EPFL

Thèse n° 8864

Rebooting Virtual Memory with Midgard

Présentée le 22 août 2023

Faculté informatique et communications
Laboratoire d'architecture de systèmes parallèles
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Siddharth GUPTA

Acceptée sur proposition du jury

Prof. C. Koch, président du jury
Prof. B. Falsafi, A. Bhattacharjee, directeurs de thèse
Dr G. Loh, rapporteur
Dr J. Gandhi, rapporteur
Prof. S. Kashyap, rapporteur

École
polytechnique
fédérale
de Lausanne

2023

अति का भला ना बोलना, अति की भली ना चुप ।
अति का भला ना बरसना, अति की भली ना धूप ।
— कबीरदास

Excess of speaking is bad, and excess of silence is also bad.
Just like excess of rain is bad, and excess of sun is also bad.
Excess of anything is bad, balance is key.
— KabirDas

To my mentors...

Acknowledgements

My PhD journey has been the most intensive task I have ever undertaken. Striving for a PhD has significantly altered various aspects of my personality and underlined the value of perseverance, teamwork, balance, and patience. Throughout my PhD journey, I have immensely enjoyed working on various research projects and met extremely talented colleagues from diverse academic and cultural backgrounds, which has motivated and taught me much about collaboration. And just like every other PhD student, I have also had various dips in my motivation during the challenging parts, which left me questioning various tradeoffs I had made. I would not have been able to reach the finish line without the support of the wonderful people around me, to whom I owe my sincere gratitude.

First, I am grateful to my advisors, Babak Falsafi and Abhishek Bhattacharjee. Babak is famous for being a tough taskmaster, which is fueled by his deep commitment to the success of his students. There have been multiple instances where my motivation faltered, but he tenaciously and unceremoniously pushed me past the finish line. While I have undoubtedly questioned his methods during the process, I can now understand the great value he provides to his students. Babak is extremely smart and has an excellent knack for spotting technology trends, which results in fruitful research directions for his students. He seeks perfection in everything and has always shone a spotlight on my shortcomings and the "wing-it" attitude, forcing me to improve as a well-rounded researcher and individual. He inculcates a strong lab culture which leads to long-lasting ties and friendships among students in and across generations. Being his student has connected me to a large family of super talented and successful individuals who drive the future of technology, which is a great perk on its own. Finally, Babak and his lab's culture are solely responsible for me finding a great love of my life — skiing — which I hope

will continue till my knees give out. Thank you, Babak!

Abhishek is one of the kindest persons I have ever met, and cares deeply about the well being of his students. For all the deadlines we have endured together, he has never failed to check up on me and offer words of encouragement. He has always offered me a different perspective on things compared to the Babak school of thought, which has helped me in exploring new ideas in research and presentation. Being one of the leading experts on Virtual Memory with concrete industry-scale contributions, he has been instrumental in shaping my understanding of the area and the output of research projects. Being Abhishek's student gave me access to another academic family of great colleagues, even though we only met at conferences. I am very glad that Abhishek agreed to advise me.

I am also grateful to my silent advisor, Boris Grot, who has been helping me even before I started my PhD journey. Boris is a great blend of interpersonal skills and technical knowledge, and has been a great source of inspiration for me. He is single-handedly responsible for me diving into a 6-year-long PhD program with Babak at the naive age of 22. I had worked with him for only four months when he convinced me to go for a PhD instead of a shorter Masters program as was done by many of my colleagues, and my current research standing is a direct result of his intuition. I am forever indebted to Boris for giving me my big break as an intern to work in a high-profile research lab during my Bachelors, and for teaching me my first lesson in research and presentation. I am very excited that I will have the opportunity to work with Boris again at my next venture.

Next, I would like to thank Gabriel Loh, Jayneel Gandhi, Sanidhya Kashyap, and Christoph Koch for serving on my PhD thesis committee, and for invaluable advice that has shaped the final thesis. Yuanlong Li, Yunho Oh, Atri Bhattacharyya, Qingxuan Kang, and Mathias Payer have been close collaborators in the work done as part of my thesis. Thank you for all your help and support!

The strong group culture at PARSA (my home away from home) has been a highlight of my PhD journey. I would like to start by thanking Mario Drumond, Rishabh Iyer, and Ognjen Glamočanin (in the order of increasing hair on their head) who have been my closest friends

and brothers. Mario is an extremist — but not the kind you see on TV — with an out-of-the-world perspective on almost every topic. The best thing about Mario is his direct unfiltered opinions, but they might take some time to get used to. While he was still in the lab and was forced to listen to my random research ideas, he only gave reactions falling in "this is the greatest thing ever" or "everyone in the room is now dumber" bucket. Fortunately or unfortunately, I have adopted Mario's way of dealing with bad and nervous situations, which typically involves saying "it could be worse, you could be missing a leg", or "you can cry throughout the presentation and just change the slides, and your paper will still get published". It helped a lot!

While Rishabh is not in PARSA, he is considered an honorary member because of historical reasons. Rishabh is extremely competitive, which shows up in his work, games, fitness, and asks of a tougher final exam. He also has a strictly hate relationship with bureaucracy, keys, cards, etc. which often ends up in rescue requests or stones being thrown at windows. We have spent countless hours walking along the lake and discussing research ideas, which has been greatly helpful. Having Rishabh around has definitely motivated me to perform better as a researcher and at many other activities. Now that he is changing continents, it will be much harder to come up with crazy activities on my own.

Ogi has been my office mate for the past few years, and has endured various outbursts followed by calm-down conversations. His love for cleanliness has resulted in a display of sanitizers in our office, and hate for bed bugs has been a constant source of amusement. He has this amazing ability where he can take the simplest event, and start spiralling and imagining the worst outcome possible. While I have enjoyed the days where I only had his company, we have always needed to open a window from time to time. All in all, Ogi has been a great friend and office mate, and it is certain that without him, my PhD experience would be much worse. In various permutations, we have all shared a lot of laughs together, which has made my time at EPFL considerably easier. Thank you guys!

Moving on to the rest of the lab, I will first thank the alumni — Alex Daglis, Yunho Oh, Arash Pourhabibi, Mark Sutherland, and Javier Picorel — who have already tasted the sweet nectar

of graduation. Alex was the senior-most student in the lab when I started my PhD and we overlapped for only one year, which shocks him each time we meet. I was extremely fortunate to submit my first paper with Alex because he gladly took that opportunity to show me the ropes of how things were done in the lab, in particular how to match the fonts in the figures with the ones in the text, something that I have never done because "the fonts are too pointy". I can honestly say that meeting Alex and discussing random research topics for hours is one of the main reasons I go to conferences in the first place.

Yunho has been one of the most positive people I have met, where the positivity is typically succeeded by a comment about how I am doomed. We have the weirdest conversations as none ends without death threats, references to the Godfather, talks about adopting and smuggling me to Korea, or weird GIFs being sent around. I cannot go to KFC anymore without remembering the great times we have had. He also strangely keeps visiting Switzerland just to hang out in Lausanne, and skips all other interesting parts of the country. God help him!

Arash is one of the most diligent people I have met, and the phrase "slow and steady" applies very well to him, while I am more of a "move fast and break things" person. He is an Apple fanatic, and I do not know anyone else with the same degree of obsession. Finally, the image of him saying "you have got to listen" while pulling his ear has been burned into my brain, and I don't think that I can ever erase it, even though I might not end up listening anyways.

Mark is one of the most detail-oriented people I have met, and his motivation to chase down technical nitty-gritties or conduct a super-thorough evaluation of an idea has been inspiring. He is also very disciplined, and his ability to regularly come to the office early in the morning and leave at a good hour to maintain a work-life balance is something to be adopted.

Javier is the biggest party-person I know, and is really fun to be around. I am really thankful to him for hiring me for a five-month cool-down period in Munich, which was punctuated with long beer-drinking sessions, and barbecue sessions along the Isar river. I really needed it! I hope we get to work together again in the future.

I am also very grateful to Atri Bhattacharyya, Simla Harma, Yuanlong Li, Shanqing Lin, Shashwat Shrivastava, and Ayan Chakraborty for their support. Atri's attitude to handle life as it comes is inspiring, or in his words "carefree wanderings". Simla is a very warm and cheerful

person and I have always seen her smiling, which is a very rare quality today. Yuanlong is super smart and hardworking, and I am sure that he will be successful with taking Midgard forward. Shanqing is super fun to be around, and is never afraid to ask the "why" questions, especially when they result in hilarious responses. Shashwat and Ayan have suffered through many skiing days with me. Shashwat has a great ability to fall randomly, but then get up immediately and start skiing again without losing confidence. Ayan seems to be the only person who has matched my craziness for skiing in the lab, and I really hope that he can pass it to other lab members, and keep the skiing culture going strong. I would also like to thank Dmitrii Ustiugov, Dina Gamaleldin, Ali Ansari, and Buğra Eryılmaz for their constant feedback and support.
Finally, I would like to sincerely thank Stéphanie Baillargues, without whom the list of PARSA members is incomplete. Stéphanie is a wizard when it comes to administrative affairs, and does not let the bureaucracy get to the PhD students in the lab. While I have loved spending the lab's money, I have always needed to battle Stéphanie who is responsible for spending the Swiss taxpayer's money reasonably. Needless to say, the Swiss taxpayers chose a great champion to represent them.

Beyond PARSA, I would like to thank a few more people for their help and support. Effi Georgala has been very supportive and motivating whenever she is around. Teca Glamočanin has been a great help when selecting gifts (which is typically beyond my scope of expertise), and has sometimes succumbed to the skiing pressure applied by me. Negar Foroutan and Kyveli Short have always been great fun to hang out with. Sahand Kashani, Mahyar Emami, Ahmet Yüzügüler, and Xinrui Jia have been frequent lunch buddies, and have provided a safe space to vent about the typical PhD blues. Sahand shares my aversion to heat and love for the cold, and accompanied Ahmet and I for the amazing trip to Swedish laplands during Christmas in 2022. Ahmet has been a master of not giving a damn for things he does not care about. Pakshal Bohra has been a great friend, and has helped a lot in balancing Rishabh out, especially during WWE gaming marathons or eating out in restaurants. Poulami Das and Diya Joseph have been great conference friends, and we have had incredible trips to Heidelberg and Portugal together. Phil and Paola have been my flatmates for the last couple of years,

and we have had great fun in cooking together, watching movies, having intense spicy noodle eating contests, and even a few skiing lessons which were never repeated. Finally, the YUVA organization at EPFL has been a great source of keeping in touch with the Indian folks around, and enjoying Indian festivals far away from home.

I would like to sincerely thank my close group of friends from BITS - Pratyush Patel, Rohith (Madara) Ramakrishnan, Pratik Singhal, and Mukul Kothari. Patel, Madara, Singhal, and I (the OG OSFS) worked on various unsupervised projects during BITS (such as building an OS from scratch), which I believe was a key in developing my interest towards computer architecture and systems. Patel, Madara, Singhal, Mukul, and I starting hanging out daily in my second year at BITS, which immediately led to a plunge in my GPA, something they will not let me forget easily (especially LCS and Philosophy). We have done a lot of crazy stuff together, in and after BITS, and hopefully will continue to do the same. I am glad to say that we all are still in regular contact, something that typically does not happen when everyone is in a different physical location. Srishti Sahani, Priyanshi Nanda, Aishwarya Anand, Furquan Uddin, Shray Agarwal, Srishti Chaudhary, Rachit Jawrani, and Peshal Agarwal have been my (school) friends for about 15 years now. The best part about our friendship is that we might not talk for months, but when we do talk or meet, then things are exactly the way they were last time. Thank you for being in my life all this time.

Last but not the least, I would like to thank my family from the bottom of my heart for their unconditional love and support. It is the environment and upbringing that I received from my mom, dad, uncle, and aunt, that has moulded my character to its current form and is directly responsible for any success I have in this life. My younger brothers — Harsh, Shantanu, and Yash — have been a great company to grow up with, and I am very proud of you. I don't have enough words to describe the sacrifices made by my family to build a better future for me and my siblings. I am extremely fortunate and grateful to have them. Finally, even though it is recent, I would like to thank Anvita and my in-laws for wholeheartedly welcoming me into their lives. No backsies! I am excitedly looking forward to a future together.

I thank the various funding sources that have enabled this thesis. My PhD research has been partially supported by EPFL, a Qualcomm Innovation Fellowship on *Rebooting Virtual Memory with Midgard* (CMS Contract No. ECO-461760), and the *Hardware/Software Co-Design for In-Memory Services* project (200020B_188696) of the Swiss National Science Foundation.

Lausanne, August 15, 2023 Siddharth Gupta

Abstract

Virtual Memory (VM) is a critical programming abstraction that is widely used in various modern computing platforms. With the rise of datacenter computing and the birth of planet-scale online services, the semantic and capacity requirements from memory have evolved dramatically and pose various challenges for VM. The traditional VM implementations cannot scale with the increasing memory capacity present in modern datacenter servers, and the adoption of heterogeneous memory hierarchies stresses the synchronization mechanisms in VM implementations. The increasing degree of multi-tenancy in datacenter servers requires lean virtualization mechanisms built using VM as a foundation, while the adoption of new paradigms such as microservices and serverless computing lead to increased contention and performance loss in VM implementations. Finally, the increasing requirement of confidentiality in various computing domains also requires improvements in VM implementation as it forms the basis of modern security mechanisms.

This thesis aims to holistically reinvent the VM implementation using OS, architecture, and microarchitecture co-design, which can help solve the above challenges. Our redesign of VM is based on the following key insights: (i) we can use the existing abstraction of Virtual Memory Areas (VMAs) to optimize address translation while ensuring that the application programming model is still POSIX-compliant so no changes are required, and reuse of existing abstractions provides an easy adoption path for the OS developers; (ii) we can divide the traditional address translation into a fast and slow step where the fast step is required on the critical path of all memory accesses and thus mandates specialized microarchitectural support, while the slow step is invoked infrequently and can be supported using the existing resources present in the cache hierarchy without requiring significant additional silicon; and (iii) instead

of encapsulating complicated functionalities completely in the microarchitecture, we can partly offload it to the OS using a simple OS and microarchitecture co-design while helping the OS developers ensure correctness by providing formal specifications of the required behavior. We realize the above insights by introducing an intermediate address space called *Midgard* between the virtual and physical address spaces. The Midgard address space is used to index the cache hierarchy and coherence domain while ensuring that the physical address space is only required when accessing the physical memory device. The introduction of the Midgard address space enables lean VMA-based virtual-to-Midgard address translation, providing fast access control and access to cache hierarchy while requiring little microarchitectural support. The page-based Midgard-to-physical address translation is only required for capacity management when accessing physical memory and can be performed using the resources already present in the cache hierarchy instead of requiring specialized microarchitectural support. To handle the exceptions generated by the delayed Midgard-to-physical address translation without requiring significant silicon resources, we introduce a novel OS + microarchitecture co-design to partly offload complicated microarchitectural functionality to be easily performed in the OS and provide formalism to help the OS developers ensure correct behavior. We evaluate the address translation in Midgard using full-system trace simulation and show that Midgard can eliminate address translation overhead by using incoming large-capacity cache hierarchies. We also build a full-system RISC-V prototype with Linux to model exception handling behavior for Midgard and show that the resulting design is both performant and correct. Overall, Midgard reinvents the traditional VM implementation and allows it to scale with the increasing memory capacity requirements in modern datacenter servers.

Résumé

La mémoire virtuelle (VM) est une abstraction critique pour la programmation et est largement utilisée dans diverses platesformes informatiques modernes. Avec l'essor de l'informatique des centres de données et la naissance de services en ligne à l'échelle planétaire, les exigences sémantiques et de capacité de la mémoire ont considérablement évolué et posent divers défis à la VM. Les implémentations de VM traditionnelles ne peuvent pas évoluer avec la capacité de mémoire croissante présente dans les serveurs de centres de données modernes, et l'adoption de hiérarchies de mémoire hétérogènes met de la pression sur les mécanismes de synchronisation dans les implémentations de VM. Le degré croissant de multi-location dans les serveurs de centres de données nécessite des mécanismes de virtualisation allégés construits à l'aide de VM comme base, et l'adoption de nouveaux paradigmes tels que les microservices et l'informatique "serverless" entraînent une augmentation des conflits et une perte de performances dans les implémentations de VM. Enfin, l'exigence croissante de confidentialité dans divers domaines informatiques nécessite également des améliorations dans l'implémentation des machines virtuelles car elles constituent la base des mécanismes de sécurité modernes.

Cette thèse vise à réinventer de manière holistique l'implémentation des VM en co-concevant le système d'exploitation, l'architecture, et la microarchitecture, ce qui peut aider à résoudre les défis ci-dessus. Notre refonte de VM est basée sur les idées clés suivantes : (i) nous pouvons utiliser l'abstraction existante des zones de mémoire virtuelle (VMA) pour optimiser la traduction d'adresses tout en garantissant que le modèle de programmation pour les applications est toujours conforme à POSIX de sorte qu'aucune modification logicielle ne soit nécessaire, et la réutilisation des abstractions existantes fournit un chemin d'adoption facile pour le

développeurs de système d'exploitation ; (ii) nous pouvons diviser la traduction d'adresse traditionnelle en une étape rapide et une étape lente où l'étape rapide est requise sur le chemin critique de tous les accès à la mémoire et nécessite donc un support microarchitectural spécialisé, tandis que l'étape lente est invoquée rarement et peut être prise en charge en utilisant le ressources existantes présentes dans la hiérarchie de cache sans consommer beaucoup plus d'espace ; et (iii) au lieu d'encapsuler complètement des fonctionnalités compliquées dans la microarchitecture, nous pouvons les décharger en partie sur le système d'exploitation en utilisant une simple co-conception du système d'exploitation et de la microarchitecture tout en aidant les développeurs du système d'exploitation à garantir une execution correcte en fournissant des spécifications formelles du comportement requis.

Nous réalisons les idées ci-dessus en introduisant un espace d'adressage intermédiaire appelé *Midgard* entre les espaces d'adressage virtuel et physique. L'espace d'adressage Midgard est utilisé pour indexer la hiérarchie de cache et le domaine de cohérence tout en garantissant que l'espace d'adressage physique n'est requis que lors de l'accès au dispositif de mémoire physique. L'introduction de l'espace d'adressage Midgard permet une traduction allégée des adresses virtuelles vers Midgard basée sur VMA, offrant un contrôle d'accès rapide et un accès à la hiérarchie de cache tout en nécessitant peu de support microarchitectural. La traduction d'adresse Midgard en adresse physique basée sur des pages n'est requise que pour la gestion de la capacité lors de l'accès à la mémoire physique et peut être effectuée à l'aide des ressources déjà présentes dans la hiérarchie de cache au lieu de nécessiter un support microarchitectural spécialisé. Pour gérer les exceptions générées par la traduction retardée de l'adresse Midgard vers l'adresse physique sans nécessiter beaucoup plus d'espace, nous introduisons une nouvelle co-conception OS + microarchitecture pour décharger en partie les fonctionnalités microarchitecturales complexes à exécuter facilement dans le système d'exploitation et fournir un formalisme pour aider les développeurs de systèmes d'exploitation à garantir un comportement correct. Nous évaluons la traduction d'adresses dans Midgard à l'aide d'une simulation de trace complète du système et montrons que Midgard peut éliminer les frais généraux de traduction d'adresses en utilisant des hiérarchies de cache de grande capacité prochainement disponibles. Nous construisons également un prototype RISC-V

complet avec Linux pour modéliser le comportement de gestion des exceptions pour Midgard et montrer que la conception résultante est à la fois performante et correcte. Dans l'ensemble, Midgard réinvente l'implémentation traditionnelle des machines virtuelles et lui permet d'évoluer avec les besoins croissants en capacité de mémoire dans les serveurs de centres de données modernes.

Contents

List of Figures

List of Tables

1 Introduction

Our society is undergoing a paradigm-shifting digital transformation where online services have become an integral part of our daily life, e.g., social media, video streaming, online shopping, web search, etc. Data is the ultimate currency which is fueling this transformation, especially in areas such as Artificial Intelligence and Machine Learning, and allows customizing and automating various services and increasing their usefulness for the end users. The requirement of storing and processing large amounts of data has given birth to datacenters, which are large deployments of thousands of servers that work together to host modern online services. Popular online services today have billions of users spread throughout the planet, which in turn generate increasing amounts of data that needs to be processed and stored in datacenters. At the time of writing, Microsoft claims about 100 million while Google claims more than 1 billion daily active users in the search domain [123]. In the social media domain, Facebook is the leader with about 3 billion monthly active users, while Youtube closely follows with about 2.5 billion monthly active users [104]. Similarly, in the online shopping domain, Amazon claims to have over 300 million active user accounts [31].

Such planet-scale online services also have the requirement of answering the requests received from their users under a given threshold of time. Such tight latency constraints are important for providing a high-quality experience to the users, in absence of which the users might turn to a competitor service, thus resulting in customer loss. For example, for every 100ms increase

in latency, Amazon has claimed to have lost 1% in sales. Similarly, Google has claimed that a 500ms latency increase in the search results leads to a 20% traffic loss [81]. Therefore, to meet these latency requirements, the datacenter operators typically host a significant fraction of the overall data in memory devices such as DRAM to benefit from its low latency. The same can be illustrated by modern cloud platforms which provide TBs of DRAM per server [59, 112]. As DRAM is expensive [47], it is important to use the available DRAM capacity efficiently. Virtual Memory has been used in computing systems for decades to provide ease of development to the programmers while allowing the OS to efficiently manage the underlying memory capacity.

1.1 Virtual Memory in Datacenters

Virtual Memory (VM) is a classic programming abstraction that provides an abstract representation of memory for easier programming without exposing the underlying physical constraints of memory capacity and technology, and simplifies the programming model by obviating the need for programmer-orchestrated data movement among memory devices and persistent storage [19], offers “a pointer is a pointer everywhere" semantics across multiple CPU cores [95] and accelerators (e.g., GPUs [94], FPGAs [71], NICs [86], and ASICs [69, 96]). VM also forms the foundation of access control and memory protection mechanisms ubiquitous to modern computer systems security and are the building blocks of other advanced mechanisms like trusted execution environments.

With the rise of datacenter computing, the services provided by large-scale datacenters are becoming increasingly popular and are serving up to billions of users across the planet. With the increasing number of users, the memory requirements of such datacenter services are increasing as well. Therefore, systems designers are building cache and memory hierarchies with increasing capacity to capture the ever-increasing working sets of datacenter workloads and improve the performance of datacenter services [15, 63, 64, 95], as evidenced by recent work on die-stacking, chiplets, DRAM caches [60, 61, 125], and non-volatile byte-addressable memories [28, 50]. However, these high-capacity cache and memory hierarchies also shift the performance bottleneck to virtual-to-physical address translation, which can pose an

overhead of about 10-30% on the overall system performance [15, 64, 95, 96]. Therefore, VM today is plagued with crippling performance and complexity challenges that undermine its programmability benefits.

Hosting the data from online services in memory comes with five key challenges for VM in the post-Moore era.

1. *Increasing memory capacity*: as the online services become popular and the number of users increases, the online services require increasing memory capacity in the datacenter servers hosting the service. Increasing the overall memory capacity in a server increases both the material and energy cost associated with the server, and also puts additional performance pressure on the hardware and software subsystems (especially VM) as they need to manage a larger memory capacity. Finally, with the end of Moore's law, increasing the memory capacity using DRAM directly is prohibitive and requires innovative solutions.

2. *Heterogeneous memory hierarchies*: with the increasing difficulty in increasing the memory capacity using DRAM, datacenter operators have started exploring Heterogeneous memory technologies to supplement the memory capacity. For example, the introduction of commercial Storage Class Memory [45, 56] has made it possible to increase the memory capacity for a fraction of the overall memory cost. Similarly, the introduction of memory pooling technologies such as RDMA and CXL [33, 86, 99] allow using the memory of neighboring servers as an extension of the local memory. However, such heterogeneity in the memory hierarchy introduces challenges regarding data movement and coherence among the memory devices and servers, as now multiple servers are required to have a coherent view of shared memory using a collection of memory devices. In particular, VM requires introducing new synchronization primitives that can ensure coherence among the various copies of VM-associated metadata, which might be present in multiple places in the same server, or even across servers.

3. *Virtualization in the cloud*: with the increasing demand for resources by third-party

online service providers, datacenter operators have resorted to hosting multiple users/services per server (multi-tenancy) in order to increase the overall utilization of the datacenter servers and reduce the total cost per user. However, co-locating multiple users on the same physical machine results in both performance problems and security concerns among the users. Therefore, datacenter operators have tried to build primitives to provide the illusion of isolation to users. VM is the basis of modern virtualization techniques as it provides isolation among various processes running in the same physical server, and has been extended to provide isolation among virtual machines co-located on the same physical server as well. However, such VM-based techniques have their own performance overheads [15, 39, 64] which scale poorly with the overall users and resources (such as memory capacity) in the system.

4. *Emerging workloads*: with the growth in the number of users of online services, and the increasing cost of datacenter servers, online service providers along with datacenter operators have started exploring new ways of deploying online services in order to decrease the cost of hosting the online services and increase the ease of deployment. This search has led to paradigms such as microservices [137] and serverless computing [124]. However, these paradigms have their own performance challenges as they allow co-locating multiple microservices together, or running short-lived serverless jobs which leads to contention because of tracking increased address translation entries in the current VM implementations, compared to traditional long-running monolithic workloads.

5. *Confidential computing*: the increase in the number of users of online services and the services/users co-located on a physical datacenter server has accelerated the development of confidential computing platforms such as SGX [57] which have already become key in hosting various sensitive services in the domains of healthcare, finance, streaming services, etc. All modern confidential computing techniques are built on top of VM, as VM provides the basic memory isolation primitives among processes running in the same system. However, as the data processed by such platforms increases, it places

a significant performance burden due to increased memory capacity when providing such confidential services using VM.

