from secondary memory to RAM. Third, a *replacement policy* to migrate non-useful pages back to secondary storage. The memory subsystem associated a "use" bit with each page in main memory, which was set when the processor issued a memory reference to a page. Periodically, an operating system program cleared out all the "use" bits. This information was employed to select a candidate page to move back to secondary memory (upon requiring making space for a new page in main memory).

In summary, decoupling the addresses of a program from their actual location in main memory is the core idea behind virtual memory. Implementing such abstraction led to a set of hardware and software mechanisms that has remained largely unchanged until today.

2.1.2 Multiprogramming

The major commercial vendors of the 1960s quickly realized the huge potential of virtual memory and included it in their most popular mainframes of the time, such as IBM 360/67, Burroughs B5000, CDC 7600, and General Electric GE645. All these mainframes time shared resources among different users and thus heavily relied on multiprogramming. Operating systems that support this feature allow several programs to simultaneously reside in main memory, and when the program executing in the CPU stalls due to waiting for a long-latency event (e.g., an input output operation), the CPU would switch to executing another program. Multiprogramming allowed to increase the CPU utilization and the machine's overall throughput.

Commercial vendors discovered that the virtual memory abstraction would simplify multiprogramming considerably, as it would provide an elegant way of sharing the memory. The operating system would organize a different mapping table of addresses and their physical location for each program. Hence, each program would only be allowed to access its own address space. Furthermore, virtual memory allows to define permissions for addresses (e.g., read only, read-write), providing fault tolerance by preventing the program from overwriting code sections.

2.1.3 Thrashing

Unfortunately, the early commercial systems that provided support for virtual memory were plagued with an unforeseen problem: *thrashing*. A computer system experiencing thrashing essentially makes no forward progress when the multiprogramming level reaches a certain point [54]. Preliminary analysis on such systems concluded that a thrashing behavior on a relatively well performing system starts with the activation of one additional program. Activating that additional program leads the system into a state of near-zero throughput. More thorough analysis revealed that the root cause of thrashing was that all active programs in the system were waiting for a page to be moved from disk to main memory. Soon after the page was in main memory, each program will again reference a disk-resident page, waiting once more for the page to be moved to main memory. In other words, the system enters into a state where it is paging forever, dramatically hindering throughput.

The thrashing problem pushed system designers to come up with new resource allocation techniques for main memory. In 1968, Peter Denning developed a new model, the *working set model* [54]. This model dictates that every process has a working set (WS), the collection of its most-recently used pages which the process requires in RAM to execute efficiently. Furthermore, the WS varies little between consecutive references to pages. Essentially, a computer system cannot enter into a thrashing state as long as every active program in the system has its working set in main memory. Interestingly, not only early experiments confirmed the WS model in programs but it stimulated research on the principle of locality, the underlying program behavior behind idea of a working set.

2.1.4 Cache Memory

Virtual memory's illusion of a computer system with a single memory with the capacity of a secondary memory and the speed of main memory stimulated system designers. In 1965, Maurice Wilkes proposed a two-level hierarchy for main memory, the slave and core memory [175]. The slave memory will later be known as a cache memory. In his paper, Wilkes proposes moving words of data, instead of pages, from the core memory to the slave memory due to smaller access latency differences between the two RAM memories; much smaller than between a disk and RAM. Furthermore, the principle of locality allowed for a small slave memory to filter many of the memory references, allowing the CPU to operate at the speed of the slave memory with the cost of the core memory.

The idea of integrating a small and fast memory between main memory and the CPU rapidly materialized in commercial products. In 1968, IBM released its 360/85 mainframe, which integrated a cache memory between main memory and the processor. Today, cache memories have become a standard feature in any computer system at any scale, from datacenters to handheld devices; all of these computer systems integrate several levels of cache memory.

2.1.5 Object-Oriented Virtual Memory

In the early 1960s, virtual memory was a common abstraction to avoid overlays and allow sharing the memory across multiple programs without interference. Nevertheless, system designers quickly realized that virtual memory could offer even more features to increase programmability. In the context of interactive time-sharing systems, designers believed that a powerful feature would be to share, reuse, and modularize parts of a program, such as arrays, procedures, structures. All these individual modules, program objects, would be accessible from anywhere in the system and linked together on demand. In other words, a program would be composed of a set of bricks, program objects, with explicit names, which simplifies reuse, sharing, and protection. These virtual memory systems were called object-oriented virtual memories.

The first example of object-oriented virtual memory appeared in the Burroughs B5000 in 1961. This virtual memory implementation presented the address space as a collection of multiple address spaces of variable size, *segments*. Each segment was the placeholder of a program object. Burroughs B5000's Algol compiler generated program segments containing procedures and data segments containing arrays of data. The program's virtual addresses were of the form (i,x), which indicated segment i and line address x. A mapping table was employed to locate the physical location of the segment's base address in main memory, while the size of the segment was explicitly stored in the aforesaid table to prevent out-of-bound accesses for the line address x.

In 1965, the Multics project at MIT went further than the Burroughs compiler. Multics allowed the programmer to define logical segments. In a Multics program, referencing for the first time a variable X in segment S would trigger a linkage fault interrupt. The fault handler would convert the symbolic segment S into a segment number s and line address X to an offset x. An address in

the Multics CPU was a pair (s,x), indicating segment s and line offset x. This link-on-demand procedure was a radical departure from the common practice of using a linking loader at run time to bind program objects together into a single address space. Interestingly, Multics implemented segmentation on top of paging, comprising a two-level mapping. The main reason of this design choice is that paging simplifies the virtual memory subsystem as memory is managed in chunks—pages—of equal size. The segment number s would select an entry in the segment mapping table (as in the Burroughs B5000 machine). Then, the aforesaid entry would point to a per-segment page mapping table, decomposing the line offset x into a page number and line number.

Though the Multics project produced a myriad on innovative ideas on sharing, reuse, and access protection of program objects, little of these ideas remained in future virtual memory subsystems. Programmers were satisfied with a single and large address space and a moderate number of program objects binded to it. The single and large address space provided by paging has become mainstream in today's virtual memory implementations.

2.1.6 From PCs until Today

Though virtual memory became a standard feature in mainframes from the early 1960s and firmly settled in the 1970s, the first personal computers (PCs) did not include features such as multiprogramming and virtual memory. For example, the operating system of the original Apple Macintosh personal computer employed physical addressing. Similarly, the first operating system for the IBM personal computer, DOS, did not support virtual memory or multiprogramming. It is not clear the main reason why early PC designs distanced themselves from the concepts that had worked so well in mainframes. Nevertheless, system designers of personal computers quickly encountered the same issues that resulted in the creation of the virtual memory abstraction, making it increasingly more appealing.

Ultimately, PC manufacturers such as Apple and IBM included virtual memory and multiprogramming in their operating systems. Adding virtual memory was possible because major chip vendors included virtual memory support in their processors. For instance, Intel added support for virtual memory starting with its 80286 chip in 1982, and Motorola extended the ISA of its 68000 chip to support virtual memory in the same year. Apple included multiprogramming in System 5, while virtual memory came in System 7. IBM offered the features of both multiprogramming and virtual memory in OS/2. Similarly, Microsoft delivered multiprogramming in Windows 3.1 and virtual memory in Windows 95 (though the first operating system from Microsoft to support virtual memory was Windows NT).

Today, nearly all computer systems rely on the virtual memory abstraction. From any sort of personal computer such as a desktop, laptop, or smartphone, to largest and greatest datacenter, all include virtual memory as a standard feature. It has become so ubiquitous that programmers do not think about it anymore when writing code. Abstracting the programmer from the available memory capacity, enforcing isolation among processes, and enabling portable codebases, are

desired features that seem to stand well the test of time, making virtual memory a key part of any modern computer system.

2.2 Virtual Memory Usage

The virtual memory abstraction arised as a way of avoiding programming overlays and automating solutions to the problem of migrating data blocks between the levels of the memory hierarchy. However, virtual memory has also been used to provide a large number of other features:

- *Automatic storage allocation*: Virtual memory creates the illusion of an always sufficiently large memory regardless of the available capacity of main memory, avoiding programming with overlays. It transparently automates the movement of blocks of data from secondary memory to main memory. Taken away from the programmer the burden of managing the allocation of data between the levels of the memory hierarchy greatly increased programmability. The feature of automating the storage allocation was the inception of the virtual memory abstraction.
- *Process isolation*: Virtual memory exposes a private memory address space to each process. Only the program objects mapped in the process' address space are accessible, guaranteeing that programs do not overwrite each other's content. Virtual memory proved to be an elegant solution to share the memory in multiprogramming environments.
- *Access control*: Within a process's address space, the program objects can be restricted to allow only certain reference types (e.g., read-only, read-write). An attempt to write to a read-only object will cause a fault in the program. This feature provides fault tolerance by for instance preventing a program to overwrite its procedures.
- *Relocation*: The core insight behind the virtual memory abstraction is the distinction between an address in the program and its physical location in memory. This distinction allows programmers to create independent program modules as tool chains (e.g., compilers, linkers) generate code employing virtual addresses. Programmers are relief of any prior knowledge of the system's memory organization or manually linking all the modules within an address space; enabling programs to be composed of reusable and sharable program modules.
- *Sharing*: Virtual memory allows for sharing a program's objects such as procedures and data arrays among multiple processes. Sharing is simply done by mapping the object's virtual addresses to the same physical memory location in all the processes. The sharing feature has been widely exploited to share libraries.
- *Metadata*: The virtual memory subsystem collects reference statistics per virtual memory allocation unit (e.g., dirty and recently referenced metadata). This information is employed by the operating system for evicting non-useful blocks back to secondary storage or by high-level languages to help at garbage collection.

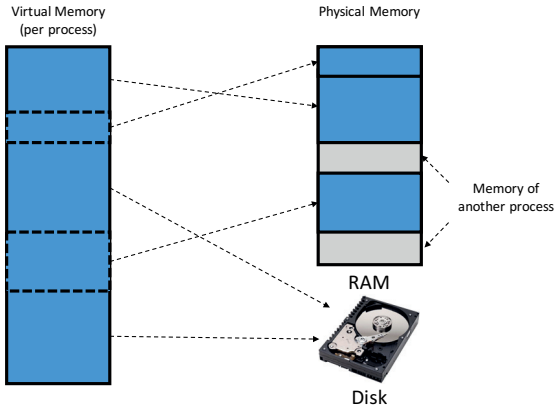


Figure 2.2 – Overview of the virtual memory abstraction.

In summary, the virtual memory subsystem provides far more features than its original purpose, making it an indisputable component of any contemporary computer system.

2.3 Structure of Virtual Memory

In this section, we will detail the internals of the virtual memory subsystem. Virtual memory's core insight is the distinction between an address in the program and its physical location in the memory. Figure 2.2 presents an diagram of the virtual memory abstraction. There are two different mechanisms to implement virtual memory, *paging* and *segmentation*. In paging, the virtual address space is split into fixed-size blocks of data, called pages, which are ISA dependent. For instance, the ubiquitous x86_64 employs pages of 4KBs, while SPARCv9 uses 8KB pages. In contrast, in segmented systems the virtual address space is comprised of a collection of address spaces of variable size, called segments. An example of a segmented system is the Burroughs B5000 machine. As having memory allocation units of a fixed size simplifies the storage allocation problem (e.g., by avoiding external fragmentation [55]), segmentation is usually implemented on top of paging. These systems implement a two-dimensional virtual address space where each of the segments is then split into pages. The protected mode of x86 (32 bits) is an example of a system that implements segmentation on top of paging.

2.3.1 Paging

Although there exist many combinations of operating system and processor architecture that implement paging, we will describe paging in the context of the ubiquitous x86_64 architecture and the Linux operating system.

Address Space

Every program must fit all its procedures and data inside the virtual address space. Assuming the addresses are k bits, the address space consists of 2^k bytes, comprised of the following range of address $\{0, 2^{k-1}\}$. Although the theoretic limit of the x86_64 is 64-bit virtual addresses, current implementations support up to 48 bits [47], while 57-bit implementations are on the way [70]. The most popular operating systems such as Linux and Microsoft Windows split the virtual address space into user- and kernel-space. In Linux, the virtual address space of 2^{48} (i.e., 256 TB) is partitioned into two halves of 128TB each. Hence, all the kernel's code and data structures must fit into its partition. Similarly, each process's code and data structures must also fit into the 128TB of virtual memory available for the user.

The size of the physical addresses dictates the maximum capacity for main memory in a computer system. Current x86_64 machine implementations support up to 48-bit physical addresses. Therefore, a machine could theoretically integrate 256TB of physical memory.

Memory Management Unit

The memory-management unit (MMU) is a hardware block in the CPU in charge of translating the virtual addresses employed by the software into their physical location in main memory. Though the MMU is a hardware block, it also interacts with several software components of the operating system. We describe all the hardware and software components involved in the address translation process below.

Page Table: The page table is a software mapping table that holds the translation information between the virtual and physical address spaces. The page table stores the translation information at the granularity of a page, which in x86_64 is of 4KB in size. A virtual address contains two parts, the *virtual page number* (VPN) and the *page offset*. The page offset of the virtual address indicates the byte within the page. As a virtual page is 4KB, the page offset is the 12 least significant bits of the virtual address. The rest of the bits of the virtual address compose the virtual page number. As x86_64 employs 48-bit virtual addresses, the virtual page number are the 36 most significant bits of the virtual address. The physical address space (i.e., the physical memory) is also divided into pages, called physical pages or page frames, of 4KB. Similar to a virtual address, a physical address is composed of two parts, the *page frame number* (PFN) and the page offset.

A page table stores the mapping between virtual page numbers and page frame numbers. Note that between a virtual address and its physical address the page offset bits remain unchanged. A page table contains entries, called page table entries (PTE), one entry per virtual page. Each page table entry holds at least the following information: (1) a *present* bit set to 1 when the page is in main memory, (2) a *read/write* bit for access control to the page, set to 1 when the page allows reads and writes (otherwise the page is read-only), (3) a *user/supervisor* bit set to 0 when a virtual page only allows accesses in the supervisor mode (otherwise the page can be accessed

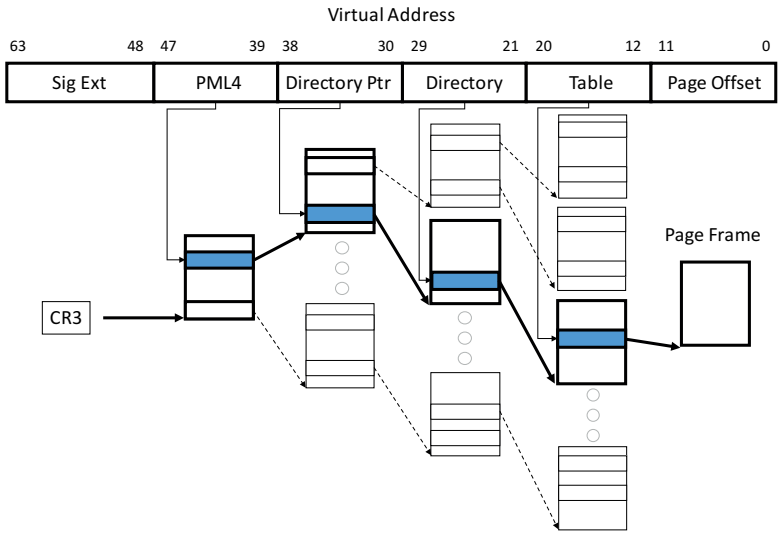


Figure 2.3 – Page walk operation in x86_64.

in any mode), (4) an *accessed* bit set to 1 when the program references the page, and (5) a *dirty* bit set to 1 when the program writes to the virtual page. The size of a page table entry (PTE) in x86_64 is 8 bytes.

There are many page table implementations in the literature, although the x86_64 employs a hierarchical page table of four levels; there exists a page table per process. Each level of the tree is indexed using a fixed set of bits of the virtual page number. The last level stores the page table entries, while the intermediate levels store pointers to the next levels. Hierarchical pages tables can be sparsely populated and therefore are space-efficient under any scenario, either in densely or sparsely populated virtual address spaces. The main drawback is that locating the mapping information takes multiple memory references (4 in x86_64). However, we will later explain techniques to reduce this overhead.

Page-Table Walker: Figure 2.3 presents the structure of the hierarchical page table employed in the x86_64 instruction set architecture. As explained above, all the levels of the hierarchy are traversed to locate the page table entry (PTE) that holds the translation information (e.g., the page frame number). The operation of traversing the page table is called a *page table walk*. The page table walk operation is done in either software or hardware. The software approach involves executing an operating system handler that walks the page table. For instance, the MIPS and DEC Alpha instruction set architectures adopted this approach. In contrast, the hardware approach integrates a finite-state machine in the MMU which performs the walking operation. Most of the contemporary ISAs such as x86_64 or ARMv8 employ a hardware page table walker.

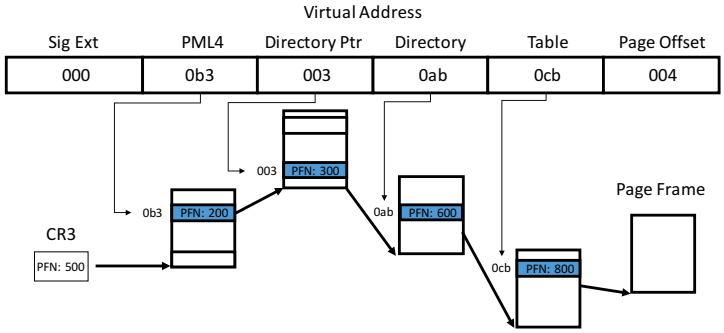
Let us now discuss the page walk operation in more detail. A virtual address is split into different sections: Page Offset, Page Table, Page Directory, Page Directory Pointer, and Page Map Level 4. The rest of the virtual address is sign extended up to the bit 63. Each of the address section (except the Page Offset) is employed to index a different level of the page table. Each section is

of 9 bits because each page is 4KB and each entry is 8B, and therefore there could be at most 512 valid entries per page. First, the hardware page table walker (PTW) must know the location of the root page table, the Page Map Level 4 (PML4). A control register called CR3 in the CPU holds the page frame number of the PML4. Note that the CR3 register must hold a physical address; otherwise the address would require a translation. Linux is in charge of updating the content of CR3 when switching process contexts. Then, the page walker references an entry in the PML4 by adding the PML4 bits of the virtual address to the page frame number stored in CR3. The PML4 entry is examined, it could either be invalid or contain the page frame number of a Page Directory Pointer (PDP) page. If the entry is valid, the PTW references a PDP page by adding the PDP bits of the virtual address. This procedure repeats recursively until an invalid entry is found or until the last level of the page table is reached and the Page Table Entry (PTE) is referenced. If the PTE contains a valid entry, the Page Offset bits are added to the page frame number to create the full physical address.

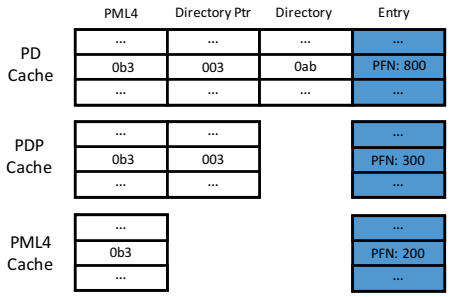
Overall, a page walk operation involves four memory references in the x86_64 hierarchal page table design. Furthermore, larger virtual address spaces, for example 57 and 64 bits would require 5 and 6 memory references respectively. However, there are techniques to avoid walking the page table for every translation operation or skipping levels of the page table walk. We will discuss such optimizations in later parts of this section.

Page Fault: It is possible that the page table walker references entries that are invalid. An invalid entry means that the following levels of the hierarchy are not instantiated or the reference violates the access control bits (e.g., a store operation to a read-only page). In such cases, the memory management unit raises a synchronous interrupt or exception called *page fault*, and stores the virtual address causing the page fault in the control register CR2. The exception triggers the execution of a Linux software handler in the CPU to resolve the page fault. There could be many possible actions. The page fault handler may allocate a page frame and populate the following levels of the hierarchy. If allocating the page frame does not involve a disk access, the page fault is called minor. For example, the first access to a page in the program or when the page frame is already in main memory. If the allocated page frame required reclaiming a page from the secondary store, the page fault is called major. The page fault handler may also send a signal to terminate the program in cases where there is an access right violation or when the faulting virtual address is not part of the process' virtual address space. If the page fault is successfully resolved, the CPU returns to executing the program and the MMU retries the translation.

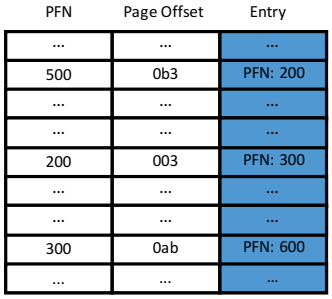
Translation Look-aside Buffer: The memory management unit integrates another hardware block besides the page table walker, the translation look-aside buffer or TLB. The TLB is a small high-speed memory to cache the most recently used translations from the page table walker. Without a TLB, every translation would require a page table walk. An entry in the TLB contains the virtual page number and its corresponding page table entry. On each memory reference, the MMU checks its TLB. Upon a hit, the page table entry is used to generate the full physical address. Upon a TLB miss, the page table walker traverses the page table. Furthermore, modern TLBs also tag the entries with the address space identifier (ASID) bits of the process to avoid



(a) Page walk example



(b) Intel's paging-structure caches



(c) AMD's page walk cache

Figure 2.4 – Example of a page walk and the populated page table caches for Intel and AMD.

flushing the TLBs on context switches. Contemporary CPUs usually integrate MMUs with a TLB hierarchy of two levels. The first-level TLB usually contains several tens of entries while the second level holds from several hundreds to a few thousand entries. For instance, a modern CPU of the x86_64 ISA, Intel Broadwell, integrates a 64-entry first-level TLB and a 1536-entry second-level TLB. Though translations that hit in the TLB hierarchy are fast, walking the page table adds a significant overhead.

Page-Structure Caches: To reduce the overhead of page walks, commercial vendors integrate dedicated caches for the rest of the levels of the page table (besides the TLBs which hold entries of the first level only). The observation is that the temporal locality in page walks is higher for the upper levels of the page table. For example, a page walk to two consecutive pages in the virtual address space will likely share the same three entries of the upper levels (i.e., PML4, PDP, and PD). If we could cache these entries, the second page walk would skip the first three levels, requiring a single memory access for the last level to locate the page table entry (which would be cached in the TLB). Hence, the two major chip vendors of the x86_64 instruction set architecture, Intel and AMD, integrate dedicated caches for the upper levels of the page table. We explain both designs below.

Intel's terminology for these caches is paging-structure caches (PSCs). Just like the TLBs, these caches are accessed with virtual addresses. There is a dedicated PSC for each upper level of the page table (i.e., PML4, PDP, and PD). A PSC for the last level, PML4, contains entries of the

PML4 page, whereas a PSC for the second level, PD, contains entries of PD pages. Figure 2.4a shows an example of a page walk operation. The content of the PSCs after the walk is shown in Figure 2.4b. Each entry in the cache contains a tag, colored in white, and its associated data, colored in blue. The tags are required to uniquely identify an entry in the page table, which varies depending on the page table level. For instance, only the PML4 bits are needed for the PML4 cache as there is a single PML4 page. In contrast, a PD page requires all three sections of the virtual address (i.e., PML4, PDP, and PD bits) to uniquely identify a PD entry. The data of a PSC entry is the page entry (e.g., a PD entry) identified by the tag.

Upon a TLB miss, prior to the page walk operation, the MMU checks all three PSCs. The MMU then selects the entry from the cache hit with the longest tag. For example, if there is a hit in both the PML4 and PD caches, the MMU prioritizes the page entry from the PD cache. A hit in a paging-structure cache provides the physical address of the next level, starting the page walk operation from aforesaid level and skipping all previous levels. A page walk that hits in the PD cache requires a single memory reference, instead of the initial four. Similarly, page walks that hit in the PDP and PML4 caches require two and three memory references respectively.

AMD CPUs implement a page-walk cache (PWC). A PWC tags its entries with physical addresses. In this design, all the three upper levels of the page table share the same PWC. The content of a page-walk cache after the page walk operation (of Figure 2.4a) is show in Figure 2.4c. The tag of a PWC entry, colored in white, is the full physical address of its associated page entry. For example, the first valid entry's tag contains the physical address of the PML4 entry. The physical address is composed by adding the PML4 bits $0b3$ to the page frame number of the root page (stored in CR3). The data content of the entry, colored in blue, points to the page frame number of the page in the next level (i.e., a PDP page).

In AMD's implementation, the page walk operation starts from the root page as in the original page walk. The reason is that the PWC is tagged with physical addresses. Upon a TLB miss, the MMU generates the physical address of the target PML4 entry. Instead of accessing the memory, the MMU first check the page-walk cache. Upon a hit, the page frame number of the next level is obtained from the cache. The physical address of the next level is subsequently composed by adding the aforesaid page frame number the bits of the PDP bits. The MMU checks again the PWC before accessing the memory. This operation continues recursively until the page frame number of the page table entry is generated or a miss in the PWC arises, falling back to the conventional page walk and issuing a reference memory. In the ideal, all the entries of the upper levels of the page table reside in the PWC, requiring a single memory reference to resolve the page walk.

In short, both major x86_64 chip vendors integrate dedicated caches for the upper levels of the hierarchical page table. Intel integrates page-structure caches (PSCs), which are partitioned per level and are accessed with virtual addresses. Therefore, one could look at the PSCs as TLBs for the upper levels of the page table. In contrast, AMD integrates a page walk cache (PWC), which is shared among all the upper levels and is accessed with physical addresses. Therefore,

one could look at a PWC as a dedicated data cache; much like an L1 data cache for translations.

Huge Pages: A memory management unit probes the translation look-aside buffer (TLB) on every memory reference. Hence, TLBs are usually kept small to ensure fast access and affordable power dissipation levels. To increase the translation coverage of the TLB or *TLB reach* (i.e., the number of entries times the page size), commercial chip vendors have introduced huge pages. For instance, x86_64 and ARMv8 ISAs support 2MB and 1GB pages in addition to the 4KB base page size. A single TLB entry of a 2MB page can potentially provide the same reach as a TLB of 512 entries of 4KB pages. Hence, huge pages increase the TLB reach without increasing the number of the TLB entries. Furthermore, using huge pages decreases the number of memory references of page walks. The reason is that huge pages are generated by allocating contiguous base page sizes. For example, a 2MB page requires a Page Table page (PT), fully populated of PTEs pointing to a contiguous range of physical addresses. Because the page frame pointed by the PTEs are contiguous, the physical address of the page can be obtained from a single PTE. Therefore, the page of the previous level (i.e., page directory or PD) can directly point to the page frame stored in the first PTE (which points to a contiguous page of 2MB), skipping the last level and achieving a page walk of three memory references. Similarly, a 1GB page requires a PD page fully populated of entries pointing to a contiguous range of 2MB pages. In this case, the previous level (i.e., page directory pointer or PDP) can directly point to the page frame stored in the first PD entry (which points to a contiguous page of 1GB), skipping the last two levels and achieving a page walk of two memory references.

Furthermore, the existence of multiple page sizes complicates the design of the TLB. TLBs are tagged with the virtual page number, which depends on the size of the page. Unfortunately, the MMU does not know a priori the size of the virtual page, making the standard set-associative indexing of memory structures hard. Computer systems have traditionally solved this problem by either integrating an TLB per page size or by implementing a fully associative TLB. For instance, Intel CPUs (x86_64) integrate a TLB per page size, whereas Oracle SPARC CPUs (SPARCv9) implement a fully associative TLB. Alternatively, one can virtualize different page sizes in the same set-associative TLB by probing the cache with multiple virtual page numbers (one per page size). However, this approach significantly increases the TLB hit latency as the virtual page numbers have to be probed sequentially. An example of such design is the second-level TLB in the Intel Haswell and Intel Broadwell processors, which stores pages of 4KB and 2MB simultaneously. Nevertheless, both processors still integrate a TLB per page size for the first level of their TLB hierarchy.