As the above challenges explicitly rely on VM for functionality, we need to ensure that VM performance can scale with the increasing demands in modern datacenters. In this thesis, we focus on the increased memory capacity requirement which is the underlying issue in all the above challenges. To improve VM performance, computer and system architects continuously design complex address translation hardware and Operating System (OS) support that requires significant silicon resources and sophisticated implementation. Such complex support poses verification burdens despite which design bugs are still common [78]. Individual CPU cores (and recent accelerators) integrate large two-level TLB hierarchies with thousands of entries, separate TLBs at the first level for multiple page sizes, and skew/hash-rehash TLBs at the second level to cache multiple page sizes concurrently [30, 88, 108]. Such large TLB hierarchies and multiple page sizes necessitate a staggering amount of OS logic to defragment memory and create huge pages [92, 93, 134, 135] and heuristics to determine when to create, break, and migrate them [73, 85, 117, 118]. As huge pages can lead to performance pathologies and hence are not a panacea, processor vendors also integrate specialized MMU cache per core to accelerate the page-table walk [13, 18]. Specialized per-core TLBs and MMU caches, in turn, necessitate sophisticated coherence protocols in the OS (i.e., shootdowns) that are slow and buggy, especially with the adoption of asynchronous approaches to hide shootdown overheads at a higher core and socket counts in modern servers [10, 72, 79].

1.2 Thesis Goals

The primary goal of this thesis is to remove the performance bottleneck caused by address translation in case of large memory capacity in datacenter servers. We aim to improve the performance VM subsystem while ensuring POSIX compatibility, thus making sure that the developers do not need to rewrite their application code. Moreover, as memory capacity is both limited and expensive, we do not want to rely exclusively on techniques such as huge pages which might lead to internal memory fragmentation compared to a fine-grained

management of memory using fixed-size pages. If we can improve the performance of address translation for large memory capacities, then it will in turn benefit the overall VM performance for all the five key challenges explained before.

We intend to introduce a holistic redesign of the VM subsystem while prioritizing software compatibility and efficient memory capacity utilization. We start by focusing on the management of both the virtual and physical address spaces as is done in the traditional systems. We observe that the virtual address space inherently contains data in the form of flexible and logical data sections called Virtual Memory Areas (VMAs), which are a modern incarnation of age-old segments [132]. In contrast, the physical address space is managed in terms of fixed-size pages which allow utilizing the overall memory capacity efficiently. The mapping of each VMA into multiple fixed-size pages results in the creation of a large amount of address translation metadata which has to be consulted every time the virtual-to-physical address translation is required, thus creating a performance bottleneck. To avoid this bottleneck, we propose dividing the traditional virtual-to-physical address translation logic into a fast and slow component. The fast component represents only the memory management done in the logical address spaces (such as virtual address space) and can be used to perform common case permission checks and cache hierarchy accesses, while the slow component represents the capacity management performed in the physical address space representing the backing memory devices. Such a division in the address translation can be caused by introducing an intermediate address space in the system.

Thesis Statement:

A VMA-based intermediate address space can be used for common case access control and cache hierarchy accesses while requiring the page-based physical address space for only capacity management.

1.3 Thesis Contributions

In this thesis, we aim to circumvent the performance problems of the existing VM implementations by using the following key insights:

1. Redesign microarchitecture for address translation using existing programming abstractions such as VMAs that already exist in the traditional OS and application toolchains in order to maintain POSIX compliance, thus ensuring no effort for the application developers and quick adoption from the OS developers.

2. Decouple address translation into two steps where the first step is invoked for all memory accesses, and therefore mandates (lean) specialized microarchitecture for high performance, while the second step is invoked infrequently and can rely on the existing cache hierarchy resources instead of requiring specialized structures.

3. Instead of completely delegating complicated functionalities to the microarchitecture (in particular exception handling), we can offload some of the functionality to the OS using an OS + microarchitecture co-design. We can also provide formal specifications to the OS developers to help ensure correctness in order to prevent hard-to-find bugs such as the ones that have plagued TLB shootdown mechanisms [9] for years.

We propose the Midgard — the original name for the middle realm between Asgard and Helheim in Norse mythology — as an intermediate address space between the virtual and physical address spaces, which relies on fusing the existing OS abstraction of VMAs into hardware for the first time. As applications view memory as a collection of a few flexibly-sized VMAs with specific permissions, it is possible to create a unique Midgard address space where VMAs of various processes can be mapped after deduplication. The unique Midgard address space serves as a namespace for all data in the coherence domain and cache hierarchies, and thus all cache hierarchy accesses must require a translation from the program's virtual address to the unique Midgard address. Translation from Midgard to physical addresses is only required when the required cache block is not found in the cache hierarchy and has to be

fetched from physical memory.

The Midgard idea is inspired in part by prior work on virtual cache hierarchies [23, 24, 43], which reduce address translation pressure by deferring the need for physical addresses until physical memory access. Unfortunately, they also create implementation complexity by exposing synonyms/homonyms or compromising programmability due to reliance on inflexible fixed-size segments. While approaches like single address space OSes [70] tackle some of these problems (i.e., removal of synonyms/homonyms), they require the recompilation of binaries to map all data shared among processes into a unified virtual address space. Instead, to achieve our goal, we provide a programmer-transparent but OS-managed intermediate address space, where the OS ensures the resolution of any synonym or homonym using the existing abstraction of VMAs. Thus, the intermediate address space can be used to index the cache hierarchy without the shortcomings of homonyms/synonyms or segments [132], and also enables a fast translation from virtual addresses to the intermediate addresses using VMAs and requires a slow translation from intermediate addresses to physical addresses using pages only when accessing the physical memory.

With the introduction of Midgard as an intermediate address space, the translation from virtual to Midgard addresses can be accomplished using translation caching structures much smaller than TLBs because there are far fewer frequently-used VMAs (~10) than pages in real-world workloads. In contrast, the translation from Midgard to physical addresses still remains slow, but is required infrequently and allows utilizing the trends of increasing cache hierarchy capacity. Larger cache hierarchies traditionally amplify VM overheads, but instead, they can mitigate VM overheads by absorbing most of the page-based translation activity and obviate the requirement of specialized translation hardware. While large 1000-entry (~16KB SRAM) per-core TLBs fail to meet the performance requirements for address translation in traditional systems and require impractically larger structures, Midgard allows the page-based translation to be performed using zero or very few TLB entries, thus reclaiming the 100KB-to-MB of SRAM dedicated towards TLBs in modern manycore systems [128] which anyways fails to meet the performance requirements of address translation today. Therefore, instead of

amplifying the address translation overheads, larger cache hierarchies can now be leveraged to *reduce* them. Our trace-based evaluation shows that for large-capacity cache hierarchies, Midgard's address translation overhead drops to near zero as the working sets fit in the cache hierarchy, while traditional TLBs suffer even higher degrees of address translation overhead. Overall, the introduction of the Midgard address space can provide the benefits of virtual cache hierarchies without homonym/synonym problems.

The introduction of the Midgard address space delays the page-based address translation near memory, which can give rise to complicated exception handling scenarios in infrequent cases. In particular, the traditional microarchitecture design relies on detecting any VM-related exceptions in the pipeline itself in a *precise* manner as the address translation and permission check is tightly integrated with the pipeline. The same assumption allows the microarchitecture implementation to retire stores before completion which leads to a significant performance boost in the pipeline and forms the basis for modern relaxed memory consistency models. However, when address translation is performed away from the cores, then the above assumption breaks and the same optimizations cannot be naively applied. While there are well-understood post-retirement speculation techniques that can handle such exceptions without losing performance, they also impose an overhead of 10-20KB SRAM per core to maintain the state for optimistically retired stores. Unfortunately, such silicon requirements will undermine the benefits received from removing the traditional TLBs. Therefore, to maintain the original system performance using early store retirement without spending significant silicon resources, we propose handling the exceptions *imprecisely* while offloading part of the microarchitectural functionality to the OS which can be carried out easily in the infrequent cases when exceptions are generated by the page-based translation. As such a OS + microarchitecture co-design will interact with complicated microarchitectural concepts such as memory consistency models, ensuring correctness is of primary importance here to prevent hard-to-find bugs such as the ones that have plagued TLB shootdown mechanisms [9] for years. Therefore, to help the OS developers ensure correct behavior, we develop a complete formalism and proofs to show that the design for imprecise exceptions is compatible with popular memory consistency models such as Processor Consistency (PC) and Weak Consis-

tency (WC). We then describe the detailed hardware-software co-design required to bring the imprecise exception idea to fruition. However, the formalism allows other system designers to come up with their own design while ensuring correctness.

Overall, this thesis explains the various modifications required to support Midgard in the OS, requirements for the virtual-to-Midgard address (logic-side) translation, and requirements for the Midgard-to-physical address (memory-side) translation, along with the OS + microarchitecture co-design for imprecise exception handling. The OS requires tracking the VMAs for each process, mapping all the unique VMAs as MMAs in the Midgard address space, maintaining architectural VMA tables that track VMA-to-MMA mappings along with the application permissions, and maintaining system-wide page tables that track the page-granularity mappings from the Midgard to physical address space along with the OS-specified permissions. The logic-side translation requires hardware caching support to store the mappings for recently-used VMAs, fast access time for both the address translation mappings and the cache hierarchy, and supporting precise exceptions in case of faults in the logic-side and memory-side translations. The memory-side translation instead relies on the cache filtering for infrequent invocations and requires techniques for performing the memory-side page-table walk with or without specialized address translation caches, along with coherence among the memory-side page-table entries.

We evaluate Midgard by quantifying its performance characteristics over cache hierarchies ranging in size from tens of MBs to tens of GBs and show that even modest MB-scale SRAM cache hierarchies filter the majority of memory accesses, leaving only a small fraction of memory references for translation from Midgard to physical addresses. We characterize VMA counts as a function of dataset size and thread count and confirm that low VMA counts mean a seamless translation from virtual to Midgard addresses. Using average memory access time (AMAT) analysis, we show that LLC capacities in the tens of MBs comfortably outperform traditional address translation and that at hundreds of MBs, they even outperform huge pages. In our evaluation, Midgard reaches within 5% of address translation overhead of traditional 4KB-page TLB hierarchies for a 16MB LLC and breaks even with 2MB-page TLB

hierarchies for a 256MB LLC. Unlike TLB hierarchies exhibiting higher overhead with larger cache hierarchies, Midgard's overhead drops to near zero as secondary and tertiary data working sets fit in the cache hierarchies. Finally, we show that even for pessimistic scenarios with small LLCs, Midgard can be augmented with modest hardware assistance to achieve competitive performance with traditional address translation. To showcase the imprecise exception handling idea, we further develop a full-system RISC-V prototype of an out-of-order core that supports generic imprecise exceptions and runs Linux. We run litmus tests for memory consistency models to show that our implementation is compatible with the RISC-V memory consistency model. Then we provide an end-to-end evaluation of various benchmarks to show that the imprecise exceptions can work successfully, and are able to maintain similar overall performance as in the traditional system.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 explains the importance of the VM abstraction, the traditional implementation of the VM subsystem, scaling problems with large memory capacities, and various classes of previous proposals that have aimed to improve VM performance. Chapter 3 to chapter 7 present the contributions of this thesis.

- Chapter 3 provides a brief overview of Midgard along with the high-level design and the interface required to support Midgard.
- Chapter 4 explains the exception handling logic required in Midgard.
- Chapter 5 explains the VMA-granularity logic-side address translation in Midgard that provides application-level access control and access to the cache hierarchy.
- Chapter 6 explains the page-granularity memory-side address translation in Midgard that provides capacity management in the physical address space and provides access to the memory device.
- Chapter 7 describes our evaluation that quantifies the benefits of Midgard compared to

traditional systems.

Finally, chapter 8 describes future research directions with Midgard and concludes the thesis.

1.4.1 Bibliographic Notes

This thesis was conducted under the supervision of my advisors, Babak Falsafi and Abhishek Bhattacharjee. Portions of it are a product of collaboration with Yuanlong Li, Qingxuan Kang, Atri Bhattacharyya, Yunho Oh, and Mathias Payer. The design, implementation, and evaluation of Midgard is based on a conference paper published in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)* in 2021 [44]. The chapter 4 on the exception handling in Midgard is based on a conference paper published in *Proceedings of the 50th International Symposium on Computer Architecture (ISCA)* in 2023 [46].

2 Virtual Memory

As computing has evolved over the last six decades, Virtual Memory (VM) [19] has played an essential role as a programming abstraction for computing systems. VM allows programmers to write programs without being concerned about the underlying memory subsystem, forms the basis of modern security and virtualization technologies, and is responsible for the efficient utilization of memory capacity present in a system. VM has been so successful as a programming abstraction that today it is ubiquitous in computing systems of all scales such as datacenter servers, desktops, mobile devices, and even various forms of embedded computing. With the rise of datacenter computing [14] and continued scaling of memory capacity [59, 112] coupled with the end of Moore's law [32, 122], the existing implementation of VM is expected to result in a high performance overhead [15, 39, 64] which will dwarf the programming benefits it provides, therefore requiring a careful redesign to reduce the implementation overhead. This chapter explains the benefits of VM, its current design and implementation, the overheads associated with scaling VM implementations along with the growing memory capacity, and various classes of proposals that have tried to mitigate the overheads associated with VM.

2.1 A Fundamental Programming Abstraction

VM is the de-facto programming abstraction in almost all computing systems today because of the benefits it provides. We now explain the various features of VM that make it so successful

as a programming abstraction.

2.1.1 Programmability and Portability

VM allows the programmers to write programs while ignoring the specifics of the underlying memory subsystem. VM provides the abstraction of a *virtual address space* to the programmers which is logical and not tied to the physical memory capacity of the system. The virtual address space is designed to have a large capacity (e.g., modern CPUs support 57-bit address spaces [55] that can store up to 2^{57} bytes of data) which is considerably more than the typical memory usage of programs today. Therefore, the programmer (in co-operation with the linker and compiler) is able to use the whole virtual address space freely without accounting for the capacity restrictions of the underlying physical address space dictated by the amount of physical memory in the system. The availability of more than explicitly-required memory capacity allows the programmer to easily construct a variety of logical data sections (e.g., heap, stack, code, files, libraries) which can be placed in different parts of the virtual address space. Such a placement also allows special programming properties such as the stack starts at a high address and grows down while the heap starts at a lower address and grows up, while being practically ensured to never collide with each other. Finally, as the programs are written using the virtual addresses, they can be easily ported to other machines without any memory-addressing-related modifications because every machine can provide the abstraction of a virtual address space irrespective of the varying physical capacity of memory.

2.1.2 Isolation and Protection

VM forms the basis of modern security mechanisms in computing systems as it provides memory security inside and among various processes running on the same system. In modern systems, each process has a private virtual address space and can only generate addresses belonging to its own virtual address space. Therefore, while multiple processes might be concurrently running in a system, no process can generate virtual addresses to access data belonging to another process in the system unless the data is explicitly shared using files or

shared memory regions, providing every process an isolated environment as if it is the only process running in the system. However, the bugs present in the code running in a process can still cause it to malfunction and destroy critical data. Morever, while memory isolation directly provides security and ease of programming, there are various other interfaces that allow the processes to interact with other processes in the system, or even those running on other machines. Such interactions can turn malicious if the external processes try to abuse the provided interface to exploit bugs and security vulnerabilities present inside a particular process. To counter such security issues, VM further provides protection mechanisms that can limit the type of memory operations that can be performed on a particular virtual address. VM does so by enforcing permission checks with each virtual address, where the permissions can be provided by the programmer in cooperation with the toolchain. For example, to protect against attacks that attempt to inject malicious code in the process, the code section of the process is given only executable and read permissions, thus ensuring that the code can never be re-written while the process is running. Similarly, the data-containing sections such as heap and stack are typically not given the executable permissions so that malicious code disguised as data cannot be executed from such sections. Overall, VM provides security mechanisms both inside and among virtual address spaces.

The isolation and protection mechanisms provided by VM also form the basis of virtualization [22], which is a foundational technology required for cloud computing today. Using further extensions in OS and hardware, VM provides the foundation of memory virtualization where the guest virtual machine is given the abstraction of a physical memory device, while the underlying physical memory is managed only by the host OS. The guest OS runs its own VM implementation inside the guest virtual machine, maintains its own guest virtual address spaces and maps them to a guest physical address space which is provided by the host OS. The host OS treats the guest physical address space similar to its other virtual address spaces and maps data from it to the actual physical address space. Therefore, while the guest OS sees an abstract physical address space, only the host OS can control and manipulate the actual physical address space that represents the physical memory present in the machine. In this way, VM ensures complete memory isolation even when running multiple virtual machines

on the same physical machine while sharing the underlying physical memory.

2.1.3 Memory Management and Over-Subscription

With the rise of TB-scale memory hierarchies in datacenter servers [59, 112], memory accounts for a significant portion of overall costs [7], making it even more important to utilize the memory capacity well and without wastage. VM allows the OS to manage the memory capacity efficiently without programmer intervention while running a variety of concurrent processes. Modern VM implementations manage the underlying memory by dividing the overall memory capacity into small units called pages that can be mapped to virtual addresses when the process requests additional memory. Using a page-based VM implementation ensures that even if there is a single unused page present in memory, it can be used independently of which process or virtual address it has to be mapped to. VM also allows for the over/under-subscription of memory, which means that processes can both under utilize the memory capacity, or allocate more memory than the total capacity of memory present in the system. In the case of memory over-subscription, the extra data that cannot be contained in memory is transparently spilled to storage devices such as disks, or can be compressed such that it takes less space [74]. For example, modern OSes use storage devices as swap devices where they store cold data that has not been accessed from memory in a while. Similarly, the OS allows processes to memory map a file which might be present in a storage device, and then the data from the file is transparently brought to memory only on demand when the process wants to access it. Therefore, the VM abstraction can transparently extend the memory capacity using storage devices, though at the cost of overall performance as storage devices accesses are slower than memory accesses.

2.1.4 Performance Optimizations

VM also allows the OS to perform various performance optimizations transparently to the application. In particular, the OS can detect the data that is frequently used by the applications and transparently move it to memory, while moving the cold data to storage devices. Doing

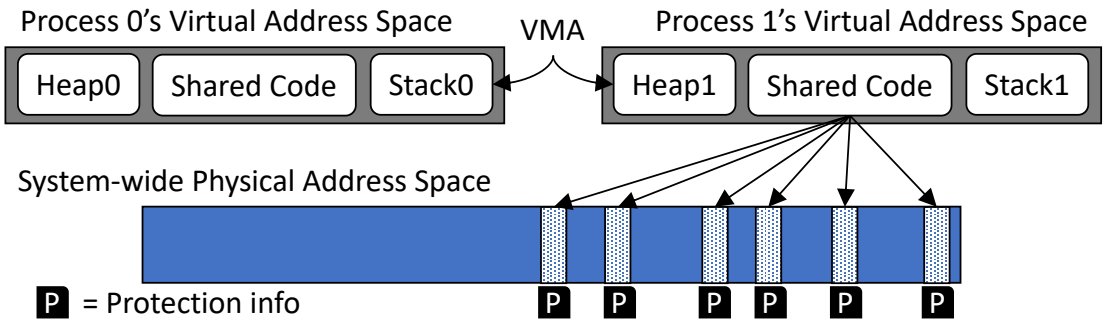


Figure 2.1: VMAs in the virtual address space consist of fixed-size pages, which may or may not be mapped to frames in the physical address space by the OS.

so ensures that common case data accesses are served from memory, which provides high application performance. With the projections of the end of memory scaling [47] and the expected proliferation of heterogeneous memory devices [33, 47, 54, 113] in future datacenter servers, such techniques will be heavily used to move data between faster memory devices such as DRAM, and slower memory devices such as Persistent Memory [45, 134] or Flash [47]. The OS can also predict the memory access patterns, and prefetch the required data to memory, thus benefitting performance. Overall, VM allows the application to ignore the layout of the underlying memory subsystem, while moving data among memory and storage devices transparently to benefit the application performance.

2.2 Virtual Memory Implementation Today

As explained in section 2.1, existing OSes (Linux, MacOS, Windows) implement VM by providing a private virtual address space to each process, where the size of the address space is independent of the actual physical memory capacity present in the system. The memory capacity of the system is exposed only to the OS as a capacity-constrained physical address space, and is not directly exposed to the applications, giving them the abstraction of near-infinite memory capacity. Thus, the VM abstraction is appropriate for programmability as the typical programmers do not directly need to cater to the resource constraints in the system.

2.2.1 Virtual Address Space

The OS provides a private virtual address space to each process in the system. The maximum size of the virtual address space is dictated by the underlying CPU architecture. For example, modern x64_64 Intel CPUs provide a virtual address space of 57 bits [55] that can contain a maximum of 2^{57} bytes. The underlying CPU architecture in co-ordination with the OS ensures that any generated virtual memory addresses are constrained to the address space of the corresponding process. Therefore, each process can only see one large, private virtual address space, creating the abstraction that it is the sole process in the entire system, thus providing isolation. As each process can only access addresses in its own virtual address space and therefore cannot access the virtual address space of any other process, VM provides security through isolation among processes. VM also allows the processes to specify access permissions for data, which are then checked during execution, thus preventing illegal memory accesses due to bugs or malware.

The OS in co-ordination with the toolchain (e.g., compiler, linker) organizes the data in the virtual address space in terms of Virtual Memory Areas (VMAs), as shown in Figure 2.1. Every VMA represents a logical data section that represents semantic data useful to the program. Here is a list of commonly used VMAs in programs today:

- Code: contains executable instructions
- Text: contains constant data used in the program
- Data: contains initialized data variables
- BSS: contains uninitialized data variables
- Stack: contains data allocated in the scope of an executing function
- Heap: contains data allocated dynamically during program execution
- Files: memory-mapped files that might be in use by the program
- Libraries: each library in use can correspond to multiple VMAs

Based on the source of the data in the VMAs, they can be classified into two categories: file and anonymous VMAs. VMAs such as code, text, and files load their data from some file stored in the system, and therefore are categorized as file VMAs. In contrast, VMAs such as data, bss, heap, and stack contain data that is generated dynamically and is only relevant to the execution of the program without any association to a backing file, are called anonymous VMAs. Every VMA is represented using a base and bound virtual address which indicates its size and position in the virtual address space. Every VMA also incorporates permission bits (r - read / w - write / x - execute) specified by the program for its semantic correctness that are exposed by the OS to the CPU, and are checked for every memory access to the corresponding VMA. While some VMAs maintain their size and position throughout their lifetime (e.g., code, text, data, bss), other VMAs can grow during the program execution (e.g., heap, stack, files). In particular, as most of the data generated by the program is placed in the stack and heap, they need to be positioned in the virtual address space such that they do not collide while growing. To achieve the same, the heap is placed at lower virtual addresses and grows towards high virtual addresses, while the stack starts at high virtual addresses and grows towards lower virtual addresses. While modern virtual address spaces are large enough to avoid collisions in most scenarios, collisions can still occur in cases when there are multiple threads associated with the same process and every thread has its own stack. In such a case, there can be a collision between the heap and a stack, or two stacks themselves [115]. Modern OS and compilers try to detect such problems by checking addresses during malloc(), sbrk(), and mmap() system calls, or by placing guard pages between VMAs to identify overflows. Overall, placement of VMAs in a logical address space is not a trivial problem, but can be solved using known heuristics and rigorous checks.

2.2.2 Physical Address Space

In contrast to per-process private virtual address space, there is only one physical address space in a system as it represents the underlying physical memory devices. The physical address space is only visible to, and completely controlled by the OS (or both the host and

guest OS in case of virtualization). The capacity of the physical address space reflects the aggregate memory capacity present in the system, and therefore is much smaller than the capacity of the virtual address spaces. For example, a system with 1TB of overall memory capacity will have a physical address space of 40 bits, which can thus contain a total of 2^{40} bytes worth of data. As the physical address space is constrained in capacity, it is very important to efficiently utilize the entire space. However, there are two types of inefficiencies that arise when managing the physical address space: external and internal fragmentation.

Earlier incarnations of VM implementations featured segments [19, 132] which were data sections defined by the programmer and toolchain for organizing the virtual address space, and were conceptually similar to VMAs. In such incarnations, the segments were directly mapped to the physical address space, therefore requiring that there is enough contiguous free space present to map the whole segment. However, as the segments can grow and shrink during the program execution, such an implementation gives rise to external fragmentation. External fragmentation indicates the wastage of physical memory space when there is a small space left in between two mapped segments, but that space is not enough to map an individual segment on its own, and is thus rendered unusable. To avoid the problem of external fragmentation, VM implementations have now moved towards page-based techniques.

Modern VM implementations use a page-based approach when managing the physical address space. In such an approach, the physical address space is divided into fixed-size units called pages or physical frames, and then every physical page can be mapped to a virtual page whenever the corresponding application asks for more memory. Page-based VM guarantees that even if there is a single page free somewhere in the entire physical address space, it can be mapped to a virtual page when needed and therefore prevents external fragmentation by design. However, if the page size is chosen to be large, then there might be internal fragmentation as the program might not be able to use the whole page efficiently. But decreasing the page size too much to minimize internal fragmentation will result in too many pages which will lead to a management and bookkeeping issue. In modern platforms, the page size has been typically adopted to be 4KB [128, 130], while there also exist commercial products which

use a larger page size such as Apple M1 [1] using 16KB pages. Apart from these standard page sizes, modern commercial products also support using larger page sizes such as 2MB and 1GB, both in the OS and in hardware [128, 130].

2.2.3 Address Translation

VM implementations require the program to access data in memory using virtual addresses, but the CPU must be able to access the corresponding data in memory using physical addresses. To do so, modern systems incorporate the process of address translation for every memory access where virtual addresses are translated into corresponding physical addresses, and the associated permission bits are checked. However, it is the responsibility of the OS to maintain mappings between virtual and physical addresses which can then be exposed to the CPU for address translation. To do so, VM allows the OS to perform memory management without requiring application support.

As mentioned before, the virtual address space contains data in terms of flexible VMAs, where each VMA represents a logical data section and has the same properties throughout. In contrast, to efficiently utilize the limited memory capacity in the system, the OS divides the physical address space into fixed-sized pages to avoid external fragmentation while encountering very little internal fragmentation. To map the VMAs to physical pages, the VMAs are further divided into the same fixed-sized pages, while ensuring that the VMA size is always a multiple of the page size for ease of implementation. Then the OS maps the virtual pages to physical frames as required (as shown in Figure 2.1). The OS tracks the virtual-to-physical mapping information for each process separately by using process-private page tables in existing systems. The page table is constructed as a radix tree where each tree node is itself a page, and contains entries representing sub-trees in case of intermediate nodes or pages in case of leaf nodes. The main benefit of the radix tree structure is that it allows saving space by not representing unallocated virtual addresses in the virtual address space of a process. Finally, VM also facilitates communication among various devices using memory as different parts of the virtual address space for each process can be mapped to different memory devices.

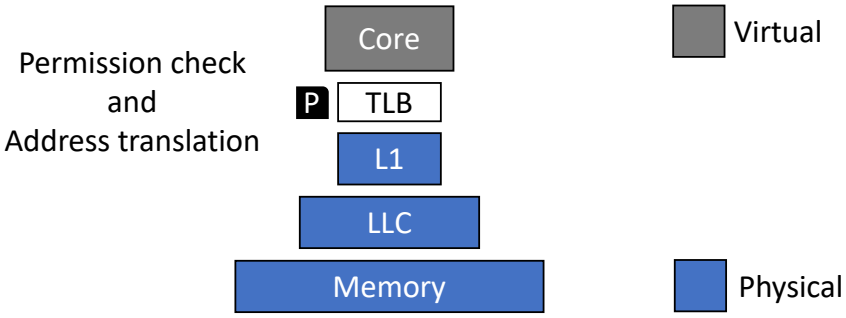


Figure 2.2: Existing systems use physical addresses to index the cache and memory hierarchy.

For example, GPUs today use a unified virtual address space where each virtual address can either be mapped to the GPU-private memory or the host-CPU memory, and pages can be moved between these memory devices for faster performance.

Apart from the translation from virtual to physical addresses, the OS also needs to track the permissions to be enforced for each memory access. In a shared memory system, each process can have different permissions attached to the same data, therefore the OS tracks the permissions in the per-process radix page tables as well. In determining the memory permissions per page, the OS accounts for the permissions associated with the VMA the physical page is mapped to. However, the OS also requires considering additional memory permissions that are generated because of the memory management. For example, if the OS swaps out a page, then the associated permissions become invalid irrespective of the permissions associated with the VMA. Similarly, if the OS is performing the copy-on-write optimization with a particular page, then the associated permissions are kept as *ro* or read-only, even though the VMA permissions might permit writes using *rw* or read-write. Therefore, the final permissions enforced by the OS are the intersection of the VMA permissions and the memory management permissions. These permissions are represented in the leaf page table entries along with the physical addresses.

In systems today, while the core only works with virtual addresses (e.g., pointers for loads and stores), the obtained virtual address needs to be translated to the corresponding physical address before the cache hierarchy or memory can be accessed (as shown in Figure 2.2).

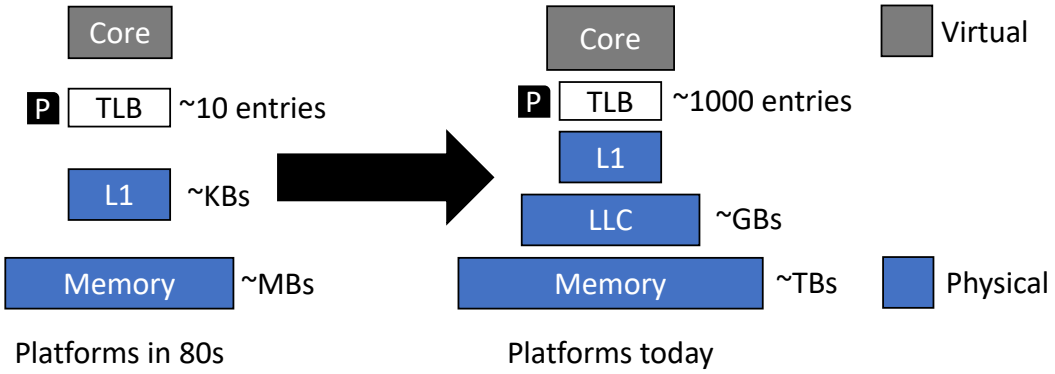


Figure 2.3: While 10s of TLB entries were enough to cover KBs and MBs of cache and memory in the 80s, TLBs today contain 1000s of entries but cannot provide enough coverage for the GBs and TBs of cache and memory available today.

Therefore, all the components in the cache/memory hierarchy work directly with the physical addresses, for example, the tags present in the cache hierarchy contain physical addresses. As the virtual-to-physical address mappings are stored in the radix page tables by the OS, each translation might require performing page-table walks, thus typically generating up to four or five [55] memory accesses for each data access. In order to accelerate this translation step, the CPU employs Translation Lookaside Buffers (TLBs), which are hardware-managed caching structures to store a subset of virtual-to-physical mappings for fast access. If the TLBs contain the required translation, the page tables do not need to be consulted; otherwise, a page-table walk is required to fetch the required entry and obtain the corresponding physical address. Thus, well-functioning TLBs can eliminate the excessive memory accesses to the page tables and provide fast translation for most memory accesses.

2.3 Scaling Problems

With the growth in modern data-oriented services, the applications are becoming increasingly memory-intensive. As online services hosted in the datacenter become popular, they have to scale to serve billions of users across the planet. As the growing users of such online services generate data rapidly, the memory capacity of servers operating on these datasets is growing as well, already reaching 10s of TBs [59, 112]. The cache hierarchy capacity in datacenter servers

is also increasing proportionally along with the overall memory capacity. As shown in Table 2.1, Intel Knights Landing [113] features a 16GB off-chip DRAM cache, AMD Zen3 [128] features 256MB of SRAM cache capacity shared among 64 cores, Intel Kabylake [130] features 128MB of eDRAM cache on the memory side, and AMD Ryzen 7-3D [8] uses 3D integration of SRAM to provide a 96MB L3 vertically-integrated with the cores. Moreover, the upcoming Intel Sapphire Rapids [54] promises to provide 64GB of stacked HBM2 memory using 4×16GB HBM2 stacks, where the HBM2 can be configured as a hardware-managed cache (similar to Intel Knights Landing) in conjunction with a backing DDR5 memory pool. However, the cache and memory capacity growth will be futile if the VM implementation cannot scale accordingly to allow fast access to cache and memory.

As the memory capacity increases, the overall number of pages present in the system increases, thus requiring more TLB entries to capture the hot translations. Reflecting the increase in the memory and cache hierarchy capacity, the TLBs present in the existing cores have also become bigger (as shown in Figure 2.3). While Intel Pentium 4 [2] featured a 64-entry TLB to cover its 256KB LLC, existing CPUs such as Intel Kabylake [130] and AMD Zen3 [128] provide 1.5K and 2K TLB entries respectively to cover their 128-256MB cache capacity. The Apple M1 Firestorm [1] provides 3K TLB entries per core. However, when using the standard 4KB pages, even 3K entries can provide total coverage of only 12MB. Unfortunately, while the cache hierarchy and memory capacity reach GBs and TBs, the TLBs cannot continue scaling because of the limited SRAM resources due to the end of Moore's law [32]. The mismatch between the memory capacity and resources dedicated to VM results in a significant fraction of time spent on VM activities (e.g., frequent TLB misses lead to page-table walks, thus wasting CPU cycles), thus resulting in high a 20-40% overhead [15].

With the increase in the overall memory capacity and the number of pages, the process-private virtual address spaces also need to grow to accommodate more physical pages. E.g., Intel has recently announced that they are increasing the virtual address space size from 48 bits to 57 bits [55] to accommodate the emerging TB-scale memory capacity servers. Larger virtual address spaces require larger page tables with more levels in the radix tree. E.g., going from 48

Product	Year	Cores	Cache capacity	TLB size	Coverage
Intel P4 [2]	2000	1	256KB SRAM	64	256KB
Intel KabyLake [130]	2016	4	128MB eDRAM	1536	6MB
Intel Knights Landing [113]	2016	72 (36x2)	16GB HBM	256	1MB
Apple M1 [1]	2020	8 (4+4)	16MB SRAM	3096	48MB*
AMD Zen3 [128]	2021	64 (8x8)	256MB SRAM	2048	8MB
AMD Ryzen 7-3D [8]	2022	8	96MB 3D-SRAM	2048	8MB
Intel Sapphire Rapids [54]	2023	56 (14x4)	64GB HBM2	2048	8MB

Table 2.1: Comparison of cache hierarchy and TLB capacity in various generation of commercial CPU products. *Apple M1 uses 16KB pages that increases its coverage, while all other products use 4KB pages.

to 57 bits requires five levels instead of four levels, thus resulting in longer page-table walk latencies and increased miss penalty for under-provisioned TLB hierarchies.

Finally, with the end of Moore's law, the increase in the memory hierarchy capacity is sustained by introducing heterogenous memory devices (DRAM, Persistent Memory, High-Bandwidth Memory, Flash). E.g., the introduction of Intel 3D-Xpoint [45] allows storing data at a lower price per byte but also results in longer access latency. Therefore, to efficiently use the heterogenous memory devices, the applications typically store their overall dataset in slower-but-cheaper devices and migrate the data to faster devices (such as DRAM) when required for higher performance. The same principle applies when migrating data to the high-bandwidth GPU memory for fast access. However, frequent page migrations among memory devices correspondingly lead to page-table modifications, leading to global broadcast-based TLB shootdowns. Recent studies have shown that frequent TLB shootdowns can become a critical performance bottleneck [72].

Unfortunately, scaling physical memory to TBs results in linear growth of the translation metadata. With 4KB as page size, 1TB of physical memory consists of 256M pages and requires individual mapping for each page. Even with a high locality in the memory access patterns, where 3% of the data captures most of the memory accesses, we still require frequent accesses to 8M mappings. While existing systems already provide thousands of TLB entries per core [1],

we still fall short by three orders-of-magnitude, with the end of Moore's law ceasing any further silicon scaling. Moreover, we cannot provision thousands of TLB entries for every piece of heterogeneous logic that comes in various sizes (small cores, GPUs, and accelerators). This mismatch of requirements and availability makes virtual memory one of the most critical bottlenecks in memory scaling.

2.4 Previous Proposals

With the rise of datacenter computing and continued scaling of memory capacity coupled with the end of Moore's law, the existing implementation of VM has become too costly for the programming benefits it provides, therefore requiring a careful redesign to reduce the implementation overhead. In this section, we go over the various classes of previous proposals aimed at optimizing VM to attain a better cost-performance ratio.

2.4.1 Contiguity in the Physical Address Space

As explained in section 2.3, the resources dedicated to VM cannot continue scaling to catch up with the growing cache and memory capacity. The high overhead of VM implementations might disrupt the VM tradeoffs, where the VM overheads are much greater than its benefits. Therefore, to maintain the address translation performance, previous proposals have focused on increasing the TLB coverage by typically increasing the memory covered per TLB entry. Increasing the memory coverage per entry requires creating larger blocks that are contiguous in both the virtual and physical address spaces, namely huge pages or direct segments, as shown in Figure 2.4.

Huge pages are larger pages with a fixed size and therefore can be understood as a set of standard pages contiguous in the physical address space where each page has the same properties. If such a condition holds, then the whole set of pages can be represented by just one entry, thus utilizing the contiguity in the physical address space (if any). Typically, huge pages of 2MB and 1GB are naturally supported by the radix page tables [19], while other

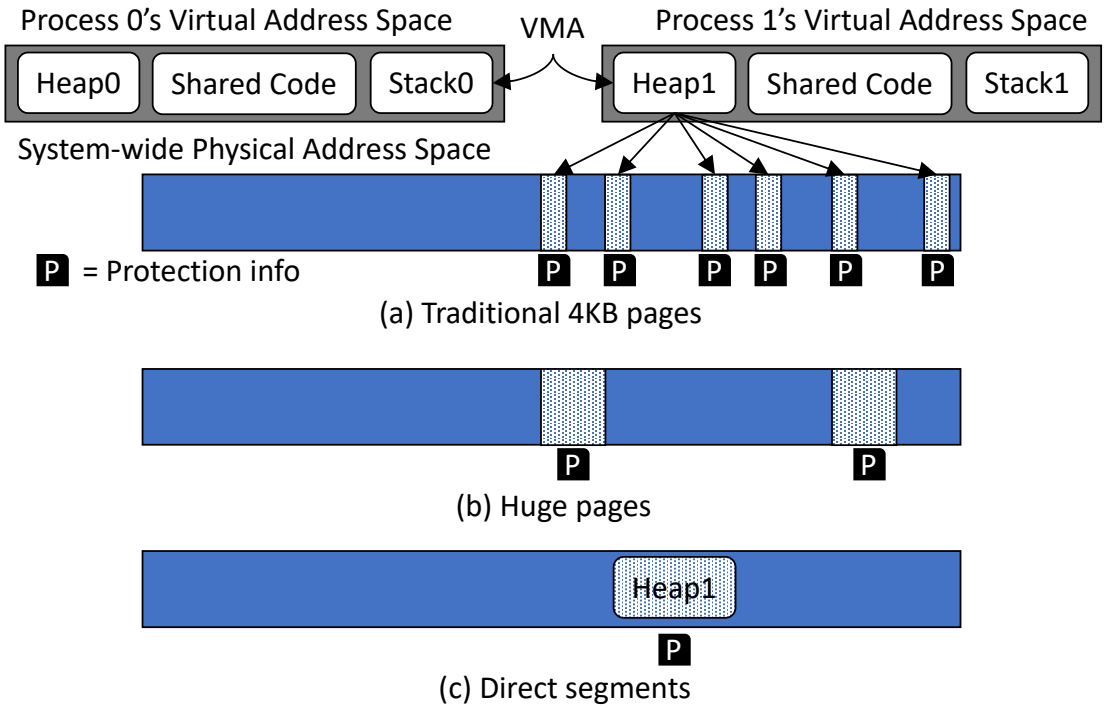


Figure 2.4: Proposals relying on contiguity in the physical address space either require maintaining (b) huge pages or (c) direct segments / VMAs in the physical address space itself.

proposals and platforms have suggested different page sizes [85, 117, 118]. Recent proposals also advocate for any-power-of-two size pages [48], which increases flexibility with which huge pages can be used, at the cost of hardware and software complexity. There are two popular policies concerning huge pages: when are the huge pages allocated to the application, and how is the huge page created? While huge-page-aware applications can be helpful where the application explicitly demands a huge page, existing OSes typically allocate huge pages transparently (eagerly or lazily) and combine the smaller pages in use by the application to huge pages. Similarly, while there have been techniques where the total number of huge pages is predecided and created at boot time, existing OSes create huge pages dynamically by moving small pages around to create the required contiguous space in the physical address space. The same huge page can also be disintegrated into smaller pages if required. There also have been proposals on directly using TLBs to exploit the contiguity present in the physical address space itself [92, 116], along with techniques to use identity mappings between the virtual and physical address space to optimize TLB lookups [50].

Direct segments generalize the huge page idea based on the contiguity in the physical address space to variably-sized regions. Such segments typically represent a logical region in the application's virtual address space, such as VMAs or large allocated regions. Direct segments can be much bigger depending on the application usage, so successfully creating a segment provides much more coverage per TLB entry. Direct segments can also be implemented to evolve (grow or shrink) as the application executes, thus ensuring that the segment mapping always exploits the maximum contiguity as exposed by the VMA/region in the virtual address space. Creating a direct segment requires moving standard pages around to create enough contiguity in the physical address space to accommodate the direct segment. Similar to huge pages, direct segments can also be maintained statically or dynamically [15, 64, 135], and can be exposed to the applications as well depending on the OS implementation.

While utilizing the contiguity in the physical address space might help increase the coverage of each TLB entry, the contiguity constructs such as huge pages and direct segments also cause memory management issues. Accommodating allocation units of different sizes together in the physical address space brings us back to a variant of the external fragmentation problem that we intended to avoid in the first place by using uniform page sizes. Huge pages can cause external fragmentation as other pages might be occupying the physical space required for allocating a new huge page even though there is enough physical space available in total. E.g., if a small page pollutes a contiguous area in the physical address space, it needs to be moved elsewhere (causing overheads) to allow the allocation of a huge page. The same problem applies to direct segments, albeit in a pronounced manner. Therefore, similar to external fragmentation, it might not always be possible to allocate a huge page because the space in the physical address space is not cleanly organized and can therefore cause wastage of space. Finally, huge pages and direct segments also inhibit finer granularity management of the physical memory, e.g., during migration across memory devices.

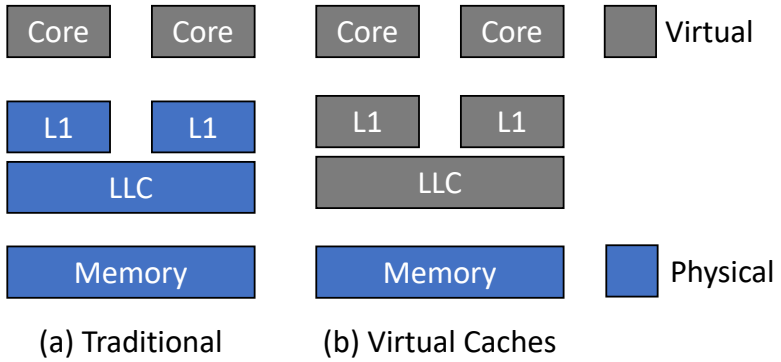


Figure 2.5: Existing systems (a) use physical addresses to index the cache, while virtual cache hierarchy proposals (b) use virtual addresses to index the cache hierarchy.

2.4.2 Virtual Cache Hierarchies

As seen in subsection 2.4.1, while the majority of the proposals and products today rely on creating or maintaining contiguity in the physical address space for better address translation performance, another class of proposals focuses on postponing the requirement of address translation away from the cores to reduce its overhead. This subsection explains the proposals on Virtual Cache Hierarchies [23, 24, 43] and Single Address Space Operating Systems [70] as they attempt to delegate the address translation away from the cores towards memory instead of requiring it at the cores.

Virtual Cache Hierarchies is a generic term for proposals that require addressing the cache hierarchy using the virtual address issued by the cores. The translation from virtual to physical addresses is only required when the cache hierarchy does not contain the required block, and the block must be fetched from memory (as shown in Figure 2.5b). The main benefit of such a design is that the slow address translation step is required only for memory accesses and therefore is not on the critical path of each memory access. For a well-provisioned cache hierarchy, the memory accesses are rare, thus rarely requiring the translation of virtual addresses to physical addresses as most memory instructions can access data directly from the cache hierarchy by simply using the virtual addresses. The long-latency overheads of a page-table walk required for virtual-to-physical address translation can be better tolerated with

relaxed latency requirements at the bottom of the memory hierarchy than near the cores where all the structures are required to be fast and performant. Finally, more silicon resources can be provided for the virtual-to-physical address translation near memory because of relaxed power and space constraints compared to the cores.

However, using the virtual address to index the cache hierarchy also comes with problems regarding synonyms and homonyms. In existing OSes, two different virtual addresses belonging to the same or different virtual address spaces that map to the same physical address and thus refer to the same data are called synonyms. In contrast, the same virtual addresses belonging to different virtual address spaces that map to different physical addresses and thus refer to different data are called homonyms. In Virtual Cache Hierarchies, synonyms create a coherence problem as they should refer to the same cache block but are present as different cache blocks with different virtual addresses. Similarly, homonyms also create a coherence problem as they end up referring to the same cache block while referring to different cache blocks. The problems in cache coherence because of synonyms and homonyms have been the main impediment in adopting Virtual Cache Hierarchies. As there is no easy way to get rid of synonyms and homonyms in the software itself, architecture proposals have focused on filtering synonyms and homonyms dynamically in the cache hierarchy itself [136], but such proposals have typically not been adopted because of their implementation complexity. Therefore, while the Virtual Cache Hierarchies have been adopted in some domains with restricted programming environments, they have not been adopted in mainstream processors.

Single Address Space Operating Systems (SASOS) is an OS-based solution to eliminate synonyms and homonyms, thus enabling the adoption of Virtual Cache Hierarchies. The main principle behind SASOS is that instead of providing a separate virtual address space to each process, the OS only maintains a single virtual address space and accommodates the data from all the processes in the same address space. Any data shared among processes is thus mapped to the same virtual address, while private data is mapped to separate virtual addresses. In this manner, the design eliminates the synonyms and homonyms, thus allowing the virtual addresses to be directly used to index the cache hierarchy. However, enabling the use of a

	Core side			Memory side	
	Access Control	Programmability	Scalability	Implementation	Optimized Page-table Walks
Virtual Cache Hierarchy [23, 24, 43]	×	✔	✔	HW × SW ✔	×
Single Address Space OS [70]	×	✔ *	✔	HW ✔ SW ×	×
In-Cache Address Translation [132]	✔	×	✔	✔	✔
Enigma [139]	✔	✔	×	✔	×
Virtual Block Interface [49]	✔	×	✔	HW ✔ SW ×	×
Midgard [44]	✔	✔	✔	✔	✔

Table 2.2: Comparison of core-side and memory-side translation attributes among various previous proposals and Midgard. (*) indicates that Single Address Space Operating Systems have restricted programmability because they do not provide a private virtual address space to each application, therefore, the applications do not have the flexibility to map data at any required address.

single virtual address space across all the processes requires a significant amount of change in the applications and the toolchains themselves, which has been a significant hurdle in the adoption of SASOS. Moreover, SASOS disables the isolation among processes provided by the VM design by simply not allowing any process to be able to generate a virtual address belonging to a different process.

2.4.3 Intermediate Address Spaces

As explained in subsection 2.4.2, Virtual Cache Hierarchies can offer many performance benefits for VM implementations but typically face adoption problems because of synonyms and homonyms. Some previous proposals have introduced mechanisms to counter such problems, thus allowing to achieve the benefits of Virtual Cache Hierarchies without dealing with the synonym/homonym problems. These proposals, namely In-Cache Address Translation [132], Enigma [139], and Virtual Block Interface [49], introduce an intermediate address space to implement a practical version of Virtual Cache Hierarchies and achieve similar benefits without

the problems. Table 2.2 compares all the discussed proposals in terms of the core-side and memory-side implementation of various address translation properties.

In-Cache Address Translation assumes a typical segmentation-based system where the virtual address space of each process consists of four segments, and the baseline system requires each segment to be further mapped to pages in the physical address space similar to the existing systems. Their proposal introduces an intermediate address space that is larger than the private virtual address spaces and can accommodate the segments across all the processes. The shared segments across processes are mapped to the same segment in the intermediate address space, resolving any synonyms and homonyms. Finally, each memory access generated from the cores requires translating the virtual address to the corresponding intermediate address to access the cache hierarchy, while the intermediate address is translated to the physical address only if the required cache block is not found in the cache hierarchy. As there are only limited segments per process, translating the address of a virtual segment to an intermediate segment is significantly easier than the virtual-to-physical address translation. The intermediate address space also enables optimizations like reading a page-table entry directly from the cache hierarchy instead of requiring a page-table walk. However, the proposal has faced adoption problems because of its dependence on segments that are an old programming abstraction and are not prevalent anymore.

Enigma is another proposal based on intermediate address spaces that relies on dividing the virtual address space into huge pages (256MB as specified in the proposal) and mapping these pages to the intermediate address space. The huge pages shared across processes are mapped to the same huge page in the intermediate address space, resulting in no synonyms and homonyms. The intermediate addresses are used to index the cache hierarchy, and only if the cache hierarchy does not contain the required cache block then the intermediate-to-physical address translation is required for accessing the memory. While huge pages are a popular abstraction compared to segments, dividing existing 48-64 bit address spaces into reasonably-sized huge pages will still result in a large number of virtual-to-physical mappings that do not scale with the memory capacity. Therefore, the core-side translation is still not

guaranteed to be fast.

Virtual Block Interface is a proposal that provides a new programming abstraction of virtual blocks instead of using outdated segments in order to create an implicit intermediate address space. While using virtual blocks avoids the problem of having too many mappings, introducing a new programming abstraction requires application developers to significantly rewrite their application to comply with the new virtual memory implementation. Instead, we maintain that using an existing programming abstraction to optimize the virtual memory design will result in an easier adoption as it does not require the application developers to modify their applications, thus allowing using existing application ecosystems.

3 Midgard: An Overview

As the cache and memory capacity in servers scales to tens of GBs and TBs, respectively, the existing VM techniques fail to provide enough coverage and become a critical performance bottleneck for memory-intensive services as explained in section 2.3. While Virtual Cache Hierarchies [23, 24, 43] can provide significant performance benefits by delaying the address translation away from the cores, they suffer from the implementation complexity of synonyms and homonyms. In this chapter, we introduce Midgard [44] as a novel VM architecture that introduces an intermediate address space indexed using VMAs and provides the performance benefits of Virtual Cache Hierarchies without encountering their implementation complexity. Our evaluation of Midgard shows that it enables the VM overhead to reduce as the cache hierarchy capacity increases, thus resulting in near-zero VM overhead for GB-scale cache hierarchy capacity.

3.1 A VMA-Based Intermediate Address Space

As explained in section 2.1, the application address space is organized in terms of Virtual Memory Areas (VMAs), where every VMA is a logical data section and represents semantic data useful to the program. Memory permissions required for semantic correctness of the program are also defined for every VMA, e.g. the code VMA has read-only permissions, while the heap and stack VMAs have read-write permissions. While data is logically organized in

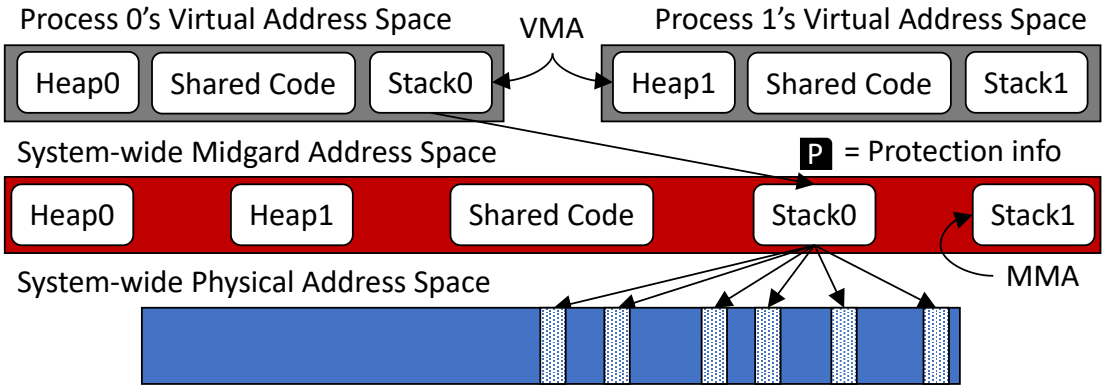


Figure 3.1: VMAs in the virtual address space are mapped to Midgard Memory Areas (MMAs) in the Midgard address space. The protection information is also specified at the VMA granularity. Finally, the MMAs are divided in pages which might be mapped to physical frames in the physical address space.

terms of VMAs in the virtual address space, existing systems map the VMAs into pages for efficient management of the physical memory capacity. As the overall memory capacity in the system increases, the metadata required for tracking virtual-to-physical address mappings also increases, and thus cannot be captured by TLBs that are constrained in capacity because of the end of Moore's law.