Overview: Figure 2.5 presents an overview of all the hardware components of the MMU and its interaction with the OS data structures (i.e., the page table). In summary, a address translation process involves the following workflow. when the process running on a CPU core references a memory address, the core sends the virtual address to the memory management unit. The MMU then probes the first level TLB. Upon a hit, the page frame number is appended to the page offset bits to form the full physical address which is then returned to the core. A TLB miss triggers a probe operation in the second-level TLB, the S-TLB. An S-TLB hit results in the same action

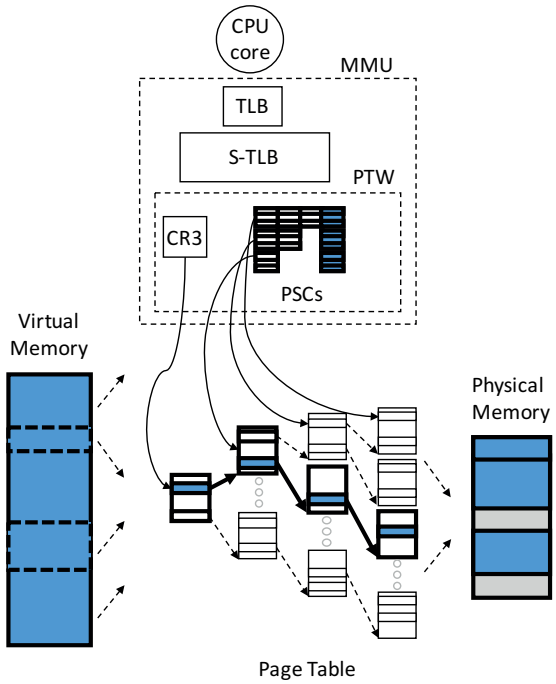


Figure 2.5 – Overview of the address translation process.

as a TLB hit. Upon an S-TLB miss, a page walk operation begins. First, the MMU probes the page-structure caches (PSCs). Upon a hit in the any of the levels, the page walker starts walking the page table from the appropriate level (the level of the PSC that returns a hit), and skipping the previous levels. In the case that probing the PSCs results in no hits, the page walk operation starts from the root page of the page table. The bits of the virtual address that index the root page are added to the the content of the CR3 register. Then, the page table walker (PTW) references an entry of the root page or PML4. The PTW recursively performs the same operation in all the levels of the page table until an invalid entry is found, or the last level is reached and the PTE is obtained. If the PTE entry is valid, the page frame number stored in it is appended to the page offset bits to form the physical address. Then, the MMU returns the physical address to the core. As shown in the workflow, translating an address involves a significant overhead when the translation information is not found in the TLBs.

2.3.2 Segmentation

As nearly all computer systems today employ paging, we will briefly explain the other implementation of virtual memory, segmentation.

In a system that implements segmentation, the virtual address space comprises a collection of variable-sized address spaces called segments. Different segments usually accommodate semantically disparate parts of a program such as procedures, heap, stack, etc. For example, the Algol compiler of the Burroughs B5000 generated segments containing procedures and data

arrays. In segmented systems, a virtual address is a pair of the form (i,x), where i indicates the segment identifier and x is the line address within the segment. Analogous to the page table in paged virtual memory, the translation information is stored in a segment table. The segment table is indexed with the segment identifier. Each entry is the segment table contains the physical address of the start of the segment, called *base*, the maximum value of the line address within the segment, called *limit*, and control access information. Upon a memory access, the memory management unit references the segment entry. If the line address x is within the limit value, the full physical address is generated by adding the line address bits to the base address. Instruction set architectures that support segmentation usually provide hardware registers to store segment entries and avoid referencing the segment table.

Segmentation simplifies the task of sharing modularized program components (e.g., procedures) across different programs, by simply mapping a segment into a program's address space. In contrast, paged-based virtual memory exposes a single address space broken down into pages, which lack any semantic information of the program, making the sharing of components harder. Unfortunately, managing the physical memory at variable-size granularity complicates the storage allocation problem. Indeed, segmentation has shown to exhibit the external fragmentation problem [55]; where though there is enough physical memory available, there is no single contiguous chunk of physical memory of the segment's size. In turn, though early systems implemented pure segmentation, such as Burrough's B5000, current ISAs (e.g., PowerPC, x86) only support segmentation on top of paging. We will now explain the x86 (32 bit) ISA virtual memory implementation.

x86 Segmentation

Intel's segmentation on top of paging implementation first appeared in the Intel 80386 processor. The x86 (32 bit) ISA supports two different modes of operation: real mode and protected mode.

The real mode does not support virtual memory and is used at early boot time or to support legacy code. This mode allows for a linear physical address space of 20 bits (up to 1MB of addressable memory). The 20-bit virtual addresses are stored in segment registers. The 16 most significant bits of the 20-bit address space is the segment identifier. Each segment in real mode is of 64KB in size. Hence, to compute the linear physical address, a 16-bit offset is always added to the segment identifier.

The protected mode implements segmentation on top of paging. In this mode, a virtual address consists of a 16-bit segment identifier and a 32-bit segment offset (up to 4GB of addressable memory). Segment identifiers are stored in the segment registers. Starting from Intel 80386, the x86 ISA integrates six segment registers: CS, DS, ES, SS, FG, and GS. Although segment registers can be explicitly referenced in the program, the ISA selects registers by default (unless other registers are explicitly referenced). For example, for an instruction fetch the code segment (CS) is selected by default, while any stack operation automatically selects the stack register (SS).

In x86, there are two segment tables, a local descriptor table (LDT) and a global descriptor table (GDT). The LDT is supposed to contain segments which are private to a program, while the GDT usually contains global segments. The 13 most significant bits of the bit segment identifier are used to index the segment tables. The 2nd bit of the segment identifier is used to decide which of the two segment descriptor tables to index. The entries in the descriptor tables contain the segment's linear base address as well as the limit and access control information. The segment registers contain both the segment identifier as well as its descriptor table entry for fast access. The segment base address is added to the segment offset to generate the virtual address. Then, the virtual address is translated using paging as explained in the previous section.

In Intel's x86 64-bit ISA, x86_64, a virtual address consists of a 16-bit identifier and a 64-bit segment offset. However, the ISA exposes a flat virtual address space by hardwiring to zero the segment registers CS, SS, DS, and ES. Essentially, the virtual address is the 64-bit segment offset, which is then translated using paging. Hence, in the x86_64 instruction set architecture paging is the only translation mechanism.

2.3.3 Virtual Memory in other ISAs

Though we have explained the virtual memory implementations of today's most ubiquitous ISA, x86_64. However, other instruction set architectures such as ARMv8 and SPARCv9 exhibit significant implementation differences of virtual memory. In this subsection, we will briefly discuss the aforesaid differences.

MIPS

The 64-bit MIPS instruction set architecture implements virtual memory with paging. The MIPS R10000 processor implemented this ISA. Though program binaries use 64-bit virtual addresses, only the 44 least significant bits are used during the translation. The top five most significant bits of a 64-bit virtual address divide the address space into regions. The top two bits idenfity user, supervisor, and kernel regions. The other three bits further divide the regions into cached, uncached, mapped, and unmapped, each with different semantics and behavior. MIPS employs a linear page table of two or three levels (depending on the operating system). The MMU does not integrate a hardware page walker. In MIPS, a TLB miss raises an exception in the processor and an OS handler traverses the table and resolves the translation. The MIPS instruction set architecture supports multiple page sizes, from 4KB up to 16MB by multiples of four.

Alpha

DEC's 64-bit Alpha instruction set architecture supports page-based virtual memory. The DEC Alpha 21164 processor implemented this ISA. Though programs generate 64-bit virtual addresses, Alpha only supports a 43-bit virtual address space and employs a 3-level hierarchical page table.

Furthermore, the ISA can address up to 1TB of memory as physical addresses are 40-bit long. Like MIPS, the MMU does not integrate a hardware page table walker. Instead, a TLB miss raises an exception in the processor and an OS handler walks the page table and fills the page table entry in the TLB. Alpha supports huge pages of 64KB, 512KB, and 4MB sizes.

Power

IBM's 64-bit Power instruction set architecture implements virtual memory with segmentation on top of paging. An example of such ISA is found in the recent IBM Power8 processor. The ISA exposes a 64-bit segmented address space where each segment is of 256MB or 1TB in size. For 256MB segments, the 36 most significant bits of the segment address (the segment address contains both the segment identifier and offset), are used to index a segment descriptor table. Similarly, for 1TB segments, the 24 most significant bits of the segment address are used to index the segment table. To ensure fast access to segment descriptor entries, the processor integrates a fully associative segment look-aside buffer (SLB). The segment descriptor entry contains the base virtual address of the segment and the limit, which the memory-management unit uses to generate a 78-bit linear virtual address.

Virtual addresses are translated to 50-bit physical addresses using paging. The MMU walks the page table in a two-step process. First, the page table walker traverses a software cache of the full page table. The software cache is implemented as an eight-way set-associative inverted page table. Upon a miss in the software cache, an exception is then raised in the processor and an OS handler walks the full page table and resolves the translation. The Power ISA supports multiple page sizes: 4KB, 64KB, 16MB, and 16GB. Furthermore, the MMU integrates a translation look-aside buffer to cache frequently referenced page table entries.

SPARCv9

Oracle's 64-bit SPARC instruction set architecture called SPARCv9 implements page-based virtual memory. Oracle SPARC M7 processor implements this ISA. Similarly to x86_64, SPARC v9 supports a 54-bit subset of the 64-bit for virtual and physical addresses. Page walks are performed in two steps as in the Power ISA. First, the page table walker traverses a software cache, called Translation Storage Buffer (TSB), which is a cache of the full page table. The TSB is implemented as a direct-mapped inverted page table. Then, and upon a miss in the TSB, an exception is raised in the processor and an OS handler resolves the translation by traversing the full page table. The SPARCv9 instruction set architecture supports multiple page sizes: 8KB, 64KB, 4MB, 256MB, 2GB, and 16GB.

ARMv8

ARM's 64-bit ARM instruction set architecture called ARMv8 supports virtual memory with paging. ARM Cortex A57 is a processes that implements such ISA. ARMv8 supports two different subsets of the 64-bit virtual address spaces and two different base page sizes, 4KB and 64KB. When using the conventional 4KB base pages, ARMv8 supports 39-bit and 48-bit virtual address spaces. The 39-bit virtual address space employs a 3-level hierarchical page table, whereas the 48-bit virtual address space uses the same page table as that of x86_64 (a 4-level hierarchical page table). With 64KB base pages, ARMv8 supports 42-bit and 48-bit virtual address spaces. The 42-bit and 48-bit virtual address spaces use a 2-level and 4-level hierarchical page table respectively. Similar to x86_64, when using 4KB pages, ARMv8 supports huge pages of 2MB and 1GB. When using 64KB as the base page size, ARMv8 support a single huge page size of 512MB. The ARMv8 ISA also integrates hardware cache for the upper levels of the hierarchical page table. For instance, the ARM Cortex A57 processor includes a page walk cache per CPU core (similar to AMD's page-walk cache).

2.4 Unified Virtual Memory

Computer systems with heterogeneous computation units must offer an efficient and familiar programming model to ensure widespread adoption. A key aspect of the programming model is the way the memory is exposed between CPUs and the rest of the execution elements in the system (e.g., GPUs, ASICs). An easy to use and efficient view of the memory is fundamental to exploit the full potential and maximize the computing efficiency of these heterogeneous systems.

Traditionally, CPUs and specialized computation units (e.g., GPUs, MPUs) have resided in separate address spaces [141, 144]. In this programming model, data needs to be explicitly copied with custom APIs, not only generating redundant data but also requiring manual data consistency maintenance. Although copying array-based data structures is trivial, pointer-based data structures (e.g., linked lists, trees) require error-prone and explicit pointer transformations. Additionally, for conventional physically addressed computation units, data needs to be pinned in memory, which can lead to poor performance [144]. Furthermore, expensive OS intervention is inevitable and may even diminish the gains of specialization [43]. In short, a separate address space and physical addressing do not provide an efficient and simple to use programming model.

Initiatives in industry such as the Heterogeneous System Architecture (HSA) Foundation, a consortium founded by AMD, ARM, Qualcomm and Samsung among others, are proposing a shift towards *unified virtual memory* between any computation unit in the system [85]. In this programming model, a pointer is equally valid on the CPU and on the other computation units, simplifying data sharing and avoiding multiple data copies. Furthermore, virtual memory allows for efficient fine-grained memory accesses (e.g., pointer chasing) and transparent storage allocation and protection mechanism. The end result is that virtual memory provides a familiar and powerful abstraction to programmers and operating systems alike. All these benefits have led

commercial GPUs to adopt an HSA-compliant unified virtual memory, beginning with AMD's Carrizo chip [174]. Given the benefits of HSA-style unified virtual memory and its early adoption into commercial products, we expect future computation units to also follow this integration path. The implication of unified virtual memory is that any computation element in the system has to efficiently translate addresses spanning a distributed physical memory of hundreds of gigabytes to a few terabytes.

As explained in earlier parts of this chapter, CPUs support virtual memory by integrating per-core memory-management units. Though translations served by the TLBs achieve good performance, a miss in the TLB incurs in an expensive page table walk. Unfortunately, as large memories are beyond the reach of contemporary TLBs [22, 107], a significant fraction of translations require a page walk. As page table entries are arbitrarily scattered across memory chips, page table walks require expensive cross-chip traffic for each of the levels of the page table [70], making page walks a severe performance bottleneck.

Specialized computation units currently support virtual memory differently. For instance, the most recent integrated GPUs [174] are fully compliant with the HSA specification and therefore support virtual memory. Integrated GPUs reside in the CPU chip and translate virtual addresses through a centralized I/O memory management unit (IOMMU), recently introduced in commercial CPUs to allow devices to translate addresses. These IOMMUs integrate IOTLBs and a hardware page table walker. Although hits in the IOTLBs achieve a relatively low translation overhead, a page walk is an order of magnitude higher than in the CPU cores [166]. The reason is that the IOMMU's page walker cannot access the CPU's cache hierarchy when walking the page table. As every memory reference during a page walk involves an access to main memory, page walks are an even more severe performance bottleneck than in CPUs.

Other specialized architectures such as FPGAs have a basic form of virtual memory support. The recent prototype of Intel-Altera's Heterogeneous Architecture Research Platform (HARP) employs a static 1024-entry TLB with 2MB pages (up to a TLB reach of 2GBs) to support virtual memory for user-defined logic blocks [153]. Such a static TLB approach requires pinning pages in memory, while TLB refills mandate running an expensive kernel driver in the CPU. This approach performs poorly when the TLB reach does not cover the entire memory. Alternatively, and similarly to the integrated GPUs mentioned above, FPGAs (or ASICs) could employ the IOMMU of the CPU to walk the page table. While this approach removes the overhead of running the kernel driver and avoids the need for a custom API, page walks would still be expensive (due to the reasons explained above). Furthermore, specialized computation units residing outside of the CPU chip (i.e., discrete computing elements) would require expensive cross-chip traffic, further increasing the overhead of page walks.

A more sound approach to support virtual memory is to integrate a complete memory management unit (MMU) per specialized computation unit. Each MMU with its own hierarchy of TLBs, a page walker, and page table caches. Though an improvement over the previous approaches, its performance is still limited by the frequency of page walks. Unfortunately, as large memories are

beyond the reach of contemporary TLBs [22, 107], a significant fraction of translations require a page walk. The distributed nature of modern large memory systems, makes page table walks require costly cross-chip traffic for each of the levels of the page table [70].

In summary, unified virtual memory provides a global view of a large memory across all the computing elements in the system, offering an efficient and familiar programming model. Unfortunately, modern memory capacities along with the distributed nature of the memory are stressing the traditional virtual memory mechanisms that have served us so well in the past. Therefore, we identify page walks as the critical performance bottleneck in computer systems that support unified virtual memory.

2.5 This Thesis

In this thesis, we identify page walks as a critical performance bottleneck in virtually addressed execution units such as CPUs, GPUs, or MPUs, and propose mechanisms to eliminate the page walk overhead. We focus on page-based virtual memory as nearly all the systems implement it, and the x86_64 instruction set architecture as it is the de facto ISA in datacenters. Furthermore, the evaluation of our translation mechanisms focus on memory-side processing units (MPUs) as MPU-capable systems stress the conventional translation mechanisms the most; though our translation mechanisms are widely applicable to any other context such as CPUs.

3 Associativity in Virtual Memory

Page-based virtual memory splits the address space into fixed-size virtual pages. Each virtual page is mapped on demand into a page frame in physical memory. The process of figuring out the page frame that a given virtual page maps to is called address translation. A priori, there is no restriction on the virtual to physical mapping; a virtual page can potentially map to any page frame. In the 1960s, when virtual memory was invented, a machine could integrate a few MBs of memory. For instance, in a machine integrating 2MB of physical memory (e.g., the General Electric GE645) and assuming 4KB base pages, a virtual page could be found in any of the 512 possible locations. Today, as DRAM has continuously improved in density and cost at a rate similar to Moore's law [69], a computer system today can perfectly integrate few TB of physical memory (e.g., HP DragonHawk integrates 6TB). For example, in a 2TB computer system, a virtual page can be found in any of 512 million possible locations! Though the number of possible locations has increased by 6 orders of magnitude, we still use the same extreme mapping flexibility as when virtual memory was invented back in the 1960s.

Though extremely flexible, this fully associative mapping places translation on the critical path of every memory access. As any mapping is possible, a memory access cannot begin until the translation finishes. Interestingly, several studies have recently exploited operating system facilities such as huge pages, buddy allocators, and memory compactors that by construction allocate contiguous page frames to contiguous virtual pages [139, 140]. The existence of natural contiguity generated by the OS is a first indication that the page placement flexibility provided by full associativity is not fully employed. Hence, we believe that the traditional full associativity of virtual memory should be revisited.

The other end of the spectrum is to completely eliminate the associativity of virtual memory and enforce a direct mapping between virtual pages and page frames. Since now a virtual page can only map to a single page frame, the address translation and data fetch are independent and can proceed in parallel. Unfortunately, conventional wisdom dictates that using a direct mapping creates an excess of page faults, due to multiple virtual pages mapping to the same page frame. However, this belief is merely intuition, as until this date there exists no study on VM

Table 3.1 – Workload description.

Workload	Description
Cassandra	NoSQL data store running Yahoo's YCSB.
Memcached	Cache store running Twitter-like workload [121].
TPC-H	TPC-H on MonetDB column store (Q1-Q21).
TPC-DS	TPC-DS on MonetDB column store (Queries of [114]).
MySQL	SQL DBMS running Facebook's LinkBench [62].
RocksDB	Store engine running Facebook benchmarks [63].

associativity, unlike caches, which share some organization aspects and for which such a study has existed for three decades [83]. Hence, it is clear that a study of VM's associativity has the potential to uncover significant insights and optimization opportunities for address translation.

In this chapter, we perform an associativity study of VM (i.e., the number of locations that a virtual page can map to in the physical memory) across a variety of scenarios by classifying the page faults using the classic 3C model originally developed for caches [83]. The study demonstrates that the full associativity of VM is unnecessary, and only a significantly modest associativity is required. We conclude that capacity and compulsory misses—which are unaffected by associativity—dominate, while modest associativity is sufficient to eradicate conflict misses. More specifically, for working sets that are memory resident, compulsory misses dominate when the VM associativity equals the number of processes executing in the system. For working sets that exceed the memory size, capacity misses dominate and grow as a function of the working set size. In contrast, conflict misses become scarce once the associativity of virtual memory matches the number of processes in the system, and virtually disappear after the first few additional ways.

Essentially, we make the observation that one can think of memory as just a fully associative software-managed virtually addressed cache, where the tags are the page table entries and the data are the page frames. Fundamentally, our associativity trends for VM match prior work on set-associative caches [35, 83]. To provide much faster translation times architects could exploit the modest associativity requirements of virtual memory. Though in a completely different context, our work is in spirit similar to that of three decades old work on caches [81]; showing that set-associative or direct-mapped caches can provide nearly all the flexibility benefits of full associativity with much faster cache access times.

The rest of this chapter is organized as follows. Section 3.1 introduces the methodology of our VM associativity study. Section 3.2 presents the associativity results. We then discuss the implications of the results in Section 3.3 and finally conclude with a summary of the chapter in Section 3.4.

3.1 Methodology

3.1.1 3C Model

In this chapter, we employ the 3C model—initially developed for caches [83]—to study VM associativity. In this context, associativity means the number of possible locations—page frames—a given virtual page can map to. This model classifies misses (i.e., page faults) into three categories: conflict misses, capacity misses, and compulsory misses. Conflict misses arise due to too many active pages mapping to a fraction of the memory sets. Capacity misses arise due to a fixed memory size. Compulsory misses inevitably occur on the very first access to a page.

We calculate the contribution of each of these components as follows. First, the conflict miss rate is the memory's miss rate less the miss rate of a fully associative memory of the same size. Second, the capacity miss rate is the fully associative memory's miss rate less the miss rate of an infinite memory (one that never replaces a page). Finally, the compulsory miss rate is the infinite memory's miss rate, which is never zero because the first reference to a page still generates a miss (i.e., a page fault).

We assume virtual pages and page frames of 4KBs (like in x86_64), LRU replacement policy, and the standard method of computing the index by selecting the least significant bits of the virtual page number. We employ this methodology for the results of Section 3.2.

3.1.2 Workloads

To simulate real-world scenarios in our study, we select a set of representative server workloads, summarized in Table 3.1. We include two cloud workloads from CloudSuite [66], Cassandra and Memcached, an online transaction processing (OLTP) workload [62], MySQL, two online analytical processing (OLAP) workloads [29], TPC-H and TPC-DS, and a widely-used storage system workload [57], RocksDB.

3.1.3 Traces

We collect memory traces using the Pin binary instrumentation tool [124]. For workloads with fine-grained transactions (i.e., Memcached, RocksDB, MySQL, and Cassandra), the traces contain the same number of instructions as the application executes in 60 seconds without Pin. For analytics workloads (i.e., TPC-H and TPC-DS), we instrument the entire execution. We extract the ASID bits and the virtual address of each memory reference, concatenate both [23, 178],[1] and use it to probe a set-associative memory structure, to observe and classify the misses.

For the associativity experiments in Section 3.2, we tune all the workloads to have a resident set size (RSS) of 8GB. In other words, the allocated physical memory for all the processes of

[1] The final virtual address consists of: ASID $\oplus$ VPN $\oplus$ Page Offset.

a given workload is 8GB. For single-process runs, a single process has an RSS of 8GBs. For two-process runs, each of the processes has an RSS of 4GB. The same scaling applies for the runs with four and eight processes. This way we guarantee that only the increase in the number of processes impacts the miss rate. To vary the memory size to dataset size ratio, we vary the size of the set-associative memory structure. The reason the datasets are not larger than 8GBs is to make sure that the workloads have enough time to touch a significant fraction of the memory throughout the trace.

We collect the traces on a dual-socket server CPU (Intel Xeon E5-2680 v3) with 256GB of memory, using the Linux 3.10 kernel and Google's TCMalloc [75]. Address space randomization (ASLR) is enabled in all experiments.

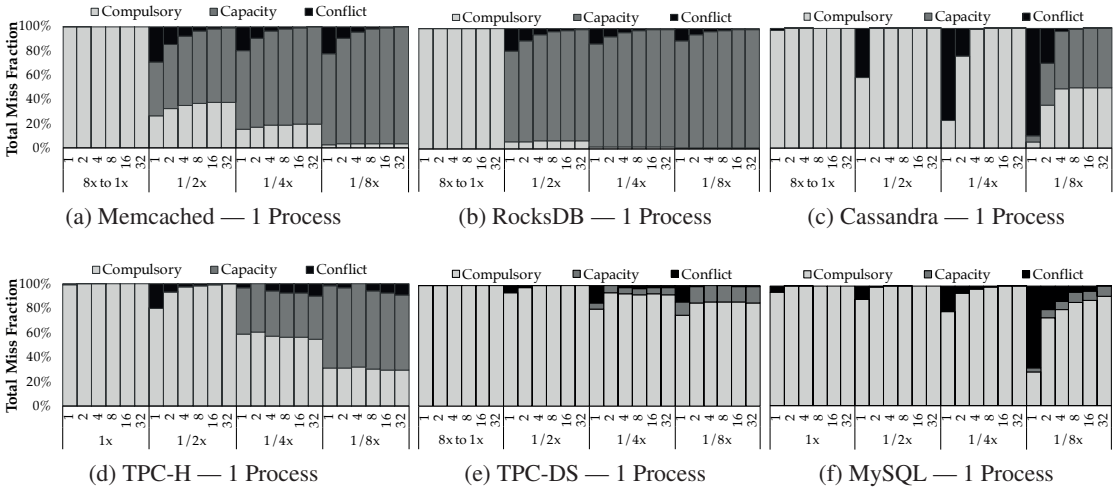
3.2 Evaluating Associativity

We now study the associativity in virtual memory across a wide range of scenarios. First, we consider the case where there is a single process taking up all the available physical memory [22, 79]. This scenario is typical of deployments for first-party workloads with tight latency constraints. We refer to this scenario as *single-process in-memory*. Then, we examine an scenario where there is a single process whose working set is larger than the physical memory [56, 57, 113]. This scenario is typical of the backends of first-party workloads. For example, Facebook employs an RDBMS with a storage engine optimized for SSDs [57]. We refer to this scenario as *single-process out-of-memory*. We then consider the case where there are multiple processes sharing the physical memory. This scenario is typical of cloud server systems executing third-party workloads (e.g., Amazon AWS). The aggregated working sets of the third-party workloads can either be memory-resident or larger than memory. We refer to these two scenarios as *multi-programming in-memory* and *multi-programming out-of-memory* respectively.

3.2.1 Single-Process In-Memory

Fig. 3.1 shows the associativity study's results for three single-process workloads: Memcached, RocksDB, Cassandra, TPC-H, TPC-DS, and MySQL. The y-axis breaks down the total misses into the three distinct miss classes. Each category on the x-axis corresponds to the ratio between the size of the physical memory and the size of the application's working set. For example, 8× indicates that the memory is eight times larger than the application's working set. Similarly, 1/2× means that the application's working set is twice the size of the memory. In this study, we collapse the results for 8× to 1× because their high similarity results in visually identical representations; each case represents a fully in-memory scenario. Furthermore, within each working set category, the x-axis sweeps through different VM associativities, from direct-mapped (i.e., one way) to 32-way associative.

Even with a simple direct-mapped translation, compulsory misses represent 99.9% of all the



(a) Memcached — 1 Process (b) RocksDB — 1 Process (c) Cassandra — 1 Process

(d) TPC-H — 1 Process (e) TPC-DS — 1 Process (f) MySQL — 1 Process

Figure 3.1 – Overall miss ratio broken down into compulsory, capacity, and conflict misses.

misses (except for MySQL which is 95%). There are naturally no capacity misses as the working set fully fits in memory. Conflict misses are extremely scarce. For example, Memcached using the direct-mapped configuration achieves a conflict miss rate in the order of one miss per 10^8 memory accesses. Additionally, using 2 ways removes all Memcached's conflicts for the $8\times$, $4\times$, and $2\times$ cases, while 4 ways are required for the $1\times$ case (where the memory size is equal to the size of the working set).