The key insight in Midgard rests on two key observations:

1. Performing address translation at a VMA granularity is the best for performance because VMAs are the most coarse-grained representation of data. However, representing VMAs directly in the physical address space is not practical because of the contiguity required by VMAs, as explained in subsection 2.4.1.

2. Physical addresses are not necessarily required to index the cache hierarchy. Using another sparse and unique address space will ease the address translation at the cores and work well with cache coherence protocols, as explained in subsection 2.4.3.

Based on the above observations, we present an insight that if we can introduce a VMA-based intermediate address space in the system, we can optimize the translation to the intermediate addresses and delay the translation to the physical addresses. Therefore, we introduce an

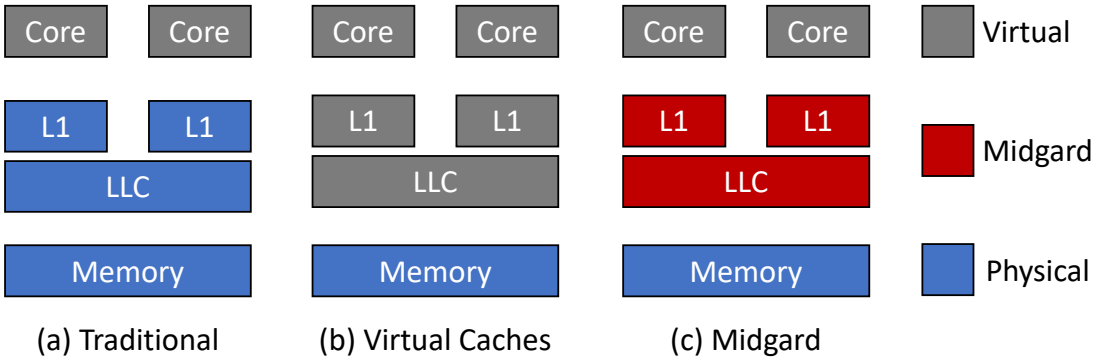


Figure 3.2: Existing systems (a) use physical addresses to index the cache, virtual cache proposals (b) use virtual addresses to index the cache hierarchy, while Midgard (c) uses the Midgard addresses to index the cache hierarchy.

intermediate address space (called the Midgard address space) in the system which can be used to index the cache hierarchy (as shown in Figure 3.2), while the virtual address space is still used at the cores and the physical address space is used to access the memory device. The Midgard address space is a logical, unique address space in the system where the data is organized similarly to the virtual address spaces at a VMA granularity, while each unit is called a Midgard Memory Area (MMA) and refers to unique data, thus ensuring no synonyms or homonyms. The main advantage of Midgard is that the frequent virtual-to-Midgard address translation takes place in terms of VMAs, while the infrequent Midgard-to-physical address translation takes place in terms of pages (as shown in Figure 3.1). The OS manages the Midgard address space and maps VMAs from the virtual address spaces to MMAs in the Midgard address space. The OS also breaks the MMAs into pages and maps them to the physical frames in the physical address space while allowing the possibility that some pages in the MMAs are not mapped to physical frames as the data might be unallocated or swapped out to storage devices (as with demand paging in existing systems). Therefore, the VMA-based translation that is required in the critical path of every memory access is fast, while the slow page-based translation is only required in case the cache hierarchy does not contain the required data.

3.2 Two-Step Address Translation: Logic-side and Memory-side

As shown in Figure 3.2, the virtual-to-Midgard address translation happens at the core side (called *logic-side translation*) and is required for each memory instruction before the data can be accessed from the cache hierarchy. As the virtual-to-Midgard address translation takes place at a VMA granularity, it is much faster and simpler than page-based address translation as each application typically only has ~100 VMAs, while only ~10 of them are hot and accessed frequently [80, 135]. The memory access permissions are also represented and checked at a VMA granularity instead of being replicated for each page. Therefore, only a tiny amount of silicon resources are required at the core side to represent the hot VMAs, thus reducing the typical silicon overhead and simplifying the logic-side translation design.

In contrast, the Midgard-to-physical address translation happens at the memory side (called *memory-side translation*) and is only required if the cache hierarchy does not contain the required cache block, which then has to be fetched from memory. As a well-provisioned cache hierarchy can serve data for the majority of the memory instructions, the Midgard-to-physical address translation is required infrequently. As the cache hierarchy capacity increases, the frequency of memory-side translation and memory accesses decreases. Moreover, as the slow page-based memory-side translation is done away from the cores, it has more latency slack (10-100 ns) because of infrequent occurrence than typical page-based translations, which have to work at ns-scale latencies of the cores. As memory-side translation happens away from cores, it can potentially be provided with more silicon resources with ease than the silicon resources for existing core-side TLBs because of the power density constraints near cores. However, as the cache hierarchy absorbs most of the data locality present in the requests, there is little locality left for memory-side translation, thus requiring only a few resources for caching translations. Finally, using the logical Midgard address space to index the cache hierarchy also allows optimizing the page-table walks for the memory-side translation by directly serving page-table entries from the cache hierarchy (if present) instead of requiring the complete radix-tree traversals as in existing systems.

3.3 Challenges and Opportunities

Midgard requires decoupling the traditional address translation process into two parts - the logic-side address translation, and the core-side address translation. This decoupling is made possible by the insertion of an intermediate address space in the system that can provide access control and cache hierarchy access using address translation at a VMA granularity, thus requiring the capacity management functionality of the physical address space only if the cache hierarchy does not contain the required data. Designing such a system requires introducing a new interface for address translation, while solving various challenges in both the OS and the hardware design.

On the OS side, accommodating Midgard requires decoupling the traditional address translation and memory management implementation into a logic-side translation that is performed at a VMA granularity, and a memory-side translation that is performed at a page granularity. Managing the virtual and Midgard address spaces at a VMA granularity requires tracking the mappings using new data structures per process that can work well with the variable-sized VMAs. Also, the OS is required to accommodate the VMAs across all processes in the system-wide Midgard address space, which requires it to use allocation or memory management algorithms that can work well with the variable-sized VMAs. Finally, the OS has to map the MMAs present in the Midgard address space to the required pages present in the physical address space. While the OS can continue to use the traditional radix page tables for tracking these mappings, the overall memory management code in the OS requires considerable change as there is only a unique system-wide page table instead of the typical per-process page tables. Along with tracking the mappings, the OS also needs to redesign various other memory management-related components such as page caches and direct mappings, that currently work with the page abstraction and make them compatible with the VMA abstraction.

On the hardware side, the microarchitecture needs to be changed to accommodate the two layer translation. The logic-side translation applies to any units that require VM support to perform memory operations, such as cores, accelerators, and peripheral devices. Each unit has

to perform its own logic-side translation to obtain the relevant Midgard address which can then be used to probe the overall coherence domain. Therefore, Midgard addresses can be used to fetch data from anywhere in the coherence domain, which can be in the same or remote socket/chiplet. As the logic-side translation is in the critical path of every memory access, it needs to be low latency so that it does create a performance bottleneck. The memory-side translation applies to only memory controllers that grant access to the underlying physical memory device, such as DRAM, High-Bandwidth Memory, or Persistent Memory. As the memory-side translation is applicable system-wide, the microarchitecture needs to support any synchronization mechanisms that might be needed for ensuring correctness in translation. Finally, while the memory-side translation is not required in the common case, it still requires performance optimizations such as caching to ensure that it leads to a low overhead.

While there are challenges that need to be mitigated, Midgard also opens up various opportunities to optimize various other components of the system. By supporting an intermediate address space, Midgard allows decoupling the memory-side translation both logically and physically. Therefore, we can envision a system where the memory-side translation is not performed as part of normal OS functionality, but similar to a Flash Translation Layer that exists as the logic bundled with the memory device itself. Midgard also allows opportunities to optimize memory pooling across various servers as the Midgard address space does not need to be necessarily local to just one server, but instead can be shared among various other servers using coherent interfaces. Finally, the abstraction of the intermediate address space can also optimize virtualized systems as the translation between the guest and host OSes that takes place using physical address spaces can instead be done using their Midgard address spaces. As such translation can take place in terms of VMAs instead of pages, it is expected to bring significant performance benefits.

3.4 Interface for Address Translation

The address translation interface consists of three components: the address translation tables, the register containing the address of the address translation tables, and primitives for enforc-

ing coherence manually. In traditional systems, the address translation tables are presented in the form of page tables which contain the mappings between VMAs residing in per-process virtual address spaces and pages present in the system-wide physical address space. The structure of the page table also influences the design of the MMU which needs to walk the table to find the required translation, e.g., traditional page tables have a fixed radix-tree structure which is known to the MMU. To architecturally expose the per-process page tables, each core contains a register that contains the base address of the page tables, e.g., *CR3* in x86, *Page Table Base Register (PTBR)* in ARMv8 systems. Finally, while traditional systems have caching support for page tables in form of TLBs, the per-core TLBs are not coherent with each other, and therefore require OS support to enforce manual coherence. Therefore, traditional systems have TLB shootdown primitives which can be used to discard a particular address translation mappings from all TLBs in the system.

Following the address translation interface in the traditional systems, we would need to introduce a similar interface in case of Midgard. However, as Midgard has two levels of address translation, we would require a separate interface for each of those translation levels. In the case of logic-side translation, as the translation is performed at a VMA granularity, the OS will need to track VMA-to-MMA translations in a VMA table instead of a traditional page table. However, as each process has its private virtual address space, it would also need to have a private VMA table containing the mappings from its private virtual address space to a system-wide Midgard address space. The structure of the VMA table can be fundamentally different from the traditional page tables because the page tables track fixed-size pages, while the VMA tables will track variable-sized VMAs and map them to variable-sized MMAs. As each core can be running a different process, we need a separate per-core register to indicate the base address of the VMA table belonging to the running process so that the MMU can walk the table to find the required translation. Finally, similar to traditional systems, as each core can have private TLB-like caching structures for the VMA-table entries, we would also need to provide shootdown primitives to maintain coherence manually. However, as there are much fewer VMAs than pages, it is also possible to design novel hardware mechanisms to ensure coherence among all the VMA table caching structures automatically.

In the case of memory-side translation, the translation is performed at a page granularity, similar to the translation present in the traditional systems. Therefore, the OS can create Midgard page tables similar to the traditional page tables based on radix trees to track the mappings from MMAs present in the Midgard address space to the pages present in the physical address space. As the Midgard page tables have a well-known radix tree structure, the MMUs can walk the page tables using the same algorithms as present in the traditional system. As Midgard page tables contains address translation mappings between a system-wide Midgard address space and a system-wide physical address space, there is only a unique set of Midgard page tables in the system. As there is only one set of Midgard page tables in the system, we logically require only one Page Table Base Register for the entire system, instead of the per-core registers as required in the traditional systems. Finally, memory-side translation can also have specialized caching support for the Midgard page tables, and therefore can require a separate set of shootdown primitives than those required for the logic-side translation.

4 Memory-side Exception Handling

As explained in 3.2, the traditional address translation is decoupled into two parts: the logic-side translation and the memory-side translation. The logic-side translation is a fast VMA-granularity virtual-to-Midgard address translation performed at the cores similar to the traditional address translation. In contrast, the memory-side translation is similar to the traditional page-based translation but is performed away from the cores and closer to the physical memory. The logic-side translation performs permission checks along with translation, and can generate protection fault exceptions due to permission mismatch, or segmentation fault exceptions due to access to unallocated memory addresses. As the logic-side translation is tightly integrated with the pipeline, any exceptions generated by the same are detected and contained in the speculative window of the pipeline, and thus can be handled precisely [111]. The memory-side translation also performs permission checks related to capacity management, and can generate exceptions such as page fault if the referred page is unallocated/unmapped, or can generate protection fault in case a write was attempted to a page that was temporarily made read-only, such as in case of copy-on-write or kernel same-page merging optimizations. However, any exceptions in the memory-side translation are generated away from the cores, and therefore are not guaranteed to be contained in the speculative window of the core. If the exceptions cannot be contained in the speculative window, then they are not guaranteed to be precise, and might be handled *imprecisely*.

Such imprecise handling of exceptions is a direct consequence of the traditional implementation of out-of-order cores. Modern out-of-order cores implement a speculative window which allows them to execute younger instructions before the older instructions have completed. Such speculative execution allows hiding the execution latency of arithmetic and floating-point operations, or memory access latencies with useful work. However, even though the instructions might be finish executing out of order, they are always retired in order to ensure that any resulting exceptions or interrupts are handled precisely at the point of detection with sequential instruction execution semantics. However, modern out-of-order cores also assume that any exceptions can be generated only by execution units or address translation units covered in the speculation window. Once the execution units or tightly-integrated address translation (logic-side address translation in our case) completes successfully, then it is assumed that no more exceptions are possible. Building on the same assumption, modern out-of-order cores retire stores after their address and data is confirmed, and place them in the store buffer for completion. This optimization allows the pipeline to retire younger instructions without waiting for the store to complete, thus significantly benefitting performance, and forms the basis of modern relaxed memory consistency models [84].

However, in case of Midgard, it is possible that the memory-side translation generates exceptions even after the logic-side translation has completed successfully. Therefore, if memory-side exceptions are caused by stores that were retired before completion, it is possible that the exception is detected by the core when the store has already retired and is awaiting completion in the store buffer, and the pipeline might have retired further instructions. In such a scenario, the exception handler cannot be triggered at the PC of the corresponding store as the pipeline has already moved on, and therefore the exception handling is imprecise. In the rest of this chapter, section 4.1 will provide more detailed explanation about the conventional precise handling of exceptions, and what behavior can make the exception handling precise. Then, section 4.2 will explain that post-retirement speculation techniques can be used to force the precise handling of such exceptions, but at a prohibitively high silicon cost. The remaining section 4.3, section 4.4, and section 4.5 explain how we can handle such exceptions imprecisely and develop a formalism to prove that the resulting behavior is compatible with modern

memory consistency models which can be consequently used by OS developers to ensure correctness, provides an OS + microarchitecture co-design to implement imprecise store exceptions, and then finally develop a full-system RISC-V prototype to prove the feasibility and performance implications of our design, along with correctness using litmus tests.

4.1 Exception Handling Background

In this section, we describe the challenges posed by post-retirement store exception detection on the notion of precise exceptions. We then discuss various "obvious" ways to address these challenges and their shortcomings, particularly in the context of relaxed memory consistency models.

4.1.1 Precise exceptions

Precise exceptions [111] are a de-facto abstraction assumed in modern CPUs as they simplify software at the cost of reasonable hardware complexity. Precise exceptions allow programmers to assume a simple sequential execution model where only one instruction executes at a time, and any exceptions are detected and handled before the corresponding instruction executes.

CPUs implementing precise exceptions require exceptions to be triggered precisely at the corresponding instruction only after older instructions completed and successfully modified process state, and before younger instructions modified process state. All exceptions must be detected and handled before instruction retirement. Microarchitecturally, exceptions are implemented by 1) executing instructions such that they modify the process state sequentially, but disabling out-of-order execution and its performance benefits, or 2) employing speculative execution to discard the effects of any younger instructions along with the exception-generating instruction, benefiting the performance in the common case without exceptions. Importantly, this approach extends to interrupts, with the caveat that interrupts are generated asynchronously by external devices and can be triggered on any instruction.

Virtual memory is an example of a common source of exceptions as it requires every load/store

Fault	Fetch	Control protection exception, Code page fault, Code-segment limit violation
	Decode	Invalid opcode, Device not available, Debug
	Execute	Divide by zero, Bound range exceeded, FP error, Alignment check, SIMD FP exception, Invalid TSS
	Memory	Segment not present, Stack-segment fault, Page fault, General protection fault, Virtualization exception
Trap		Debug, Breakpoint, Overflow
Abort		Double fault, Triple fault, Machine Check

Table 4.1: Classification of x86 exceptions [58].

instruction to perform address translation before it can be applied to the cache/memory hierarchy. If an exception (e.g., page fault) is detected during address translation, then the corresponding load/store and all the younger instructions are flushed, and the exception handler is triggered precisely. After the OS handles the exception and reschedules the process, the load/store instruction is re-executed and completes successfully.

While precise exceptions are dominant today, there are cases where imprecise exceptions have been adopted. In the past, imprecise exceptions were generated by long-latency arithmetic or floating-point operations implemented on co-processors [53, 62, 126]). Today, machine checks [114] (e.g., ECC errors in the cache/memory hierarchy) are the only example of imprecise exception in modern CPUs because they are non-restartable and cause the OS to terminate the process or even crash. Finally, emerging accelerators such as GPUs [119] adopt imprecise exceptions because precise exceptions pose a dramatic performance overhead or require significantly more silicon area and power. As computing systems evolve, the cost of supporting precise exceptions and their programming implications are being gradually revisited with new constraints. This thesis studies the cost of supporting precise exceptions coming from compute units embedded in the cache/memory hierarchy.

4.1.2 Long-Latency Exceptions can be Imprecise

Today, cores are the only component in a CPU that can generate exceptions, while all other components perform simpler operations and cannot generate exceptions. Table 4.1 represents the x86 exceptions and their point of origin. Except for machine checks, all exceptions are generated in the fetch, decode, execute, or memory stages and caught synchronously in the reorder buffer (ROB). But with the integration of compute capabilities in the deep cache/memory hierarchies [38, 105], the integrated compute units can also generate exceptions when responding to load/store instructions executed by the cores. As exceptions can originate from the cache/memory hierarchy, exception detection can even include address translation and memory latency (100s of cycles and growing).

The above problem generally appears in proposals that require software intervention when servicing memory requests, such as 1) software handling of cooperative/distributed shared memory [26, 35, 51, 75, 100, 106, 131]; 2) informing memory operations [52] and software techniques for cache-miss handling [21, 47, 82]; 3) virtual cache hierarchies [23, 24, 43, 70], or intermediate address space [44, 49, 132, 139] designs where virtual memory exceptions are generated in the cache/memory hierarchy; and 4) accelerators that can generate exceptions when executing additional functionality for memory requests performed by the cores [107]. Fortunately, this problem does not apply to accelerators invoked using an explicit request-response programming model [4, 76, 137, 138] where any generated exceptions are treated as interrupts by the cores. While we already illustrated the problem with Midgard in the beginning of the chapter, we further provide an additional example to broaden the scope of the problem:

täkō [107] is a semi-general-purpose accelerator connected to the L2 and LLC slice of each core to perform user-defined data transformations, such as compression and encryption. Users can configure täkō to compress data when evicting it from the cache and write the compressed version to memory. Upon reading the compressed data from memory on a cache miss, täkō will install the decompressed version in the cache. Such a design allows exceptions

(e.g., page fault, divide-by-zero) to be generated by the accelerator when processing memory requests received from a core because täkō relies on the virtual memory abstraction to allow users to define data transformation logic using software-defined callbacks. For example, when a core executes a store instruction resulting in a cache miss, täkō will fetch and decompress data from memory and can potentially encounter a page fault. The page fault will result in a delayed notification of the exception being triggered on the corresponding store instruction.

Modern CPUs use aggressive optimizations [34, 84] that remove the store from the ROB (called retirement) once it becomes the oldest instruction but is yet to write its value to the L1 cache (called completion). As stores do not produce a register value for the younger instructions, they are retired to unblock the pipeline and wait for completion in the store buffer. Subsequently, if the store triggers an exception as in the examples above, the resulting exception cannot be precise because it cannot be triggered at the corresponding store instruction as it has already retired, and the pipeline might have further retired younger instructions. The same problem was also identified by Qiu and Dubois [97] in the context of out-of-order cores where the speculative window can cover the exception detection latency in most cases, which is not the case today. Fortunately, such impreciseness is limited to only store instructions as all the load instructions produce register values for the younger instructions and cannot retire before completion.

4.1.3 Forced Precise Exceptions Kill Performance

To imitate the sequential strategy [111], we can disable the store buffer optimization described above to force precise exceptions. Disabling the store buffer would ensure that all load and store instructions await completion in the pipeline before they retire, forcing any exception detection to happen in the ROB itself and enabling triggering the exception handler precisely at the required instruction. However, the store buffer is the cornerstone of relaxed memory consistency models [84] such as Processor Consistency (PC) and Weak Consistency (WC), and even techniques such as end-to-end Sequential Consistency [110].

Relaxed memory consistency models (or memory models) allow memory reorderings to

significantly improve single-thread performance. Such memory reorderings require the long-latency stores to be retired and put aside in the store buffer, allowing any younger instructions to retire, leading to a high pipeline throughput. Disabling the store buffer optimization also disables such reorderings, effectively reverting to Sequential Consistency (SC) and exposing the long latency of stores to the pipeline. Doing so voids the typical 20-30% single-thread performance improvement due to relaxed memory models, thus providing precise exceptions at a significant performance cost.

In this thesis, we discuss the following two solutions to maintain the performance gains of relaxed memory models. 1) Using the speculation strategy [111], we demonstrate that speculation can cover the long latency of memory operations to provide precise exceptions with the performance of relaxed memory models but at a much greater silicon cost than required for ROB speculation. 2) We demonstrate that hardware-software co-design can efficiently implement imprecise handling of store exceptions where the exception handler is triggered at an unrelated instruction instead of the corresponding store instruction, thereby imitating an interrupt.

4.2 Precise Exceptions with Speculation

Post-retirement speculation has been shown to match and even exceed the performance of relaxed memory models at the cost of additional silicon. We now detail how these speculation techniques can be used to maintain the abstraction of precise exceptions even for long-latency store operations.

4.2.1 Post-Retirement Speculation

As described in subsection 4.1.3, relaxed memory model implementations utilize the store buffer to accommodate retired stores without blocking the pipeline. If such a store triggers an exception after retirement, then the exception handling cannot be precise as the pipeline has potentially retired younger instructions. For enforcing precise exceptions, the store must

Core	16× ARM Cortex-A76 [129] 4-way OoO, WC, 128-entry ROB, 32-entry SB
TLB	L1(I,D): 48 entries, L2: 1024 entries
L1 Caches	64KB 4-way L1D, 64KB 4-way L1I 64-byte blocks, 2 ports, 32 MSHRs 2-cycle latency (tag+data)
L2	1MB/tile, 16-way, 6-cycle access, non-inclusive
Coherence	Directory-based MESI
Interconnect	4 × 4 2D mesh, 16B links, 3 cycles/hop
Memory	80 cycle access latency (default)

Table 4.2: System parameters for simulation on QFlex [89].

		Instruction mix (%)				WC speedup
		Store	Load	Sync	Others	
GAP [17]	BFS	11	22	<1	67	1.53
	SSSP	3	22	1	74	1.06
	BC	25	25	0	50	3.24
Tailbench [68]	Silo	7	13	2	78	1.15
	Masstree	14	13	<1	73	1.60
Cloudsuite [36]	Data Caching	11	24	<1	65	1.12
	Media Streaming	9	13	<1	78	1.16
	Data Serving	9	24	<1	67	1.10

Table 4.3: We list the evaluated benchmarks, their instruction mix (%), and the WC speedup over SC.

		Speculation state requirement (KB)		
		Baseline	2× memory latency	4× latency skew
GAP [17]	BFS	14	14	17
	SSSP	21	21	21
	BC	18	18	18
Tailbench [68]	Silo	18	18	25
	Masstree	16	16	16
Cloudsuite [36]	Data Caching	17	17	22
	Media Streaming	14	14	17
	Data Serving	14	17	23

Table 4.4: We list the evaluated benchmarks, their speculation state requirements (in KB) to achieve the full WC performance benefits in the baseline SC system, a system with 2× memory latency, and a system with 4× store-to-load latency skew.

await completion and detect all exceptions before retiring, forcing the legal execution to obey SC without a store buffer.

Post-retirement speculation proposals [20, 25, 41, 42, 98, 127] applied to SC can match and even exceed the performance of relaxed memory models such as PC and WC. These proposals obtain performance benefits through speculative memory reorderings while ensuring that other cores cannot observe them by using checkpointing mechanisms to roll back the core to a legal SC state if there is interaction with other cores. As there is little interaction among cores, such proposals significantly benefit from speculative reorderings. The same reasoning applies to exceptions as they are infrequent, and speculative reorderings can provide performance benefits in the common case, while checkpoints can be used to restore the core to a legal SC state when triggering exceptions. However, such post-retirement speculation techniques have a significant silicon cost attached to them.

4.2.2 Case Study: ASO

We adopt ASO [127] for imprecise exceptions. Other designs such as SC++ [42] require more silicon than ASO, while Invisifence [20] reduces silicon usage by limiting checkpoints, but also limits the overall achievable performance. In ASO, when the core is stalled due to an

ordering requirement, it creates a checkpoint that allows it to ignore the ordering requirement speculatively. ASO uses a scalable store buffer to record the program order of all the speculative stores while using the L1 cache to store the latest speculatively read and written values which younger loads can then use, where the speculatively-written values in L1 are not globally visible. The speculation is successful when the core obtains the write permissions for all speculatively written blocks and atomically drains all of them to L2, making them globally visible.

The number of supported checkpoints is pre-decided as part of the ASO implementation. Each checkpoint requires a map table to record the physical registers representing the core's legal state before speculatively executing the checkpointed instruction. While in traditional ROB-contained speculation, the physical register denoting the state prior to the instruction execution is freed when the instruction retires, ASO requires keeping the physical registers until the speculation succeeds and the checkpoint is freed. For imprecise exceptions, each store miss requires a new checkpoint as the missing store can potentially trigger an exception. Once the store miss is resolved without exception, the corresponding checkpoint is merged into the previous checkpoint and the relevant physical registers are freed. Hence, the number of checkpoints reflects the number of outstanding store misses. Speculation fails if an exception is detected on a speculated store, causing the core state to be rolled back using checkpoints, followed by a precise invocation of an exception handler.

4.2.3 Quantifying the Speculation State

We use cycle-accurate full-system simulation with QFlex [89] to measure the performance benefits of ASO for imprecise exceptions. Table 4.2 details system simulation parameters. Table 4.4 lists the evaluated server benchmarks from GAP [17], Tailbench [68], and Cloudsuite [36], along with the instruction mix and the IPC speedup on a WC system. As the PR, CC, and TC in the GAP benchmark suite have <1% stores and no performance benefits from WC, we do not evaluate them further.