For in-memory scenarios, page conflicts arise because the virtual address space of server applications is particularly sparse; there are many virtual segments scattered all over the address space. For example, Java processes exhibit many virtual segments due to the dynamic nature of the JVM. This behavior is best observed in the case of Cassandra, which exhibits direct-mapped miss rates in the order of one miss per 10^7 and 10^6 memory accesses for the $8\times$–$2\times$ cases and the $1\times$ case respectively. However, Cassandra's conflict misses drop rapidly as associativity increases and using 4 ways removes all the conflicts for the $8\times$ and $2\times$ cases, while 8 ways are required for the $1\times$ case. MySQL also exhibits a slighly higher fraction of conflict misses with direct-mapped as it heavily references several virtual segments (e.g., heap, buffer pool). MySQL's conflict miss rates are in the other of one miss per 10^8 and 10^6 memory accesses for the $8\times$–$2\times$ cases and the $1\times$ case respectively. However, the conflict rates of MySQL drop fastly as 2 ways removes all the conflicts for the $8\times$ and $2\times$ cases, while 8 ways are required for the $1\times$ case.

Overall, for single-process in-memory scenarios, compulsory misses clearly dominate all the misses even with the direct-mapped VM configuration. More specifically, direct-mapped makes compulsory misses represent 99.9% (95% for MySQL) of all misses. Furthermore, conflict misses become negligible with the addition of the first few ways, achieving with 4 ways a virtually zero conflict miss rate in the order of one miss per 10^8 memory accesses in the worst case across all workloads. The observation that conflict misses drop rapidly with associativity has also been shown for caches [35, 83].

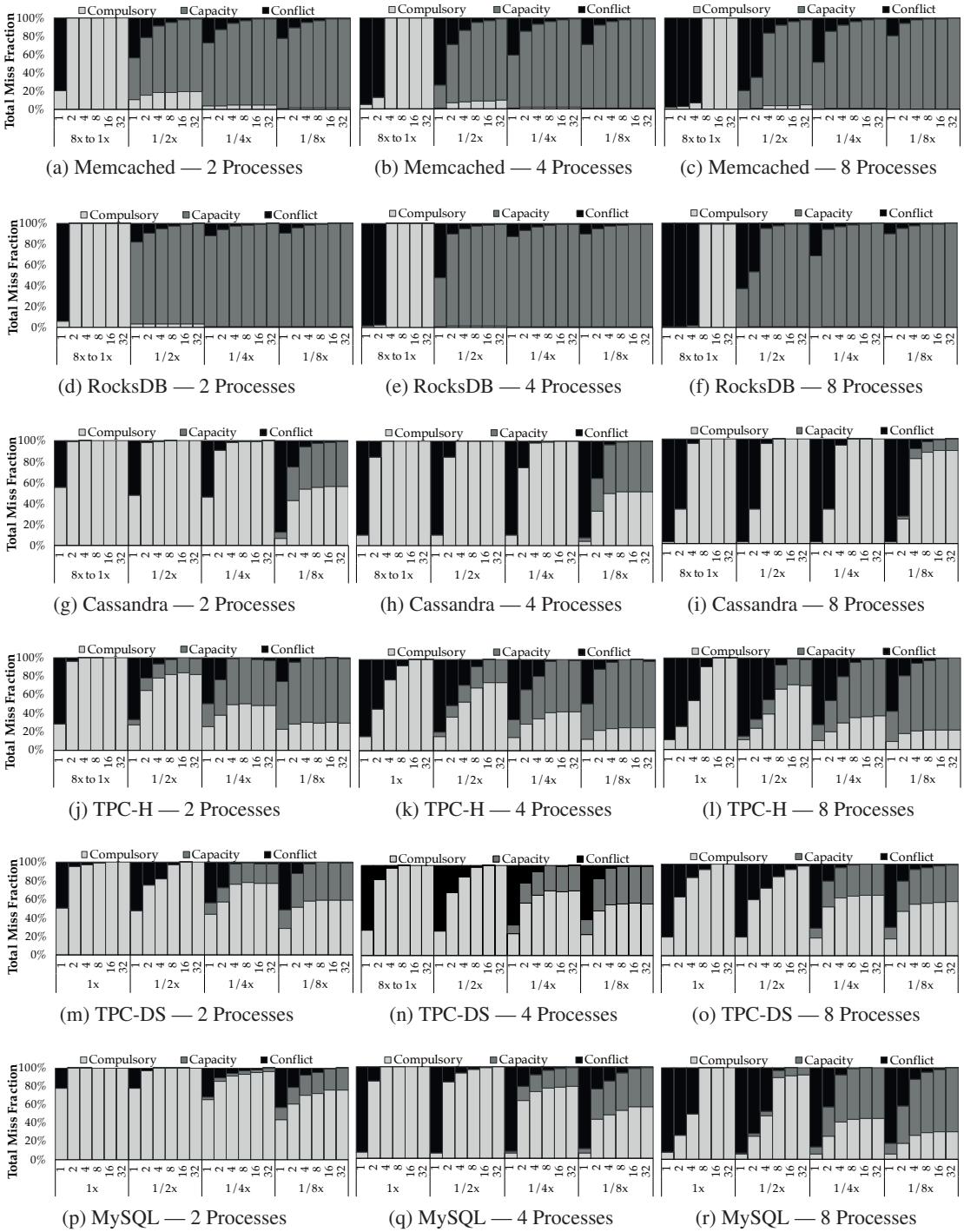
3.2.2 Single-Process Out-of-Memory

In contrast to the in-memory scenarios, when the working sets do not fit in memory (1/2×, 1/4×, 1/8× cases) capacity misses grow with the dataset size. Naturally, the fraction of compulsory misses becomes less significant and its contribution to the miss ratio drop. Although conflict misses are more significant than in the in-memory scenarios, conflicts drop sharply after the addition of 2-4 ways, and become generally less significant as the dataset size increases—when capacity misses grow.

For Memcached and RocksDB, though more significant than in the in-memory scenarios, conflict misses drop fast after 2-4 ways, while the contribution of conflict and compulsory misses drop as the dataset sizes increases. Naturally, when the ratio between dataset size and memory size significantly grows, pages do not fit in memory with any associativity. Cassandra and MySQL follow similar trends: The contribution of compulsory misses drops as capacity misses arise, while conflict misses drop sharply after the addition of few ways, 2-8 ways in this case. The only difference with Memcached and RocksDB is that the contribution of conflict misses increases as the dataset size grows. Nevertheless, the reason is that each transaction in Cassandra and MySQL is an order of magnitude longer, due to the more complex software stacks, and therefore only a fraction of the total dataset is referenced during the simulated memory trace. In the impractical case of simulating a longer trace, all the pages would be touched, triggering capacity misses and making the trends look exactly the same as for Memcached and RocksDB. In any case, the trace is long enough to show that conflict misses drop sharply with the addition of a few ways.

TPC-H and TPC-DS, the only analytics workloads, show similar trends: The contribution of compulsory misses drops as capacity misses arise, while conflict misses drop sharply after the addition of few ways, 2-4 ways in this case. Furthermore, the contribution of conflict misses drops as the dataset size grows. Interestingly, TPC-H and TPC-DS, both running on MonetDB, exhibit an anti-LRU behavior: Conflict misses tend to slightly grow as associativity increases. In our case, we classify capacity misses as the increase in misses with respect of the best performing case (which is full associative for almost all cases). For example, 2 ways for TPC-H in the 1/4× case and 4 ways for TPC-DS in the 1/8× case. Nevertheless, these results also confirm that full associativity is not beneficial. Note that the results for the conflict misses are conservative as we run the TPC-H and TPC-DS queries once. Hence, all the database tables referenced a single time during the memory trace that end up being evicted from the memory, will be classified as capacity misses the next time the aforesaid tables are accessed. Lowering the contribution of conflict misses even more.

Overall, although conflict misses are more significant than in the in-memory scenarios, conflicts drop sharply after the addition of 2-4 ways and become less significant as capacity misses grow. With 16 ways, conflict misses represent $\sim 1\%$ of all the misses in the worst case across all workloads. Fundamentally, all these results corroborate seminal work on caches [83].



(a) Memcached — 2 Processes (b) Memcached — 4 Processes (c) Memcached — 8 Processes

(d) RocksDB — 2 Processes (e) RocksDB — 4 Processes (f) RocksDB — 8 Processes

(g) Cassandra — 2 Processes (h) Cassandra — 4 Processes (i) Cassandra — 8 Processes

(j) TPC-H — 2 Processes (k) TPC-H — 4 Processes (l) TPC-H — 8 Processes

(m) TPC-DS — 2 Processes (n) TPC-DS — 4 Processes (o) TPC-DS — 8 Processes

(p) MySQL — 2 Processes (q) MySQL — 4 Processes (r) MySQL — 8 Processes

Figure 3.2 – Overall miss ratio broken down into compulsory, capacity, and conflict misses.

3.2.3 Multi-programming In-Memory

Fig. 3.2 presents the associativity results for multi-programming scenarios. We take each of the workloads, Memcached, RocksDB, TPC-H, TPC-DS, MySQL, and Cassandra, and vary the number of processes, from 2 to 8, while keeping the aggregate dataset size the same with respect to the single process scenarios. For example, for the $1\times$ case in Fig. 3.1, the dataset of a single process equals the size of the memory. In Fig. 3.2, for two processes, each of them shares the same physical memory, and hence the working set of each process is half the size of the physical memory. The same applies with four processes, each of their working sets consumes only a quarter of the physical memory. We thus guarantee that only the increase in the number of processes has an impact on the associativity.

For the in-memory scenarios ($8\times$–$1\times$ cases), once the associativity equals the number of processes, compulsory misses represent 99.9% of all the misses for Memcached, RocksDB, Cassandra, and MySQL. For TPC-H and TPC-DS, 2 ways per process make compulsory misses 99.9% of all the misses. Hence, 1 or 2 ways per process (depending on the workload) makes conflict rates match that of the single process workloads in a direct-mapped configuration. Increasing the associativity further makes conflicts virtually disappear after the first few additional ways. For instance, for Memcached, with 8, 16, and 32 ways, the conflicts are in the order of a single miss per 10^9 accesses for 2, 4, and 8 processes, respectively. Cassandra requires 4, 8, and 8 ways to achieve a miss conflict rate of one miss per 10^9 accesses for 2, 4, and 8 processes, respectively. The trends are similar for TPC-H, TPC-DS, and MySQL. For example, even though two ways are required for TPC-H to make compulsory misses 99.9% of all the misses, similarly to Memcached and Cassandra, 4, 16, and 32 ways achieve a miss conflict rate of one miss per 10^9 accesses for 2, 4, and 8 processes, respectively. To put these miss rates into perspective, the time required to execute a billion memory accesses is in the order of seconds.

Although Fig. 3.1 and 3.2 show results that use the standard method of computing the index by using the least-significant bits of the VPN, we also studied an alternative proposal whose strength is distributing address more evenly over sets [147]. It did not reduce conflicts significantly due to the fact that page conflicts arise when many pages map to the same memory set, even when assuming a uniform page distribution. Changing the index function uniformly changes the distribution, and therefore does not mitigate the issue. Once again this behavior corroborates prior work on caches [83].

Overall, once the associativity equals the number of processes, compulsory misses vastly dominate. More specifically, a direct-mapped configuration (for Memcached, RocksDB, Cassandra, and MySQL) or 2 ways (for TPC-H and TPC-DS) makes compulsory misses represent 99.9% of all misses. Furthermore, conflict misses become negligible with the addition of the first few ways, achieving with 4 ways per process a virtually zero conflict miss rate in the order of one miss per 10^8 memory accesses in the worst case across all workloads.

3.2.4 Multi-programming Out-of-Memory

For the cases where the datasets do not fit in memory, the trends are similar to the single-process out-of-the memory scenarios. Capacity misses become more significant as the datasets grow, making the contribution of conflict and compulsory misses less significant. Furthermore, similarly to the multi-programming in-memory scenarios, once the associativity equals the number of processes, the contribution of conflict misses to the total becomes marginal. More specifically, a single way per process for Memcached, RocksDB, Cassandra, and MySQL, and 2 ways for TPC-H and TPC-DS, make the contribution of page conflicts similar to the direct-mapped single process cases. Again, conflicts drop rapidly as in the other cases and with 4 ways per process, conflict misses are within 1% of the total misses in the worst case for all the workloads.

Interestingly, though single-process out-of-memory scenarios require 16 ways in the worst case, the multi-programming ones require 4 ways per process only. The reason is that 16 ways are required for the worst-case scenario, which is MySQL in the 1/8× case. As explained earlier, MySQL exhibits a large number of compulsory misses even when the workload's dataset is larger than the physical memory. This behavior arises due to the complex software stack that each transaction needs to traverse, which precludes MySQL to touch all its dataset within the duration of the trace. In the multi-programming scenarios, each MySQL process is concurrently accessing its dataset and the memory, increasing the contribution of capacity misses to the total miss ratio. Hence, 4 ways per process are enough to keep conflict misses within 1% of the misses in the worst case.

Additionally, TPC-H and TPC-DS present an increase in capacity misses with respect to the single-process scenarios. We have observed that the size of the dataset touched by the queries increases with respect to the total dataset size as the scale factor of the database drops. Hence, the aggregated dataset referenced by the TPC-H and TPC-DS queries increases as the number of processes grow. One of the reasons is that MonetDB uses dictionary compression for strings, which compresses better for larger scale factors [60].

Overall, similarly to the in-memory case, once the associativity equals the number of processes, the contribution of conflict misses to the total becomes marginal. More specifically, direct-mapped or two ways per process (depending on the workload) makes the fraction of conflict misses match the single-process out-of-memory cases. Conflict misses drop rapidly as in the other cases, and with 4 ways per process, conflict misses remain within 1% of the total in the worst case for all the workloads.

3.3 Observations and Implications

Overall, the results demonstrate that the full associativity of virtual memory is unnecessary. Most importantly, this insight holds across scenarios that span single process, multi-programming, in-memory and out-of-memory datasets. The trends clearly indicate that compulsory and capacity

misses dominate, and conflict misses, which associativity alleviates, drop rapidly as the associativity increases. More specifically, for in-memory scenarios, compulsory misses dominate when associativity equals the number of processes in the system. For out-of-the-memory scenarios, capacity misses dominate and become more important as dataset sizes increase. In this case, conflict misses become scarce once associativity matches the number of processes, and virtually disappear after the first few additional ways.

To understand how modest the associativity requirements are, we should take a look at what modern systems provide. For instance, the latest server specification from the Open Compute Project [164] stipulates 128GB as the minimum memory capacity. Assuming 4KB pages, a fully associative VM would provide $32M$ ways. As conventional systems are reported to have around 100 processes concurrently running after booting [9], assuming a system with 128 processes and an associativity of 16 per process—which is extreme as we have seen that an associativity of 2–4 is sufficient—the total associativity requirements will not exceed $2K$ ways. Even for a server with a small amount of memory, the $2K$ requirement is $16K\times$ less than what full associativity provides. We expect this associativity gap between what is needed and what is provided to widen in the future as servers with TBs of memory are becoming ubiquitous [69], while emerging NVM technologies [1] are expected to be exposed as part of virtual memory (instead of using the IO block layer) [105].

In this work, we observe memory as just a fully associative software-managed virtually addressed cache (where the tags are the page table entries and the data are the page frame). While the context is completely different, our associativity trends fundamentally match prior literature on caches [35, 81, 83]. In spirit, this part of our work is similar to that of Mark Hill three decades ago [81]. Just as the set-associative or direct-mapped caches provide nearly all the flexibility benefits of full associativity with much faster access times, a set-associative VM can provide all the benefits of full associativity with much faster translation times.

3.4 Summary

This chapter presented an associativity study of VM across a variety of scenarios. The study demonstrates that the full associativity of virtual memory is unnecessary, and only a significantly modest associativity is required. By classifying the page faults using the classic 3C model, we conclude that capacity and compulsory misses—which are unaffected by associativity—dominate, while modest associativity is sufficient to eradicate conflict misses. More specifically, for working sets that are memory resident, compulsory misses dominate when the VM associativity equals the number of processes executing in the system. For working sets that exceed the memory size, capacity misses dominate and grow as a function of the working set size. In contrast, conflict misses become scarce once the associativity of virtual memory matches the number of processes in the system, and virtually disappear after the first few additional ways. Much like seminal work on set-associative caches, the modest associativity requirements could be exploited to provide nearly all the flexibility benefits of full associativity with much faster translation times.

4 Set-Associative Virtual Memory

In this chapter, we exploit the modest associativity requirements in virtual memory to design an efficient translation mechanism in the context of memory-side processing units (MPUs).

In light of computing demands growing at a pace comparable to Moore's law, the end of Dennard scaling, and the slowdown of increasing silicon density have pushed system designers towards specialized architectures to boost computing efficiency and density. At the same time, the complexity of increasing the density of conventional DRAM chips, has made memory vendors to leverage three-dimensional integrated circuits and stack DRAM dies on top of logic. Two well-known commercial products of such 3D memory technology are Micron's Hybrid Memory Cube [127] and Hynix's High Bandwidth Memory [95]. The combination of substantial benefits in bandwidth and immense reductions in data movement has created a huge traction with a myriad of proposals of custom memory-side processing units (MPUs) for various computations types [10, 71, 111, 130, 145, 146, 176]. Along with the existence of an economic driver for more efficient processing, there is evidence pointing towards the future adoption of architectures with MPU-capable memory chips.

A key challenge in exploiting the full potential of these heterogeneous systems is the memory management between CPUs and MPUs. Industry initiatives such as the Heterogeneous System Architecture (HSA) Foundation are proposing to unify virtual memory (VM) between CPUs and any other processing unit in the system [85]. In this model, a pointer is equally valid on the CPU and MPU, simplifying data sharing and eliminating the need for explicit data copying and manual data consistency maintenance. In addition, VM enables efficient fine-grained memory accesses and transparent memory allocation and protection. Essentially, VM provides a familiar and powerful abstraction to programmers and operating systems alike. All these benefits have led HSA to adopt unified virtual memory into commercial GPUs, beginning with AMD's Carrizo chip [174]. Given the benefits of HSA-style unified virtual memory and its early adoption and appearance in commercial products, we expect future computation units such as MPUs to also follow this integration path.

Unfortunately, the benefits of VM come at a significant performance cost. Hardware support for

address translation in commercial CPUs encompasses both per-core MMUs [47] and centralized IOMMUs for devices [90], which rely on TLB hierarchies to achieve low translation overhead. However, large modern memories are beyond the reach of today's TLBs [22, 139, 140], resulting in frequent long-latency page walks and valuable CPU time lost. Due to the arbitrary distribution of page table entries across all the memory chips, page walks involve several chip-to-chip transfers of tens of nanoseconds per hop [87, 106, 165] for each level of the page table [70], resulting in unacceptable page walk overheads.

This chapter proposes SAVAgE (Set-Associative VirtuAl mEmory), an efficient translation mechanism for MPUs that eliminates most of the overhead of page walks. Our translation mechanism builds on the modest associativity requirements and characteristics of our network of memory chips. SAVAgE restricts the associativity of VM so that a virtual address uniquely identifies a memory chip and partition. This characteristic allows MPUs to access the memory as soon as the virtual address is known. Each memory partition integrates an MMU, which includes a TLB hierarchy and page table, that translates the virtual address and fetches the data—both the translation and data are always located in the MMU's local memory partition. Translation is fast due to four reasons: First, the translation is completely localized in the partition where the page frame resides, which allows the data fetch request and its translation always target the same partition and completely overlap. Second, a memory access within a memory partition is faster than accessing remote memory partitions, therefore achieving low-overhead page walks. Third, as the page frame resides in the same partition as data, the data fetch can immediately start after translation finishes. Last, as a memory partition offers an associativity in the order of hundred of thousand ways, the number of page faults is virtually identical to full associativity. SAVAgE achieves low-overhead page walks as the page walk and data fetch operations overlap almost entirely.

Essentially, SAVAgE turns memory into a virtually addressed software-managed cache, just that the memory is set-associative as opposed to fully associative. Again, one can think of memory as a cache, where the data array is the set of page frames, and the tag array is the page table.

The rest of this chapter is organized as follows. Section 4.1 presents background on MPU-capable memory chips. Section 4.2 introduces the architecture of SAVAgE and Section 4.3 discusses further considerations. We then describe our methodology in Section 4.4 and evaluate SAVAgE in Section 4.5. We conclude with a summary of SAVAgE in Section 4.6.

4.1 MPU-capable Memories

3D memory is an emerging promising technology for the memory system. This memory technology that vertically stacks multiple DRAM dies on top of a logic layer within a single package, by leveraging low-latency/high-bandwidth through-silicon vias (TSVs). Two well-known incarnations of such 3D memory technology are Micron's Hybrid Memory Cube (HMC) [127] and JEDEC's High Bandwidth Memory (HBM) [95]. Fig. 4.1a illustrates the anatomy of a 3D

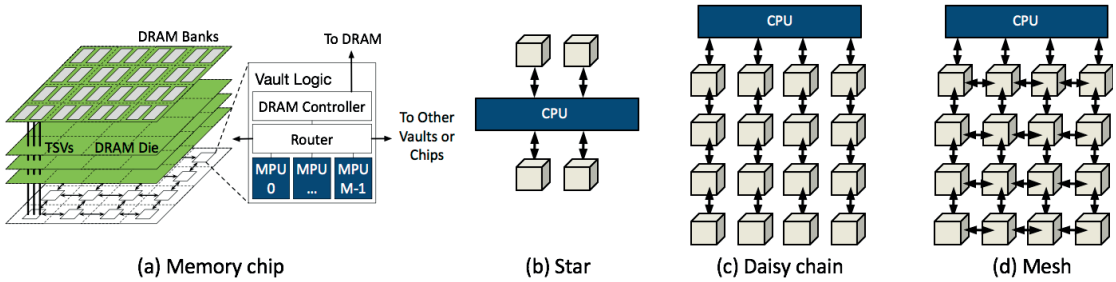


Figure 4.1 – Overview of a 3D memory architecture (a), and proposed and existing memory network topologies (b), (c), and (d).

memory chip. The chip consists of multiple vertical memory partitions, called vaults. Each vault is similar to a conventional DDR channel, with its own DRAM controller and signals, and hence can be accessed independently. Each DRAM controller connects to all the DRAM dies through a TSV bus. Similar to previous studies, the memory-side processing units (MPUs) are scattered across the vaults [10, 71, 111, 146], while a network-on-chip (NoC) connects all the vaults to each other and to the off-package links.

Architectures that deploy 3D memories consist of a pool of CPUs and memory chips. Fig.s 4.1b, 4.1c, and 4.1d show the memory organizations considered in this paper. The CPU is connected to multiple memory chips using high-speed point-to-point SerDes links and a packet-based communication protocol. Fig. 4.1b depicts a star topology where the CPU is connected to a small number of memory chips [68, 149]. Larger memory systems interconnect dozens of chips in a daisy chain (Fig. 4.1c), which minimizes the number of links [71, 146], or a mesh (Fig. 4.1d), which minimizes the number of hops [10, 112].

In these systems, a memory access from an MPU within its memory partition is much cheaper than accessing a remote one, as the latter involves traversing expensive NoC and cross-chip interconnects. Fig. 4.2 compares the average end-to-end memory access latency depending on the target data's location: 1) same partition, 2) different partition, same chip, and 3) any chip in the network. The three cases are labeled as *Partition*, *Chip*, and *Network* respectively. Unsurprisingly, the location of the data significantly affects latency. Data access in the local partition is the fastest; accessing a remote partition within the same chip is around 1.4× slower, while accessing remote memory chips increases the latency by 3.5–10×, depending on the network and memory chip count. In summary, these results indicate that optimizing for memory access latency requires MPUs to localize memory accesses within their partitions.

In this thesis, we illustrate and evaluate the benefits of our novel virtual memory subsystem in the context of memory-side processing units (MPUs). Though we can evaluate our VM subsystem in any context (e.g., CPUs), we believe MPUs stress the address translation mechanism the most and hence are the most challenging scenario for the following three reasons. First, the large number of memory chips and the arbitrary distribution of page table entries make page walks involve expensive cross-chip traffic. Second, the lack of deep cache hierarchies limits the caching

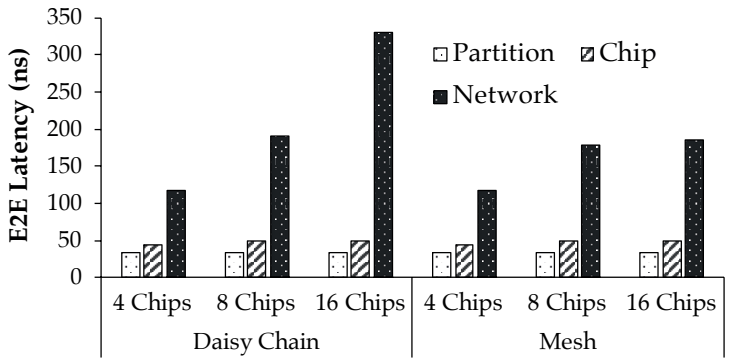


Figure 4.2 – End-to-end latency of memory accesses.

of page table entries close to the MPUs. Third, the lean nature of the MPU cores (due to the tight power and area constraints) precludes integrating expensive hardware to overlap page walks with useful work. All these aforesaid characteristics of MPU architectures contributes for making page walks a severe performance bottleneck. As a result, we focus on memory-side processing units (MPUs) for the rest of this thesis, although our insights and translation mechanisms are widely applicable to any other context.

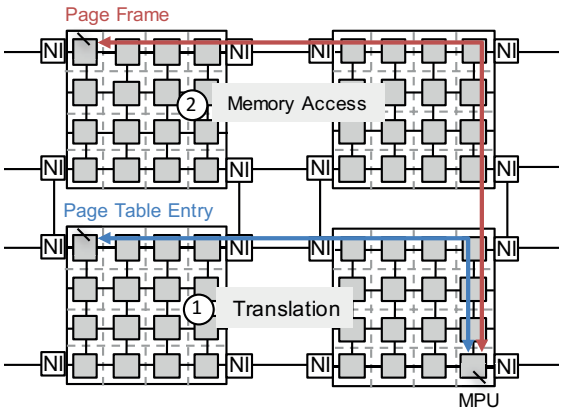
4.2 The SAVAgE Architecture

In this section, we investigate the design space for providing an efficient translation mechanism for the MPUs, given the modest associativity requirements and the characteristics of our network of memory chips. Then, we propose and describe our solution, SAVAgE.

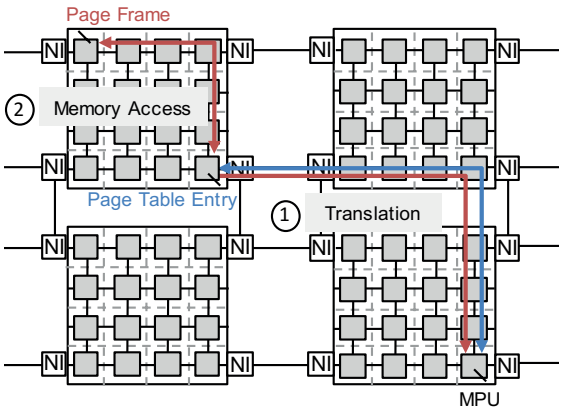
4.2.1 Placement

As explained before, virtually addressed MPUs require hardware support for virtual-to-physical address translation. A conventional design integrates an MMU in each MPU, each MPU probes its MMU on each memory access, and upon a miss in its TLBs, a page walk process beings. Fig. 4.3a shows a page walk that references a page table entry which resides in another memory chip and partition. In a system with multiple chips and partitions, this will be the common case. Blue arrows indicate memory messages related to translation, whereas red arrows are part of the data fetch. In other words, memory accesses to page table entries are colored in blue, whereas memory accesses to page frames are red.

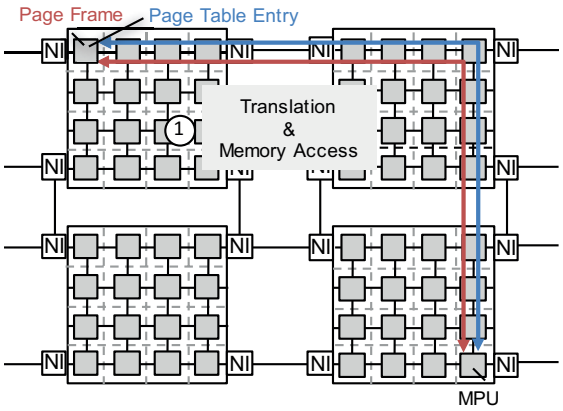
There are several problems with the baseline. First, the page table entry that contains the physical address can be anywhere in the system, involving expensive interconnect traversals that make page walks costly. Second, the data fetch cannot begin until the translation operation finishes, adding the translation latency to the data fetch latency. Third, the translation process does not finish until the page table entry returns to the MPU, even if the page table entry and the target



(a) Baseline.



(b) Per-Chip MMU.



(c) Per-Partition MMU.

Figure 4.3 – Different translation schemes on 4-chip floor plans. Large squares represent memory chips. Small gray squares are memory partitions. Translation and memory access messages appear in blue and red, respectively.

page frame were in the same chip or memory partition. The fundamental problem is that there is no direct relation between virtual page numbers and page frames, something inherent to fully

associative virtual memory. The consequence is that there is no expected correlation between the location of page table entries and page frames, precluding fetching the data before the page table entry returns to the MPU. Although this design makes sense for fully associative VM, it is not optimal for a system where VM is set associative.

We now exploit the insight of low-associativity requirements and consider that the memory set only spans a memory chip. In other words, a virtual address identifies a memory chip uniquely, and hence we know that the target page frame is somewhere in that chip. Under this constraint, we can utilize a centralized per-chip MMU, along with a page table and a set of TLBs, as shown in Fig. 5b. Upon a page walk operation from an MPU, the virtual address is used to access the per-chip MMU. If the translation is not in the per-chip TLBs, a page walk in the memory chip begins. When the page walk finishes, the MMU in the memory chip starts the data fetch immediately, referencing the page frame (which can reside in any of the partitions in the chip). When the data returns to the centralized MMU, both the page table entry and the data return to the MPU.

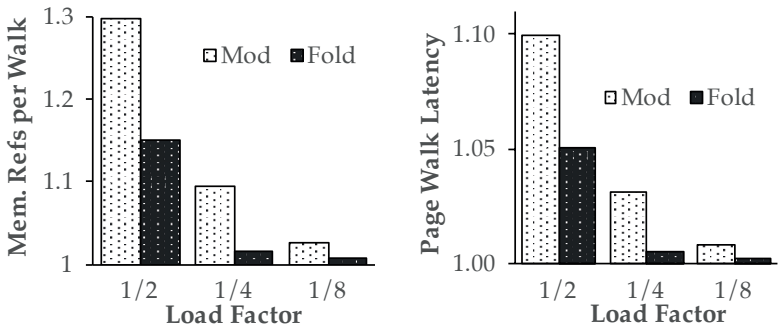
This method is an improvement over the baseline for the following two reasons. First, the translation and data fetch operations overlap the latency on the interconnect to reach the target memory chip, as only the virtual address is used to locate the memory chip. Second, the translation process finishes as soon as the page walk within the memory chip finishes. However, there is room for improvement. As the virtual address only identifies the memory chip, page frames and page table entries can be in any memory partition. This placement creates two problems. First, the centralized per-chip MMU can be located anywhere in the chip, and upon a page walk, the page table entry could be located in a different partition. Second, the target page frame might be located in any other partition. Based on the latency results of Fig. 4.2, the overhead could be significant.

A step further in designing an address translation mechanism tailored to the characteristics of our system is to co-locate the page table entries and the page frames in the same memory partition, where memory accesses are faster. In this case, shown in Fig. 5b, a memory set only encompasses a memory partition. The locality-aware placement of the MMU delivers the best performance among all the cases. First, the translation and data fetch operation overlap the latency to reach the memory partition—because both requests go to the same location. Second, the translation finishes as soon as the page walk finishes (or upon a TLB hit), and since the target page frame is within the same partition, the data fetch can begin immediately.

Now that we have decided upon the placement of the memory-side MMUs, we need to decide on what page table structure and TLB hierarchy to employ.

4.2.2 Page Table

Now that we know the location of the memory-side MMUs, the next step is to decide on the page table structure. There are many types of page tables in the literature, although there are many, the



(a) Memory references per page walk. (b) Page walk latency normalized to DRAM latency.

Figure 4.4 – Inverted page table performance.

most popular ones are hierarchical and inverted [177]. We choose to use an inverted page table for the four following reasons. First, inverted page tables do not need to be dynamically resized, so they are allocated once and pinned contiguously in memory [93]. Second, all the processes whose pages map to the partition share the same page table. Third, inverted page tables avoid the need for page walk caches [177]. Finally, when there are no collisions in the inverted page table, page walks only require a single memory reference.

One of the key parameters of an inverted page table is the ratio of page table entries to the total number of page frames, also called the load factor [46]. The load factor directly dictates the rate of collisions (i.e. two virtual page numbers mapping to the same page table entry). Importantly, given a load factor, the collision rate is not affected by the working set size or access patterns, assuming uniform hashing [46]. In Fig. 6a, we compare a conventional modulo hash to a stronger k-bit XOR folding [147], to demonstrate the effects of hashing on page table collisions. As we have not seen significant differences in workloads, process counts, and memory size to dataset size ratios, we present an average of all the results. The results indicate that an inverted page table with a load factor of 1/4 and the fold hash faction practically removes all page conflicts. To achieve similar conflicts with the modulo hash function we would need a load factor of 1/8. Therefore, our design uses inverted page table with a k-bit XOR folding hash function and a load factor of 1/4.

Fig. 6b shows the page walk latency normalized to the latency of a single DRAM access. For our configuration, the page walk latency is within 1.005× of a DRAM access. The reason the page walk latency does not directly correlate with the number of memory references per walk is that we use open addressing for resolving conflicts [177]. In open addressing, upon a collision, we probe the next entry in the inverted page table, rather than dereferencing a pointer to a new page table entry, exploiting the locality in the DRAM row buffer [147], and thus reducing the latency overhead of collisions. Additionally, open addressing removes the pointer per page table entry required to resolve collisions. Each entry in our page table holds a 36-bit virtual page number (VPN), a 12-bit address space identifier (ASID), the 36-bit page frame number (PFN), and 12

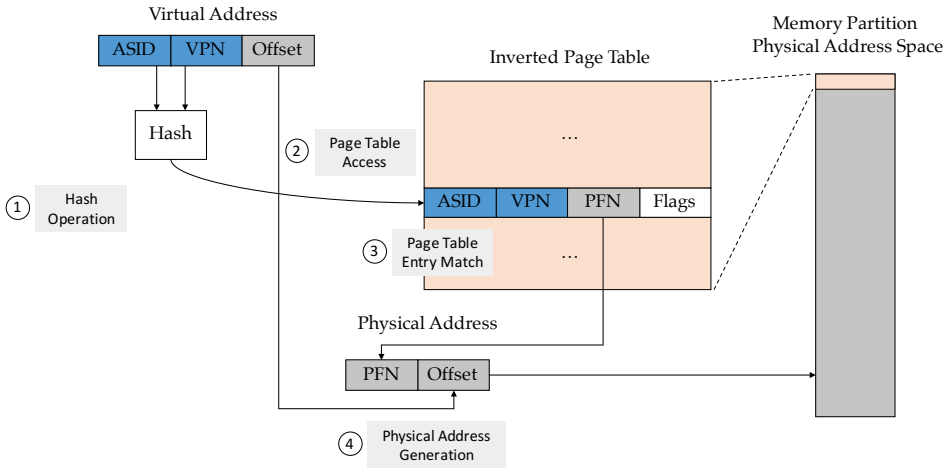


Figure 4.5 – Inverted page table structure and the page walk operation.

bits for page flags, fitting in 16-byte entries. We assume 48-bit virtual and physical address as in today's mainstream architectures, x86_64 and ARMv8. As we allocate four 16B entries for each 4KB page, due to the 1/4 load factor, the inverted page table consumes a modest 1.6% of the physical memory.

Figure 4.5 shows the structure of the inverted page table and page walk operation. At the beginning, the concatenated ASID and VPN bits are hashed to calculate an index in the inverted page table ①. The index is employed to locate an entry in the inverted page table ②. Upon a match (otherwise the next index in the page table is accessed) the page frame number is extracted ③ and concatenated with the page offset bits to generate the physical address ④. The physical address points to a location in the memory partition.

4.2.3 TLB Hierarchy

Although address translation requires a single memory access in the common case, architects still want to minimize memory accesses as a matter of principle and therefore conventional MMUs incorporate a hierarchy of TLBs to cache frequently used page table entries. Fig. 4.6 shows how the TLB hit ratio scales as the number of entries increase across different process counts. As our translation performance is less dependent on the TLB's hit ratio, due to the reduced cost of page walks, we average the results across all workloads and scenarios, i.e., in-memory resident and larger than memory working sets. However, we do break down the average results across process counts as all the processes with pages mapping to the same partition would share the same MMU (more details on the methodology are found in Section 4.4). Although we expect way less than 8 processes running concurrently on the MPUs,[1] we see that even with eight different processes, a TLB with 64 entries, which is the usual size of a first-level TLB, achieves a hit ratio in excess of than 80%. Additionally, increasing the number of entries beyond 1024 gives

[1] All MPUs running in the same address space are a single process.

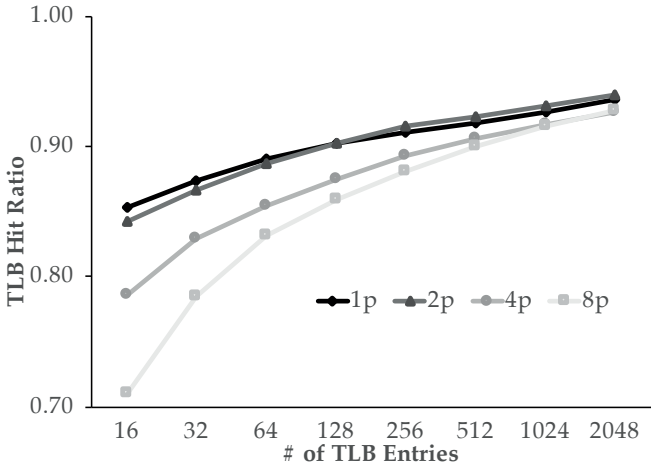


Figure 4.6 – TLB hit ratio sensitivity analysis.

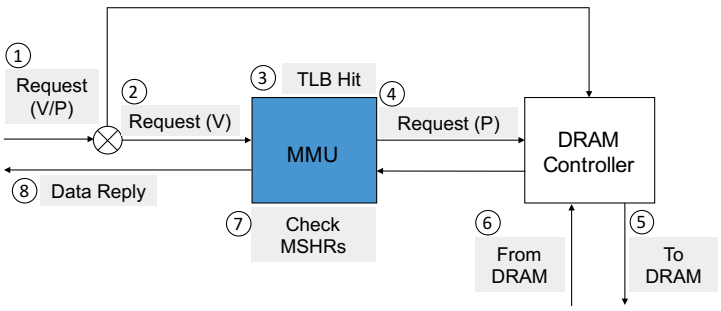
diminishing results, which again matches the size of a conventional second-level TLB. Therefore, a conventional two-level TLB hierarchy, a first level with 64 entries and a second level with 1024 is a sound approach. As we distribute the TLBs across MMUs, each MMU has its own, the TLB performance of an MMU is not affected by accesses to other partitions or chips. Hence, the TLB performance is robust across any memory chip and partition counts.

4.2.4 Putting Everything Together

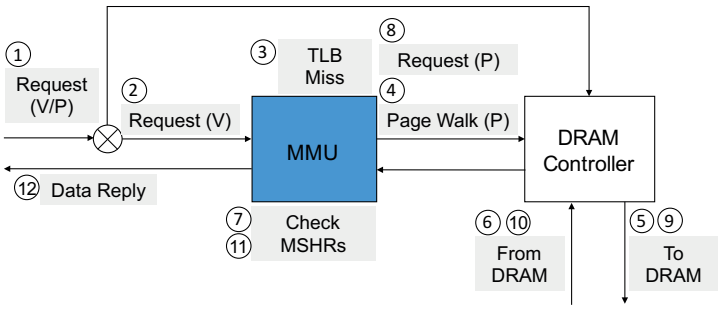
Now that we have defined the placement of the MMUs, the page table, and the TLB hierarchy, we provide a walkthrough of the memory-side MMU's operation in Fig. 4.7, for both TLB hits and page walks.

In Fig. 4.7a, we look at the case where the TLBs in the memory-side MMU experience a hit. A request from an MPU arrives at the router of the target partition which forwards it to the MMU ①. A multiplexer checks whether the message uses virtual or physical addresses ②. Memory requests that use physical addresses either come from a CPU core, DMA, or MPU that hit in its TLB hierarchy.[2] In the case of physical addresses, the request is sent directly to the DRAM controller. In this example, the request uses virtual addresses, hence arriving at the MMU, and the TLB hierarchy is probed, resulting in a TLB hit ③. Note that the hit requires both the VPN and ASID bits to match, which are in the requests. The MMU then appends the offset bits to the page frame number to create the physical address, creates an MSHR entry tagged with this physical address, and sends a request to the DRAM controller ④. When the DRAM controller finds the address, it sends DRAM commands to fetch the appropriate cache block ⑤. When the reply comes back ⑥, the DRAM controller forwards the reply to the MMU. The MMU checks the MSHRs to see if there is a match ⑦, as it might be a reply to a request that used physical addresses, and then sends the reply back to the MPU, which contains both the data and translation

[2]MPUs tag requests that use virtual addresses with a cookie.



(a) TLB hit operation.



(b) TLB miss operation.

Figure 4.7 – Operation flow of memory-side MMUs.

information ⑧.

In Fig. 4.7b, the memory-side MMU experiences a miss in its TLB hierarchy, triggering a page walk. A virtual request from an MPU arrives at the multiplexer ① and gets forwarded to the memory-side MMU ②. The MMU probes its TLB hierarchy, but the page table entry is not cached ③. The MMU generates the physical address of the page table entry by adding the output of the hashing function on the request's virtual address to the base address of the inverted page table. The MMU then creates an MSHR entry tagged with this physical address and sends the request (for the page table entry) to the DRAM controller ④. When the DRAM controller finds the address, it sends DRAM commands to fetch the appropriate page table entry ⑤. When the reply comes back ⑥, the DRAM controller forwards the reply to the MMU. The MMU checks the MSHRs for a match ⑦. The matching MSHR entry indicates that it was a page walk, and therefore the MMU generates the request for the actual cache block ⑧, and repeats the steps shown for the TLB hit operation: ⑨, ⑩, and ⑪. Finally, it sends the reply back to the MPU with the data and translation ⑫.

Note that most of the functionality of the MSHRs in the MMU is performed by the request queues at the memory controller. We could extend the state of the queues and provide the same functionality. However, we avoid the complexity of extending the memory controller due to the modest hardware requirements for the MSHRs, 64 to 128 entries [119] assuming overprovisioning for the worst case, we prefer to avoid the complexity of modifying the memory

controller. Additionally, to further reduce the already low memory bandwidth requirements of the page walks, as the TLBs achieve high hit ratios, we indicate to the the memory controller that the request for the page table entry is 16 bytes, instead of the conventional 64-byte requests. Current memory controllers of 3D memories already support different request sizes [128].

4.3 Discussion

In this section, we discuss possible concerns and further considerations regarding our translation mechanism.

4.3.1 Page faults

Upon a page fault triggered by an MPU, we choose to interrupt the CPU to run a handler, as MPUs may not be capable of running an OS. The memory-side MMU notifies the MPU of the fault, and then the MPU places a request in a memory-mapped queue indicating the faulting virtual address and MPU's id. Then, it raises an interrupt on the CPU. The handler running on the CPU resolves the page fault and updates both the CPU's and memory-side MMU's page table. All MMU state is exposed through memory-mapped IO with an uncacheable memory policy. Once the fault is serviced, the handler notifies the appropriate MMU, which resumes execution and retries the faulting address. Such page fault processing is also employed in today's integrated GPU processors [166].

4.3.2 TLB shootdowns

Many ways exist to maintain the memory-side page tables coherent upon TLB shootdowns initiated by the CPU. Our approach is similar to those used in integrated GPUs: an OS driver monitors any changes on virtual address spaces shared with MPUs, triggering update operations for the affected page table entries. Note that the inverted nature of the page tables eliminates any global coherence activity in the memory network as a virtual page maps to only one partition, and consequently, page table. Additionally, modifying the desired page table entry requires a single memory access in the common case, accounting for at most a few hundred nanoseconds. This additional overhead is not as significant as the TLB shootdown operation itself, which removes the stale entries in the TLBs of MPUs and CPU cores, and already takes tens of microseconds [135].

4.3.3 Cache hierarchy

In this work, we assume MPUs look like conventional cores, integrating physical caches and an MMU with TLBs. Upon a TLB miss, the memory-side MMUs reply with the page table entry and cache block. Then, the MPU stores them in the TLB and data cache respectively. However, a more natural design, which also avoids TLBs and TLB shootdowns [167] is to use virtual

caches. Recent practical designs for virtual cache hierarchies [138, 178] would be a perfect fit for the memory-side MMUs. In this approach, MPUs access the cache with virtual addresses, and upon a cache miss, the request is propagated to the memory-side MMUs to translate and fetch the corresponding block. The memory-side MMU would only reply with the data cache block, without the page table entry, simplifying our design.

4.3.4 Synonyms

In our current design, all the synonyms of a particular page frame need to map to the same memory set or partition. We believe this is not a significant limitation as it has been already included in commercial systems [41]. Nevertheless, our page table entries contain the whole page frame address, and hence with a slight modification in our design, we could enable synonyms to map anywhere. The only potential consequence is a drop in performance as a synonym page might map to a page frame residing in another memory partition. However, prior work has already shown that synonyms are both scarce and infrequent [23, 138, 178], and therefore this overhead should be negligible.

4.3.5 Multi-level memories

Although prior work on memory-side processing assumes a single level [10, 11, 71, 146], memory can be organized as a hierarchy, with a die-stacked cache [149, 168] backed up by planar memory. For hardware-managed caches, the memory-side MMU performs the translation and accesses the partition, and in case the page frame is not there, the page is fetched from planar memory as part of the standard cache miss operation. Once the page arrives into the partition, the data is sent back to the MPU. Note that moving the page from planar memory to the 3D memory does not affect the page table entry. In software-managed hierarchies [149], MPUs rely on the software API for explicit migration of pages into the die-stacked memories, as MPUs cannot access planar memory directly.

4.3.6 Kernel memory

Our simulation infrastructure captures user-level instructions, although we argue that MPUs will accelerate only user-level not kernel-level code, like all existing custom hardware (e.g., GPU, FPGA). Nevertheless, our technique should be a great fit for the kernel, as Linux's memory usage is almost entirely direct-mapped [126] and memory resident. The tag matching logic of the TLBs would then require the following simple change. As all the processes share the same kernel virtual addresses, the logic needs to ignore the ASID bits. This modification is trivial as Linux partitions the address space into two halves, and hence the tag matching logic just needs to check the virtual address's MSB.

4.3.7 OS support

The OS only needs to guarantee that the virtual page number and the page frame number map to the same memory set. OSs that support virtual caches already provide this capability (e.g., Solaris [41] and MIPS OS [161]). MIPS OS traverses the free list of pages and returns a page frame that resides in the same memory set as the virtual page. Alternatively, one can associate a separate free list of pages per memory set, similarly to the facilities that Linux implements for its libnuma support.

4.4 Methodology

Like most recent work on VM [19, 22, 25, 137, 139, 140, 152], we use trace-driven functional and cycle-accurate simulation.

4.4.1 Performance

This study does not cover the problem of partitioning applications into CPU-side and memory-side execution, or specializing the latter in hardware, or interfacing and synchronizing the two executions [10, 17, 71]. Such problems are out of the scope. To avoid such problems we evaluate the applications running entirely on the MPUs.

Full-system simulation for events such as TLB misses and page faults is not practical, as these events occur less frequently than other micro-architectural events (e.g., branch mispredictions). Hence, we resort to the CPI models often used in VM research [27, 137] to sketch the performance gains. These prior studies report performance as the reduction in the translation-related cycles per instruction. As CPI components are additive, this metric is valid irrespective of the workload's baseline CPI. We further strengthen this methodology by studying the CPI savings on all memory cycles, not only on translation stalls (as we overlap translation and data fetch operations). Our model thus captures both the translation cycles and data fetch cycles, which together constitute the largest fraction of the total CPI in server workloads [66]. Hence, our results are more representative of the end-to-end benefits of each technique. The CPI is measured by feeding the memory traces into our cycle-accurate simulator.

4.4.2 Traces

We collect memory traces using the Pin binary instrumentation tool [124]. For workloads with fine-grained transactions (i.e., Memcached, RocksDB, MySQL, and Cassandra), the traces contain the same number of instructions as the application executes in 60 seconds without Pin. For analytics workloads (i.e., TPC-H and TPC-DS), we instrument the entire execution. We extract the ASID bits and the virtual address of each memory reference, and concatenate both to form the final virtual address [23, 178].

The traces are used for the page table experiments of Section 4.2. We feed the traces into our inverted page table modeling tool, which allows us to model different load factors and hash functions. For the TLB experiments of Section 4.2.3 and the performance experiments of Section 4.5, we tune the workloads to employ datasets of size 32GB and 64GB, depending on the size of the network. We use 32GB for 4- and 8-chip configurations, and 64GB when using 16 memory chips.

We collect the traces on a dual-socket server CPU (Intel Xeon E5-2680 v3) with 256GB of memory, using the Linux 3.10 kernel and Google's TCMalloc [75]. Address space randomization (ASLR) is enabled in all experiments.

4.4.3 Simulation Parameters

We use the Flexus cycle-accurate simulator [172], with detailed core, MMU, memory hierarchy, and interconnect models. Following prior work on memory-side processing, which assumes single-issue in-order cores [10, 71, 146], we model the MPU cores after ARM Cortex A7 [15]. We provision the baseline with a high-end MMU similar to Intel Xeon Haswell [47, 77], with multi-level TLBs and MMU caches [18, 25]. We assume a 4-level hierarchical radix tree page table [93] with 48-bit virtual and physical addresses (as in ARMv8 and x86_64). The MMU supports 4KB, 2MB, and 1GB pages. Page table entries are transparently allocated in the L1-D cache like in commercial systems [89]. For simplicity, we probe the cache with physical addresses for the baseline and with virtual addresses when using SAVAgE. We verify that TLB misses never reference a cache-resident block, and therefore virtual and physical caches behave identically.

Without loss of generality, we model our 3D memory stack as following the organization of the Micron Hybrid Memory Cube with eight 8Gb DRAM layers and 16 vaults [128], totalling 8GB of memory per chip. We conservatively estimate the die-stacked memory timing parameters from publicly available information and research literature [71]. For SAVAgE, we employ a conventional two-level TLB hierarchy. The SRAM overhead per memory chip is less than 256KB (partitioned across vaults) for an area of $0.3mm^2$ in 22nm, corresponding to less than 0.2% of the area of an 8Gb DRAM die (e.g., $226mm^2$ [154]). As there are 16 partitions, 512MB each, SAVAgE provides a VM associativity of 128K ways.

Last, we conservatively assume that page conflicts always generate a page fault to an SSD, taking $32\mu s$ to resolve [44]. Table 4.1 summarizes the used system parameters.

4.5 Evaluation

In this section, we perform a quantitative study of the performance of different translation mechanisms. Additionally, we provide a qualitative discussion on how our translation mechanism compares with other prior proposals.

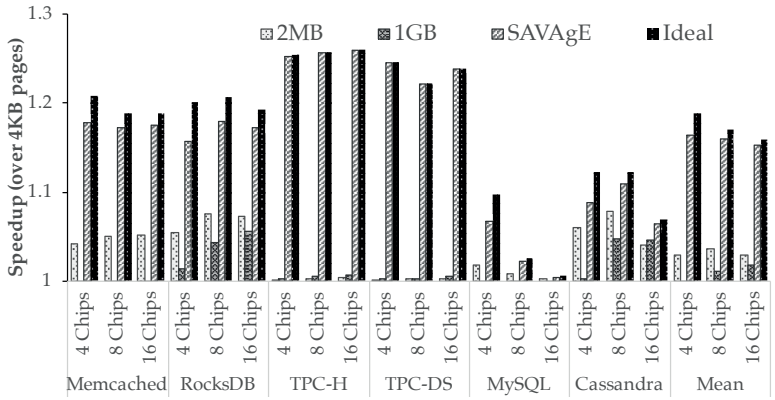
Table 4.1 – System parameters.

MPU logic	**Description**
Cores	Single-issue, in-order, 2GHz
L1-I/D	32KB, 2-way, 64B block, 2-cycle load-to-use
MMU	**Description**
TLB	4KB pages: 64-entry, 4-way associative 2MB pages: 32-entry, 4-way associative 1GB pages: 4-entry, fully associative
STLB	4KB/2MB pages: 1024-entry, 8-way associative
Caches	L4: 2-entry, fully associative [25] L3: 4-entry, fully associative [25] L2: 32-entry, 4-way associative [25]
Memory	**Description**
MPU chip	8GB chips, 8 DRAM layers x 16 vaults
Networks	4, 8, 12, and 16 chips in daisy chain and mesh
DRAM	t_{CK} 1.6ns, t_{RAS} 22.4ns, t_{RCD} 11.2ns t_{CAS} 11.2ns, t_{WR} 14.4ns, t_{RP} 11.2ns
Serial links	2B bidirectional, 10GHz, 30ns per hop [106, 165]
NoC	Mesh, 128-bit links, 3 cycles per hop
SAVAgE	**Description**
TLB	4KB pages: 64-entry, 4-way associative
STLB	4KB pages: 1024-entry, 8-way associative

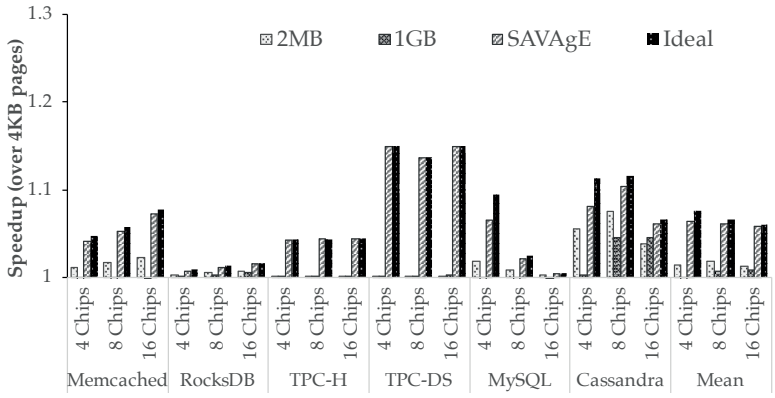
4.5.1 Performance Analysis

The paper discusses many different workload scenarios and system configurations. As there are way too many performance points, we will show only the scenarios that have significant performance differences. First, we will show two workload scenarios, in-memory, where the memory size is equal to the dataset size (i.e., the 1x ratio), and out-of-memory, where the dataset is eight times larger than the memory size (i.e., the 1/8x ratio). Second, as the performance across different process counts does not vary significantly, we are showing the average across runs. Third, the two topologies, daisy chain and mesh, behave similarly in terms performance, and therefore we also present the average across both topologies. Last, we present the results for different memory chip counts, 4, 8, and 16.