Table 4.3 shows the amount of speculation state required per core to obtain the speedup

equivalent to that of WC. The speculation state includes the scalable store buffer, per-word valid and Speculatively Written (SW) bits in L1D, the Speculatively Read (SR) bits in the L1D and L2 cache, the additional physical registers required to store the legal SC state before speculatively-retired instructions, and the map tables to track the physical registers. Each entry in the scalable store buffer is 16B, while each checkpoint can require up to 32 extra physical registers (256B), resulting in a larger physical register file. Finally, each map table contains 32 logical-to-physical register mappings while storing 8-10 bits as the register index in a 256-1024 entry physical register file. Increasing the number of supported checkpoints increases the total required speculation state. Table 4.3 demonstrates that post-retirement speculation can indeed match the performance of corresponding WC implementations. As most of our workloads have <1% synchronization instructions, we do not achieve better performance from fence speculation. Unfortunately, the required performance gain comes at a high silicon cost of up to ~25 KB per core. Moreover, most of the required speculation state is part of the physical register file, which is a critical structure in the pipeline and cannot be scaled easily.

We also perform additional studies to quantify the impact of hardware scaling trends on the amount of speculation state required. To this end, we quantify the impact of memory latency and the latency skew between stores and loads on the required speculation state. As the memory capacity in the system scales, the average memory latency becomes higher because of frequent remote memory accesses across sockets and due to denser memory technologies such as persistent memory. Moreover, modern systems feature multiple sockets/chiplets per CPU [128], which can lead to remote cache accesses from far-off sockets. In such a system, cache coherence protocols require extra hops for servicing stores than they need for servicing loads because stores require invalidating all copies of the block, which might reside in far sockets/chiplets.

Table 4.3 depicts two additional systems with 2x the baseline memory latency, and 4x the baseline store-vs-load latency skew respectively. As demonstrated, a system with 2x the memory latency requires about the same amount of speculation state to reach the WC speedup

as the baseline system. The reason is that increasing the memory latency affects both the loads and stores, and as loads typically outnumber the stores, they quickly become the performance bottleneck. In contrast, the required speculation state can increase considerably in a system where the stores take 4x longer latency to complete than the loads. The reason is that as the skew between store and load latencies increases, the pending stores in the store buffer take more time and prevent further stores from retiring, thus blocking the pipeline. Increasing the time required to drain the store buffer also has performance side effects for other subsystems, such as the *tlbi* instruction for performing TLB shootdowns. Overall, while speculation can indeed provide the performance of WC, it requires a significant amount of SRAM resources that are not trivially obtained in the post-Moore era.

4.3 Imprecise Store Exceptions

Maintaining precise exceptions with SC either leads to significant performance degradation or requires post-retirement speculation mechanisms to provide WC performance using tens of KBs of expensive per-core speculation state. Instead, we propose a hardware-software co-design to implement PC/WC systems with the relaxed semantics of imprecise store exceptions. In this section, we provide the formalism for memory models with imprecise store exceptions and demonstrate that with appropriate hardware and OS support, imprecise store exceptions are compatible with both PC and WC.

4.3.1 Brief Description

Similar to modern CPUs, we require that the stores are retired and put in the store buffer to await completion. Subsequently, if they trigger an exception, the exception handler must be triggered imprecisely on the oldest instruction in the ROB. However, the faulting stores (i.e., stores that trigger exceptions) present in the store buffer can neither be drained to the cache hierarchy nor rolled back to be re-executed later as they have already retired and younger instructions have further modified the architectural state. With exception handling latencies of several μs (e.g., lazy memory allocation) to tens of ms (e.g., demand paging), the stores

cannot stay in the store buffer because they will eventually clog the store buffer and prevent the core from executing younger instructions.

We create a new architectural interface to supply faulting stores along with their address, data, byte mask, and the required exception code to the OS so that we can free up the microarchitectural resources occupied by the store. When triggering an imprecise exception handler, the OS first reads the faulting stores using the architectural interface and then resolves the exception. If the exception is recoverable (e.g., page fault), the OS applies the obtained faulting stores to the corresponding addresses, and the program resumes execution. If the exception is irrecoverable (e.g., segmentation fault), the faulting stores are discarded, and the program is terminated.

As the OS may take up to tens of *ms* to resolve the exception and apply the faulting store, the store is effectively reordered after younger operations in the global memory order, which might seem to violate the underlying memory model. While previous research proposals [77, 102] studied the interaction between memory operations executed by the core and MMUs, there are no studies on such OS-induced reorderings. We claim that we can make these reorderings transparent to user programs with a formally-defined memory model incorporating store exceptions and a hardware-software co-design that conforms to required constraints.

4.3.2 Formal Definition of Memory Models

We first describe the standard formalism [84] of PC and WC using notations defined in Table 4.5. We study PC and WC because they are the prevalent models in ISAs today (x86, AMD, ARM, RISC-V). We use PC to represent Total Store Order (TSO) as they are identical in modern cache-coherent systems [3].

PC relaxes the store-to-load ordering and is formally defined with the following rules:

$$L(A) <_p S(B) \implies L(A) <_m S(B)$$

$$L(A) <_p L(B) \implies L(A) <_m L(B)$$

Notation	Definition
L(A)	Load latest value from address A
S(A), S(A, D)	Store data D to address A
S_{OS}(A), S_{OS}(A, D)	OS stores data D at address A
F	Fence as a memory ordering primitive
X <p Y	Operation X happens before operation Y in program order on the same core
X <m Y	Operation X happens before operation Y in the global memory order
PUT(S(A))	Send S(A) to the architectural interface.
GET	Retrieve one faulting store from the architectural interface
DETECT	Detect an exception
RESOLVE	Resolve the exception and resume execution
$MAX_{<m}$ ({}$_{S(A)}$)	Return the latest value in memory order from the set of stores to address A

Table 4.5: Memory consistency formalism notations [84].

$$S(A) <_p S(B) \implies S(A) <_m S(B)$$

$$S(A) <_p F <_p L(B) \implies S(A) <_m F <_m L(B)$$

$$L(A) = MAX_{<m} (\{S(A, D) \mid S(A, D) <_m / <_p L(A)\})$$

WC relaxes various other orderings present in PC and is formally defined with the following rules:

$$L(A)/S(A) <_p F \implies L(A)/S(A) <_m F$$

$$F <_p L(A)/S(A) \implies F <_m L(A)/S(A)$$

$$L(A) <_p L'(A)/S(A) \implies L(A) <_m L'(A)/S(A)$$

$$S(A) <_p S'(A) \implies S(A) <_m S'(A)$$

$$L(A) = MAX_{<m} (\{S(A, D) \mid S(A, D) <m/<p\ L(A)\})$$

where L'(A)/S'(A) depicts another load/store to address A. Overall, WC relaxes all orderings except the ones involving fences and memory operations to the same address.

We further require additional operations to handle imprecise store exceptions. Assuming that S(A) triggers an exception, the DETECT operation indicates the detection of the exception. Once the exception is detected, then S(A) should be supplied to the architectural interface using PUT(S(A)) operation, and consequently, the OS will read the faulty store using GET operation and will apply the faulting store to the address A using $S_{OS}(A)$ operation. Finally, the OS can finish the exception handling and resume the program execution using the RESOLVE operation. Overall, these new operations strictly happen in the global memory order as:

$$DETECT <m PUT(S(A)) <m GET <m S_{OS}(A) <m RESOLVE$$

4.3.3 Observing the Memory Order

Independent of the underlying memory model and the presence of imprecise exceptions, programs can infer the order among memory operations only by detecting value changes at the corresponding addresses. Assume that an application applies two stores, S(A,1) and S(B,1), to two zero-initialized memory locations, A and B. To detect the memory order between S(A) and S(B), the program requires two *observer loads* L(A) and L(B) that should be executed on a different core to detect the change in values at addresses A and B. The only way that the application can infer S(A,1) <m S(B,1) is if L(A) <m L(B), L(A) reads 1, and L(B) reads 0, corresponding to the execution S(A,1) <m L(A) <m L(B) <m S(B,1). Note that these requirements create a total order among the four operations.

Similarly, the only way to infer S(B,1) <m S(A,1) is if L(B) <m L(A), L(A) reads 0, and L(B) reads 1, corresponding to the execution S(B,1) <m L(B) <m L(A) <m S(A,1). If any of the above requirements are not fulfilled, then the order between S(A,1) and S(B,1) cannot be inferred solely based on the value read by observer loads because there is no dependency chain among the

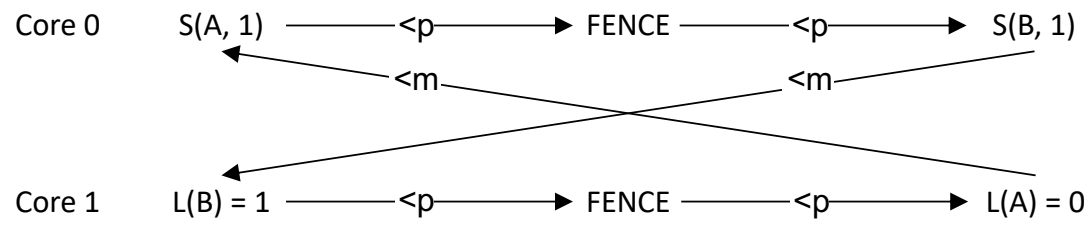


Figure 4.1: Violation in the message-passing litmus test.

stores and loads. In summary, the order among memory operations can only be inferred by the composition of preserved program orders [5] (memory orders enforced from program order based on the memory model, e.g., program order between two stores in PC) and specific change in value that loads can detect. Consequently, programs cannot infer the memory order if the required value changes are not detectable.

Using the above rules, certain combinations of detectable value changes can lead to violations of the memory model. Consider the message-passing litmus test [84] shown in Figure 4.1, where Core 0 communicates B to Core 1 by first setting A to indicate the ready status of B. For simplicity, we explicitly insert two fences between two stores and two loads to make WC identical to PC. Out of four possible results, only the execution with L(B) reading 1 and L(A) reading 0 is prohibited because it indicates that both S(A, 1) <m L(A) and L(A) <m S(A, 1) hold simultaneously, which is impossible. The basis of our formalism is that the applications cannot detect any violations due to faulting stores because the required value changes to infer the memory ordering are either presented in the correct order or are not detectable.

4.3.4 Contract Among the Cores, Interface, and OS

Next, we describe the contract required between the cores, the architectural interface, and the OS to adhere to the underlying memory models, as shown in Table 4.6. Imprecise exception handling requires an architectural interface to supply the faulting store to the OS, and the interface is required to guarantee that the OS retrieves the faulting stores in the same order as the core sends them. While we do not require a total order among the faulting stores from all cores, we require enforcing a per-core order for PC and do not require any order for WC.

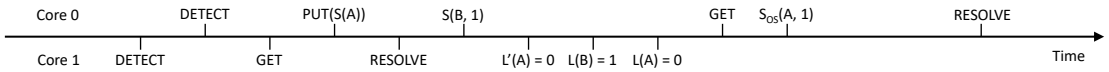
When using the interface to supply faulting stores, the cores must ensure that the stores are supplied in the order the underlying memory model prescribes. In PC, the faulting stores in the store buffer should be supplied to the interface in the FIFO order of the store buffer so that the OS can apply all the stores to memory in the required order after successful exception handling. However, in WC, as there is no order required among stores in the store buffer except for stores to the same address that are already coalesced, the order of supplying the faulting stores is irrelevant.

After successful exception handling, the OS retrieves the faulting stores using the interface and applies them to their target addresses. The order in which the faulting stores are applied is critical for the correctness of the memory model, and requires the OS to obey the following constraints. First, similar to precise exceptions, the program or thread that triggered the imprecise exception can only resume after the exception has been successfully handled. Second, all the retrieved faulting stores must be applied to their target addresses for the imprecise exception handling to be complete. Third and last, the OS must ensure that the faulting stores are applied to memory in the same order as retrieved from the interface. The last rule applies only to PC which requires a strict order among per-core stores. In the case of WC, the OS does not need to enforce any order among stores as the memory model does not mandate it.

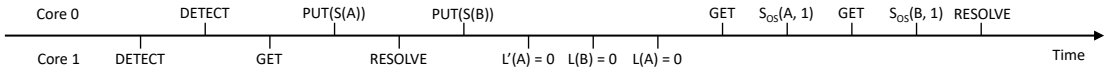
The above constraints also imply that if there is a precise exception on a load or if an imprecise exception is pinned on an atomic or a fence instruction because it blocks the ROB waiting for the store buffer to drain where a store in the store buffer generates the imprecise exception, then the load/atomic/fence instruction will be re-executed only after successful exception handling indicated by RESOLVE <m L(A)/Atomic/F . The overall intuition behind these constraints is to ensure that the microarchitecture communicates the order required for obeying the underlying memory model to the OS, which then applies the faulting stores in the required order to complete exception handling successfully before resuming the program execution.

Component	Requirements for PC
Cores	Supply faulting stores to the interface in the serial order dictated by the store buffer
Interface	Supply faulting stores to the OS in the same order as received from the core
OS	1) Program resumes only after exception handling 2) Apply all faulting stores during handling 3) Apply the faulting stores in the interface order

Table 4.6: The contract among the cores, interface, and OS.



(a) Execution with split stream.



(b) Execution without split stream.

Figure 4.2: Race condition between the GET operation on Core 1 and the PUT(S(A)) operation on Core 0.

4.3.5 Formalism with Split Stream

When handling imprecise store exceptions, there are two approaches to treat the non-faulting stores present along with faulting stores in the store buffer. The non-faulting stores can be directly written to memory or supplied to the interface along with faulting stores. The first approach results in a formalism that treats non-faulting stores and faulting stores as two separate streams of operations, while the second approach results in all the stores being treated as the same stream. We provide the formalism for both approaches and describe their differences.

When a faulting store is in the store buffer, the other non-faulting stores can still be drained to memory. We call this case split stream because the stream of faulting stores is treated differently from that of non-faulting stores. In PC, both streams should obey program order, while in WC, the order of stores in each stream is irrelevant. In both models, the non-faulting stores will be drained to memory immediately, while the faulting stores will be applied later by

the OS, potentially breaking the required global memory order among all the stores. We show this effect using the following formalism. Assume that S(A) is faulting so that it is supplied to the interface, while another non-faulting S(B) that happens after S(A) in program order is drained to memory as usual. Eventually, an exception handler is triggered, which retrieves S(A) from the interface and applies it to memory, resolving the exception. We can formally represent the scenario as follows:

$$\begin{aligned} S(A) <_p S(B) \implies & \text{DETECT} <_m \text{PUT}(S(A)) <_m S(B) <_m \\ & \text{GET} <_m S_{OS}(A) <_m \text{RESOLVE} \end{aligned}$$

As shown, S(B) can appear before $S_{OS}(A)$ in memory order and violate the memory ordering requirements for PC. To be compatible with PC, the hardware and software need to ensure that observer loads cannot observe this violation, which requires that $S_{OS}(A) <_m \text{RESOLVE} <_m L(A)$ so that L(A) cannot detect the value change on A that leads to the violation. While it might seem like this condition is easy to satisfy when address A is faulting, a subtle race condition exists in this case. Consider a program where Core 0 executes $S(A,1) <_p S(B,1)$, and Core 1 executes $L'(A) <_p L(B) <_p L(A)$. Assume that both A and B are zero-initialized, and the accesses to address A will result in exceptions at both S(A) and L(A), corresponding to the following two concurrent executions:

$$\begin{aligned} \text{Core 0:} \quad & \text{DETECT} <_m \text{PUT}(S(A)) <_m S(B) <_m \\ & \text{GET} <_m S_{OS}(A) <_m \text{RESOLVE} \end{aligned}$$

$$\begin{aligned} \text{Core 1:} \quad & \text{DETECT} <_m \text{GET} <_m \text{RESOLVE} <_m \\ & L'(A) <_m L(B) <_m L(A) \end{aligned}$$

There can be a race between PUT(S(A)) on Core 0 and GET on Core 1 such that at the time GET completes, it cannot see PUT(S(A)). As $S_{OS}(A)$ might not be applied before Core 1 performs its RESOLVE, it can result in an execution where L(B) reads 1 and L(A) reads 0, as shown in Figure 4.2a. While such execution is legal in WC, it violates PC because Core 1 can infer that S(B) has taken place, but S(A) has not, even though it comes earlier in program order. For the

split stream formalism to work in real designs, the hardware and OS should together ensure that the core executes PUT(S(A)) before the OS performs the final GET to avoid any data races. Such a design would require a barrier or synchronization between the hardware and software so that no further PUT(S(A)) can occur after the GET. Though possible, building such designs is difficult because of the complexity and performance overhead of implementing such barrier or synchronization mechanisms.

To prove that S(A) <p S(B) $\implies$ S(A) <m S(B), we consider the following four cases:

1. Both S(A) and S(B) are not faulting,
2. Only S(B) is faulting,
3. Both S(A) and S(B) are faulting,
4. Only S(A) is faulting.

Assume that S(B) is in the store buffer when S(A) is drained to memory or supplied to the architectural interface.

Case 1 is the original PC case. The store buffer drains both S(A) and S(B) to memory in their program order.

In case 2, the store buffer drains S(A) to memory and supplies S(B) to the interface. The OS then retrieves S(B) from the interface and applies it. This case represents the following execution:

$$\begin{aligned} \text{S(A) <p S(B)} &\implies \text{S(A) <m DETECT <m PUT(S(B)) <m} \\ &\quad\ \text{GET <m } S_{OS}\text{(B) <m RESOLVE} \\ &\implies \text{S(A) <m } S_{OS}\text{(B)} \end{aligned}$$

In case 3, the store buffer supplies both S(A) and S(B) to the interface. The OS then retrieves both stores from the interface and applies them in the retrieved order. This case represents the following execution:

$$\begin{aligned} \text{S(A) <p S(B)} &\implies \text{DETECT <m PUT(S(A)) <m} \\ &\quad\ \text{PUT(S(B)) <m GET <m } S_{OS}\text{(A) <m} \\ &\quad\ \text{GET <m } S_{OS}\text{(B) <m RESOLVE} \\ &\implies S_{OS}\text{(A) <m } S_{OS}\text{(B)} \end{aligned}$$

Case 4 is same as case 3 because S(B) is supplied to and applied by the OS as it follows the faulting S(A).

If S(B) is not in the store buffer because it has not retired, then S(A) can either be applied to memory (case 1 and 2), or trigger an exception (case 3 and 4) where the OS applies S_{OS}(A) to memory and executes the RESOLVE operation. In all cases, S(B) is guaranteed to retire only after S(A) or S_{OS}(A) has been already applied to memory.

Proof 4.1: Store-store ordering rule of PC.

4.3.6 Formalism without Split Stream

We now describe the second approach, where the faulting and non-faulting stores are treated as the same stream. In PC, stores in the store buffer should be applied to memory in program order. In the presence of faulting stores, instead of sending younger non-faulting stores to memory as another stream, they are supplied to the architectural interface along with the faulting stores. The faulting and younger non-faulting stores are still supplied to the interface in the FIFO order of the store buffer, allowing the OS to retrieve and apply them in the correct program order when handling the imprecise store exception. The intuition behind this approach is that if the faulting and any younger non-faulting stores are supplied to the OS and applied by the OS in their original program order, then there are no potential PC violations that the OS needs to hide as in the previous case. Assuming that S(A) is faulting while another younger S(B) is not, the core will supply both stores to the architectural interface in the program order to be applied by the OS. We can formally specify this scenario as follows:

$$\begin{aligned} S(A) <_p S(B) \implies & \text{DETECT} <_m \text{PUT}(S(A)) <_m \\ & \text{PUT}(S(B)) <_m \text{GET} <_m S_{OS}(A) <_m \\ & \text{GET} <_m S_{OS}(B) <_m \text{RESOLVE} \end{aligned}$$

As shown, the OS always maintains the correct order between $S_{OS}(A)$ and $S_{OS}(B)$ that comes from the program order of S(A) and S(B). Considering the race condition discussed in the previous subsection, even though the race between PUT(S(A)) and GET still exists, as long as L(B) reads 1, which indicates that $S_{OS}(B) <_m L(B)$, L(A) must also read 1 because the OS enforces the order between $S_{OS}(A)$ and $S_{OS}(B)$ such that $S_{OS}(A) <_m S_{OS}(B) <_m L(B) <_m L(A)$, as shown in Figure 4.2b. On the other hand, if L(B) reads 0, then L(A) can either read 0 or 1, both of which are not PC violations. We can formally prove that the same-stream approach obeys all five rules for PC defined in subsection 4.3.2. Due to space limitations, we only show the proof for the store-store rule $S(A) <_p S(B) \implies S(A) <_m S(B)$ in Proof 4.1. Other rules can be proved in a similar manner.

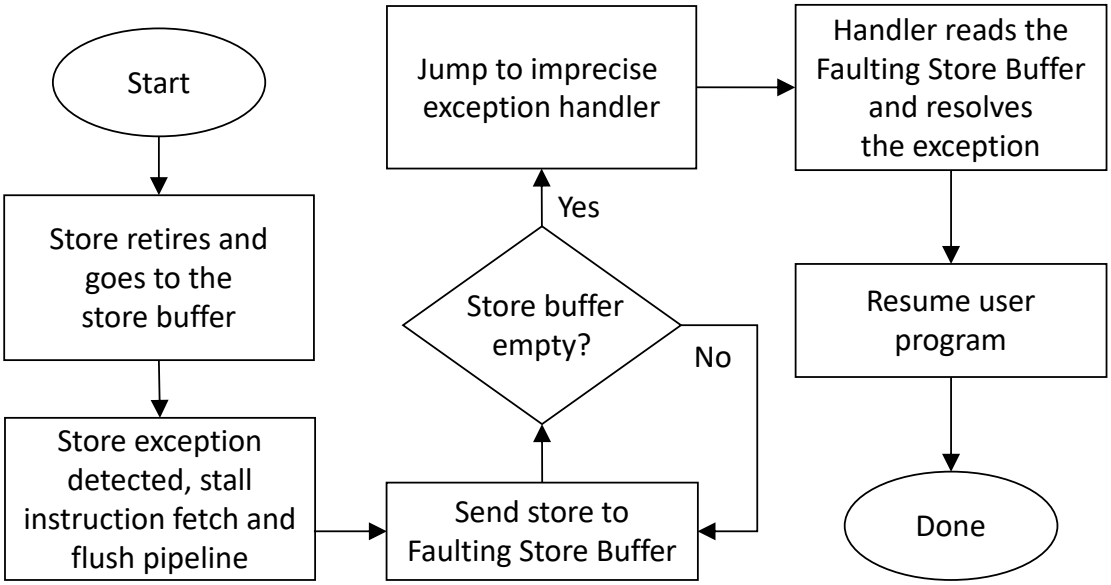


Figure 4.3: Imprecise store exception handling flow.

4.4 Design

This section presents a detailed hardware-software co-design of the architectural interface to supply faulting stores to the OS. We assume a generic multicore system as shown in Figure 4.4. We also assume that the imprecise store exceptions are generated by a generic hardware component situated in the cache hierarchy and away from the cores.

4.4.1 Exception Detection

Figure 6.4 depicts the detection and handling flow for imprecise store exceptions. When the store buffer receives a retired store from the ROB, it sends a memory request for the store to the cache hierarchy. In the case of a faulting store, the memory request cannot find the required cache block, and eventually reaches the required hardware component which generates an exception and sends a response with an embedded error code back to the requesting core. The response message backtracks through the cache hierarchy while freeing the occupied resources, such as MSHRs allocated for the request. The response is received by the L1-D, which then relays it to the corresponding store buffer entry, completing the detection of the

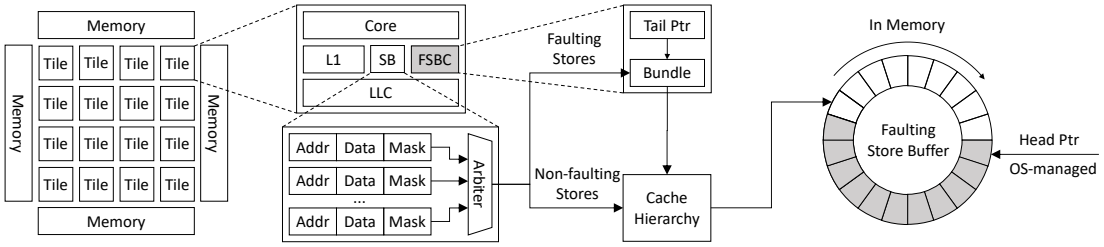


Figure 4.4: Modifications to handle imprecise exceptions in a generic multicore system.

imprecise store exception. Then, the store buffer drains the store address, data, byte mask, and the error code into the *Faulting Store Buffer* (FSB), which is the backing storage of the architectural interface that temporarily accommodates faulting stores before supplying them to the OS, as specified in subsection 4.3.4.

4.4.2 Faulting Store Buffer and Controller

Similar to the x86 virtualization mechanism where the core drains the architectural state of the guest virtual machine to the Virtual Machine Control Structure upon #VMExit [58], the store buffer drains faulting stores to the FSB for imprecise store exceptions. The FSB is a per-core ring buffer located in the main memory with a head and tail pointer, as shown in Figure 4.4. The FSB is similar to the ring buffers used in OS/virtualization like io_uring [29], XEN [12], and VirtIO [103], or hardware interfaces/protocols like NVMe [87] and RDMA [99] that facilitate uni-directional order-preserving communication. The order among faulting stores is encoded in their relative positions in the FSB. We use a *Faulting Store Buffer Controller* (FSBC) to control the order in which the faulting stores are written into the FSB. Each core has a private FSBC co-located with the store buffer, as shown in Figure 4.4. After detecting an exception, the store buffer sends the faulting stores to the FSBC in the order mandated by the memory model. The FSBC then writes them to the tail pointer position of the FSB. After each store draining completes, the FSBC increments the tail pointer and sends a completion response back to the store buffer.

The FSBC is exposed to the OS using four per-core system registers in the ISA: *base, mask, head pointer*, and *tail pointer*. The OS configures the base and mask to specify the address of the FSB

in memory, which is allocated by the OS and is not visible to the application. The tail pointer is written by the FSBC and read by the OS, specifying the position to drain the next faulting store. The head pointer is written by the OS and read by the FSBC, specifying the position of the oldest faulting store in the FSB. The OS can retrieve the oldest faulting store by reading the entry at the head pointer. The OS increments the head pointer to mark the faulting store as read and retrieves the next faulting store (if present). Once the head pointer matches the tail pointer, all faulting stores have been handled. The FSB is sized according to the number of store buffer entries, representing the maximum number of already retired stores that might need to be drained. Our proposal does not require any further changes to the existing core microarchitecture. The load/store queues and store buffer can still have their original design and capacity. In the common case when there are no imprecise store exceptions, the core works traditionally with the store buffer providing WC performance benefits over SC. The control and data paths of FSBC are activated only after the store buffer detects an imprecise store exception.