Fig. 4.8a shows the speedup of four translation mechanisms over the conventional MMU using 4KB pages, for the in-memory scenario. The techniques are the conventional MMU using 2MB and 1GB pages, SAVAgE, and an ideal translation with zero overhead. We label the techniques as 2MB, 1GB, Savage, and Ideal. First, we see that SAVAgE clearly outperforms 4KB, 2MB, and 1GB pages. In some cases by a large margin, up to 25% over 4KB, 2MB, and 1GB for TPC-H. In other cases more moderately, down to 3%, 2%, and 5% over 4KB, 2MB, and 1GB pages respectively for MySQL. Most importantly, SAVAgE is consistently on par with the ideal translation that incurs zero overhead for translation. Overall, SAVAgE improves the performance by 17% on average over 4KB pages and stays within 1.5% of the ideal translation on average.



(a) In-memory scenario.



(b) Out-of-memory scenario.

Figure 4.8 – Speedup over conventional MMU with 4KB pages for 2MB and 1GB pages, SAVAgE, and ideal translation.

Fig. 4.8b presents the speedups for the out-of-memory scenario. As expected, the speedups are less significant that in the in-memory case. The reason is that all the cases incur the slowdown of resolving page faults, and therefore there are fewer accesses that can be accelerated. Still, SAVAge systematically performs better than the conventional MMU, 6% on average over 4KB pages, and stays within 1.5% of the ideal translation mechanism on average.

We model a greatly optimistic case when using huge pages as we assume all pages are huge, with no generation overhead or fragmentation in any scenario. Additionally, we do not account for the excess in IO traffic generated by writing back dirty huge pages. Therefore, we expect further improvements with realistic overhead. Furthermore, we are conservatively assuming that all page faults are major, and hence involve an access to secondary storage, which might not always be the case (in cases of minor page faults). Hence, in a real system, our speedups would likely be significantly higher.

4.5.2 Comparison with Other Proposals

As explained in Section 2.4, there are a myriad of recent proposals on address translation for CPUs [22, 107, 139, 140]. All these techniques aim to enhance the reach of TLBs by exploiting the contiguity available in the virtual and physical address spaces. These techniques are orthogonal to our focus on the memory-side MMUs as we reduce the page walk overhead, and consequently, the TLB miss penalty.

Nevertheless, we believe that these techniques would be only effective for in-memory scenarios, and perform poorly when the datasets do not fit in memory. In out-of-memory scenarios, direct segments [22] would not perform well as a single virtual segment is mapped contiguously in physical memory at start up, precluding the most frequently used pages from staying in memory. Redundant Memory Mappings (RMMs) [107] would likely not be able to allocate contiguous chunks of physical memory, as the system is under heavy memory pressure. CoLT [140] and Clustered TLBs [139] would probably achieve similar performance to the conventional translation, given the limited contiguity available in memory.

4.6 SAVAgE Summary

This chapter proposed the Set-Associative VirtuAl mEmory (SAVAgE), a new translation mechanism that eliminates most of the overhead of page walks. Our translation mechanism builds on the modest associativity requirements and characteristics of our network of memory chips. SAVAgE restricts the associativity of VM so that a virtual address uniquely identifies a memory chip and partition. This characteristic allows MPUs to access the memory as soon as the virtual address is known. Each memory partition integrates an MMU, which includes a TLB hierarchy and page table, that translates the virtual address and fetches the data—both the translation and data are always located in the MMU's local memory partition. SAVAgE achieves low-overhead page walks as the page walk and data fetch operations overlap almost entirely.

5 Eliminating Associativity in Virtual Memory

In this chapter, we exploit the lack of associativity requirements in virtual memory when there is a single process taking up all the physical memory in the system. This case is typical of first-party workloads with tight latency constraints [22, 79].

Large-scale IT services are increasingly migrating from storage to memory because of massive data consumption rates [16, 53, 179]. Online services such as web search, social connectivity, and media streaming exhibit stringent response time requirements which mandate memory-resident data processing. Similarly, business intelligence over analytical pipelines are increasingly memory resident to minimize query response times. Furthermore, first-party workloads deployed by IT giants such as Microsoft and Google often run in dedicated servers, taking up all the system resources to ensure performance isolation and predictability [22, 79]. The end result is that memory has become pivotal to server design, which aims to maximize the throughput per server to minimize the total cost of ownership of datacenters.

The slowdown in Dennard scaling has further pushed server designers towards memory access efficiency. A large spectrum of IT services exhibit modest computational requirements but vast energy footprints due to moving data from memory all the way to the processor cores [50]. For instance, the energy cost of fetching a word of data from off-chip DRAM is almost four orders of magnitude (i.e., 10000×) more expensive than the processing cost [78]. To increase memory density, several memory vendors such as Hynix and Micron are vertically stacking multiple DRAM dies on top of a logic layer within a single package [95, 127]. The ample benefits in bandwidth and the proximity to the data—which minimizes time-consuming and energy-hungry data movement—have stimulated the emergence of custom memory-side processing units (MPUs) on the logic chip for many different types of computation.

To follow the expected integration path of a unified virtual memory between CPUs and MPUs (HSA-style) [85], MPUs must integrate an address translation mechanism. Unfortunately, the programmability benefits of VM come at a significant performance cost. Conventional hardware support for address translation relies on TLB hierarchies to achieve low translation overhead. Unfortunately, modern memory sizes are beyond the reach of today's TLBs [22, 139, 140],

resulting in frequent long-latency page walks and valuable CPU time lost. Due to the arbitrary distribution of page table entries across all the memory chips, page walks incur in overheads of hundreds of nanoseconds. Furthermore, SAVAgE, a novel technique to eliminate the overhead of page walks (presented in the previous chapter) provides close to ideal average performance, but leaves significant room for improvement on the table in certain scenarios (i.e., single-process in-memory workloads). As we focus on an important class of workloads—first-party IT services—maximizing the per-server throughput has the potential to bring large benefits back in the total cost of ownership of datacenters.

This chapter proposes DIPTA (Distributed Inverted Page Table), an address translation mechanism that completely eliminates the overhead of page walks. Our translation mechanism builds on the observation that the associativity of VM can be virtually eliminated for in-memory workloads. DIPTA restricts the associativity so that a page can only reside in a few number of physical locations which are physically adjacent—i.e., in the same memory chip and DRAM row. Hence, all but a few bits remain invariant across the virtual-to-physical mapping, and with a highly-accurate way predictor, the unknown bits are figured out so that address translation and data fetch are completely independent. Furthermore, to ensure that the data fetch and translation are completely overlapped, we place the page table entries next to the data in the form of an inverted page table, either in SRAM or embedded in DRAM. Hence, DIPTA completely eliminates the overhead of page walks for in-memory workloads.

Essentially, DIPTA turns memory into a virtually addressed hardware-managed cache. This cache's tag array is implemented as a distributed inverted page table and is accessed in parallel with the data array [96, 147].

The rest of this chapter is organized as follows. Section 5.1 presents an in-depth study on VM associativity for in-memory scenarios. Section 5.2 introduces the architecture of DIPTA, and Section 5.3 discusses further considerations. We describe our methodology in Section 5.4 and evaluate DIPTA in Section 5.5. We conclude the chapter with a summary of DIPTA in Section 5.7.

5.1 Revisiting Virtual Memory

Page-based virtual memory (VM) is an essential part of computer systems. At the time of its invention, the memory requirements of all active processes in the system exceeded the amount of available DRAM by orders of magnitude. A page table, which is a *fully-associative* software structure, was employed to maximize allocation flexibility by allowing any virtual page to map to any available *page frame*. Interestingly, this architecture has barely changed and has only incorporated hardware structures to cache page table entries like TLBs [48] and multi-level MMU caches [18, 25].

With storage devices in recent decades dramatically lagging behind processors and memory in performance, and DRAM continuously improving in density and cost, many online services and

Table 5.1 – Workload description.

Workload	Description
Cassandra	NoSQL data store running Yahoo's YCSB.
Memcached	Cache store running a Twitter-like client workload [121].
TPC-H	TPC-H on MonetDB column store (Q1-Q21).
TPC-DS	TPC-DS on MonetDB column store (Queries of [114]).
MySQL	SQL DBMS running Facebook's LinkBench [62].
Neo4j	Graph DBMS running neighbor search on MusicBrainz [4].
RocksDB	Store engine running Facebook benchmarks [63].

analytic engines are carefully engineered to fit their working set in memory [5, 6, 22, 32, 66, 79, 117, 136, 145, 179]. Due to rare page swapping, contiguous virtual pages are often mapped to contiguous physical pages [139, 140], and hence the conventional page placement flexibility provided by full associativity remains largely unused. As such, we believe that the traditional *full associativity* of virtual memory (i.e., the flexibility to map any virtual page to any page frame) should be revisited in search for a simpler and more efficient mapping.

5.1.1 Revisiting Associativity

Limiting associativity means that a page cannot reside anywhere in the physical memory but only in a fixed number of locations. For instance, a direct-mapped configuration maps each virtual page to a single page frame. Note that multiple virtual pages could map to the same physical frame, resulting in *page conflicts*. Increasing the associativity adds more flexibility to the page mapping and reduces conflicts. To simulate real-world scenarios in our study, we select a set of representative server workloads, summarized in Table 5.1. We include two cloud workloads from CloudSuite [66], Cassandra and Memcached, an online transaction processing (OLTP) workload [62], MySQL, two online analytical processing (OLAP) workloads [29], TPC-H and TPC-DS, a neighbor search workload on a graph DBMS, Neo4j, and a widely-used storage system workload [57], RocksDB. We collect long memory traces of the server workloads using Pin [124]. We extract the virtual address and address space identifier (ASID) of each memory reference and use it to probe a variably associative memory structure, to observe and classify the misses. A detailed description of our methodology is found in Section 5.4.

Table 5.2 (left) shows the page conflict rate as associativity varies. As shown, little associativity is enough to eliminate all page conflicts and match a fully-associative VM. On the one hand, Memcached and RocksDB do not exhibit frequent conflicts due to great contiguity in their virtual address space, as subsequent virtual pages are mapped to subsequent sets, never causing conflicts within a segment. Memcached maps the majority of its dataset into a single virtual segment, and hence a few extra ways eliminate all the conflicts. In contrast, RocksDB maps hundreds of memory-mapped files, each mapping to its own segment. Although there is high

Table 5.2 – Impact of associativity on page conflict rate and page conflict overhead.

	Page conflict rate (page conflicts per million accesses)					Page conflict overhead norm. to memory latency ($rate \times penalty/memory latency$)				
	DM	2-Way	4-Way	8-Way	16-Way	DM	2-Way	4-Way	8-Way	16-Way
RocksDB	8.4×10^{-2}	3.0×10^{-2}	2.5×10^{-2}	2.2×10^{-2}	0	2.8%	0.98%	0.83%	0.73%	0
TPC-H	1.0	1.6×10^{-1}	1.7×10^{-3}	0	—	33.68%	5.2%	0.06%	0	—
TPC-DS	1.4×10^{-1}	2×10^{-4}	0	—	—	4.71%	0.01%	0	—	—
Cassandra	1.1	3.7×10^{-2}	3×10^{-4}	0	—	37.2%	1.22%	0.01%	0	—
Neo4j	3.9×10^{1}	2.8×10^{-2}	0	—	—	1300.8%	0.93%	0	—	—
MySQL	2.4	1.7×10^{-3}	0	—	—	80.48%	0.06%	0	—	—
Memcached	5.0×10^{-2}	7.8×10^{-3}	0	—	—	1.65%	0.26%	0	—	—

contiguity across these segments, it creates conflicts, requiring 16 ways to completely eliminate page conflicts. Neo4j and Cassandra exhibit a large number of conflicts for a direct-mapped configuration because of their numerous randomly-placed JVM segments which conflict with each other. However, conflicts drop fastly, and 4 and 8 ways eliminate all the page conflicts for Neo4j and Cassandra respectively. Last, MySQL exhibits significant conflicts for the direct-mapped configuration as it heavily references the heap and buffer pool virtual segments. However, with a 2-way configuration the conflicts become seldom. The reason for page conflicts is two-fold: (i) the virtual space is not fully contiguous, and (ii) the software is unaware of the set-associative organization. Fortunately, the virtual space exhibits enough contiguity so that even unmodified software stacks can tolerate limited associativity.

Figure 5.1 plots the number of page conflicts as a function of associativity, normalized to the direct-mapped case. For most workloads, 2-way associativity provides the highest drop in the number of conflicts, a behavior also seen in caches [82]. Nevertheless, higher associativity may still be needed to avoid all conflicts for certain workloads, such as RocksDB and TPC-H, which heavily use memory-mapped files. Page conflicts decrease rapidly beyond a few ways, making an associativity larger than 8 exhibit marginal reductions in conflicts, which corroborates prior work on set-associative caches [35, 82].

Table 5.2 (right) estimates the average memory access time (AMAT) increase due to page conflicts. Here we conservatively assume that MPU's DRAM accesses are always local (the lower the memory latency, the higher the relative overhead of page conflicts). We also conservatively assume that page conflicts generate a page fault to an HDD, taking 10ms [2]. Overall, limiting the VM associativity to 4 ways introduces virtually zero overhead (e.g., adding less than 0.1% to the AMAT in the worst case). This overhead and the required associativity would further decrease in the presence of faster SSD storage or a small fully-associative software victim cache (as proposed before in the context of direct-mapped hardware caches [100]).

5.2 DIPTA

We exploit limited associativity to design a novel and efficient near-memory address translation mechanism. We first present simple SRAM-based implementations of DIPTA and show how to organize memory pages into DRAM rows to enable efficient address translation. We then present

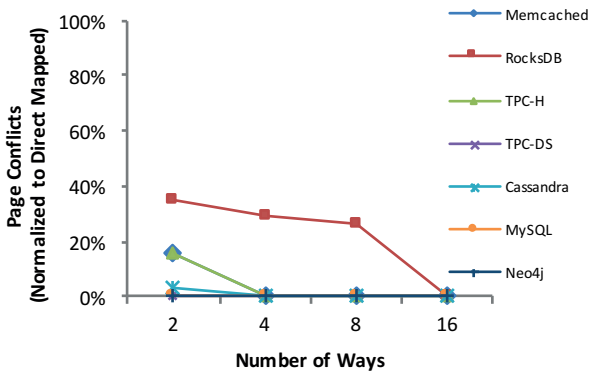


Figure 5.1 – Impact of associativity on page conflict rate.

scalable implementations where the translation information is embedded in memory. Finally, we conclude with a discussion of implementation costs and overheads.

5.2.1 SRAM-based DIPTA

To remove address translation completely from the critical path, it is *necessary* and *sufficient* to ensure that translation never takes more time than data fetch. Conventional hardware translation does not meet this requirement, as it can take an unpredictable amount of time, and the translation and data fetch latencies are rarely balanced.

As an effective way to ensure that the translation time never exceeds the data fetch time, we propose to distribute the translation information and co-locate it with the data, fetching them together so as to not expose the translation latency. In the proposed architecture each DRAM vault keeps the information about the virtual pages it contains in an SRAM structure, in the form of an inverted page table. The resulting Distributed Inverted Page Table (DIPTA) is looked up in parallel with the data fetch.

The inverted page table entries (per vault) can reside in a cache-like SRAM structure which is either direct-mapped or set-associative, depending on the associativity of VM. Assuming a 2GB MPU chip and 4KB pages, DIPTA would contain 512K entries; one entry per page frame. Each entry would hold the VPN of the page residing in the corresponding frame (36 bits) and the rest of the metadata, including 12 bits for the address space identifier (ASID) and 12 bits for page flags, totaling less than 8B for 48-bit virtual addresses (e.g., x86_64, ARMv8).[1]. The size of the table would be 4MB per MPU chip. Assuming 16-32 vaults [127, 162], the per-vault SRAM storage overhead totals 128KB-256KB. For illustration purposes we assume 4KB DRAM rows.

A direct-mapped implementation is trivial as the DIPTA SRAM lookup can proceed in parallel with the data fetch. The reason is that the virtual address enables direct indexing of both DIPTA and DRAM, as the address uniquely identifies the DRAM row and column of the target cache

[1] Note that memory requests coming from MPUs contain both the VPN and ASID bits.

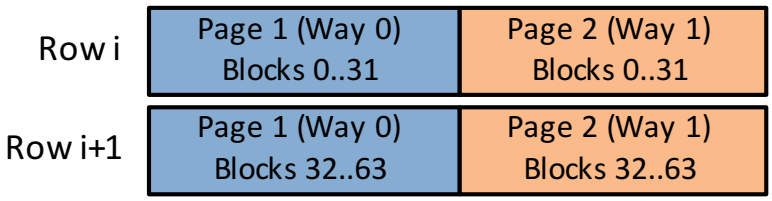


Figure 5.2 – Data layout of two consecutive DRAM rows.

block. To illustrate how data fetch and translation complete together, let us assume an MPU issues a load operation to virtual address VA_i. With direct mapping, both the physical address PA_i and the location of the corresponding DIPTA entry are immediately determined to be in $Vault_i$. A request for both data and translation is issued to $Vault_i$. It may take many cycles to reach the target vault through the memory network, after which the data fetch and translation can be issued independently and in parallel, the former to the vault's DRAM and the latter to its SRAM DIPTA partition. As the per-vault SRAM structure is small, fetching the translation is always faster than fetching the data from DRAM. By the time the cache block arrives, the translation metadata has already been fetched, and checked against the memory request, taking translation off the critical path.

Supporting associativity is not trivial because data from different ways in a set could reside in different DRAM rows, and the actual way is not known a priori. A set-associative DIPTA lookup and VPN/ASID comparison is required prior to data fetch to identify the row where the data resides. Once the row is determined, the page offset is used to access the data within the row. Unfortunately, while such a solution is simple, the serial DIPTA lookup puts the translation back on the critical path, which particularly hurts local accesses and does not scale with the chip capacity.

We address the set-associative DIPTA lookup bottleneck in two ways. First, to perform DRAM row activation in parallel with the DIPTA lookup, we propose to interleave memory pages within DRAM rows. Figure 5.2 illustrates one possible data placement for two 4KB page frames into two 4KB DRAM rows for a 2-way set-associative DIPTA. Because a DRAM row of 4KB cannot store two 4KB page frames, we strip the data across DRAM rows by splitting each page into two parts. Even rows store the first half of each way, whereas odd rows store the second half. The target DRAM row is determined by the highest order bit of the page offset, and the position of the block within that row is determined by the rest of the offset bits. Besides keeping both ways for each block in the same row, the proposed placement also preserves spatial locality by keeping consecutive blocks together. This example can be easily generalized to any associativity, page frame size, or DRAM row size.

While stripping pages allows for a single row activation (the page offset determines the target row), each row now contains multiple ways and the page offset cannot identify which block to access. To avoid waiting for translation, a naive solution would read all the ways from DRAM at once and in parallel with translation, which would waste bandwidth and energy proportional to

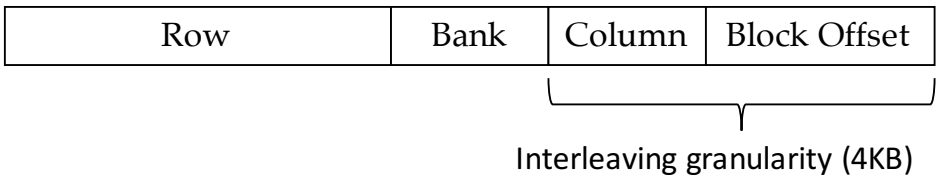


Figure 5.3 – DRAM mapping interleaving policy.

the associativity [96, 147]. To avoid such overheads, we employ lightweight but highly accurate near-data way prediction. This approach eliminates the need to fetch all the ways in parallel, while ensuring translation and data fetch happen independently, yet in a single DRAM access.

We employ a simple way predictor per vault integrated within each memory controller. We design an address-based predictor as they have been shown to achieve high accuracy for pages [96, 33, 143]. Our way predictor exploits both spatial and temporal locality, and is organized as a tagless array of 2^k entries indexed by a k-bit XOR hash of those VPN bits that determine the set within the vault (i.e., the bits that determine the vault are excluded). In the case of a 4-way associative system with 2GB 16-vault chips and 4KB page frames, there are 13 bits that determine the set within each vault. For a way predictor of 32 entries, we construct a 5-bit XOR hash (k=5) for indexing. Each entry encodes the last accessed way in the set with two bits. In this case, the total storage for way prediction is only 4B per vault, and covers 32 sets or 128 local pages in each vault. Because way prediction can be done during the row address strobe (RAS) phase, it is always off the critical path.

The combination of page stripping and way prediction allows for fully overlapping the translation in the common case of a predictor hit. Interleaving also minimizes the misprediction penalty; a second column address strobe (CAS) command to an already opened DRAM row. Moreover, with a distributed predictor, the prediction accuracy in one vault is not affected by accesses to other vaults (unlike TLBs), boosting the spatial and temporal locality within vaults.

Figure 5.3 shows the DRAM mapping interleaving policy we assume; essentially how a physical address maps into DRAM. For a direct-mapped configuration, the DRAM mapping interleaving policy is the same as the widely-used page-based policy, in which pages are split across different banks [158]. The page-based policy performs well for workloads that either stream often or exhibit high locality at the page level. However, this policy can sometimes limit parallelism and create unfairness across different threads as memory controllers prioritize requests to already opened DRAM rows [109]. For a 4-way configuration, as a DRAM page is split across 4 different banks, the DRAM mapping policy is similar to a recently proposed policy which aims to balance locality and parallelism [109]. In this last configuration, there are 16 cache blocks for each of the 4 DRAM pages that occupy the DRAM row. Overall, a myriad of address-mapping schemes exist, none of which is optimal across all workloads. Our mapping schemes are similar to recently proposed techniques, and therefore are expected to perform similarly to them in terms of average latency and bandwidth.

5.2.2 In-Memory DIPTA

The SRAM solution is simple, but requires 4MB (16MB) of SRAM for a 2GB (8GB) memory stack. The area overhead is higher than conventional translation hardware, and grows quickly with the chip capacity, leaving less space for MPUs. Dedicating a small fraction of a vault's DRAM to store DIPTA could completely eliminate the SRAM overhead. However, arbitrarily storing DIPTA in DRAM can make DIPTA lookups unpredictably long and difficult to overlap with the data fetch. Moreover, DIPTA lookups would contend for bandwidth with data fetches.

A recent die-stacked DRAM cache proposal [96] solves the tag-data co-location problem by dedicating the first 64-byte block in each DRAM row to store the tag metadata for the page residing in that row and fetches the tag information and the data block in a single access. Thanks to the limited associativity, the data location is independent of the tag content, hence the tag and read can be overlapped on cache hits [96, 147] by pipelining two separate back-to-back column address strobes (CAS), one for the tag and the other for the data. Unfortunately, such a solution has an impact on the page size. Assuming a 4KB DRAM row, reserving 64B for metadata leaves 4032B for the data, which may be acceptable for hardware DRAM caches, but it is certainly not acceptable for OS pages.

Instead, we propose a novel DRAM data layout illustrated in Figure 5.4, where 63 4KB page frames are stored in 64 consecutive DRAM rows. For simplicity, we assume 4KB DRAM rows, although the solution can be trivially generalized to any page frame and DRAM row size. Because the first 64B in each row are reserved for metadata for all pages residing in that row, the last block of the first page cannot fit and is stored in the subsequent row. In this example, each page spans exactly two DRAM rows, and each DRAM row contains blocks from at most two different pages.

In Figure 5.4, the first block of *Page 0* is placed in the second block of *Row 0*; we denote its position as *Offset 1*, while the first block (*Offset 0*) holds the metadata. The first half of the metadata block contains metadata of the page that ends in the current row (i.e., the page that could not fit in the previous row), whereas the second half contains metadata of the page that starts in the current row. Because *Row 0* contains only one page, the first part of the metadata block is empty (denoted as *X*). The last block (*B63*) of *Page 0* also occupies the first available data slot (*Offset 1*), but in the subsequent row (*Row 1*). The first block of the next page, *Page 1*, occupies the block at *Offset 2* in *Row 1*, whereas the last block occupies the same position in the subsequent row, and so on. *Page 62* starts at the very end of *Row 62*, and occupies the entire *Row 63*. Because no page starts in *Row 63*, the second metadata slot in this row is empty. The layout of *Row 64* is identical to the layout of *Row 0*; rows 0-63 form a cycle. This solution incurs no SRAM overhead and requires dedicating 64B per DRAM row for metadata. The DRAM overhead is $data\_line\_size/DRAM\_row\_size$ and decreases with the DRAM row size. For a 4KB DRAM row the overhead is 1/64 or $\sim$1.5% of DRAM capacity.

The target DRAM row for a given address is computed with minimal logic as: $block\_address/(k-1)$, where k is the number of 64-byte blocks per row, in this example 64. The exact position of the

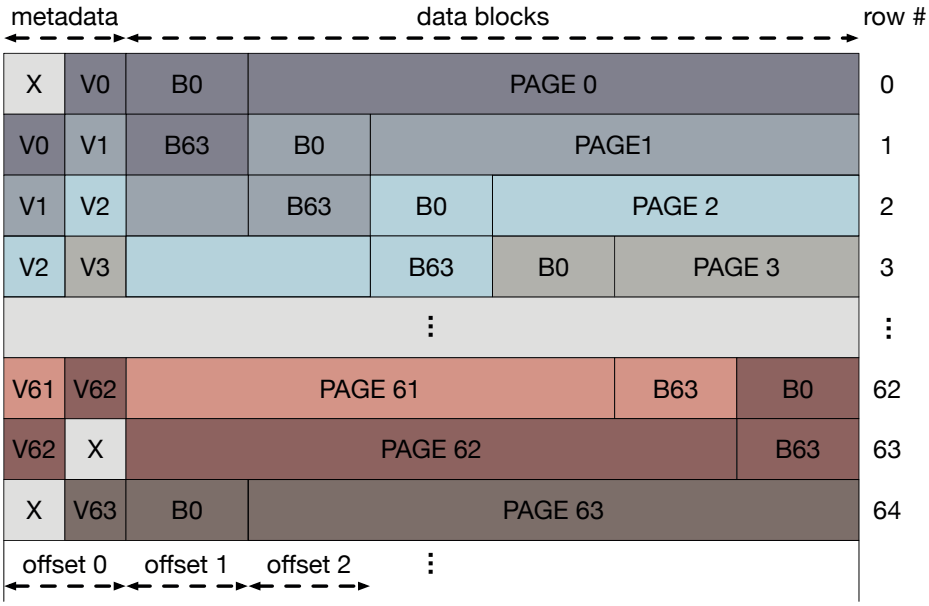


Figure 5.4 – Metadata integration with 4KB pages & rows.

requested data block is given by $block\_address \bmod (k-1) + 1$. Both formulas can be computed by a single and fairly simple hardware unit [147].

Much as in SRAM DIPTA, the set-associative implementation follows the layout in Figure 5.2. The first block in each row contains the metadata belonging to all the ways. The drawback of supporting high associativity in the DRAM-based implementation is that the storage overhead grows with associativity. For example, dedicating one block for metadata in a 2-way associative organization leaves an odd number of blocks for the data to store two ways, wasting a block to ensure symmetry, resulting in a DRAM overhead of $associativity \times data\_line\_size/DRAM\_row\_size$. However, the unused blocks could be used for more functionality (e.g., to store coherence directory entries, or to tag memory [134]).

Co-locating and fetching translations and data from the same DRAM row minimizes bandwidth, latency, and energy overheads, similar to DRAM caches [96, 147]. Translations require $8B \times associativity$ to be transferred (locally) from DRAM to the vault controller. Assuming a 16B internal bus and a 4-way associative DIPTA, the transfer latency overhead is two bus cycles, during which the VPN/ASID comparison is performed. Though the bandwidth overhead of fetching the page table entries can reach up to 50% (as fetching a 64B cache block takes 4 bus cycles), the effective overhead tends to be rather small. The reason is that, due to DRAM timing constraints, the DDRx data bus is often underutilized, even during periods in which many pending requests are queued at the memory controller [51, 91, 92, 125, 150, 151]. This behavior translates into many idle cycles in the data bus between memory transactions, even in cutting-edge DDR4 interfaces [157]. As a result, DIPTA utilizes the otherwise idle data bus cycles, without compromising the application's effective bandwidth. The energy overhead—a second column

address strobe (CAS) command to an already open DRAM row—is negligible as most of the DRAM access energy goes into opening the DRAM row [73, 169]. In contrast, conventional translation incurs significant energy overheads due to more frequent off-chip link traversals and DRAM accesses upon TLB misses; traversing a single off-chip link can be as expensive energy wise as a DRAM access [12].

5.3 Discussion

In this section, we discuss possible concerns and further considerations regarding our translation mechanism.

5.3.1 Page faults

Although infrequent, the system must handle page faults triggered by MPU memory accesses. We choose to interrupt the CPU to run a handler, as MPUs may not be capable of running the OS. Upon a page fault, DIPTA responds to the MPU, notifying it of the fault. The MPU then places a request in a memory-mapped queue indicating the faulting virtual address and the responsible MPU's ID, and interrupts the CPU. After the missing page is brought into the memory, the handler updates the affected DIPTA entry with the new translation information. The DIPTA state can either be mapped in memory, or accessed through custom logic in the memory chips, configurable through a set of memory-mapped registers. Once the fault is serviced, the handler notifies the appropriate MPU, which resumes its execution and retries the faulting address. Such page fault processing is also employed in today's integrated GPUs [166].

5.3.2 TLB shootdowns & flushes

There are many ways of maintaining DIPTA entries coherent upon TLB shootdown and flush operations (initiated by the CPU). Our solution is similar to those used in integrated GPUs [166]: an OS driver monitors any changes on virtual address spaces shared with the MPUs, triggering update operations on DIPTA for the affected entries. Note that the inverted nature of DIPTA eliminates any global coherence activity in the memory network, because updates to DIPTA are fully localized to a single entry in the affected vault. In contrast, conventional translation hardware would extend the coherence domain to include all MPU units, in which case the coherence overhead would scale not only with the number of CPUs [167], but also with the number of memory chips. Moreover, because DIPTA entries include the address space identifier (ASID) bits, requests to flush all entries of a given address space are never received.

5.3.3 Memory-mapped files

Limiting associativity does not prevent a process from mapping files in its address space. On demand, the OS would bring page-size chunks of the file to its page cache [126]. As with memory allocation, the OS would only need to guarantee that the page frames are mapped to the same set as their virtual page counterpart. The OS would then populate the CPU page tables and DIPTA to make the file pages available to the user process.

5.3.4 Synonyms

As with any inverted page table, synonyms are not straightforward to handle [94]. A trivial approach would enforce synonyms to either have the same virtual addresses or to map to the same set [41], and extend each DIPTA entry with extra storage. A more clever approach, inspired by Yoon and Sohi's work [178], would add a small per-vault structure populated by the OS to remap synonym pages to a single leading virtual page, and consequently to a single page frame. Because synonym accesses are infrequent [23, 178, 138], the overhead would be negligible even if the leading virtual address is in a different vault or different chip. In this work, we do not extend DIPTA to support synonyms because we do not expose shared libraries or the kernel address space, which are the sources of synonyms [23, 178].

5.3.5 Cache hierarchy

In case MPUs integrate physical caches, a naive approach would add a TLB to cache frequently-used page table entries, while TLB misses—page walks—would be accelerated by DIPTA. Upon a TLB miss, the page table entry (in DIPTA) and data cache block are accessed in parallel (as part of the normal DIPTA operation), and cached in separate structures; the page table entry and the cache block are cached in the MPU's TLB and cache, respectively. Nevertheless, a more natural design, which also avoids TLBs and TLB shootdowns [167], is to integrate virtual caches. In this approach, MPUs access its cache with virtual addresses, and upon a cache miss, the request is propagated to DIPTA to translate and fetch the corresponding block. Recent practical designs for virtual cache hierarchies would be a perfect fit for DIPTA [138, 178].

5.3.6 CPU and DMA accesses

The conventional CPU address translation remains untouched. The address mapping does not affect the way CPUs or DMAs access memory—it is just an internal mapping that the vault controller uses to map cache blocks into DRAM banks. The controller is aware of the source of the memory requests and hence can treat them accordingly. Virtual addresses from MPUs identify the memory set and use the way predictor to generate a complete physical address. Physical addresses from CPU/DMA have all the bits required to identify the block. Furthermore, as DIPTA is agnostic to the execution units (DIPTA is just aware of memory requests), CPUs

can use DIPTA to eliminate the overhead of page walks for the part of the memory covered by DIPTA. While the focus of this work is on MPUs, it is an interesting direction.

5.3.7 Multi-level memories

Although prior work on memory-side processing assumes a single level [10, 11, 71, 146], memory can be organized as a hierarchy, with a die-stacked cache [149, 168] backed up by planar memory. For hardware-managed caches, DIPTA performs the translation and accesses the page frame speculatively, and in case the page frame is not in the cache, it is fetched from planar memory as part of the standard cache miss operation. Once the page frame is in the appropriate DRAM row, the data is sent back to the MPU. The DIPTA page table entries have to be embedded in both planar and die-stacked DRAM, and move with the page frame. Note that caching a page frame from planar memory into the die-stacked memory does not affect its page table entry. In software-managed hierarchies [149], MPUs rely on the software API for explicit migration of pages into the die-stacked memories, as MPUs cannot access planar memory directly. As part of the page migration operation, the DIPTA page table entries are populated accordingly.

5.3.8 Kernel memory

Our simulation infrastructure captures user-level instructions, although we argue that MPUs will accelerate only user-level not kernel-level code, like all existing custom hardware (e.g., GPU, FPGA). Nevertheless, our technique should be a great fit for the kernel, as Linux's memory usage is almost entirely direct-mapped [126] and memory resident. The tag matching logic of DIPTA would then require the following simple change. As all the processes share the same kernel virtual addresses, the logic needs to ignore the ASID bits. This modification is trivial as Linux partitions the address space into two halves, and hence the tag matching logic just needs to check the virtual address's MSB.

5.3.9 Operating system support

The operating system only needs to guarantee that the virtual page number and the page frame number map to the same memory set. OSs that support virtual caches already provide this capability (e.g., Solaris [41] and MIPS OS [161]). MIPS OS traverses the free list of pages and returns a page frame that resides in the same memory set as the virtual page. Alternatively, one can organize the allocation of memory similarly to Linux's memboot memory allocator. Linux could employ a set-associative array, storing on each set a single bit per way indicating whether the page frame is free or occupied. The virtual page causing the fault could be simply hashed to locate its location in the array; avoiding the linear search for free pages that the memboot memory allocator requires.

5.4 Methodology

Like the recent work on virtual memory [19, 22, 25, 137, 139, 140, 152], we use a combination of trace-driven functional and full-system cycle-accurate simulation.

5.4.1 Performance

This study does not cover the problem of partitioning applications into CPU-side and memory-side execution, or specializing the latter in hardware, or interfacing and synchronizing the two executions [10, 17, 71]. While important, such problems are out of the scope of this paper and would cloud the key address-translation tradeoffs in near-memory processing. To avoid such problems we evaluate the applications running entirely on the memory network.

Full-system simulation for the server workloads listed in Table 5.1 is not practical. Hence, we resort to the CPI models often used in VM research [27, 137, 152] to sketch the performance gains. These prior studies report performance as the reduction in the translation-related cycles per instruction. As CPI components are additive, this metric is valid irrespective of the workload's baseline CPI. We further strengthen this methodology by studying the CPI savings on all memory cycles, not only on translation stalls (as we overlap translation and data fetch operations). Our model thus captures both the translation cycles and data fetch cycles, which together constitute the largest fraction of the total CPI in server workloads [66]. Hence, our results are more representative of the end-to-end benefits of each technique. The CPI is measured by feeding the memory traces into our cycle-accurate simulator.

Furthermore, we evaluate a set of data-structure traversal kernels—ASCYLIB [52]—in full-system cycle-accurate simulation. ASCYLIB contains state-of-the-art multi-threaded hash tables, binary trees, and skip lists. For clarity, we present results for four representative implementations: the Java Hash Table (Hash Table), Fraser Skip List (Skip List), Howley Binary Search Tree (BST Internal), and Natarajan Binary Search Tree (BST External). We choose this specific suite because dynamic data structures are the core of many server workloads (e.g., Memcached's hash table, RocksDB's skip list), and are a great match for near-memory processing [86, 114]. The poor locality and the abundance of pointer chasing in data-structure traversals allow us to stress the translation and way prediction operations.

5.4.2 Traces and Workloads

For the associativity experiments in Sections 5.1 and 5.5.4, we collect long memory traces using *Pin* [124]. For workloads with fine-grained transactions (i.e., Memcached, RocksDB, MySQL, and Cassandra), the traces contain the same number of instructions as the application executes in 60 seconds without Pin. For analytics workloads (i.e., TPC-H, TPC-DS, and Neo4j), we instrument the entire execution. We feed the collected traces into a tool that models a set-associative memory of 8GB, 16GB, and 32GB. To stress the associativity requirements, all the

workloads are tuned to employ all the available physical memory; 8GB of physical memory for the experiments of Section 5.1, and up up to 16GB and 32GB for Section 5.5.4's experiments.

The traces are collected on a dual-socket server CPU (Intel Xeon E5-2680 v3) with 256GB of memory, using the Linux 3.10 kernel and Google's TCMalloc [75]. Address space randomization (ASLR) is enabled in all experiments.

For the performance experiments of Section 5.5, we employ the server traces of 32GB and 64GB, depending on the size of the network. We use 32GB and 64GB for the 4-chip and 16-chip configurations, respectively. For the data-structrue kernels, each workload performs uniformly distributed key lookups on its in-memory data-structure. The datasets range from 16GB to 20GB (depending on the workload) across all network configurations.

Simulation Parameters

We use the Flexus cycle-accurate simulator [172], with detailed core, MMU, memory hierarchy, and interconnect models. Following prior work on near-memory processing, which assumes single-issue in-order cores [10, 71, 146], we model the MPU cores after ARM Cortex A7 [15].

Table 5.3 – System parameters.

MPU logic	**Description**
Cores	Single-issue, in-order, 2GHz
L1-I/D	32KB, 2-way, 64B block, 2-cycle load-to-use
MMU	**Description**
TLB	4KB pages: 64-entry, 4-way associative 2MB pages: 32-entry, 4-way associative 1GB pages: 4-entry, fully associative
STLB	4KB/2MB pages: 1024-entry, 8-way associative
Caches	L4: 2-entry, fully associative [25] L3: 4-entry, fully associative [25] L2: 32-entry, 4-way associative [25]
Memory	**Description**
MPU chip	8GB chips, 8 DRAM layers x 16 vaults
Networks	4, 8, 12, and 16 chips in daisy chain and mesh
DRAM	t_{CK} 1.6ns, t_{RAS} 22.4ns, t_{RCD} 11.2ns t_{CAS} 11.2ns, t_{WR} 14.4ns, t_{RP} 11.2ns
Serial links	2B bidirectional, 10GHz, 30ns per hop [106, 165]
NoC	Mesh, 128-bit links, 3 cycles per hop
SAVAgE	**Description**
TLB	4KB pages: 64-entry, 4-way associative
STLB	4KB pages: 1024-entry, 8-way associative
DIPTA	**Description**
Configuration	SRAM, 4-way associative, 1024-entry WP

We privilege the baseline with a high-end MMU similar to Intel Xeon Haswell [47, 77], with multi-level TLBs and MMU caches [18, 25]. We assume a 4-level hierarchical radix tree page table [93] with 48-bit virtual and physical addresses (as in ARMv8 and x86_64). The MMU supports 4KB, 2MB, and 1GB pages. Note that page table entries are transparently allocated in the L1-D cache. We probe the cache with physical addresses for the baseline and with virtual addresses for DIPTA. We verify that TLB misses never reference a cache-resident block, and therefore virtual and physical caches behave identically.

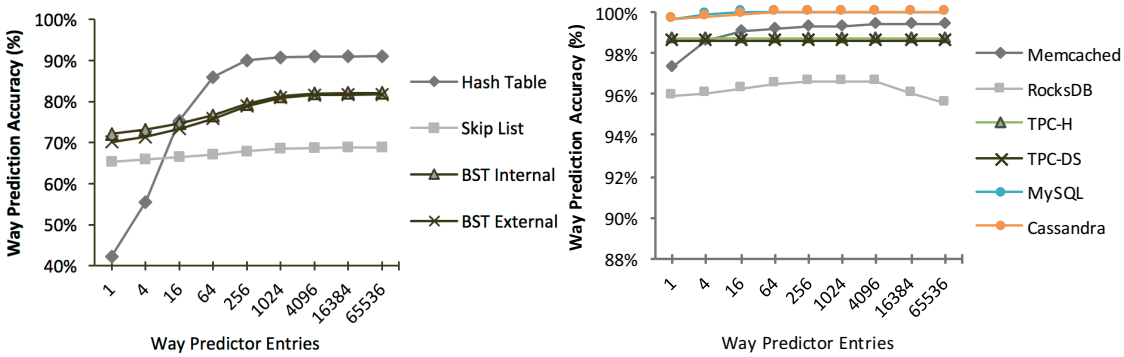
Without loss of generality, we assume the Hybrid Memory Cube organization with eight 8Gb DRAM layers and 16 vaults [128]. We conservatively estimate the die-stacked memory timing parameters from publicly available information and research literature [71]. For SAVAgE, we employ a conventional two-level TLB hierarchy. The DIPTA implementation is 4-way set-associative in SRAM, with a way-predictor of 1024 entries per vault. The DRAM and SRAM implementations of DIPTA provide almost identical results, with the tradeoff being between SRAM area and DRAM capacity. The DRAM implementation has practically no SRAM overhead (except for tiny way predictors) but occupies space in DRAM for translations. The SRAM overhead for 8GB chips is 16MB (partitioned across vaults) for an area of $20mm^2$ in 22nm, corresponding to only 9% of the area of an 8Gb DRAM die (e.g., $226mm^2$ [154]). Its access latency of 8 cycles guarantees that the memory and DIPTA accesses are overlapped. The system parameters are shown in Table 5.3.

5.5 Evaluation

In this section, we perform a quantitative study of the performance of different translation mechanisms across the set of server workloads and a set of microkernels. In the study, we vary the topology and scale of the memory network, as well as the amount of data locality within the memory chips.

5.5.1 Way Prediction Accuracy

To better stress the way predictor, we first study its accuracy on ASCYLIB, which is a worst case scenario due to the limited temporal and spatial locality (e.g., the behavior of Skip Lists is very similar to GUPS). Figure 5.5a shows the way prediction accuracy for an 8-way associative organization as the number of entries increases. With only 1024 entries, the way predictor yields very high accuracy, 69%-91%. This high accuracy comes at tiny storage cost as a single entry requires only 2 bits (for 4 ways), requiring in total only 256B per vault and 4KB per chip of storage overhead. Figure 5.5b shows the accuracy of the server workloads of Table 3.1. The temporal and spatial locality is higher in the server workloads, achieving a way predictor accuracy of 96%-99% with only 64 entries. The modest storage requirement is 16B per vault and 256B per memory chip.



(a) Data-structure kernels (b) Server workloads

Figure 5.5 – Sensitivity study on the accuracy of the way predictor.

Besides leveraging the spatial locality, the way predictor's distributed nature also boosts the temporal locality. Unlike TLBs, which centralize translations, each predictor holds the entries for its share only; the accuracy in one vault is not affected by accesses to others. To overprovision for the worst case, we assume in this work a 1024-entry way predictor per vault.

5.5.2 Where Does Time Go?

Figure 5.6 shows the execution time breakdown and CPI of the data-structure kernels for the conventional address translation with 4KB pages. We perform this experiment for 4- and 16-chip daisy chain topologies to see the impact of the network size. We also control the fraction of local data accesses—data accesses that remain within the memory chip, varying it from 25% to 100%. An application that perfectly partitions its dataset across the memory chips exhibits 100% data locality. As the figure shows, improving data locality reduces the CPI due to fewer high-latency cross-chip data accesses, but also increases the relative contribution of address translation to the total execution time—measuring up to 80% with 100% locality. This behavior is expected as the locality cannot be enforced for the page table entries, which are arbitrarily distributed across the memory chips. Hence, as locality increases, data accesses become cheaper, while the latency overhead of page walks remains the same.

The Hash Table and Skip List kernels exhibit the highest translation time as the locality in the data structures is significantly low. The tree data structures (i.e., BST Internal/External) show slightly better TLB locality compared to the Hash Table and Skip List kernels. This locality arises due to the reuse of the top tree levels across different data-structure traversals. Such locality is not present in the hash table and skip list data structures, where probes for different keys will likely access distinct pages. Additionally, given a data locality point, the translation overhead significantly increases with the network size for all the kernels, as expected.

Similarly, Figure 5.7 shows the execution time breakdown and CPI of the server workloads. As showed in the way prediction results, the temporal and spatial locality of the server workloads is

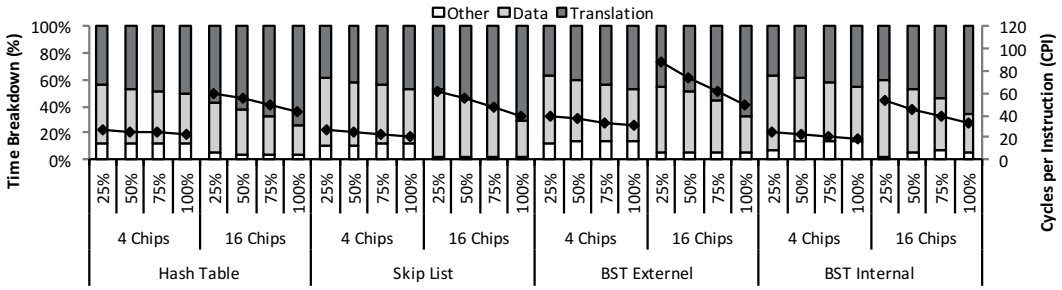


Figure 5.6 – Time breakdown and CPI for different data locality and network topologies.

higher than in the data-structure kernels, and hence the contribution of address translation to the execution time in smaller—measuring up to 42% with 100% locality. Memcached and RocksDB show similar trends to that of the data-structure kernels, the CPI reduces as the locality increases and the fraction of time spent in translation increases. In contrast, TPC-H and TPC-DS show little sensitivity to the fraction of data locality. The reason is that these data analytics benchmarks run on MonetDB, a state-of-the-art column-oriented RDBMS, which exhibit significant spatial cache locality [30]. Hence, the overhead of bringing a cache block from a remote memory chip is amortized with several accesses to the same cache block. As there is little sensitivity to the fraction of data locality (and the locality does not affect the location of the page table entries), the time breakdown remains invariant across all the locality points and memory chip counts. MySQL exhibits poor cache locality, as the CPI dramatically drops as the data locality increases. In fact, the cycles spent in data fetches greatly dominate the execution time. The reason is that MySQL organizes tables (which are comprised of tuples) in database pages stored in its buffer pool. The database pages are usually smaller than the base page size (e.g., 2KB). On every tuple access, the buffer pool page header and tuple slot index are consulted [156], which turn in high translation but low data locality. Last, Cassandra also exhibits poor cache locality, though lower than MySQL, due to compulsory cache misses as Cassandra manages large objects of 1KB. However, Cassandra also maintains a cache, as a dynamic data structure, of recently-referenced keys, which gets probed fairly frequently as the workload models a skewed key distribution. Accesses to this cache increase the contribution of translation to the total execution time.

Overall, both the data-structure kernels and server workloads exhibit significant translation time, which usually increases with the fraction of data locality and memory chip count. Therefore, reducing the cycles spent in translation has the potential to bring great performance benefits to all the workloads.

5.5.3 Performance

Figure 5.8 shows the speedups over 4KB pages with 1GB pages, SAVAgE, and DIPTA, for 4- and 16-chip mesh and daisy chain topologies, for the data-structure kernels. For clarity, we present the results only for the extreme locality points: 25% and 100%; As expected, the speedups grow as locality increases and with the size of the memory network. Furthermore, the speedups on the

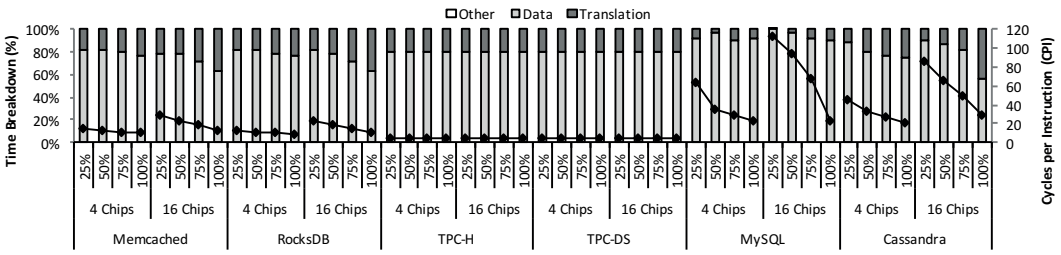
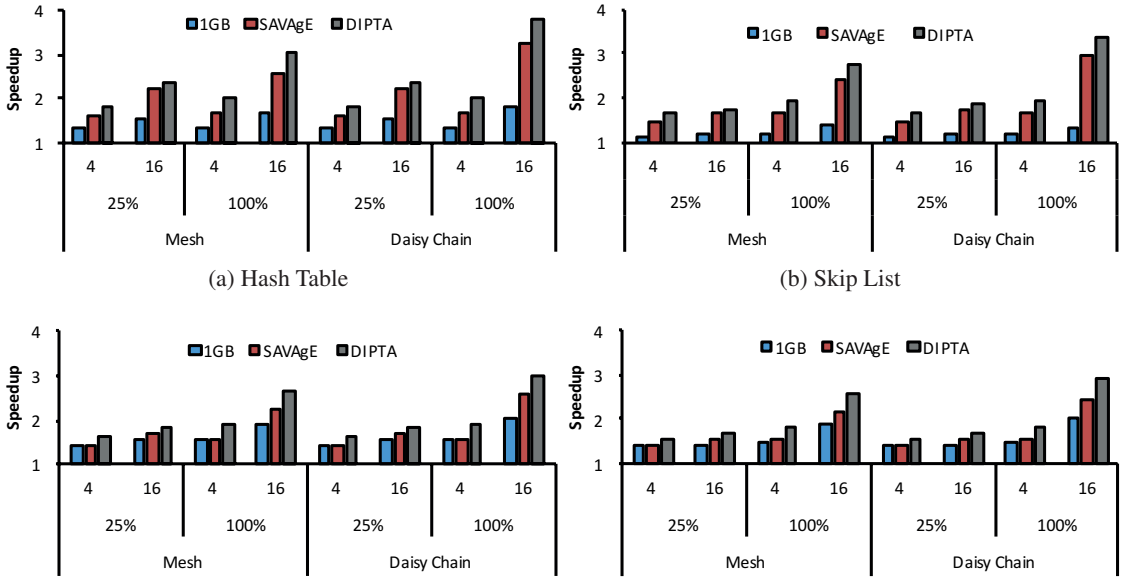


Figure 5.7 – Time breakdown and CPI for different data locality and network topologies.

daisy chain are more pronounced as the average hop count is larger than in the mesh. In the Hash Table and Skip List kernels, which exhibit the lowest TLB locality, translation cycles account for the largest fraction of the execution time among all benchmarks, which is reflected on the speedups. In contrast, the tree-based data structures exhibit better data and TLB locality, and consequently, lower speedups. DIPTA's speedups over 4KB pages range between 1.58× and 3.81×, with an average speedup of 2.11×, whereas the baseline with 1GB pages improves the performance from 1.20× to 2.02×, with an average speedup of 1.45×. Additionally, SAVAgE improves the performance from 1.40× to 3.20×, with an average speedup of 1.85×. DIPTA's speedups over 1GB pages range between 1.14× and 2.13×, with an average of 1.44×. DIPTA's speedups over SAVAgE range between 1.05× and 1.18×, with an average of 1.14×. Although omitted for clarity, we also compare DIPTA against the baseline with 2MB pages, which performs only slightly better than 4KB pages, and always worse than 1GB pages. As shown in the results, DIPTA significantly outperforms conventional address translation hardware. Note that DIPTA virtually eliminates the overhead of page walks, and hence our results are equal to an ideal MMU with zero overhead for page walks.

Figure 5.9 shows the speedups over 4KB pages with 2MB pages, SAVAgE, and DIPTA, for 4- and 16-chip mesh and daisy chain topologies, for the server workloads. Similarly to the data-structure kernels, the speedups usually grow as locality increases and with the size of the memory network. This trend does not hold for TPC-H and TPC-DS for the reasons explained in the previous section. The speedups on MySQL are fairly modest as the data cycles (and not the translation cycles) dominate the execution time. DIPTA's speedups over 4KB pages range between 1.02× and 1.75×, with an average speedup of 1.25×, whereas the baseline with 2MB pages improves the performance from 1.01× to 1.41×, with an average speedup of 1.06×. Additionally, SAVAgE improves the performance from 1.02× to 1.21×, with an average speedup of 1.06×. DIPTA's speedups over 2MB pages range between 1.01× and 1.23×, with an average of 1.44×. DIPTA's speedups over SAVAgE range between 1.007× and 1.07×, with an average of 1.03×. For clarity, we omit the results with 1GB pages, which performs better than 4KB pages, but always worse than 2MB pages. 1GB pages performs worse than 2MB pages because the number of entries in the MMU for 1GB pages is significantly limited; there are only four entries. As for the data-structure kernels, DIPTA clearly outperforms conventional translation hardware and SAVAgE, while delivering the performance of an ideal MMU with zero overhead for page walks.



(a) Hash Table

(b) Skip List

(c) BST External

(d) BST Internal

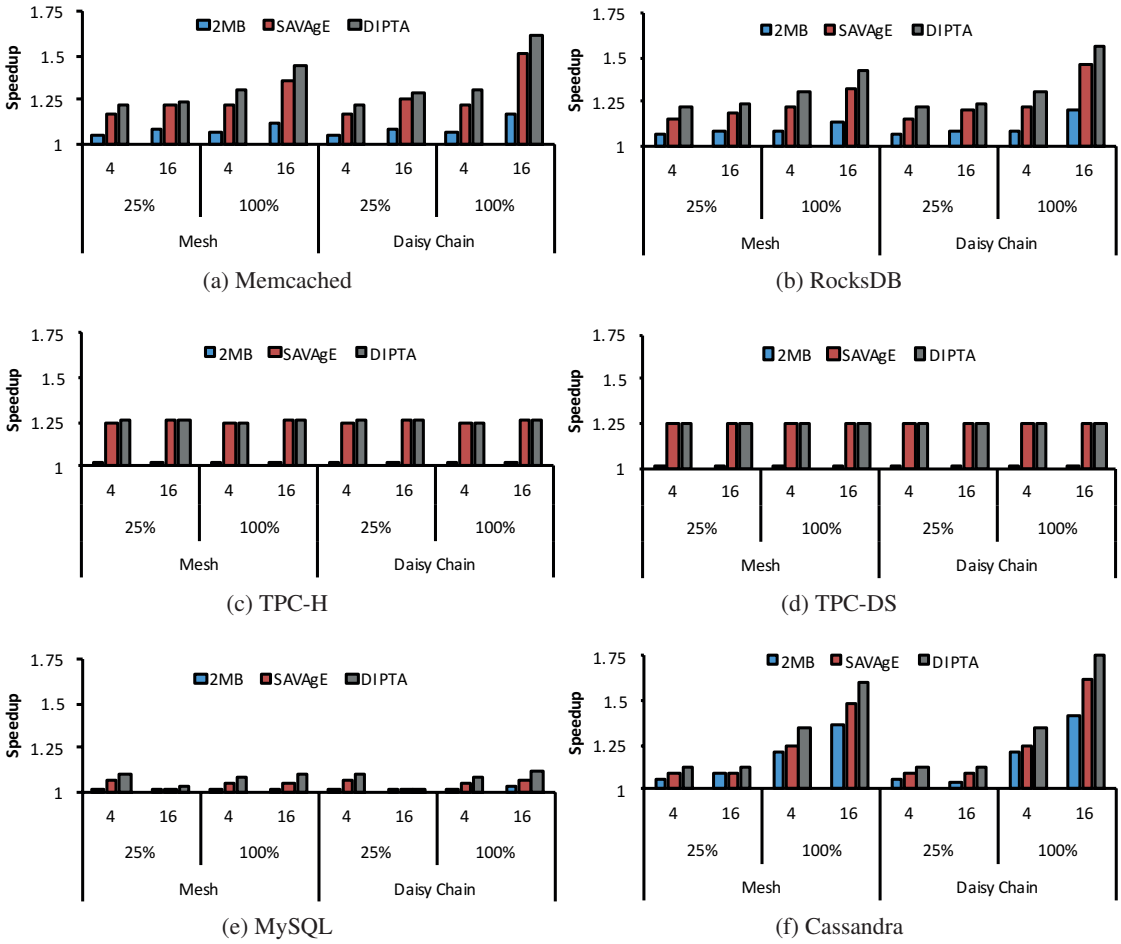
Figure 5.8 – Speedup results over 4KB pages for the data-structure kernels.