4.4.3 Exception Handling

On detecting an imprecise store exception, the core stops the instruction fetch and drains all unfinished stores present in the store buffer to the private per-core FSB without requiring any special synchronization or cache coherence transactions. After draining each store, the FSBC sends a completion response to the store buffer which then discards the corresponding entry. Once all entries are drained, the FSBC triggers an imprecise exception which is attached to the oldest uncommitted instruction in the ROB, resembling an interrupt. Consequently, all the uncommitted instructions, including the one with the attached exception, are flushed, and the core jumps to the corresponding exception handler.

The store buffer dictates the order of stores and associated imprecise exceptions to obey the underlying memory model. In PC, after the store buffer detects an imprecise store exception on a faulting store, all the younger uncompleted stores are drained to the FSB in program order, even if the coherence requests for those stores are still ongoing and potentially result in

more imprecise exceptions. Similarly, before handling any precise exception detected in the pipeline, the core drains the store buffer to detect potential imprecise store exceptions. If such an exception is detected on an older store, the core flushes the pipeline, forgoes the precise exception, and handles the imprecise exception instead. Only after the successful handling of imprecise store exceptions, the instruction that triggered the precise exception is re-executed and re-generates the precise exception again. Overall, the design ensures that all exceptions are handled in program order.

The OS can correctly identify an imprecise store exception by the dedicated exception code reserved in the ISA. In the exception handler, the OS first copies all faulting stores from the FSB to an OS-managed data structure and then starts the traditional exception handling. When the handler executes, the effects of all the committed user instructions are present in registers, memory, or the FSB, but the exact architectural state of the faulting store is lost. To the best of our knowledge, typical exception handlers do not examine the architectural state corresponding to past retired instructions but receive the necessary information as part of the exception. Similarly, for imprecise exceptions, the handler retrieves the necessary information from the FSB, such as the faulting store's address and data, and handles the exception. In case of recoverable exceptions, the OS resolves the exception, applies the faulting stores to memory in the same order as they were retrieved from the FSB, and then resumes the responsible application. In case of irrecoverable exceptions, the OS terminates the responsible application and the faulting stores are discarded.

While exception handling is serialized in time, interrupts can be detected concurrently with imprecise store exceptions. In current systems, handling interrupts does not require draining the store buffer, which makes it possible for imprecise store exceptions to be detected while the interrupt handler is executing. We rely on the *Interrupt Enable* (IE) bit defined in the ISA to prevent the exceptions from obstructing the interrupt handler. The IE bit is automatically set when triggering interrupt and imprecise store exception handlers and by the OS when it enters a non-interruptable critical section. The OS clears the IE bit when it exits from the critical section or is ready to handle new interrupts or imprecise store exceptions. By manipulating

the IE bit, the core and OS can serialize the handling of interrupts/imprecise store exceptions and the execution of critical sections. Moreover, pending/masked imprecise store exceptions can stop the OS from resuming user applications because the exception cannot be masked in user mode as the IE bit is hard-wired to zero and not effective in user mode.

In contrast to precise exceptions, an imprecise store exception can correspond to multiple faulting stores, allowing the OS to handle them in batches. Consider the case where multiple faulting stores generate major page faults due to demand paging. In the traditional case, each major page fault triggers a precise exception to let the OS schedule an IO request to load the corresponding page back. The next page fault can be triggered only after the last IO request is done and the application is resumed, forcing all IO requests to take place sequentially. In contrast, within a single invocation of the imprecise store exception handler, the OS can schedule multiple IO requests for all the faulting stores covered by the exception, effectively overlapping IO latencies and improving IO throughput. The batching effect also helps reduce the overhead of invoking the handler, where the context switch, exception dispatch, and other miscellaneous costs are only paid once per each handler invocation instead of per faulting store.

4.4.4 OS Requirements

As the FSBC controls the draining of the faulting stores into the FSB, the OS should always pin the data pages allocated to FSBs in memory, ensuring no page faults. As the FSB is sized according to the store buffer, the OS only needs to reserve a few 4K pages per core. As a minimal requirement, the OS should ensure that the imprecise store exception handler does not trigger further imprecise exceptions. Otherwise, the handler will need to support recursive exceptions, which significantly complicates the system design. Such recursion is not supported in traditional systems as well.

In cases where the OS must send some data to the accelerator (e.g., when invoking `copy_to_user` where the user buffer is allocated from the accelerator), the kernel can also generate imprecise store exceptions. In such cases, the OS can utilize fence instructions to fully contain

imprecise store exceptions and limit them from affecting other parts of the kernel. For example, after invoking `copy_to_user`, the OS can issue a fence instruction to ensure that any potential OS imprecise exceptions are properly reported and handled. The OS can enhance any function that may potentially generate imprecise exceptions in this way. As the OS does not directly use accelerators, we expect only a few OS functions to require such enhancement. Similarly, the OS should issue a fence before switching to the user mode to avoid OS imprecise exceptions affecting user applications.

4.5 Prototype and Evaluation

In this section, we introduce our full-system prototype for imprecise store exceptions and evaluate our prototype's silicon overhead, functional correctness, and performance to demonstrate the feasibility and benefit of handling store exceptions imprecisely.

4.5.1 Prototype Overview

We use XiangShan [133], an open-source, high-performance, Out-of-Order RISC-V CPU written in Chisel [11] to build our prototype. XiangShan implements the RISC-V Weak Memory Ordering (RVWMO) [101] as its memory model. We extend XiangShan's microarchitecture to support detecting and handling imprecise store exceptions as described in subsection 4.4.2. We also port our prototype to AWS cloud FPGAs for fast simulation using FireSim [67]. Due to the limited capacity of cloud FPGAs, our prototype currently only supports two minimal XiangShan cores. We use Linux 5.15 as the OS and add various device drivers and handlers to support injecting and handling imprecise store exceptions as specified in subsection 4.4.3.

We synthesize and implement our prototype using Vivado 2020.2. In the routed design, FSBC consumes 354 CLB LUTs and 763 CLB registers per core, corresponding to only 0.12% and 0.48% of the total core consumption. As FSBC is tightly integrated into the core, some core modules (e.g., the *CSR* module that manages system registers and exceptions) are also changed accordingly. The silicon overhead of these extra modifications is also minimal.

4.5.2 Error Injection and Handling

We create a hardware component *EInject* for error/poison injection to model imprecise store exceptions that accelerators might generate. EInject monitors each non-coherent TileLink-UL [109] transaction between the LLC and memory. For transactions whose addresses lie in the memory region reserved by EInject, it looks up a bitmap to check whether the targeting physical page is marked as faulting. If so, EInject terminates the transaction and generates a response to the LLC with a bus error by setting the *denied* bit.

As an MMIO device, EInject exposes two MMIO registers, *set* and *clr*, to the software to manage the bitmap. Writing an address A to these two registers sets or clears the bit corresponding to the 4KB page of that address in the bitmap. Thus, the software can dynamically inject faults into the system by setting some pages as faulting and handle these faults by setting the pages back to non-faulting. We add a device driver in Linux to allow user-level applications to *mmap* the memory reserved by EInject and control the errors on the mapped pages by *ioctl*.

We implement a minimal OS handler for resolving imprecise store exceptions. Since EInject is the only source of imprecise exceptions in the system, the handler marks the corresponding page as non-faulting through the EInject interface for each faulting store, performs the store using normal store instructions, and then increments the head pointer. The handler continues this action until the head pointer catches the tail pointer, thus indicating that all the faulting stores have been served.

4.5.3 Functionality Correctness

We use the litmus tests [37] in the RISC-V specification [101] to verify that our prototype does not violate RVWMO even with imprecise store exceptions. The test suite targets various ordering relations and constraints specified in RVWMO, as shown in Table 4.7. We modify each test to allocate the memory for consistency check from the EInject regions. Before running each test, we also intercept the *main* function to mark the allocated memory as faulting, as described in subsection 4.5.2, to inject bus errors on all load, store, and atomic

Ordering relation	Explanation	Cases covered
Dependencies	Register dependencies for addr, data, and ctrl	2366
Program order (same location)	Rd-Rd or Wr-Wr to same the address from the same core	368
Preserved program order	Instruction pairs maintained in program order (Atomic, LR/SC)	733
External read-from order	Wr-Rd to the same address from different cores	1544
Internal read-from order	Wr-Rd to the same address from the same cores	1304
Coherence order	Wr-Wr total order to the same address	747
From-read order	Rd-Wr to the same address	976
Barriers	Ordering imposed by barriers	1581

Table 4.7: Ordering rules [5] covered in litmus tests.

instructions, which generate many precise and imprecise exceptions that are silently handled by the minimal handler in Linux. We pick all (1600) 2-core litmus tests that can run successfully on QEMU and run them in a large batch on our prototype system. Our prototype does not produce any RVWMO violation for all the litmus tests. Overall, we empirically prove that our prototype does not break the underlying memory model.

4.5.4 Performance: Microbenchmark

We use a microbenchmark with injected imprecise store exceptions to evaluate the performance overhead incurred by both hardware and software of our prototype. The microbenchmark runs multiple iterations of a loop that applies 10 K stores to a 512 MB array. To stress the imprecise store exception handling, at the start of each iteration, the microbenchmark picks a random subset of 4KB pages and marks them as faulting using the EInject interface. The resulting imprecise store exceptions are then transparently handled by the minimal handler. The microbenchmark uses RISC-V's performance monitor counters to read clock cycle numbers from the hardware.

Figure 4.5 shows the breakdown of the overhead of handling a single faulting store. The overhead consists of three parts: 1) microarchitectural overhead, which contains the time spent on draining faulting stores to the FSB and ROB/pipeline flush, 2) OS overhead of applying the faulting store, and 3) other OS overheads such as context switches, exception dispatching, etc. We can see that in the case of our minimal handler, handling each faulting store consumes roughly 600 clock cycles, among which the microarchitectural overhead is only a tiny fraction. In more realistic cases where the handler has more complex OS logic, the overhead of microarchitecture and applying the faulting store can be largely ignored.

As explained in subsection 4.4.3, one imprecise store exception can correspond to multiple faulting stores if they are simultaneously present in the store buffer. If the exception rate is sufficiently high, then faulting stores are handled in batch, and the OS overhead per faulting store is reduced significantly, as shown in Figure 4.5. The microarchitectural overhead decreases because the store buffer is drained only once for multiple faulting stores. We anticipate

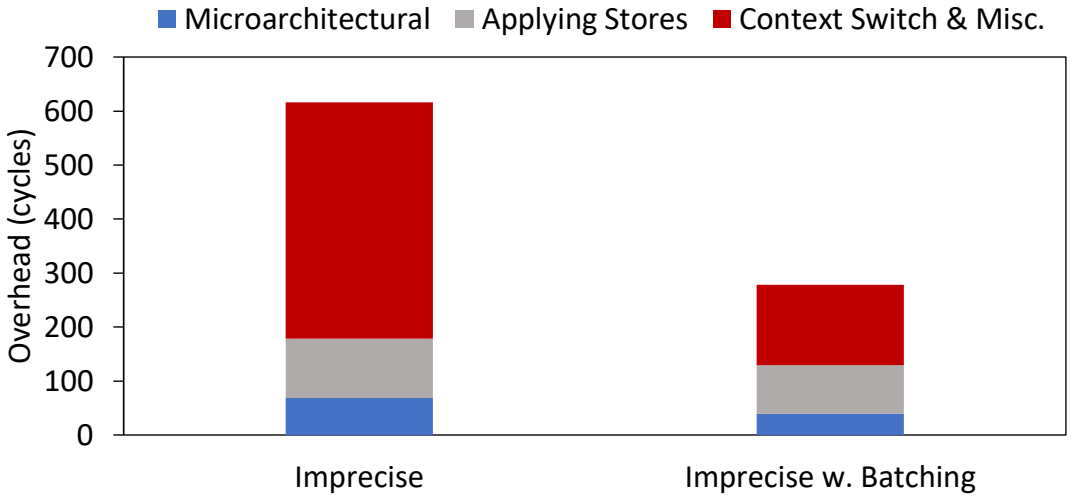


Figure 4.5: Overhead breakdown of imprecise exceptions with and without batching.

that the overhead reduction will be more prominent in realistic cases if the handler schedules batched IO operations.

4.5.5 Performance: Real Workloads

We run BFS, SSSP, and BC from GAP as well as Silo and Masstree from Tailbench to evaluate the end-to-end performance of our prototype. To inject synthetic imprecise exceptions, we modify the workloads to allocate memory for the graph (in GAP) or the request packets (in Tailbench) from the EInject region. All the allocated memory regions are marked as faulting before the workload starts. For GAP, we configure each workload to process a graph with ~1M nodes and ~8M edges to stress the handling of imprecise exceptions. We use the total execution time, including both the user and OS parts, of the computing kernel as the performance metric. For Tailbench, we run each workload in the integrated mode for a fixed duration and use the aggregated throughput as the performance metric. In both cases, the workloads run as normal Linux processes, experiencing all normal OS activities, including syscalls, task scheduling, and the handling of timer interrupts, page faults, and injected imprecise exceptions. For comparison, we run the workloads with and without imprecise exceptions, denoted as Imprecise and Baseline, respectively, to obtain the relative performance of each

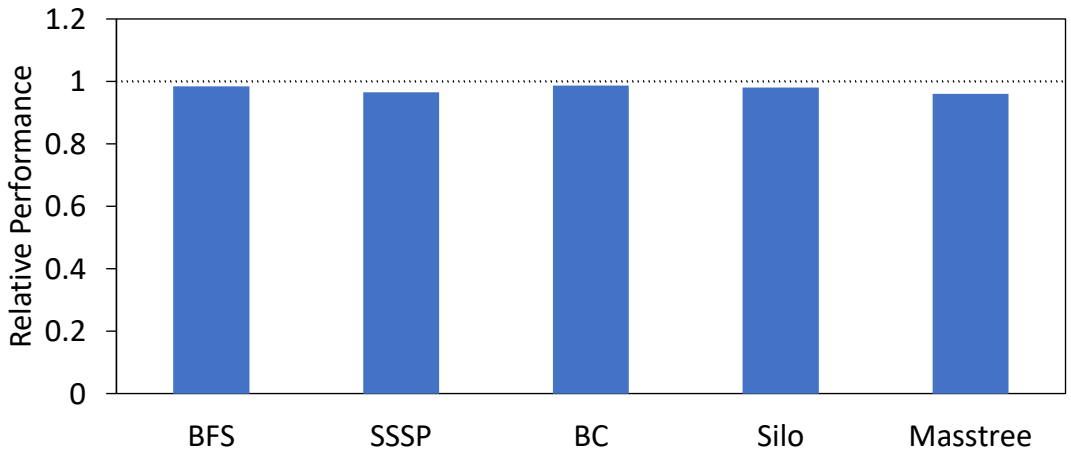


Figure 4.6: Relative performance of GAP and Tailbench workloads with imprecise store exceptions.

workload with imprecise exceptions.

All the workloads can successfully run from the start to the end. During the execution of each GAP workload, roughly 16K~32K injected imprecise store exceptions are transparently handled by Linux. For each Tailbench workload, thousands of imprecise store exceptions are injected per second, which is much larger than the frequency at which page faults are triggered in modern workloads. Our proposed prototype works and interacts flawlessly with the entire hardware/software system.

Figure 4.6 shows the relative performance of various workloads in the Imprecise case. Our prototype achieves over 96.5% of the Baseline performance for all GAP workloads. Moreover, the difference in user execution time between the Imprecise and Baseline cases is below 1%. For Silo and Masstree, handling extra exceptions absent in the Baseline case reduces the aggregated throughput by less than 4%, which is likely acceptable for the novel paradigm of imprecise exceptions. Overall, our approach enables imprecise exceptions while maintaining WC performance without excessive silicon overhead.

5 Logic-side Address Translation

As explained in section 3.2, the overall address translation in Midgard is divided into two steps: logic-side translation and memory-side translation. The logic-side translation is required to translate the virtual addresses that belong to a process-private virtual address space into Midgard addresses that belong to a unique system-wide Midgard address space that is shared by all the processes. The logic-side translation is performed at a VMA granularity and also enforces permission checks based on the VMA permissions specified by the application programmer and the toolchain. As there are typically only ~100 VMAs in typical datacenter workloads, we can easily provision silicon to capture the ~10 VMAs that are being frequently accessed, thus ensuring that the VMA-based logic-side translation attains low latency, and therefore high performance. Overall, logic-side translation serves as the fast step that is always required in the address translation process in Midgard.

5.1 OS Support

As explained in section 2.2, data in the per-process virtual address spaces is organized in terms of VMAs. To perform the logic-side translation in Midgard, the OS needs to map the VMAs present in the per-process virtual address spaces to a system-wide Midgard address space while deduplicating shared VMAs. Therefore, the OS needs to (i) track all VMAs present in the per-process virtual address spaces, (ii) perform memory management of the Midgard

address space, and (iii) expose the Virtual-to-Midgard mappings using a per-process VMA table. To perform these responsibilities, the OS needs various changes in its traditional core implementation of VM along with the design of the required tables. This section describes all the OS changes required to support the logic-side translation in Midgard.

5.1.1 Virtual Address Space Organization

The per-process virtual address spaces are fundamentally organized in terms of logical data sections, which are called Virtual Memory Areas (VMAs) in existing OSes such as Linux. Therefore, applications contain data in terms of the stack, heap, code, file, libraries, which are all represented as VMAs (as shown in Figure 2.1). Each VMA is a variable-sized region as its size can change throughout execution based on the application behavior, e.g., the stack grows in size as the application executes more functions or allocates space on the stack. For ease of implementation, the VMA size in existing systems is constrained to be a multiple of the page size, i.e., 4KB. Each VMA also has homogenous properties throughout, such as memory permissions. In case a logical data entity has multiple properties for different data pieces, multiple VMAs are used to represent it. For example, standard dynamically-linked libraries have three data portions, i.e., the execute-only code portion, the read-only initialization data such as strings, and the read-write data that contains variables to be used and modified by the application, thus requiring a separate VMA for each of the three parts. The sharing of data among processes using shared memory is also done using VMAs, i.e., each shared region is a VMA with homogenous properties. Because of such a design, VMAs are the most basic coarse-grained unit used for tracking data in the virtual address spaces. In co-ordination with the compiler and linker, the OS is responsible for managing the application's virtual address space and maintaining the list of VMAs along with their location in the virtual address space and any additional permissions and properties. In order to accommodate Midgard, we intend to expose the same VMAs as an architectural abstraction along with their virtual addresses, memory permissions, and other optional properties.

5.1.2 Midgard Address Space Organization

As explained in section 3.1, the logic-side translation requires mapping the VMAs from the virtual address spaces as MMAs in the Midgard address space. The OS requires tracking the VMAs present in each process and then mapping the VMAs across all processes to unique locations in the Midgard address space. The OS is also responsible for deduplicating shared VMAs across processes (e.g., libraries, files, or shared memory regions) and mapping them to a unique MMA in the Midgard address space. This VMA deduplication is possible because VMAs are the basic logical data blocks, and therefore any sharing explicitly takes place in terms of VMA itself. As each VMA/MMA can grow and shrink during the execution of the process, the OS needs to ensure that the MMAs mapped to the Midgard address space will rarely or never collide with each other while growing during the execution. This requirement also applies to the VMAs placed in the virtual address space (e.g., heap and stack are placed far apart so that they do not collide). As the Midgard address space is sparsely populated similar to the virtual address spaces, we have ample capacity to contain data and can lay out the MMAs with enough space between them. In rare cases when the MMAs collide while growing, they can be remapped to a different location in the Midgard address space where more space is available, or they can be split into multiple VMAs/MMAs in case of a space constraint. On one hand, remapping an MMA is an expensive operation because it requires changing the VA-to-MA and MA-to-PA mappings along with flushing the concerned data tagged with stale Midgard addresses from the cache so that it can be brought in again tagged with correct Midgard addresses. On the other hand, splitting the VMAs/MMAs will result in tracking more overall VMAs/MMAs which can potentially slow down the logic-side translation.

Overall, the total required size of the Midgard address space depends on the total size of unique data present in the system. We expect that in most cases, the total size of data in the system is bound by the total capacity of the storage devices present in the system. Typically, the storage capacity is at most 10-100x that of the memory capacity present in the system [6]. Therefore, for a system with 10TB of memory, we expect 1000TB of storage and consequently require 50 bits worth of total addresses to store/map all the data. To ensure that the Midgard

address space is sparsely populated and avoid collision between MMAs, we would expect that the Midgard address space is 10-100x overprovisioned, thus requiring the address space to be sized to 57 or 64 bits. Recent announcements by Intel [55] about increasing the size of the virtual address space because of the increasing memory and storage capacity lead us to believe that they size their virtual address spaces with similar reasoning. Sizing the Midgard address space is similar to sizing the virtual address space because in systems today, a single process can directly map all or most of the memory and storage capacity, and thus behave similar to Midgard. However, with network-attached storage systems prevalent today, large files can consume unprecedented Midgard address space if mapped. Similar to the virtual address space, these files require software solutions to be accommodated, e.g., the files can be split so that only the parts being used are mapped, thus requiring more VMAs/MMAs but consuming little Midgard space under pressure.

Even if we can ensure that the Midgard address space is sized correctly to accommodate all the data present in memory and storage, we still need to place the MMAs far apart so that they do not collide while growing in the common case. Therefore, designing and tuning the algorithm for MMA placement is important to minimize the MMA collisions. We expect that a memory management algorithm similar to the buddy allocator can be used to solve the MMA placement problem in most cases. The algorithm divides the Midgard address space into regions which will contain VMAs with properties and capacity that can be classified into various categories. For example, VMAs such as code/text/data/bss which do not grow or shrink during the program execution can be put compactly together in a region as they are not going to change. In contrast, VMAs such as stack/heap/files can be classified based on their capacity, where MMAs with less than 1MB capacity are stored in a particular region of Midgard that is managed with 1MB blocks. As the VMAs grow or shrink, they can be remapped to other regions which is managed using larger or smaller blocks, e.g. doubling the block size in the bigger regions while halving the block size in the smaller regions. As the Midgard address space is logical and sparse, we do not need to particularly optimize for the internal fragmentation in these blocks.

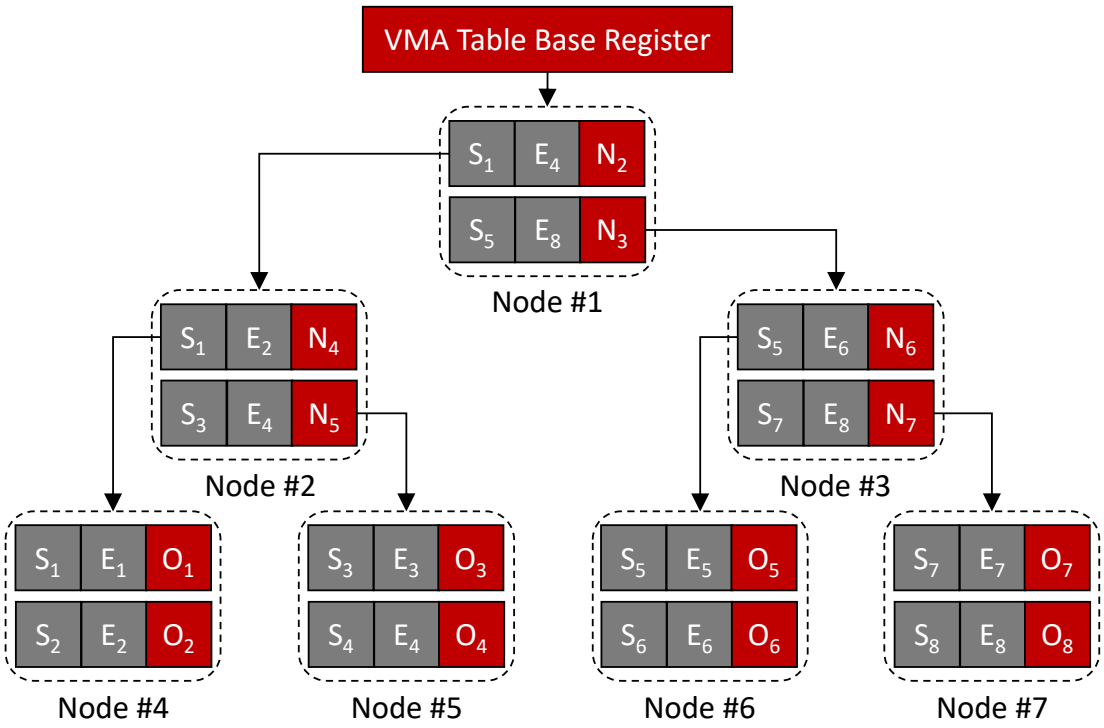


Figure 5.1: A diagram of a hypothetical VMA table where each node has two child nodes. The VMA Table Base Register is a per-core register that contains the base Midgard address of the VMA table of the currently scheduled process. Each dotted box represents a node in the VMA table, while the contained horizontal rectangles represent two entries per node. The grey boxes signify the stored virtual address, while the red boxes signify the stored Midgard address. Finally, S_i indicates the starting virtual address of the region i, E_i indicates the ending virtual address of the region i, and N_i or O_i indicates the Midgard address of the child node i, or the offset of the region i in the Midgard address space respectively. The diagram shows that leaf nodes directly track the regions corresponding to the VMAs, while the non-leaf nodes track the regions corresponding to their child nodes.

5.1.3 Tracking Virtual-to-Midgard Mappings in VMA Tables

The OS needs to track virtual-to-Midgard address space mappings per process, where the tracking structure needs to be architecturally exposed. As the mappings are created at a VMA granularity, we call this per-process architectural structure a *VMA table*. The VMA table for each process tracks all the VMAs present and their mappings to MMAs and memory permissions. As the VMA size is variable and can change throughout the execution, we implement the VMA table as a B+ tree which is typically used to track variable-sized regions. Accordingly, the ranges in Karakostas et al. [64] are tracked using a B+ tree as well. The OS stores the VMA tables for

each process as separate MMAs in the Midgard address space; therefore, any references to the nodes in the table are made using Midgard addresses. Each node in the tree contains multiple entries, where each entry contains the base and bound of the tracked region and the address of the resulting mapped region. Each entry tracks three addresses, thus resulting in 24B size, where the memory permissions and other optional hardware-exposed attributes can be stored in the unused bits.