Overall, DIPTA is effective at completely eliminating the overhead of page walks across the data-structure kernels and server workloads. DIPTA and SAVAgE clearly outperform conventional translation hardware (i.e., 4KB, 2MB, and 1GB pages) due to avoiding cross-chip page walk operations. Additionally, DIPTA significantly outperforms SAVAgE for the workloads that exhibit poor translation locality (e.g., Hash Table, Memcached), as a page walk within the vault, which requires a memory access, is still required before fetching the data.

5.5.4 Comparison with Other Proposals

Two recent proposals on address translation for in-memory workloads are direct segments (DS) [22] and redundant memory mappings (RMM) [69]. These approaches aim to increase the reach of the TLB by exploiting the abundant contiguity available in the virtual address space by mapping one (for DS) or a few (for RMM) virtual segments to contiguous page frames. As the reach of the TLB increases, the frequency of page walks decreases.

A comparison of DIPTA with DS and RMM on ASCYLIB is trivial, as these microkernels have a very simple memory layout where most of the data is mapped to a single virtual segment. Hence, we perform the more challenging comparison of these techniques on our set of server workloads by analyzing the maximum contiguity available in their virtual address space. We employ Linux's *pmap* tool to periodically scan their memory structure. The results are presented in Table 5.4. *Total segments* represents the total number of virtual segments. *99% coverage* indicates the number of virtual segments required to cover 99% of the physical address space. *Largest segment* shows the fraction of the physical address space covered with the largest virtual segment. *Largest*



(a) Memcached (b) RocksDB

(c) TPC-H (d) TPC-DS

(e) MySQL (f) Cassandra

Figure 5.9 – Speedup results over 4KB pages for the server workloads.

32 segments shows the fraction of the physical space covered with the largest 32 segments. Note that these results represent an ideal case for DS and RMM. We employ datasets of 8GB, 16GB, and 32GB. For Neo4j, we use two graphs of 8GB [4] and 16GB [7], respectively.

Table 5.4 showcases several key points. First, for some applications, such as MySQL and Memcached, a single large segment covers most of the physical memory, and therefore DS would eliminate most TLB misses (page walks) [22]. Nevertheless, other applications, such as RocksDB and MonetDB (running TPC-H and TPC-DS), exhibit a large number of segments, and hence would expose the majority of the TLB misses, and hence page walk cycles.

Second, for most applications, the total number of segments and the number of segments needed for 99% coverage are much higher than what the RMM work assumes. On average, even for the small 8GB dataset, the total number of segments is 4× higher, and the number of segments for 99% coverage is almost an order of magnitude higher than the requirements for the applications evaluated in [69]. The total number of segments places a burden on the capacity of the TLB]. For

Table 5.4 – Analysis of virtual segments as dataset scales

	Total segments			99% coverage			Largest segment			Largest 32 segments		
	8GB	16GB	32GB	8GB	16GB	32GB	8GB	16GB	32GB	8GB	16GB	32GB
RocksDB	210	370	690	160	320	640	0.62%	0.31%	0.16%	20%	9.9%	5.0%
TPC-H	280	280	290	45	45	52	24%	24%	23%	94%	94%	95%
TPC-DS	420	400	400	170	170	160	9%	3.3%	3.3%	61%	50%	50%
Cassandra	390	410	520	25	33	130	59%	33%	20%	99%	99%	68%
Neo4j	200	890	—	30	210	—	59%	40%	—	100%	51%	—
MySQL	150	150	150	2	2	2	97%	97%	97%	100%	100%	100%
Memcached	52	52	52	1	1	1	100%	100%	100%	100%	100%	100%

instance, Memcached, which exhibits a very simple memory layout, requires a TLB of 32 entries to remove almost all the TLB misses, although there is a single segment that covers almost 100% of the memory. The reason is that accesses to other segments evict the largest segment's entry. Hence, Table 5.4's last column represents the best-case 32-entry TLB coverage for each workload. The TLB used in RMM contains 32 entries and is a fully associative structure, because segment sizes vary, making the standard indexing for set-associative structures hard. The area/energy requirements of this fully associative structure alone could dwarf the area/energy footprint of simple MPUs on the low-power logic die [10, 71, 146].

Third, although there could be hundreds of segments, the associativity requirements for Section 5.1's 8GB dataset indicate that associativity can be reduced to a small number. The reason is that although segments are not fully contiguous, the OS tends to cluster the segments, and therefore nearby segments do not conflict with each other.

Fourth, some applications, such as RocksDB and Cassandra, exhibit an increase in the number of segments as the dataset grows, increasing the pressure in both the RMM TLB and the rest of RMM structures. For DIPTA, we measure the sensitivity of page conflicts to associativity as dataset scales, employing Section 5.1's methodology and tuning the workloads to utilize 16GB and 32GB. Identically to the 8GB case, conflicts drop more sharply between direct-mapped and 2-way associativity, whereas 8-way associativity practically removes all page conflicts. In all cases—8GB, 16GB, and 32GB—8 ways make the page conflict overhead less than 0.1% of a memory access in the worst case. The reason associativity requirements do not increase is that the OS clusters segments around few places (e.g., heap and mmap areas). The increase in nearby segments does not increase the conflicts. In other words, the number of conflicts is more closely related to the number of clustered areas than to the number of segments. Note that our experiments privilege DS and RMMs as we employ TCMalloc's memory allocator, which coalesces segments when possible. For instance, employing Glibc's memory allocator generates more than 800 segments for Memcached [138], while we only require a few tens of them (as also corroborated by prior work [22, 107]).

Fifth, RMM replaces the conventional demand paging policy for eager paging to improve contiguity in the physical memory. Additionally, the OS has to manage virtual memory at a variable-sized granularity, which may create the external fragmentation problem that plagued the first segment-based VM designs [55]. In contrast, DIPTA makes no disruptive changes to the OS

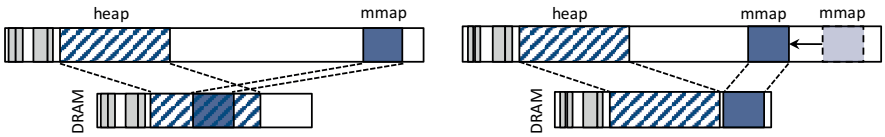


Figure 5.10 – Example of removing page conflicts in a direct-mapped configuration.

operation as it employs conventional page sizes and the default demand-paging policy.

Last, these aforesaid techniques aim to increase the reach of conventional TLBs, whereas DIPTA aims to eliminate page walk overhead, and hence the TLB miss penalty. Therefore, DS and RMM and DIPTA and orthogonal techniques.

5.6 Removing Conflicts in Software

Organizing a part of virtual memory as set-associative may introduce page conflicts. In-memory systems incur no capacity misses, and the conflicts there arise because the OS is unaware of the set-associative organization. Fortunately, we could avoid such conflicts since the virtual address space is huge (i.e., 48 bits in x86_64 [126]), and hence there is high flexibility to move segments around so that they do not cause conflicts under a set-associative mapping.

Although relocating already populated virtual segments can be difficult (e.g., pointers), segments can easily be aligned at initialization time. The OS could align any segments known at compile time (e.g., *.text*, *.rw*) when the process starts. When the memory allocator initializes the heap or the mmap segments for the first time, the OS could align them to start at non-conflicting locations in the virtual space. Note that this alignment does not necessarily require contiguity. Two segments can be placed far away from each other in the virtual space and yet be contiguous or non-conflicting in the physical space. If we want to allocate segment B next to the already allocated segment A in the physical address space, we can choose the starting virtual address for segment B using simple math that only factors in the associativity, total memory capacity, and the location of segment A. Figure 5.10 shows an example where the heap and mmap segments conflict for a direct-mapped configuration, and how a simple relocation can remove the conflict. Hence, a significant fraction of conflicts can be eliminated once the OS understands the set-associative structure of memory, which is a direction that we plan to study in the future.

5.7 DIPTA Summary

This chapter introduced the Distributed Inverted Page Table (DIPTA), a new translation mechanism that completely eliminates the overhead of page walks for in-memory workloads. DIPTA builds on the observation that the associativity of VM can be virtually eliminated for in-memory workloads. Limiting the associativity to a few ways allows for data fetch and translation operations to be performed independently, breaking the traditional translate-then-fetch serialization.

Furthermore, we place the page table entries next to the data in the form of an inverted page table, either in SRAM or embedded in DRAM, so that the data fetch and translation operations are completely overlapped. Hence, achieving the performance of an ideal MMU with zero overhead for page walks.

6 Related work

6.1 Near-Memory Processing

Several efforts in the 1990s and early 2000s examined integrating DRAM and logic in the same chip. EXECUBE [115], the first processing-in-memory (PIM) device, integrated SIMD/MIMD cores and DRAM in the same chip. IRAM [118] advocated for dedicating the on-chip transistors to DRAM in future billion-transistor chips. Active Pages [134] proposed a protocol to orchestrate the execution of code on reconfigurable logic next DRAM. FlexRAM [104] and DIVA [59] considered an array of microcontrollers next to DRAM, controlled by a host. However, these ideas never became mainstream due to technological challenges, as logic and DRAM benefit from significantly different process technologies, resulting in either sub-optimal logic or large density gaps with respect to discrete logic and memory chips.

Recent advancements in three-dimensional integrated circuits and packaging technologies have enabled memory vendors to vertically stack multiple DRAM dies on top of a logic chip within the same package [127, 162]. Each die can be implemented in a different process technology, while a plurality of Through Silicon Vias (TSVs) provide a low-latency and high-bandwidth interface between the dies. Leveraging this technology, several domain-specific architectures have emerged. NDC [146], Tesseract [10], and NDP [71] consider a network of MPU-capable memory chips populated with micro-controllers. Neurocube [111] proposes the use of programmable custom logic to accelerate neural networks. TETRIS [72] also tackles neural networks but focuses on leveraging the proximity to data to devote most of the area for processing (minimizing SRAM buffers). A few works aim to exploit the internal bandwidth of 3D memories to accelerate database operators, namely the scan operator with GPUs [145] and custom ASICs [176], and the join operator with microcontrollers [130].

6.2 Increasing TLB reach

Direct segments [22] allows for an efficient mapping between a single virtual segment mapped contiguously in physical memory. Direct segments reduces much of the pressure on TLBs for applications that allocate most of their data in a single virtual segment. However, direct segments imposes significant restrictions that limit its flexibility. First, page frames within a virtual segment cannot be swapped out, performing poorly for out-of-memory scenarios. Second, direct segments only maps a single segment, exposing most of the TLB misses for applications with complex virtual memory layouts. Third, a direct segment has to be mapped contiguously in physical memory, making the system prone to fragmentation issues.

Karakostas et al. [107] propose a fully-associative range-based TLB alongside the conventional secondary TLB, and a page table to transparently exploit the available contiguity in the virtual and physical address spaces. The technique suffers from similar limitations as direct segments; except for transparency and its support for multiple segments. The system is prone to fragmentation issues especially after the traditional Linux lazy memory allocation system is replaced by eager paging. Additionally, virtual memory is managed at a variable-sized granularity, which complicates the conventional OS facilities which assume the OS base page size.

Several studies exploited the contiguity generated by the buddy allocator and the memory compactor. CoLT [140], Clustered TLBs [139], and Sub-blocked TLBs [160] group multiple PTEs into a single TLB entry. However, packing a small number of translations per entry provides a one-time improvement, leaving the TLB reach still limited for large working sets. Huge pages generated using Transparent Huge Pages [98] and libhugetlbfs [148] increase the TLB reach by mapping very large regions to a single entry. Nevertheless, the effectiveness of huge pages is limited by the size-aligned requirement, and hence the OS can only allocate them when the available memory is size-aligned and contiguous [160]. Furthermore, if not fully utilized, huge pages significantly increase the amount of I/O traffic and the page fault service time. Additionally, huge pages are only beneficial if there is locality within the page, and may suffer in cases where locality is limited, such as pointer-intensive code.

6.3 Reducing TLB penalty

Since resolving a TLB miss can be a long-latency operation, several processor architectures integrate software or hardware structures to cache page table entries [18, 25, 89, 159]. UltraSPARC employs software page table caches such as TSBs [159], which the MMU probes before walking the page table. Page Walk Caches are dedicated caches commonly found in today's MMUs employed to cache intermediate levels of multi-level page tables, reducing the latency of page walks [18, 25]. Last, almost all mainstream commercial processors allocate page table entries in the data caches to minimize page walks referencing memory [89].

Prior work proposed prefetching page table entries into the TLBs in advance of their use to

hide the TLB miss penalty [28, 102, 152]. Bhattacharjee et al. [28] showed redundancy in TLB misses across CMP cores in multi-threaded workloads and proposed inter-core prefetching. Past work [102, 152] evaluates the efficacy of prefetchers initially proposed for data caches for prefetching page table entries into the TLB. SpecTLB [19] leverages the predictability of reservation-based physical memory allocators to speculatively assume accesses to large pages on not yet promoted pages.

6.4 Unified Virtual Memory

The most recent integrated GPUs [174] are fully complaint with the HSA specification and adopt a unified virtual memory. Integrated GPUs reside in the CPU chip and employ a centralized I/O memory management unit (IOMMU), recently introduced in commercial CPUs to allow devices to handle virtual addresses. These IOMMU integrate I/O translation Look-aside buffers (IOTLBs) and dedicated logic to walk the page table, providing support for address translation. Recent academic work [141, 144] has proposed custom TLBs and page walking logic to sustain the high translation throughput required by GPUs. Both approaches propose post-coalester TLBs. Pichai et al. employ a highly-ported TLB and a smart page walk scheduling algorithm [141]. Power et al. proposes a highly-threaded page walker to sustain bursts of TLB misses [144].

Recent integrated FPGAs have a basic form of address translation support. The recent prototype of Intel-Altera Heterogeneous Architecture Research Platform (HARP) employs a static 1024-entry TLB with 2MB pages to support virtual address for user-defined logic blocks [153]. Such a static TLB approach—there is no page walker—requires pinning pages in memory and generating large pages, while TLB refills require running an expensive kernel driver in the CPU. Spending a significant fraction of the execution time in page walks when the memory is larger than the reach of the TLB. Alternatively, FPGAs (or ASICs) could employ the IOMMU of the CPU to walk the page table; much like the integrated GPUs mentioned above. While this approach removes the overhead of running the kernel driver and eliminates the need for a custom API, page walks would still be costly for large memory systems.

The current research wave on MPUs have payed little attention to address translation [10, 71, 72, 130, 145, 146, 176]. TRETIS [72] does not disclose details on their programming model. JAFAR [176], Tesseract [10], and custom MPUs for databases [130] assume a unified address space with physically addressed MPUs. Although a unified address space avoids data replication, memory allocation requires a custom API. Custom APIs make the composability of source code more difficult, as memory allocation is not a standard C or C++ allocation (e.g., new, malloc, mmap). Furthermore, physical addressing requires pinning pages in memory, while CPUs and MPUs cannot share pointer-based data structures. Both NDC [146] and recent work integrating GPUs in 3D memories [145] assume an HSA-complaint unified virtual memory, although details on how translation is supported are not disclosed. NDP [71] also supports a unified virtual memory, with a similar implementation to that of Intel-Altera HARP. Each MPU core integrates a static 16-entry TLB with 2MB pages. A TLB miss triggers an OS kernel driver in the CPU to

resolve and refill the TLB, performing poorly when MPUs access a large memory. Alternatively, NDP could employ an IOMMU on the CPU to walk the page table and resolve TLB misses. However, as discussed earlier, misses in the IOTLB trigger page walks that are an order of magnitude higher than in CPU cores. Additionally, IOMMUs and MPUs are located in different chips, creating frequency cross-chip traffic which can account for hundreds of nanoseconds, dramatically increasing the overhead of page walks.

6.5 Page Tables

A page table stores translation, protection, attributes, and status information for one or several virtual address spaces [88, 94]. Page tables generally present the following three organizations: linear, hierarchical, and inverted. A linear page table conceptually stores all page table entries for a process in a single array. The array is indexed by the virtual page number (VPN). As they cover the whole address space, these page tables tend to be pretty large, and therefore usually reside in the virtual address space. Page faults populate the page table on demand (e.g., MIPS R4000 [103], DEC VAX-11 [120]). Hierarchical page tables store page table entries in a multilevel radix tree, with each level indexed using fixed address fields in the virtual page number. The leaf nodes store the page table entries while the intermediate node store pointers to the next level (e.g., SPARC's MMU [171], Intel Haswell [89]). Inverted page tables comprise one page table entry per physical slot in memory. Since the page frame of a virtual address is not known in advance, either an associative search [67] or a hash table is required. The most common implementation—a hash table—heavily differs from architecture to architecture. HP PA-RISC employs a single-level hash table, whereas IBM RS/6000 introduces a hash anchor table to reduce the number of conflicts at the expense of an extra memory reference. These hash tables use chaining to handle page conflicts. PowerPC 601's inverted page table packs several page table entries within a hash table's bucket, linearly searching the bucket and applying a second hash function if the search is not successful. SAVAgE and DIPTA are examples of inverted page tables.

6.6 Reducing VM Associativity

We are not the first to exploit reducing the associativity of VM. Several degrees of page coloring—fixing a few bits from the virtual-to-physical map—were proposed in the past. The MIPS R6000 used page coloring coupled with a small TLB to index the cache under tight latency constraints [161]. Page coloring has also been used for virtually-indexed physically-tagged caches [42] as an alternative to large cache associativities [76] or page sizes [99]. Alan Jay Smith [155] advocated the usage of set-associative mappings for main memory—much like another cache level—to simplify page placement and replacement.

6.7 Virtual Caches

Virtual caches are attractive as they avoid the need for address translation; addresses are only translated upon cache misses. Unfortunately, synonyms in virtual caches create multiple cache blocks for the same physical block, creating an inconsistency problem if a virtual cache block is updated (as the other virtual cache blocks will contain stale data).

There is a body of work on implementing virtual caches, which is summarized by Cekleov and Dubois [37, 38]. Software and hardware techniques have been proposed in order to deal with virtual address synonyms. On the software side, forcing shared data to reside at virtual addresses that align in the cache [41], software consistency protocols [39], flushing caches on context switches (Intel i860), lazy cache block flushing of synonyms [173], segment-level sharing [80], or single address space operating systems [40, 116] constitute the most significant approaches. On the hardware side, dual-tag stores for finding reverse translations [74], and back-pointers in L2 physical caches for finding synonyms in L1 virtual caches [170] have been proposed.

Two recent proposals provide practical implementations of virtual cache hierarchies [138, 178]. Yoon and Sohi [178] remap synonym pages of the same page frame to a leading virtual page. A table placed before the L1 access, detects and remaps the virtual address before the L1 virtual cache access. Park et al. [138] employs a synonym detector mechanism implemented as a bloom filter populated by the operating system. For non-synonym accesses, virtual addresses are employed throughout the hierarchy, while uncommon synonym addresses are translated before accessing the L1 cache. Recently, Opportunistic Virtual Caching (OVC) [23] reduces the energy of L1 references by caching safe cache blocks (e.g., no read-write synonyms) with virtual address, to avoid accessing the energy-hungry TLBs.

6.8 DRAM Caches

Prior work has shown that DRAM caches are a promising approach to bridge the latency gap between processor and memory, and to break the memory bandwidth wall [96, 97, 122, 123, 147]. Most of the approaches considered DRAM caches as a large hardware-managed cache, targeting maximizing hit rates [96, 97] and minimizing the off-chip bandwidth consumption [97, 122, 123, 147]. There are also several studies that optimize for the latency of DRAM caches. One approach is two employ a small SRAM structure to avoid exposing the tag look-up latency in case of a miss [97, 123, 147]. Additionally, to avoid the large area overhead of the tags, several approaches co-locate data and tags in the same DRAM row [96, 123, 147]. Furthermore, other designs optimize the layout of data and tags within the DRAM row to fetch both in a single DRAM access [96, 147]. Alternatively, software-managed DRAM caches either in industry products [149] and academia [135] have emerged. Software caches either expose a software API for explicit migration of pages between the cache and memory [149], or let the OS dynamically migrate the pages [135].

Essentially, SAVAgE turns memory into a virtually addressed software-managed cache. In contrast, DIPTA turns memory into a virtually addressed hardware-managed cache. In DIPTA, the cache's tag array is implemented as a distributed inverted page table and is accessed in parallel with the data array [96, 147]. Indeed, one can think of memory as a cache, where the data array is the set of page frames, and the tag array is the page table.

7 Future Work

This chapter presents interesting directions for future work.

7.1 Memory Allocation

The page placement flexibility provided by full associativity can be a burden for memory allocators (e.g., buddy allocator in Linux [126]). Limiting the VM associativity simplifies the memory allocator's job, which now has to select a page frame from a smaller pool of pages. Thus, greatly simplifying the data structures and algorithms used by memory allocators to track the free/occupied space and making the allocation/replacement process faster [8]. We believe that an interesting direction at the OS level is to understand the implications of a set-associative VM for the data structures and algorithms employed in memory allocation.

7.2 Minor page faults

In this work, we conservatively assume that page conflicts always generate a major page fault (i.e., a page fault that involves accessing secondary storage). Alternatively, and inspired by direct-mapped hardware cache designs [100], SAVAgE and DIPTA could employ a small fully associative software victim cache to store recently evicted pages. Furthermore, we could virtualize the victim cache in the data structures that the OS uses for memory allocation. Essentially, we could adopt a best-effort set-associative VM system, in which the OS tries to map virtual pages and page frames into the same memory set. If that is not possible, virtual pages are mapped to any page frame. DIPTA and SAVAgE will speculatively access the memory, and upon a page fault, the page fault handler will determine whether the page is not mapped anywhere in the memory (i.e., a major page fault) or just not mapped in the appropriate memory set (i.e., a minor page fault). Reducing the overhead of page conflicts in a set-associative VM has the potential to further reduce the associativity requirements and improve the overall performance. We believe that investigating directions for reducing the overhead of resolving page conflicts is interesting.

7.3 Non Volatile Memory (NVM)

Next-generation of emerging byte-addressable NVM technologies such as Micron's 3D XPoint [1] have the potential to increase the memory density while delivering much faster access times than high-performance secondary storage (i.e., SSDs). Consequently, NVM technologies are expected to employ the virtual memory subsystem instead of the block-based Virtual File System layer (VFS) [105]. In such systems, the number of ways provided by a fully associative VM will increase by at least an order of magnitude, widening the gap between the provided and the required VM associativity. At the same time, larger memory densities will put more pressure on the TLB, as the reach of contemporary TLBs will capture a smaller fraction of the memory, likely increasing the frequency of page walks. We believe that in such systems—an order of magnitude more associativity and more memory capacity to be covered by TLBs—reducing the associativity of VM has the potential to uncover large insights and optimization opportunities for address translation.

7.4 SAVAgE and DIPTA for CPUs

Though this thesis focuses on computer systems with MPU-capable memory chips, CPUs could also employ SAVAgE or DIPTA for systems where page walks are particularly expensive. For instance, in server systems that integrate TBs of memory and multiple sockets such as the HP DragonHawk [84], IBM Power9 [31], Oracle SPARC M7 [133], or Huawei KunLun [133]. SAVAgE and DIPTA are near-memory structures that are independent of any control flow; these structures just observe requests. However, CPUs integrate deep and large cache hierarchies, complicating the task of deciding when to employ SAVAgE or DIPTA, instead of the conventional page walker. The reason is that there could be page table entries and data cached along the cache hierarchy, which could be resolved faster with the conventional translation structures. In contrast, MPUs integrate shallow (i.e., single level) and small cache hierarchies that are well within the reach of contemporary TLBs, making every TLB miss also a cache miss. Hence, in this case, all the page walks are resolved by SAVAgE or DIPTA. Nevertheless, following recent work that shows that almost all (i.e., 98%) of page walks that access memory, the data pointed by the translation is also in memory [26], a plausible approach is to employ SAVAgE or DIPTA for page walk operations (of the conventional page walker) that miss in the last-level cache (LLC). In other words, translations and data that exhibit reuse and stay in the CPU caches are accessed with the conventional translation mechanisms, whereas the rest are handled by SAVAgE or DIPTA.

7.5 Different workloads and design environments

This thesis focuses on server systems and workloads. A different and interesting domain which is becoming increasingly popular is edge computing: the computation performed in handheld or IoT devices. These systems are likely to stress a set-associative virtual memory system as they integrate much lower memory capacities (e.g., a few GBs at most) and exhibit a very

dynamic execution environment, both in terms of the number of processes and virtual memory layouts [58]. Furthermore, as these devices are usually managed by the user, they are more susceptible to malicious applications that could exploit the associativity of virtual memory to create artificial page faults, and degrade the performance of the system. Investigating the associativity requirements of such environments is definitively an interesting future direction.

8 Conclusion

In light of computing demands growing at a pace comparable to Moore's law, the end of Dennard and slowdown of silicon density scaling have pushed system designers towards specialized architectures to boost computing efficiency. The proliferation of heterogeneous computing platforms has to be followed by an efficient and simple programming model to ensure widespread adoption. Industry initiatives such as the Heterogeneous System Architecture (HSA) Foundation are proposing to unify virtual memory (VM) between CPUs and any other computing element in the system for an efficient and easy to use programming model. The continuous demand for faster memory accesses calls for fast translations to terabytes of memory for any computing element in the system. Unfortunately, conventional translation mechanisms fall short of providing fast translations as contemporary memories exceed the reach of today's translation caches, such as TLBs, making TLB misses or page walks a severe performance bottleneck.

This thesis identifies page walks as a critical performance bottleneck in unified virtual memory implementations and proposes mechanisms to eliminate the aforesaid overhead. We provide fundamental insights into the reason why address translation sits on the critical path of accessing memory. We observe that the traditional fully associative flexibility to map any virtual page to any page frame precludes accessing memory before translating. We first study the VM associativity using the 3C model to classify misses (i.e., page faults) and show that conventional full associativity in virtual memory is a largely unnecessary feature, as the majority of misses are either classified as compulsory or capacity, hence insensitive to associativity.