Each entry in the parent nodes in the B+ tree tracks the total range and address of the child node, while each entry in the leaf nodes tracks the VMA range, address of the correspondingly mapped MMA, and memory permissions as shown in Figure 5.1. A tree traversal is required to find the required VMA mapping, where multiple entries in the same node might need to be looked up to find the correct entry leading to the child node and consequently to the final MMA address. For a given number of total VMAs, while fatter nodes can reduce the total number of levels required in the tree as each node can accommodate more children, fatter nodes also result in more entry lookups per node, therefore creating a tradeoff between node size and the number of levels in the tree. Our design selects the node size to be 128B, i.e., two cache lines, to benefit from the next line prefetching and cache line locality. Therefore, each node can support five entries, thus enabling tracking up to 125 VMAs with a three-level B+ tree. Each process-private B+ tree needs to be architecturally exposed using per-core *VMA Table Base Register* like the CR3 in x86 architecture, which will then contain the (Midgard) root address of the tree. As the B+ tree is architecturally exposed, a hardware tree walker can be easily designed to walk the tree when required.

While the B+ tree provides node lookups with $O(logn)$ complexity which is similar to the radix page tables, it is considerably more sophisticated as data structure compared to the radix page tables prevalent today. We envision that there can be a use case for a simpler data structure in case of applications that do require fast lookups, but do not require all the benefits of maintaining a B+ tree for storing their VMA mappings. Fortunately, we can easily achieve the $O(logn)$ lookup complexity using a sorted array to store all the leaf nodes that are present in the B+ tree, thus simply representing a collection of VMA mappings that are

sorted based on their starting addresses in the process-private virtual address space. In case of such a sorted array, we can use the binary search algorithm to find the required VMA with $O(logn)$ complexity. However, when using the sorted array as a VMA table, inserting a new VMA mapping entry requires shifting all the following VMA mapping entries. Therefore, such a sorted array can only be used in cases of applications which do not create or delete VMAs frequently. Overall, depending on the application VMA-usage pattern, the developer can indicate the OS to use the VMA table as either a B+ tree or a sorted array, assuming that the hardware implementation supports both formats.

5.2 Microarchitectural Support

The main advantage of having a VMA-based logic-side translation is that there are only a small number of total VMAs, while only a few of them are frequently used. Such a significant reduction in the overall translation state with VMAs allows using very little hardware support to guarantee fast virtual-to-Midgard translation, along with fast access to the cache hierarchy. Consequently, this section describes the hardware support required at the cores, accelerators, devices, and various other logic entities to enable fast logic-side translation and accommodate the Midgard address space.

5.2.1 Cores and Integrated Accelerators

In order to enable fast cache accesses, we need to ensure fast virtual-to-Midgard address translations, which are required for every memory instruction. As shown in previous papers [80, 135], while there are typically around 100 VMAs per process in existing workloads, only ~10 are frequently accessed and can be efficiently cached in hardware with little resources. Similar to the existing caching support for address translation with TLBs, we introduce Virtual Lookaside Buffers (VLBs), which cache the recently used virtual-to-Midgard mappings with VMA granularity. With only ~10 active VMAs, we expect that small but correctly provisioned VLBs with 10-20 entries can provide almost perfect caching with rare VLB misses, thus ensuring fast translations for most memory instructions. In case there is a VLB miss, or if there is no

caching support in hardware, a VMA table walk will be required to complete the translation, as explained in subsection 5.1.3. After the VMA-table walk, the obtained mapping is installed in the VLB for future accesses, where each entry consists of a base and bound address of the VMA, along with an offset to locate the corresponding MMA in the Midgard address space.

However, there is a fundamental lookup-latency-based design difference between hardware caching for page-based translations and VMA-based translations. To check if a page-based TLB entry matches the input virtual address, we need an equality comparison between the input and cached address, which can be easily done by using parallel comparators for each bit. However, in the case of VLB lookups, as each entry caches the VMA base and bound addresses, we require two inequality comparisons as the input address should fall within the base and bound addresses. The latency of the inequality comparison is typically greater than that of the equality comparison and depends on the bit-width of the address and the number of VLB entries compared simultaneously. As existing processors can typically issue at least one memory request per cycle, the VLB design should perform each lookup in one cycle as well; otherwise, we risk introducing additional stalls in the common case. We synthesize a simple VLB design with 16 entries for a 2GHz clock cycle using a 22nm CMOS library to measure the VLB lookup latency quantitatively. For 64-bit processors, the base and bound registers can be a maximum of 52 bits each as the VMA capacity is a multiple of the 4KB page size. The access time for such a VLB is 0.47ns, thus consuming almost the whole clock cycle. While it might be possible for the evaluated VLB design to meet the timing requirements in some processor designs, the VLB access time can worsen with an increase in complexity (e.g., increase in the number of ports or higher clock rate), thus making it unclear if a VMA-based VLB can perform lookups at the required rate. Therefore, we conservatively assume that the VMA-based VLB should not directly interact with the core as it might not keep up with the timing requirements of the core.

In order to combat the timing constraints, we introduce a two-level VLB design where the first level contains the existing page-granularity mappings, while the second level contains the VMA-granularity mappings, as shown in Figure 5.2. The L1 VLB is similar to a sectored cache as

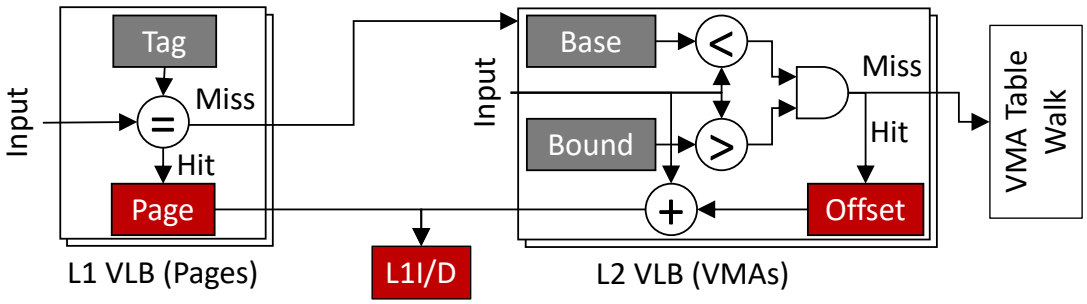


Figure 5.2: A two-level VLB design to accommodate the long latency of range lookups. The L1 VLB is similar to an existing page-based TLB where each entry represents a fixed-size page mapping, allowing fast checks with equality comparisons. The L2 VLB contains VMA granularity mappings, and therefore each entry check requires two inequality comparisons, thus incurring relatively high latency. Therefore, we suggest a two-level design where the L1 VLB acts as a sectored cache for the L2 VLB while offering fast latency lookups.

it caches page-based mappings, which are constructed dynamically from the VMA-granularity mappings present in the L2 VLB if the L1 VLB suffers a miss. As the L1 VLB can offer a high hit rate, most translation requests can be served with fast L1 latency without requiring L2 lookups. Only when the L1 VLB does not contain the required entry, the L2 VLB is looked up, and if the corresponding VMA is found, then a page-based entry is dynamically created and installed in the L1 VLB. If the required VMA is not found in the L2 VLB, then a VMA table walk is performed, the resulting VMA entry is installed in the L2 VLB, and the corresponding page-based entry is installed in the L1 VLB.

Similar to the TLBs present in existing systems, the coherence among per-core VLBs requires software support. As a process with multiple threads can have its VMA mappings cached on multiple cores, any changes to the relevant mappings require a global shootdown to eliminate the mapping from all cores. However, VMA mapping modifications (because of application behavior) are relatively infrequent compared to the page mapping modifications (because of memory management), resulting in low overhead because of the VLB shootdowns. If required, most of the proposals [9, 10, 72] for optimizing the TLB shootdown overheads are also applicable for VLB shootdowns.

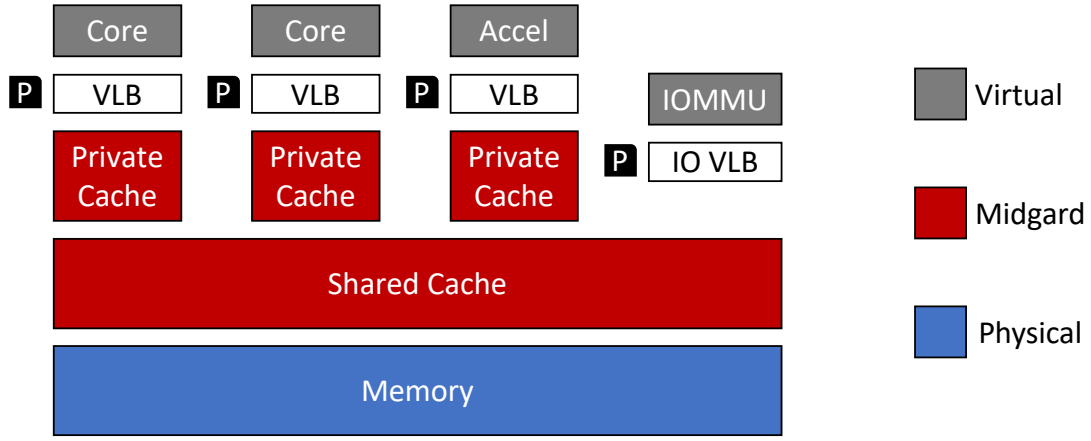


Figure 5.3: Logic-side translation is applicable for logic entities such as cores and integrated accelerators, along with any PCIe-attached devices that might communicate with the cache hierarchy and memory. All such entities require using a VLB so that they can translate virtual addresses to Midgard addresses which are required to index the cache hierarchy.

5.2.2 IO Devices and Discrete Accelerators

While the cores and integrated accelerators need a VLB to translate virtual addresses to Midgard addresses in order to access data from the cache hierarchy, IO devices and discreet accelerators also need similar functionality. Typical IO devices and discreet accelerators today are attached to the CPU using interconnects such as PCIe which has an integrated PCIe controller called 'Root Complex' on the CPU chip. The Root Complex is co-located with an IO MMU, i.e. an MMU that is specifically used for IO devices and integrated accelerators. There is only one IOMMU for each PCIe controller, and therefore it is shared across all the cores and integrated accelerators so that they can interact with the IO devices and vice versa. In traditional systems, IOMMUs also have their own set of IO TLBs which are used to cache the address translation mappings frequently used by the IO devices, and commercial IO TLBs today have thousands of entries. However, in Midgard, even the IO devices need to perform logic-side translation so that they can translate the required virtual addresses to corresponding Midgard addresses which can then be used to index the cache hierarchy, and eventually be translated to physical addresses for memory lookups if required. Therefore, each IOMMU should be coupled with an *IO VLB* instead of an IO TLB, as shown in Figure 5.3. As IO VLBs will perform address translation at a VMA granularity, we need significantly fewer IO VLB entries

compared to IO TLB entries. We expect that typical IO VLBs will require only tens of entries to support the frequent interactions with the IO devices.

5.2.3 Accessing the Cache Hierarchy

As the cache hierarchy is indexed using the physical address in the existing address translation scheme, the cache hierarchy lookups are serially dependent on the virtual-to-physical address translation. Until recently, processors supported the VIPT (Virtually-Indexed Physically-Tagged) optimization [90] which enables a partial overlap between the address translation and the cache lookup by using the virtual address bits to index the cache but using the physical address bits to check the tag, where only the tag checks are dependent on the address translation. In existing systems with page-based address translation, the last 12 bits of the address corresponding to the offset inside the 4KB page do not change after the translation, thus leading to the VIPT insight of using offset bits to index the cache. The page offset bits are then used to find the correct set in the cache, requiring the cache index bits to be contained in the offset bits, while the non-offset bits undergo address translation and are then serially used for the tag comparison. For cache blocks of 64B size, 6 bits are reserved for block offset, while the remaining 6 bits in the page offset can be used as set index bits, thus allowing up to 64 sets in the cache.

As the number of sets is limited by the page offset bits in VIPT, the cache capacity can only be increased by adding more ways that are expensive as they require extra comparators. However, because of the increasing demands on cache capacity, existing processors have stopped using the VIPT optimization to add more sets, thus requiring the address translation to be completed serially before the cache can be accessed. E.g., the recent ARM cores such as Cortex A76 [129] support 64KB 4-way associative L1 caches, thus indicating that there are 256 sets requiring eight set-index bits which cannot be accommodated in the page offset bits. Introducing the Midgard address space allows decoupling the allocation granularity in the virtual and Midgard address space from the physical address space (4KB). Therefore, the virtual and Midgard address space can have a larger allocation granularity, e.g., 2MB, by ensuring that the

VMA/MMA capacity is always a multiple of 2MB. Even if the VMA/MMA capacity is increased by 2MB, not all need to be mapped to physical pages until required. A bigger granularity increases the number of offset bits that do not change after translation, thus allowing more set-index bits and ensuring that the L1 caches can accommodate more sets with a corresponding VIMT (Virtually-Indexed Midgard-Tagged) optimization without requiring numerous ways for higher cache capacity.

One of the main advantages of introducing the Midgard address space is that the logic-side translation happens at a VMA granularity and therefore is significantly easier to support in hardware. In contrast, while the memory-side translation still takes place at a page granularity, it is only required if the requested cache block is not found in the cache hierarchy and must be retrieved from memory, thus requiring physical addresses. With the trend of increasing cache hierarchy capacities already visible in existing processors, we believe that the cache hierarchies will serve cache blocks for the majority of the memory instructions, therefore requiring the memory-side translation only in rare cases. While Intel Pentium 4 [2] featured only 256KB of LLC capacity, existing CPUs such as Intel Kabylake [130] provide 128MB of eDRAM memory-side cache capacity, and AMD Zen3 [128] provides 256MB of SRAM cache capacity spread across its chiplets with a total of 64 cores. There also are commercial products such as Intel Knights Landing [113] which features a 16GB multi-channel DRAM cache that provides data access at high bandwidth. Moreover, the upcoming Intel Sapphire Rapids [54] promises to provide 64GB of stacked HBM2 memory using 4× 16GB HBM2 stacks, where the HBM2 can also be configured as a hardware-managed cache (similar to Intel Knights Landing) in conjunction with a backing DDR5 memory pool.

With such high cache hierarchy capacities, it is most probable that the frequently used data resides in the cache, and thus most of the memory instructions can retrieve the data from the cache hierarchy itself instead of requiring physical memory accesses. Thus, well-provisioned cache hierarchies can avoid memory-side translation in most cases and eliminate the address translation overhead. Apart from simply filtering most memory accesses, the cache hierarchy also filters the temporal locality present in the remaining accesses. We expect that most of

the accesses that miss in the cache hierarchy are targeted towards a recently non-accessed block but might be located near a recently-accessed block as the spatial locality is still present in the accesses. As there is no or little temporal locality left in the accesses missing in the cache hierarchy, there is no significant benefit from provisioning large, specialized caches for memory-side translation. Assuming that Midgard addresses are 64-bit, there are 12 more bits per address than existing 52-bit physical addresses, requiring extra bits in the cache hierarchy. For a 16-core system with 64KB L1 data and instruction caches, along with a 1MB LLC slice per core, there are a total of ~320K cache blocks, thus requiring 480KB extra silicon to support the Midgard-addressed cache hierarchy.

6 Memory-side Address Translation

As explained in section 3.2, the overall address translation in Midgard is divided into two steps: logic-side translation and memory-side translation. The memory-side translation is required to translate the Midgard addresses that belong to a unique system-wide Midgard address space that is shared by all the processes into physical addresses that belong to a unique but capacity-constrained address space managed by the OS. The memory-side translation is performed at a page granularity because it is required for effective capacity management of the underlying memory device. The memory-side translation also enforces permission checks based on the page-based permissions specified by the OS for deploying memory management optimizations such as copy-on-write. The memory-side translation is also responsible for maintaining page-based usage information using access and dirty bits. While page-based translations are slow compared to VMA-based translations, the memory-side translation is performed infrequently as it is only required when a needed cache block is not found in the cache hierarchy and thus has to be retrieved from the memory. As shown in Table 2.1, modern datacenter servers feature GB-scale cache hierarchy capacity thus allowing modern online services to host their working set in the cache hierarchy, leading to only infrequent misses in the cache hierarchy that ultimately require memory-side translation. Overall, memory-side translation serves as the slow step that is only infrequently required in the address translation process in Midgard.

6.1 OS Support

As explained in section 2.2, data in the system-wide physical address space is organized in terms of pages to efficiently manage the physical address space capacity while eliminating external fragmentation. To perform the memory-side translation in Midgard, the OS needs to map the MMAs present in the system-wide Midgard address space to pages present in a system-wide physical address space. The mapping from MMAs to pages typically observes a one-to-many pattern as each MMA is broken down and mapped to various physical pages. The only exception is observed when the OS performs memory management optimizations such as copy-on-write, and then parts of multiple MMAs might be mapped to the same physical pages, thus providing a many-to-many mapping between MMAs and pages. Overall, the OS breaks down an MMA and maps it to a multitude of pages which can be potentially scattered throughout the physical address space, thus creating a large amount of metadata that has to be looked up for each memory-side translation, therefore causing performance concerns similar to traditional address translation. However, as memory-side translation is only required in case of infrequent misses in the cache hierarchy, it does not impact performance in the common case. Therefore, the OS needs to (i) track all MMAs present in the system-wide Midgard address space, (ii) perform memory management of the physical address space using pages, and (iii) expose the Midgard-to-physical mappings using a system-wide page table. To perform these responsibilities, the OS needs various changes in its traditional core implementation of VM along with the design of the required tables. This section describes all the OS changes required to support the memory-side translation in Midgard.

6.1.1 Physical Address Space Organization

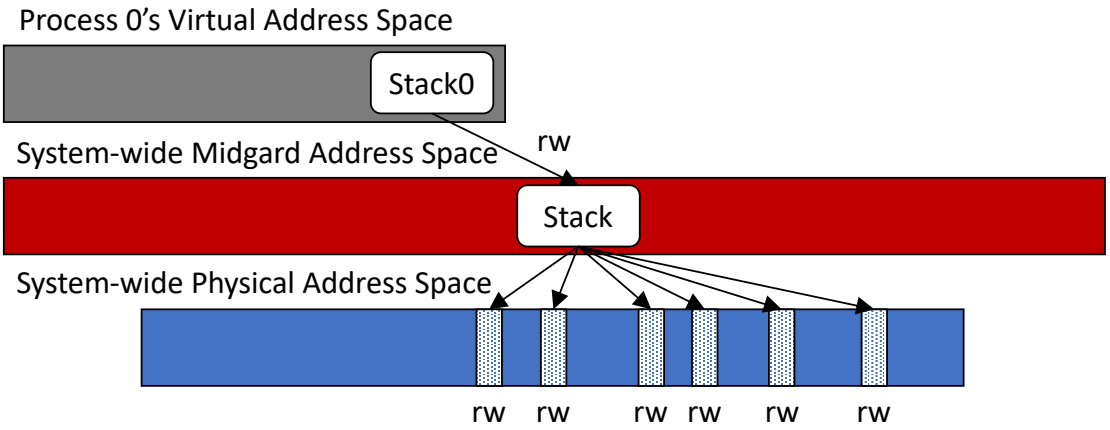
The system-wide physical address space is organized in terms of pages, which are fixed-sized units used to efficiently manage the physical memory capacity and eliminate external fragmentation, as explained in section 2.2. Therefore, while the application organizes its data in terms of VMAs, the OS transparently manages the underlying physical address space using pages (as shown in Figure 2.1) while providing various memory management features such as

over/under-subscription and swapping data to storage devices. 4KB is used as the typical page size in modern systems today. As the OS performs physical memory management in terms of pages, each page can have different permissions and attributes depending on which MMA it belongs to, or what memory management optimizations are being applied by the OS. While in traditional systems, the data sharing among processes is resolved at a page granularity, in case of Midgard, the data sharing is resolved at a VMA/MMA granularity, and therefore each page typically belongs to only one MMA unless the OS applies memory management optimizations such as copy-on-write.

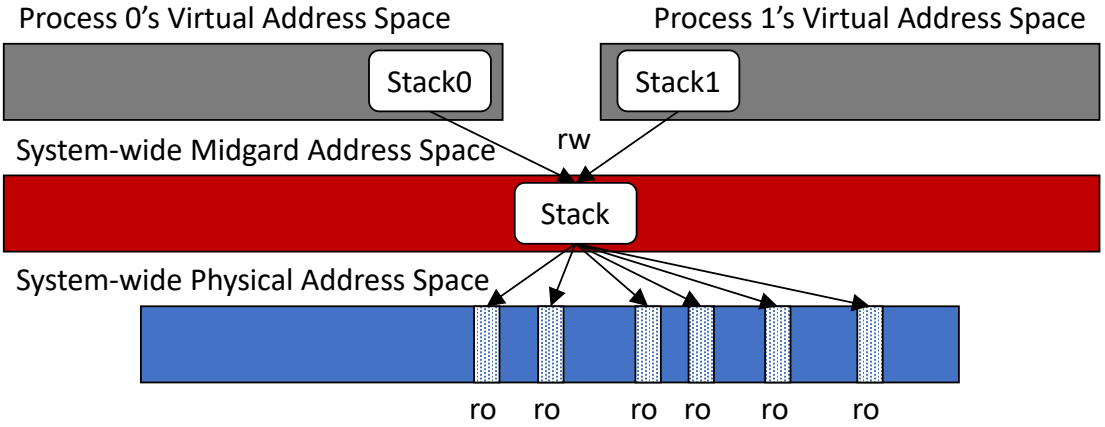
6.1.2 OS Permissions for Memory Management

Memory permissions today can be conceptually broken down into two classes: permissions specified by the applications and the OS, respectively. The application permissions are required for the semantic correctness of the program and are specified by the application and toolchains together at a VMA granularity. Using Midgard, the application permissions are architecturally exposed using the VMA tables and perform protection checks for every memory access at the core side itself. In contrast, the OS permissions are enforced for memory capacity optimizations at a page granularity by the OS, e.g., in the case of a copy-on-write optimization, the OS can deduplicate the common data across processes and have the processes use the same physical pages. In existing systems, the application and OS permissions are merged and architecturally exposed using the process-private page tables.

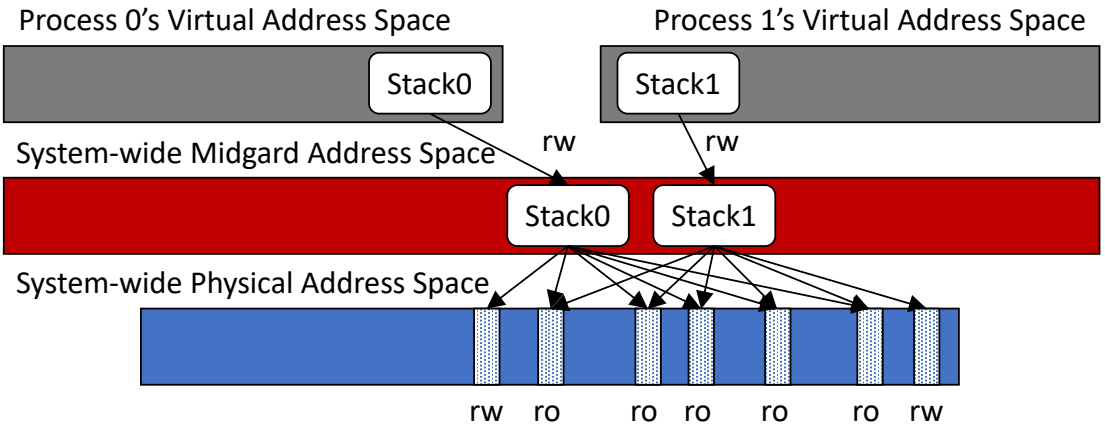
However, using Midgard, as the application permissions are specified separately at a VMA granularity, the page-granularity-based OS permissions cannot be superimposed on the same anymore. Moreover, as the OS permissions are not specific to any process but are applicable for the whole system, they should instead be exposed using the system-wide page tables containing the Midgard-to-physical address mappings. Figure 6.1 shows an example of a copy-on-write optimization with Midgard in the case of a process fork. First, the VMAs from the original process are created in the newly created process as well, but as they refer to the same data, they both point to the same MMA. Moreover, the OS permissions denoted by



(a) Before Process 0 forks, the OS has marked all pages with *rw* permissions.



(b) After Process 0 forks, the OS maps the Stack1 VMA to the Stack MMA and marks all pages with *ro* permissions.



(c) After Process 0 attempts to write to its stack, the OS receives an exception for an attempted write on a page with *ro* permissions. The OS then creates a copy of the page and remarks the page with *rw* permissions.

Figure 6.1: Copy-on-write example using Midgard and memory-side translation permissions.

memory-side translation change from *rw* to *ro* for all the pages so that a write initiated by any process causes a trap leading to the duplication of the page. Then, when any process initiates a write, an exception is triggered, and a new MMA is created for the second process as both the processes will contain different data after the write. Finally, a new physical page with *rw* permissions is allocated for the second process, and the permissions of the original page change from *ro* to *rw* enabling both the processes to write to their respective pages. As in the example, in the case of aggressive OS optimizations for memory capacity, different addresses in the Midgard address space may point to the same physical page, therefore giving rise to synonyms. However, it is ensured that when multiple Midgard addresses are mapped to the same physical page, the physical page is marked as read-only (*ro*), thus ensuring that the synonyms do not actively create cache coherence problems. As these synonyms only consume extra space in the cache hierarchy under different Midgard addresses while being contained in the same physical page, we consider these synonyms benign and use them to implement OS-based memory optimizations using Midgard.

Such permissions also allow supporting other memory management optimizations other than copy-on-write. For example, Linux currently implements Kernel Same-page Merging (KSM) [121] as an optimization to save memory capacity where the kernel identifies physical pages with the same content, and then deduplicates them to save space. KSM has been shown to provide significant benefits especially in virtualized environments where each virtual machine might be running similar kernel and libraries. Midgard can also accommodate such memory optimizations in the memory-side translation using the permission bits as for copy-on-write. For example, in case of KSM, after identifying duplicate pages belonging to two different MMAs, the OS needs to free one of the pages, and make the other page read-only while ensuring that both of the original MMAs now point to the same page. Even though mapping the same page to multiple MMAs can create synonyms in the cache hierarchy, as the memory-side permissions for the page are marked as read-only, the read-only synonyms will not create any problems with cache coherence. If the applications try to write to the concerned page, then the OS will trigger a fault and duplicate the page before modifying the contents.