Building on the modest associativity requirements of VM, we propose SAVAgE, a translation mechanism that largely reduces the overhead of page walks. SAVAgE restricts the associativity so that a virtual address identifies a memory chip and memory partition uniquely. An MMU in the memory partition, translates the virtual address and fetches the data. As a result, SAVAgE achieves low-overhead page walks as the page walk and data fetch operation overlap almost entirely. Furthermore, for the important class of first-party in-memory workloads, we propose DIPTA, a translation mechanism that completely eliminates the overhead of page walks. This mechanism builds on the observation that the associativity in VM can be virtually eliminated for

single-process memory-resident workloads. DIPTA restricts the associativity so that a page can only reside in a few number of physically adjacent locations. Hence, all but a few bits remain invariant in the virtual and physical addresses, and we speculatively predict the rest using a way predictor, decoupling address translation from data fetch. To ensure that data fetch and address translation fully overlap, we place the page table entries next to the data, completely eliminating the overhead of page walks for in-memory workloads.

In summary, reducing the associativity of virtual memory nearly eliminates the translation overhead enabling a scalable unified virtual address space.

Bibliography

[1] "3D XPoint," https://en.wikipedia.org/wiki/3D_XPoint.

[2] "Getting The Hang Of IOPS v1.3," http://www.symantec.com/connect/articles/getting-hang-iops-v13.

[3] "Google Search Statistics." http://www.internetlivestats.com/google-search-statistics/.

[4] "MusicBrainz," http://musicbrainz.org/.

[5] "Oracle TimesTen," http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/index.html.

[6] "SAP HANA," http://hana.sap.com/abouthana.html.

[7] "Stack Overflow," https://archive.org/details/stackexchange/.

[8] "The Linux Kernel Archives: Boot Memory Allocator," https://www.kernel.org/doc/gorman/html/understand/understand008.html#chap:BootMemoryAllocator.

[9] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *Proceedings of the 2012 International Symposium on Computer Architecture*, 2012.

[10] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 2015 Annual International Symposium on Computer Architecture*, 2015.

[11] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 2015 Annual International Symposium on Computer Architecture*, 2015.

[12] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked DRAM," in *Proceedings of the 2015 Annual International Symposium on Computer Architecture*, 2015.

[13] Amazon, "High Performance Computing." https://aws.amazon.com/hpc/.

[14] AnandTech, "Intel's 'Tick-Tock' Seemingly Dead, Becomes 'Process-Architecture-Optimization'," http://www.anandtech.com/show/10183/intels-tick-tock-seemingly-dead-becomes-process-architecture-optimization, 2016.

[15] ARM, "Cortex-A7 Processor," http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 2012 ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[17] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, 2014.

[18] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," in *Proceedings of the 2010 International Symposium on Computer Architecture*, 2010.

[19] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *Proceedings of the 2011 Annual International Symposium on Computer Architecture*, 2011.

[20] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, 2017.

[21] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[22] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 2013 International Symposium on Computer Architecture*, 2013.

[23] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proceedings of the 2012 International Symposium on Computer Architecture*, 2012.

[24] D. Bhandarkar and J. J. Ding, "Performance characterization of the pentium(r) pro processor," in *Proceedings of the 1997 International Symposium on High-Performance Computer Architecture*, 1997.

[25] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 2013 International Symposium on Microarchitecture*, 2013.

[26] A. Bhattacharjee, "Translation-triggered prefetching," in *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[27] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *Proceedings of the 2011 International Conference on High-Performance Computer Architecture*, 2011.

[28] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors," in *Proceedings of the 2009 International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[29] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in monetdb," *Commun. ACM*, vol. 51, no. 12, Dec. 2008.

[30] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *Proceedings of the 2005 Biennial Conference on Innovative Data Systems Research*, 2005.

[31] Brian Thompto, "POWER9 Processor for the Cognitive Era," https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-final.pdf.

[32] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: facebook's distributed data store for the social graph," in *Proceedings of the 2013 Annual Technical Conference*, 2013.

[33] B. Calder, D. Grunwald, and J. S. Emer, "Predictive sequential associative cache," in *Proceedings of the 1996 International Symposium on High-Performance Computer Architecture*, 1996.

[34] B. B. Cambazoglu and R. Baeza-Yates, "Scalability challenges in web search engines," *Synthesis Lectures on Information Concepts, Retrieval, and Services*, vol. 7, no. 6, pp. 1–138, 2015.

[35] J. F. Cantin, "Cache Performance for SPEC CPU2000 Benchmarks," 2003. [Online]. Available: http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/

[36] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 2016 International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016.

[37] M. Cekleov and M. Dubois, "Virtual-address caches part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5, 1997.

[38] M. Cekleov and M. Dubois, "Virtual-address caches, part 2: Multiprocessor issues," *IEEE Micro*, vol. 17, no. 6, 1997.

[39] C. Chao, M. Mackey, and B. Sears, "Mach on a virtually addressed cache architecture," in *Proceedings of the 1990 USENIX MACH Symposium*, 1990.

[40] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey, "Lightweight shared objects in a 64-bit operating system," in *Proceedings of the 1992 Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[41] R. Cheng, "Virtual address cache in unix," in *Proceedings of the Summer 1987 USENIX Technical Conf.*, 1987.

[42] T. Chiueh and R. H. Katz, "Eliminating the address translation bottleneck for physical address cache," in *Proceedings of the 1992 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[43] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *Proceedings of the 2016 Annual Design Automation Conference*, 2016.

[44] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the 2014 International Symposium on Microarchitecture*, 2014.

[45] E. Cooper-Balis, P. Rosenfeld, and B. Jacob, "Buffer-on-board memory systems," in *Proceedings of the 2012 Annual International Symposium on Computer Architecture*, 2012.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[47] I. Corporation, "Tlbs, paging-structure caches and their invalidation," 2008.

[48] J. F. Couleur and E. L. Glaser, "Shared-access data processing system," 1968, uS Patent 3,412,382. [Online]. Available: http://www.google.com.ar/patents/US3412382

[49] B. Crothers, "ALR first with new Pentium Pro server," https://www.cnet.com/news/alr-first-with-new-pentium-pro-server/, 1996.

[50] B. Dally, "Efficiency and Parallelism: The Challenges Of Future Computing," 2014. [Online]. Available: http://rice2014oghpc.blogs.rice.edu/files/2014/03/Dally.pdf

[51] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 2011 International Conference on Autonomic Computing*, 2011.

[52] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the 2015 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[53] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, 2013.

[54] P. J. Denning, "The working set model for program behaviour," *Commun. ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[55] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, 1970.

[56] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbocharging DBMS buffer pool using ssds," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[57] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *Proceedings of the 2017 Biennial Conference on Innovative Data Systems Research*, 2017.

[58] X. Dong, S. Dwarkadas, and A. L. Cox, "Shared address translation revisited," in *Proceedings of the 2011 European Conference on Computer Systems*, 2016.

[59] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the diva processing-in-memory chip," in *Proceedings of the 2002 International Conference on Supercomputing*, 2002.

[60] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki, "DBMS data loading: An analysis on modern hardware," in *Data Management on New Hardware - 7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2016 and 4th International Workshop on In-Memory Data Management and Analytics*, 2016.

[61] Facebook, "The end-to-end refresh of our server hardware fleet." https://code.facebook.com/posts/1241554625959357/the-end-to-end-refresh-of-our-server-hardware-fleet/.

[62] "Facebook LinkBench Benchmark," https://github.com/facebook/linkbench, Facebook Inc.

[63] "RocksDB In Memory Workload Performance Benchmarks," https://github.com/facebook/rocksdb/wiki/RocksDB-In-Memory-Workload-Performance-Benchmarks, Facebook Inc.

[64] B. Falsafi, M. Stan, K. Skadron, N. Jayasena, Y. Chen, J. Tao, R. Nair, J. H. Moreno, N. Muralimanohar, K. Sankaralingam, and C. Estan, "Near-memory data services," *IEEE Micro*, vol. 36, no. 1, pp. 6–13, 2016.

[65] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[66] M. Ferdman, A. Adileh, Y. O. Koçberber, 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 2012 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[67] J. Fotheringham, "Dynamic storage allocation in the atlas computer, including an automatic use of a backing store," *Commun. ACM*, vol. 4, no. 10, 1961.

[68] Fujitsu Limited, "White paper FUJITSU Supercomputer PRIMEHPC FX100 Evolution to the Next Generation," https://www.fujitsu.com/global/Images/primehpc-fx100-hard-en.pdf, 2014.

[69] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Range translations for fast virtual memory," *IEEE Micro*, vol. 36, no. 3, pp. 118–126, 2016.

[70] J. Gantz and D. Reinsel, "White Paper: 5-Level Paging and 5-Level EPT," 2016. [Online]. Available: https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf

[71] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, 2015.

[72] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[73] B. Giridhar, M. Cieslak, D. Duggal, R. G. Dreslinski, H. M. Chen, R. Patti, B. Hold, C. Chakrabarti, T. N. Mudge, and D. Blaauw, "Exploring DRAM organizations for energy-efficient and resilient exascale memories," in *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[74] J. R. Goodman, "Coherency for multiprocessor virtual address caches," in *Proceedings of the 1987 International Conference on Architectual Support for Programming Languages and Operating Systems*, 1987.

[75] "Google Performance Tools," https://code.google.com/p/gperftools/, Google Inc.

[76] R. N. Gustafson and F. J. Sparacio, "IBM 3081 processor unit: Design considerations and design process," *IBM Journal of Research and Development*, vol. 26, no. 1, pp. 12–21, 1982.

[77] P. Hammarlund, "4th Generation Intel ® Core ™ Processor, codenamed Haswell," http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf, 2013.

[78] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *Proceedings of the 2007 International Symposium on Computer Architecture*, 2016.

[79] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *Proceedings of the 2015 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[80] M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B. K. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. A. Wood, B. Zorn, P. Hilfinger, D. Hodges, R. Katz, J. Ousterhout, and D. Patterson, "Design decisions in spur," *Computer*, vol. 19, no. 11, 1986.

[81] M. D. Hill, "A case for direct-mapped caches," *Computer*, vol. 21, no. 12, Dec. 1988.

[82] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Computers*, vol. 38, no. 12, 1989.

[83] M. D. Hill, "Aspects of cache memory and instruction buffer performance," Ph.D. dissertation, University of California, Berkeley, 1987, aAI8813907.

[84] HP, "HP to Transform Server Market with Single Platform for Mission-critical Computing," http://www8.hp.com/us/en/hp-news/press-release.html?id=1147777#.WRs0sbyGOL4.

[85] HSA Foundation, "HSA Platform System Architecture Specification 1.0," 2015.

[86] K. Hsieh, S. M. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *Proceedings of the 2016 IEEE International Conference on Computer Design*, 2016.

[87] C.-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3d: Mitigating the numa bottleneck via coherent dram caches," in *Proceedings of the 2016 International Symposium on Microarchitecture*, 2016.

[88] J. Huck and J. Hays, "Architectural support for translation table management in large address space machines," in *Proceedings of the 1993 International Symposium on Computer Architecture*, 1993.

[89] "Intel 64 and IA-32 Architectures Software Developer's Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf, Intel Corp.

[90] "Intel Virtualization Technology for Directed I/O Architecture," Intel Corporation, 2014.

[91] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 2008 International Symposium on Computer Architecture*, 2008.

[92] B. L. Jacob, *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[93] 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 1998 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[94] B. L. Jacob and T. N. Mudge, "Virtual memory: Issues of implementation," *IEEE Computer*, vol. 31, no. 6, 1998.

[95] "High Bandwidth Memory (HBM) DRAM," JEDEC Solid State Technology Assoc., 2013. [Online]. Available: http://www.jedec.org/standards-documents/results/jesd235

[96] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *Proceedings of the 2014 Annual International Symposium on Microarchitecture*, 2014.

[97] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 2013 Annual International Symposium on Computer Architecture*, 2013.

[98] Jonathan Corbet, "Transparent huge pages in 2.6.38," https://lwn.net/Articles/423584/.

[99] N. P. Jouppi, "Architectural and organizational tradeoffs in the design of the multititan cpu," in *Proceedings of the 1989 Annual International Symposium on Computer Architecture*, 1989.

[100] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 1990 Annual International Symposium on Computer Architecture*, 1990.

[101] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg,

A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017.

[102] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings of the 2002 Annual International Symposium on Computer Architecture*, 2002.

[103] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Prentice Hall, 1992.

[104] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. V. Lam, J. Torrellas, and P. Pattnaik, "Flexram: Toward an advanced intelligent memory system," in *Proceedings of the 1999 International Conference On Computer Design, VLSI in Computers and Processors*, 1999.

[105] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the 2016 European Conference on Computer Systems*, 2016.

[106] D. Kanter, "Cavium Thunders Into Servers: Specialized Silicon Rivals Xeon for Specific Workloads," 2016. [Online]. Available: http://www.linleygroup.com/mpr/article.php?url=mpr/h/2016/11545/11545.pdf

[107] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 2015 Annual International Symposium on Computer Architecture*, 2015.

[108] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *Proceedings of the 2014 International Symposium on Workload Characterization*, 2014.

[109] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 2011 Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[110] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "Readings in computer architecture." Morgan Kaufmann Publishers Inc., 2000.

[111] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Proceedings of the 2016 International Symposium on Computer Architecture*, 2016.

[112] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 2013 International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[113] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash ≈ local flash," in *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[114] Y. O. Koçberber, B. Grot, J. Picorel, B. Falsafi, K. T. Lim, and P. Ranganathan, "Meet the walkers: accelerating index traversals for in-memory databases," in *Proceedings of the 2013 International Symposium on Microarchitecture*, 2013.

[115] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *Proceedings of the 1994 International Conference on Parallel Processing*, 1994.

[116] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architectural support for single address space operating systems," in *Proceedings if tge 1992 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[117] C. E. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, vol. 30, no. 4, 2010.

[118] C. E. Kozyrakis, S. Perissakis, D. A. Patterson, T. E. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. A. Yelick, "Scalable processors in the billion-transistor era: IRAM," *IEEE Computer*, vol. 30, no. 9, 1997.

[119] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, Jan. 2016.

[120] H. M. Levy and P. H. Lipman, "Virtual memory management in the VAX/VMS operating system," *IEEE Computer*, vol. 15, no. 3, 1982.

[121] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," in *Proceedings of the 2013 Annual International Symposium on Computer Architecture*, 2013.

[122] G. H. Loh, "Extending the effectiveness of 3d-stacked DRAM caches with an adaptive multi-queue policy," in *Proceedings of the 2009 Annual International Symposium on Microarchitecture*, 2009.

[123] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *Proceedings of the 2011 Annual International Symposium on Microarchitecture*, 2011.

[124] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 Conference on Programming Language Design and Implementation*, 2005.

[125] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Proceedings of the 2012 International Symposium on Computer Architecture*, 2012.

[126] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[127] Micron, "Micron Hybrid Memory Cube," http://www.micron.com/products/hybrid-memory-cube.

[128] Micron, "Hybrid Memory Cube Specification 2.1," 2014. [Online]. Available: http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf

[129] M. P. Mills, "Big Data, Big Networks, Big Infrastructure, and Big Power. An Overview of the Electricity Used by the Global Digital Ecosystem," https://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf.

[130] N. Mirzadeh, O. Koçberber, B. Falsafi, and B. Grot, "Sort vs. hash join revisited for near-memory execution," in *Proceedings of the 2015 International Workshop on Architectures and Systems for Big Data (ASBD)*, 2015.

[131] J. Ning, "Open Compute Project: Facebook Server Intel Motherboard V3.1 Rev 1.00," https://www.circleb.eu/wp-content/uploads/2016/04/Open_Compute_Project_FB_Server_Intel_Motherboard_v3.1_rev1.00.pdf, 2016.

[132] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proceedings of the 2015 International Symposium on Microarchitecture*, 2015.

[133] Oracle, "SPARC M7-16 Server," https://www.oracle.com/servers/sparc/m7-16/index.html.

[134] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings of the 1998 International Symposium on Computer Architecture*, 1998.

[135] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked dram," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, 2015.

[136] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: scalable high-performance storage entirely in DRAM," *Operating Systems Review*, vol. 43, no. 4, 2009.

[137] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *Proceedings of the 2015 International Symposium on High Performance Computer Architecture*, 2015.

[138] C. H. Park, T. Heo, and J. Huh, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *Proceedings of the 2016 International Symposium on Computer Architecture*, 2016.

[139] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 2014 International Symposium on High Performance Computer Architecture*, 2014.

[140] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach TLBs," in *Proceedings of the 2012 International Symposium on Microarchitecture*, 2012.

[141] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces," in *Proceedings of the 2014 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[142] B. Pichai, L. Hsu, and A. Bhattacharjee, "Address translation for throughput-oriented accelerators," *IEEE Micro*, vol. 35, no. 3, pp. 102–113, 2015.

[143] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proceedings of the 2001 Annual International Symposium on Microarchitecture*, 2001.

[144] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proceedings of the 2014 International Symposium on High Performance Computer Architecture*, 2014.

[145] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood, "Implications of emerging 3d GPU architecture on the scan primitive," *SIGMOD Record*, vol. 44, no. 1, 2015.

[146] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *Proceedings of the 2014 International Symposium on Performance Analysis of Systems and Software*, 2014.

[147] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 Annual International Symposium on Microarchitecture*, 2012.

[148] *libHugeTLBFS*, http://linux.die.net/man/7/libhugetlbfs, Red Hat Inc.

[149] J. Reinders, "Knights Corner: Your Path to Knights Landing," https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf, 2014.

[150] S. Saini, J. Chang, and H. Jin, "Performance evaluation of the intel sandy bridge based NASA pleiades using scientific and engineering applications," in *Proceedings of the 2013*

International Workshop on High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation, 2013.

[151] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *Proceedings of the 2013 Parallel Processing and Applied Mathematics*, 2013.

[152] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB preloading," in *Proceedings of the 2000 Annual International Symposium on Computer Architecture*, 2000.

[153] D. Sheffield, "IvyTown Xeon + FPGA: The HARP Program," https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf.

[154] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi, "Quantifying the relationship between the power delivery network and architectural policies in a 3d-stacked memory device," in *Proceedings of the 2013 Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[155] A. J. Smith, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Trans. Software Eng.*, vol. 4, no. 2, pp. 121–130, 1978.

[156] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 2009 International Symposium on Computer Architecture*, 2009.

[157] Y. Song and E. Ipek, "More is less: improving the energy efficiency of data movement via opportunistic use of sparse codes," in *Proceedings of the 2015 International Symposium on Microarchitecture*, 2015.

[158] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *Proceedings of the 2010 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[159] Sun Microsystems, "UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007," http://www.oracle.com/technetwork/systems/opensparc/t2-13-ust2-uasuppl-draft-p-ext-1537760.html, 2007.

[160] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," in *Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[161] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice—a low-cost high-speed address translation mechanism," in *Proceedings of the 1990 Annual International Symposium on Computer Architecture*, 1990.

[162] "Tezzaron DiRAM," http://www.tezzaron.com/products/diram4-3d-memory/, Tezzaron Semiconductor.

[163] The Economist, "Technology Quarterly after Moore's Law: Double, double, toil and trouble," http://www.economist.com/technology-quarterly/2016-03-12/after-moores-law, 2016.

[164] "Open Compute Server: Specs and Designs," http://www.opencompute.org/wiki/Server/SpecsAndDesigns, The Open Compute Project.

[165] B. Towles, J. P. Grossman, B. Greskamp, and D. E. Shaw, "Unifying on-chip and inter-node switching within the anton 2 network," in *Proceedings of the 2014 International Symposium on Computer Architecture*, 2014.

[166] J. Veselý, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software*, 2016.

[167] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramírez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[168] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat caches for scale-out servers," *IEEE Micro*, 2016.

[169] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "Bump: Bulk memory access prediction and streaming," in *Proceedings of the 2014 Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[170] W. H. Wang, J.-L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Proceedings of the 1989 Annual International Symposium on Computer Architecture*, 1989.

[171] D. L. Weaver and T. Germond, *The SPARC Architecture Manual*. SPARC International, Inc., 2003.

[172] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[173] B. Wheeler and B. N. Bershad, "Consistency management for virtually indexed caches," in *Proceedings of the 1992 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[174] K. Wilcox, D. Akeson, H. R. F. III, J. Farrell, D. Johnson, G. Krishnan, H. McIntyre, E. McLellan, S. Naffziger, R. Schreiber, S. Sundaram, and J. White, "4.8 A 28nm x86 APU optimized for power and area efficiency," in *Proceedings of the 2015 IEEE International Solid-State Circuits Conference*, 2015.

[175] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers*, vol. 14, no. 2, pp. 270–271, 1965.

[176] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proceedings of the 2015 International Workshop on Data Management on New Hardware (DaMoN)*, 2015.

[177] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016.

[178] H. Yoon and G. S. Sohi, "Revisiting virtual L1 caches: A practical design using dynamic synonym remapping," in *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*, 2016.

[179] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2010 Conference on Hot Topics in Cloud Computing*, 2010.

[180] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

Javier Picorel

javpicorel@gmail.com

Research Interests

- Hardware and software co-design for OS services
- Performance evaluation and micro-architectural characterization of cloud services
- Server architectures for datacenters
- Data analytics architectures for emerging memory technologies
- Hardware acceleration for database systems

Education

- **Ecole Polytechnique Federale de Lausanne**, Lausanne, Switzerland.
  Ph.D. in Computer Science, 2011-2017.
  Thesis: "Near-Memory Address Translation"
  Advisor: Prof. Babak Falsafi.
- **University of Zaragoza (UNIZAR)**, Zaragoza, Spain.
  Diploma in Computer Engineering (BSc + MSc), 2005-2011.
  Thesis: "Exploiting Address Deltas in Memory Streams", under the supervision of Prof. Babak Falsafi and Prof. José Luis Briz. Grade 9.5/10.

Awards & Honors

- EPFL Teaching Assistant Award, 2014.
- MICRO Best Paper Runner-Up, 2013.
- EPFL Computer Science Fellowship for the 1^{st} year of doctoral studies, 2011-2012.

Projects

- **CloudSuite 3.0**
  Directed a group of graduate students to release the third version of CloudSuite, a standard benchmark suite of online services. CloudSuite 3.0 is a major enhancement with the addition of new benchmarks, and integration into Docker as well as Google's PerfKit Benchmarker framework.
  http://cloudsuite.ch
- **QFlex 1.0**
  Directed a group of graduate students and interns to release the first version of QFlex. QFlex is a full-system cycle-accurate simulator of multi-node computer systems. QFlex is composed of three main components: QEMU, Flexus, and NS-3. QEMU is a widely-used machine emulator, Flexus provides cycle-accurate CPU and DRAM models, and NS-3 is a popular network simulator that permits modelling of real-world topologies.
  https://parsalab.github.io/qflex/

Publications

- **SAVAgE: Set-Associative Virtual Memory for Near-Memory Processing**
  J. Picorel, D. Jevdjic, A. Bhattacharjee, B. Falsafi. *Under Submission*, 2017.
- **Near-Memory Address Translation**
  J. Picorel, D. Jevdjic, B. Falsafi. *To appear in Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
- **The Mondrian Data Engine**
  M. Drumond, A. Daglis, D. Ustiugov, N. Mirzadeh, J. Picorel, B. Falsafi, B. Grot, D. Pnevmatikatos. *To appear in Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017.

- **Unlocking Energy**
  B. Falsafi, R. Guerraoui, J. Picorel, V. Trigonakis. *In Proceedings of the 21st USENIX Annual Technical Conference (USENIX ATC)*, 2016.
- **Towards Near-Threshold Server Processors**
  A. Pahlevan, J. Picorel, A. Pourhabibi Zarandi, R. Davide, M. Zapater Sancho, P. Garcia del Valle, D. Atienza Alonso, L. Benini, B. Falsafi. *In Proceedings of the 19th Design, Automation and Test in Europe (DATE)*, 2016.
- **BuMP: Bulk Memory Access Prediction and Streaming**
  S. Volos, J. Picorel, B. Grot, B. Falsafi. *In Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, 2014.
- **Meet the Walkers: Accelerating Index Traversal for In-Memory Databases**
  O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, P. Ranganathan. *In Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, 2013. (Recognized as the best paper runner-up by the program committee)
- **Scale-Out Processors**
  P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. *In Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012.

Invited Talks

- **Server Benchmarking and Design with CloudSuite 3.0**
  J. Picorel. At DATE'17, Lausanne, Switzerland, 2017.
- **Server Benchmarking with CloudSuite 3.0** (Tutorial)
  Alexandros Daglis, Mario Paulo Drumond, J. Picorel, and Dmitrii Ustiugov. At EuroSys'16, London, UK, 2016.
- **Towards Near-Threshold Server Processors**
  At ICT-Energy Workshop (co-located with HiPEAC'16), Prague, Czech Republic, 2016.
- **Server Benchmarking with CloudSuite 3.0** (Tutorial)
  Alexandros Daglis, Mario Paulo Drumond, and J. Picorel. At HiPEAC'16, Prague, Czech Republic, 2016.
- **A Framework for Hardware/Software Co-Design of Computer Systems**
  Internship wrap-up talk at Oracle Labs, Redwood City, CA, USA, 2013.

Work Experience

- **Oracle Labs**, Redwood City, CA, USA.
  Research intern, February-September 2013.
  Supervisor: Peter Hsu.
  Built a full-system emulator for an experimental hardware platform, and ported Linux and Glibc to it.
- **Ecole Polytechnique Federale de Lausanne**, Lausanne, Switzerland.
  Research intern, 2010 – 2011.
  Supervisor: Prof. Babak Falsafi.
  Investigated hardware data prefetching techniques for both traditional server and modern cloud applications.

Funding

- **Memory-Centric Server Architecture for Datacenters**, Swiss National Science Foundation (SNSF), 2016.
  PI: Babak Falsafi. Co-authored the proposal's technical part with Alexandros Daglis. Ranked at the top of six quality levels; 3-year funding.

Student Supervision

- MohammadReza Katebzadeh, Master student, on his internship project entitled "Timing-first Simulation for QFlex", 2016.
- Nora Abi Akar, Master student, on her internship project entitled "Incremental Snapshots for QEMU", 2016.
- Matthew Joshua Underwood, Bachelor student, on his internship project entitled "Extended API for Architectural Simulators for QEMU", 2015.
- Rishabh Sanghi and Nikhi Chaturvedi, Bachelor students, on their internship project entitled "NS3-based Networking for Multi-node QEMU Emulation", 2015.
- Damien Hilloulin, Master student, on his internship project entitled "ARM Trace Simulation and Determinism for QFlex", 2015.
- Jan Alexander Wessel, Master student, on his internship project entitled "QEMU and Remote Memory Controller Integration", 2015.
- John Biggs, Bachelor student, on his internship project entitled "API for Architectural Simulators for QEMU", 2014.
- Nibal Salha, Bachelor student, on his internship project entitled "ISA Instrumentation for QEMU", 2014.

Teaching Assistantships

- Computer Architecture (BS), Fall 2014, Fall 2016.
  Instructor: Babak Falsafi (2014), Mirjana Stojilovic (2016).
- Introduction to Multiprocessor Architecture (BS), Fall 2015.
  Instructor: Babak Falsafi.
- Introduction to Object-Oriented Programming (BS), Spring 2015.
  Instructors: Jamila Sam and Jean-Cedric Chappelier
- Introduction to Programming (BS), Fall 2012.
  Instructor: Marc Lecoultre.

Technical Skills

- Programming Languages: C/C++, Java, Lisp, Ruby, VHDL.
- Assembly: ARM, MIPS, SPARC, x86.
- Platforms: Docker, Linux, Solaris, Windows.
- Hardware Development: Altera Cyclone, Xilinx Spartan.
- Architectural Simulators: Flexus, QFlex, SimpleScalar.
- Machine Emulators: QEMU, Simics
- Profiling Tools: Intel VTune, Linux Perf, Solaris DTrace.
- System Administrator: FreeNAS, Linux, Solaris.
- Kernel Development: Linux.
- Version Control System: Git/GitHub.

Professional Activities

- Journal reviewer: ACM TACO, IEEE TSUSC.
- Student member: ACM SIGARCH, HiPEAC.
- Web chair: International Workshop on Design for 3D Silicon Integration (D43D), 2014.