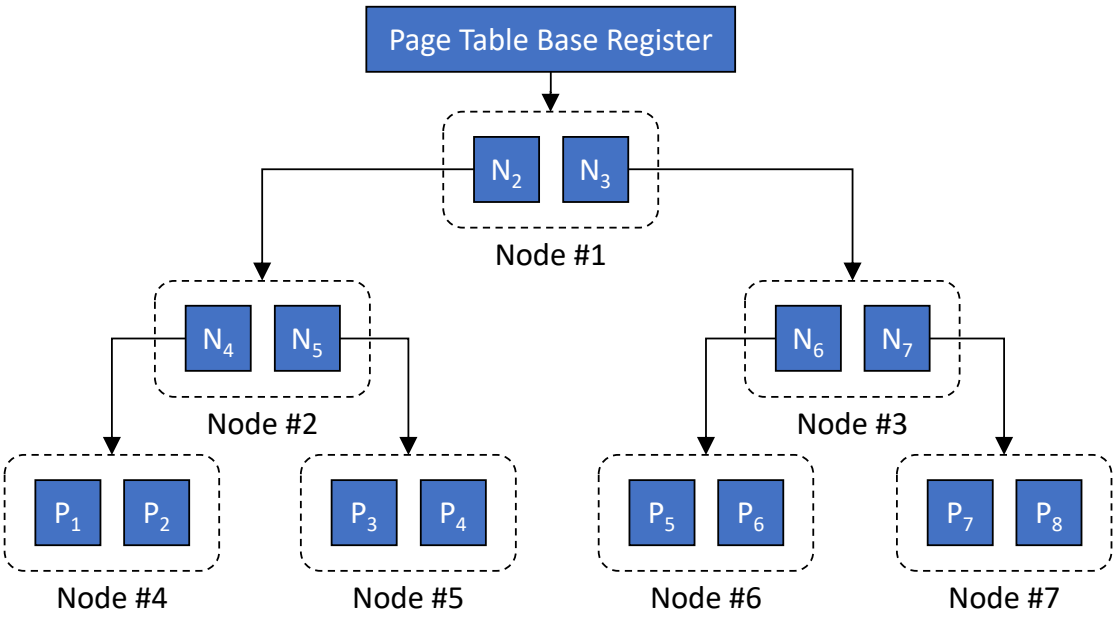


Figure 6.2: A diagram of a hypothetical page table where each node has two child nodes. The Page Table Base Register is a system-wide register that contains the physical address of the root of the system-wide Midgard page table. Each dotted box represents a node in the page table, while the contained boxes represent two entries per node. Every contained box represents an entry that stores the physical address of the child node, or the page in case of the leaf nodes. Finally, N_i indicates the physical address of the child node i, while the P_i indicates the physical address of the page i tracked by the corresponding entry. The diagram shows that leaf nodes directly track the pages, while the non-leaf nodes track the pages corresponding to their child nodes.

6.1.3 Tracking Midgard-to-Physical Mappings

As the Midgard and physical address spaces are unique for the whole system, the Midgard-to-physical address mappings are also maintained at a system level instead of a per-process level. The OS is responsible for tracking the address mappings for the whole system and storing them in an architecturally-exposed structure. As the physical address space is managed in fixed-size pages to utilize the memory capacity efficiently, the Midgard-to-physical address mappings are also created at page granularity. As in existing systems, these page-based mappings can be easily tracked with a system-wide radix page table which offers storage benefits by not allocating space for unmapped regions. We call this system-wide architectural structure a *Midgard page table*, as shown in Figure 6.2. The Midgard page table needs to be architecturally

exposed using a system-wide *Page Table Base Register* unlike the per-core registers. For a Midgard address space of 57 bits, we would need a page table with five levels, while for 64 bits, we would need a page table with six levels. Finally, similar to the existing systems, all the addresses used in the Midgard page tables are physical addresses where the non-leaf nodes contain the physical address of their child nodes, while the leaf nodes contain the address of the physical pages containing actual data.

6.2 Microarchitectural Support

After the logic-side translation completes and is followed by the cache hierarchy lookups, if the requested cache block is not found in the cache hierarchy, then we require a memory-side translation to convert the Midgard address to the physical address as the latter is required to index the physical memory and fetch the required cache block. As the memory-side translation happens away from the cores, there is an inherent resource and latency slack possible due to its infrequent occurrence because most of the memory instructions are served the requested data directly from the cache hierarchy. As both the Midgard and physical address spaces are unique in the system, the memory-side translation also pertains to the whole system instead of separate processes, allowing a myriad of optimizations as described in this section.

6.2.1 Bottom-Up Page-Table Walk

As explained in subsection 5.2.3, the cache hierarchy filters most of the temporal locality from the memory accesses that did not find the required cache block in the cache hierarchy. Therefore, there is little benefit in provisioning specialized caches for memory-side translation. If there are no specialized caches, or if the required mapping is not present in the specialized caches, then the memory-side translation requires a Midgard page-table walk similar to existing systems. However, the introduction of the Midgard address space allows optimizing the page-table walk by utilizing the logical nature of the address space.

Similar to existing systems, the page-based mappings for memory-side translation are stored

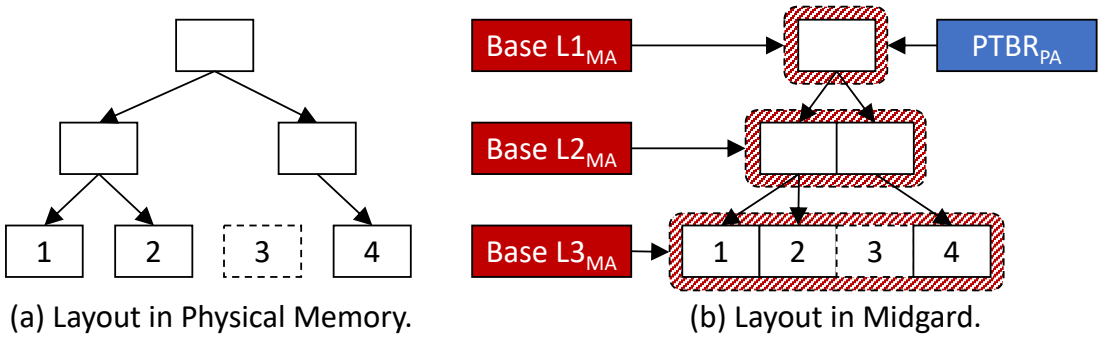


Figure 6.3: A hypothetical layout of a two-level radix page table with a radix degree of two. In the diagram, (a) shows the page-table layout in the physical memory, where every page-table node can be in different parts of the physical address space, while (b) shows the page-table layout in the Midgard address sapce where all the page-table nodes are placed contiguously, thus allowing direct offset-based lookups of the page-table entries.

in a radix page table as it provides capacity benefits by not reserving space for unallocated mappings, as shown in Figure 6.3a. Assuming a hypothetical 64-bit Midgard address space, the OS might need to track mappings for at most 2^{52} pages where each mapping occupies 8B, thus requiring a total of 2^{55}B worth of space. However, as the physical memory capacity is limited (assuming 16TB, i.e., 44 bits), only 2^{32} pages will be present, thus requiring only 2^{35}B worth of space. Therefore, a radix page table is highly beneficial for representing such a configuration as it wastes only a little space on page-table nodes that are unused and therefore requires only a little bit more than 2^{35}B space. For simplicity, we ignore the space requirement of the upper levels in the page table as their sizes are much smaller.

However, the space savings in the radix page table come at the cost of lookup latency, as each leaf node access requires the complete radix tree traversal. As the Midgard address space is logical and sparse, it allows reserving a large capacity region without being concerned about the efficient usage of the address space. Therefore, inspired by In-Cache Address Translation [132], we propose a method of mapping the page tables contiguously in the Midgard address space such that every entry can be directly looked up in the Midgard address space and consequently the cache hierarchy without requiring a full radix page-table walk. If the required page-table entry is not present in the cache hierarchy, a bottom-up page-table walk is required, which turns into a regular page-table walk as soon as a parent node is found

in the cache hierarchy.

The Midgard address space allows the existing system-wide Midgard page table to be remapped in its expanded form, as shown in Figure 6.3b. In the Midgard address space, each page table level can be completely expanded while also allocating space for nodes that are not allocated in the default version of the page table resident in the physical memory. Therefore, the last level will indeed occupy 2^{55}B of space in the Midgard address space, but as Midgard is a logical address space, we can easily spare such space for the page tables. Mapping the page table in its extended form allows accessing the required page-table entry at any level using a simple base + offset calculation. E.g., calculating the Midgard address of the N^{th} entry in a page-table level would require adding 'N × size of page-table entry' to the base address of the page-table level. We can also store all the levels contiguously with each other, therefore eliminating the need to record each level's base addresses. Using the calculated Midgard address, we can directly store and lookup page-table entries in the cache hierarchy, which is already indexed using Midgard addresses.

If the cache block corresponding to the required page-table entry is present in the cache hierarchy, then the entry can be directly accessed using the calculated address without requiring the full Midgard page-table walk. However, if the required cache block is not present, then it should be fetched from physical memory, thus requiring its physical address, which is contained in the parent page-table entry. Therefore, there will be a recursive attempt to read the parent page-table entry from the cache hierarchy to obtain the required physical address for each page-table entry that encounters a miss in the cache hierarchy. If the parent page-table entry is present in the cache hierarchy, then the physical address of the child page-table entry will be read, and the child table entry will be brought in the cache hierarchy. If none of the required parent/child page-table entries are present at any level, then the bottom-up page-table walk would revert to reading the physical address of the root node of the page table from the *Page Table Base Register (PTBR)*. After the PTBR is read, a full page-table walk is required directly from the physical memory similar to existing systems. We observe that the average Midgard page-table walk requires looking up only one or two page-table levels based

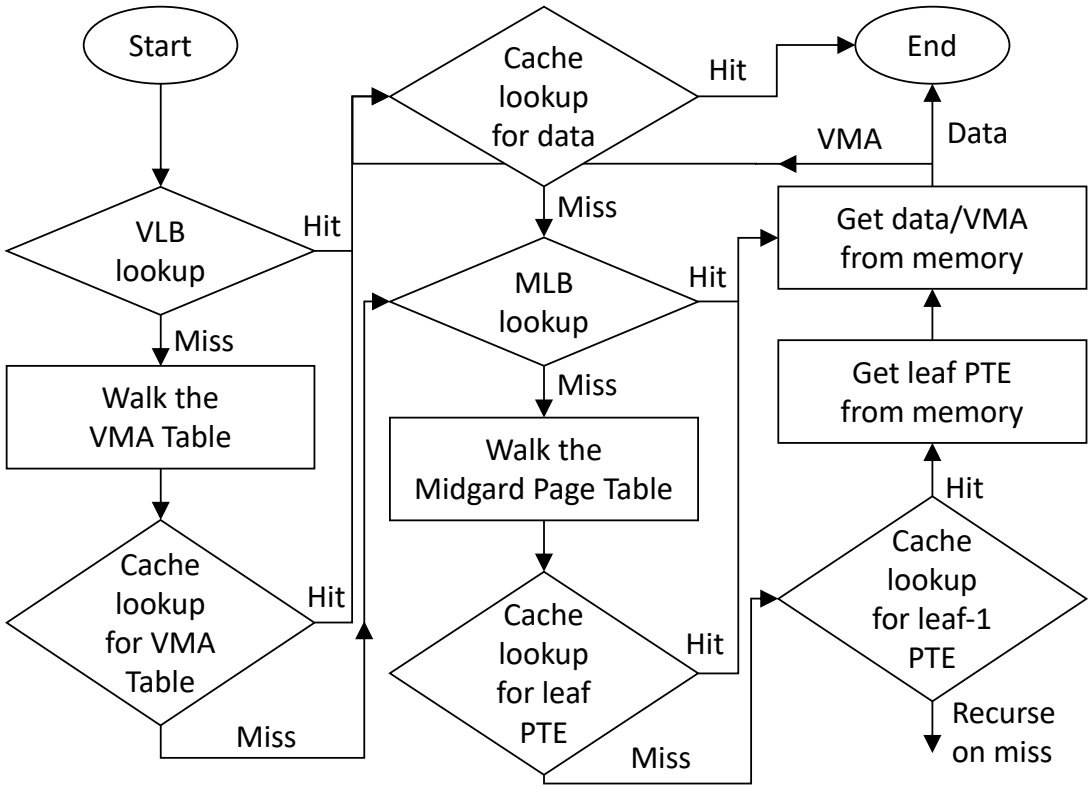


Figure 6.4: The flowchart depicts all the steps required for the completion of a memory access while considering hits and misses in the various caches provisioned for address translation and data.

on typical workload characteristics. Enabling direct accesses to any page-table level allows short-circuiting the existing page-table walk if the required entry is found, thus providing the same benefits as the MMU caches present in existing systems. Therefore, the introduction of the Midgard address space allows the generic cache hierarchy to behave as both the TLB and the MMU caches as it enables direct lookups of the leaf page-table entries and parent page-table entries without any specialized structures.

6.2.2 Explicit Caching for Page-Table Entries with MLBs

As explained in subsection 6.2.1, the memory-side translation does not necessarily require specialized caching structures and can directly read the required page-table entries from the cache hierarchies. As the memory-side translation happens near the memory, a practical

design choice would be to use the shared LLC tiles only while caching the page-table entries in the cache hierarchy. However, existing systems enjoy the fast access of page-table entries by bringing them even to the private L1 caches. Therefore, specialized fast caches can be provisioned for memory-side translation if the workload requires, which we call Midgard Lookaside Buffer (MLB). The operations for completing memory access are shown in Figure 6.4.

The introduction of the Midgard address space enables two optimizations in the design of MLBs. As the Midgard page tables are unique in the whole system, the MLBs that will cache the recently-used page-table entries will also be a system-wide hardware structure in contrast to the per-core TLBs in existing systems. Therefore, the MLBs will be implemented as a logically centralized structure even if they are implemented in physically-distributed slices, similar to the shared LLC slices present in existing systems. Being a centralized structure, the MLB can benefit from constructive interference if various cores use the same set of physical pages and, therefore, share the MLB entries.

Next, the MLB need not be necessarily implemented like existing TLBs, i.e., it does not need to tag address translation mappings with their Midgard addresses. Because of the direct lookup of page-table table entries possible in the cache hierarchy, the Midgard address of a page can be directly transformed into the Midgard address of the corresponding page-table entry. Therefore, the MLB can use the Midgard address of the page-table entry directly, thus cache the page-table entry instead of caching a mapping. Such addressing enables the MLB to be implemented as a generic cache design which is only used for caching the page-table entries.

6.2.3 NUMA Systems and Huge Pages

Implementing the memory-side translation in NUMA systems come with various challenges. In a single socket system, it is typical for the physical pages to be interleaved among memory controllers and the corresponding memory devices, as shown in Figure 6.5. For a practical design with low contention, the Midgard page addresses will also need to be interleaved among all the memory controllers. If the Midgard page maps to a different memory controller than the required physical page, it makes little sense to introduce hops among memory controllers.

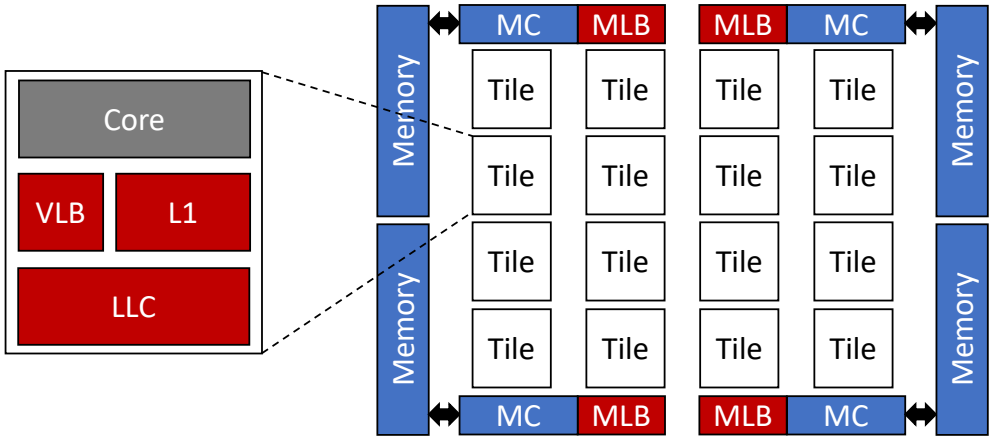


Figure 6.5: The diagram shows the application of the Midgard address in a NUMA system. Each tile consists of a core, VLB, a private L1 cache, and a shared LLC tile. The MLBs are co-located with the memory controllers (MC) which are connected to the memory devices.

Therefore, a practical design should interleave the Midgard addresses in the same pattern as the physical addresses, thus requiring that the Midgard-to-physical address mapping also respects the interleaving. E.g., a Midgard page that maps to memory controller zero should always be mapped to a physical page that also maps to the memory controller zero. As shown in previous proposals [95], such a combination requires the VM mappings to be set associative instead of fully associative. Thus, for each Midgard page, a *Home MLB slice* is defined based on its address, e.g., in a system with four memory controllers, bit 12-13 will be used as the MLB slice index.

If the memory-side translation supports huge pages, the actual 4KB pages will be distributed among all the memory controllers. The memory-side translation mappings for huge pages will have their separate interleaving policy across the memory controllers and require a separate Home MLB slice for each page size [30, 88]. Thus, the memory-side translation mapping can be read from one MLB slice, but then the data is read from a different memory controller, which is unavoidable as each mapping refers to multiple small pages spread across memory controllers. In such a system, Home MLB slices for all the huge page sizes based on the input Midgard address can either be probed serially or in parallel to find the mapping, and then the hit/miss decision is communicated to the Home MLB slice for the 4KB page, which will then

either initiate a page-table entry read in case of a miss, or directly read the physical page in case of a hit. The hops among the memory controllers can be eliminated at the cost of extra MLB space as the huge page mapping can be replicated among the MLBs as the contained pages are accessed. However, such an implementation will require a broadcast-based MLB shootdown as each page-table entry can be spread across multiple memory controllers.

Finally, if the MLB does not contain the required translation, or if the system does not support MLBs, a page-table walk is required to complete the memory-side translation. The page-table entries present in the cache hierarchy are interleaved across LLC slices based on their Midgard addresses, similar to the standard data interleaving in existing systems. Therefore, the home MLB slice will need to lookup the page-table entry in the corresponding LLC slice and then install the entry in the MLB slice.

6.2.4 Memory-side Translation Coherence

The centralized nature of the Midgard page table significantly simplifies the memory-side translation coherence compared to the TLB coherence [9, 10, 72] in existing systems. If the system does not support MLBs and the memory-side translation requires directly reading the page-table entry from the cache hierarchy, then the coherence among page-table entries is directly managed by the stock cache coherence protocol. If the OS attempts to modify a page-table entry, then the cache coherence protocol will guarantee that any other copies of the page-table entry in the cache hierarchy are invalidated before the modifications are made. Thus, no software support is required for memory-side translation coherence in such a system.

If the system supports MLBs which directly use the Midgard page address as tags (instead of the Midgard page-table entry address), then the corresponding MLB entry has to be invalidated separately as it is not part of the cache coherence protocol. Even in such a situation, because the MLBs are logically centralized, the translation coherence does not require a global shootdown broadcast as each page-table entry can only reside in a statically-determined MLB slice, thus requiring a direct shootdown. Finally, if required, the coherence can be controlled entirely in hardware as the Midgard address of a page can be directly transformed to the

Midgard address of its page-table entry and vice versa.

We mathematically analyze the average TLB and MLB shootdown latency. We assume an $N \times N$ mesh system with four memory controllers at the mesh corners. We assume that the shootdown traffic from each core is uniformly distributed among all the MLB slices. In Midgard, the MLB slice containing the required entry can be directly determined based on the address. Therefore, the average latency to access these MLB slices is $N-1$, as seen in Figure 6.5. However, a global TLB shootdown is required in traditional systems, which searches all the per-core TLBs for the required entry. Therefore, the latency is bound by the access to the farthest possible core for each shootdown. We divide the mesh into four symmetrical quadrants to calculate the average shootdown latency over all the cores. The farthest core always lies in the quadrant opposite to the core initiating the shootdown. Assuming the bottom-left core represents (i=1, j=1), and the top-right core represents (i=N, j=N), we can calculate the shootdown latency for any (i,j) pair. Let f(i,j) represent the shootdown latency for a core at (i,j) co-ordinate. Therefore, average shootdown latency:

$= \sum_{i=1}^{N/2} \sum_{j=1}^{N/2} f(i,j) / (N^2/4)$

$= \sum_{i=1}^{N/2} \sum_{j=1}^{N/2} (N-i) + (N-j) / (N^2/4)$

$= 2N - ((N/2) \sum_{i=1}^{N/2} i - (N/2) \sum_{j=1}^{N/2} j) / (N^2/4)$

$= 2N - (N \sum_{i=1}^{N/2} i) / (N^2/4)$

$= 2N - (N * (N/2) * (N/2+1)/2) / (N^2/4)$

$= 3N/2 - 1$

Thus, the MLB shootdowns have $N/2$ lower latency than the traditional global shootdowns.

6.2.5 Access Bits

Access bits are used in modern processors to track which pages have been recently touched so that the OS can ensure that the recently used pages are not kicked out of memory and

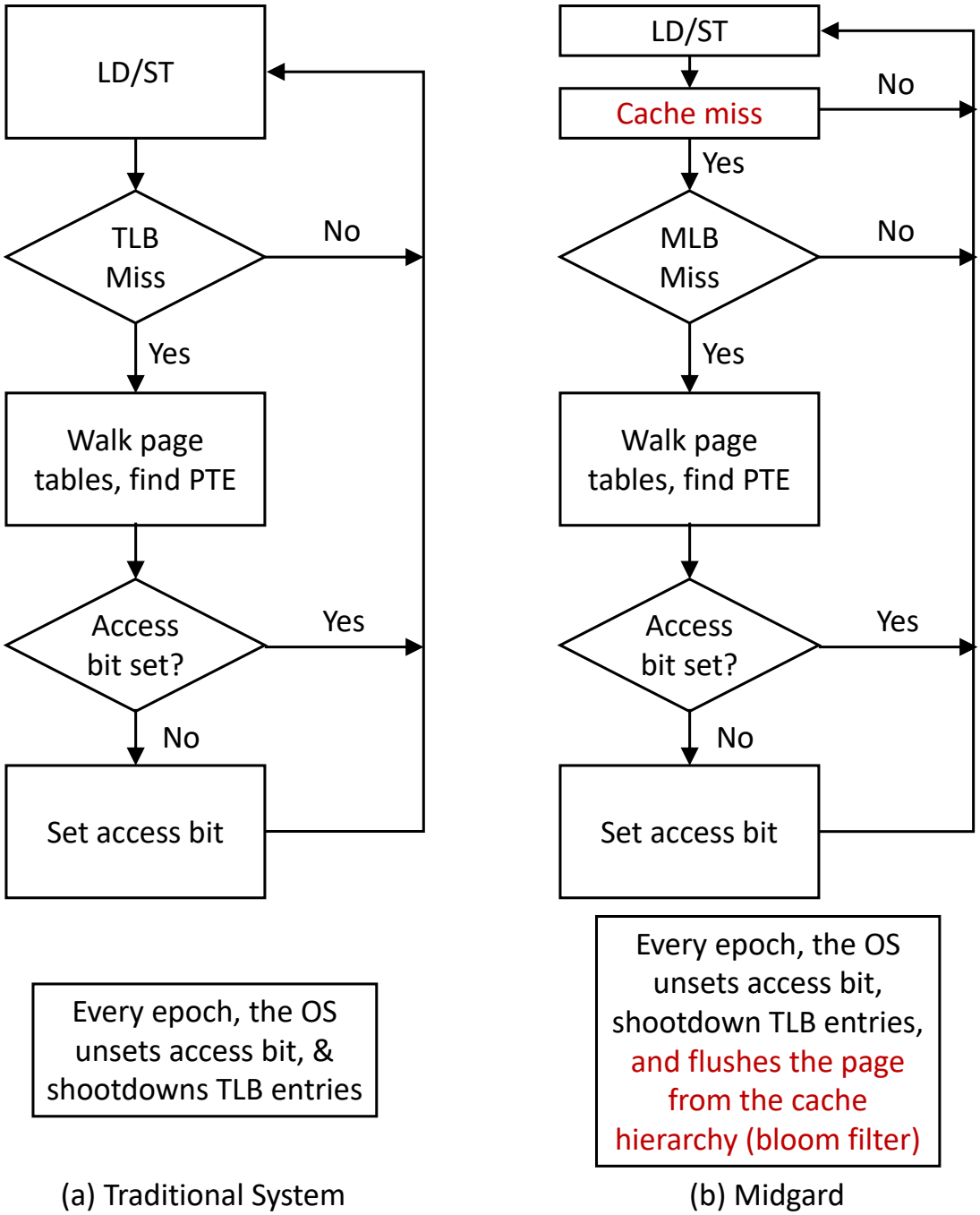


Figure 6.6: Flowcharts describing the setting of access bits by an MMU in (a) Traditional System, and (b) Midgard.

swapped to storage because they are expected to be used again. If the OS can do a good job of keeping the recently used pages in memory, then most of the memory accesses performed by the applications can be directly satisfied from memory while incurring few page faults, thus benefitting performance. As access bits are required for memory capacity management, they are required for each page and therefore are included in the leaf page table entries.

Figure 6.6 depicts the flowcharts describing the setting of access bits in traditional systems and Midgard. In traditional systems, the access bits are set by only the MMU present per core, while are reset by only the OS. Assuming that the access bit for a particular page is originally unset, the access bit is set when an application performs a memory request to the corresponding page which then causes a TLB miss thus triggering a page table walk. At the end of the page table walk, the MMU uses an atomic memory operation to set the access bit. This process is repeated every time a TLB miss is encountered for a particular page and the page table walk is performed. In conjunction with the MMU, the OS periodically resets the access bit so that it can track which pages were accessed in a particular epoch. Every time the OS resets an access bit, it also needs to perform a TLB shootdown to ensure that the very next memory access to the corresponding page will require performing the page table walk, and thus will set the access bit. Finally, the OS uses various techniques such as the clock algorithm or the second chance algorithm to identify which pages have been recently used and should be kept in memory while which pages can be replaced.

However, in Midgard, the access bits require a bit of redesign to ensure the same semantics as traditional systems. The logic-side translation in Midgard is based on VMAs, and therefore is not responsible for page-based information such as access bits. Therefore, the access bits can only be tracked during the memory-side translation. Similar to the traditional systems, the access bits in Midgard can be set when the first memory access to the page is performed, which then causes a miss in the cache hierarchy and requires memory-side translation resulting in a Midgard page table walk. The memory-side MMU can set the access bit using atomic memory operations every time the memory-side translation is performed, similar to the traditional system. The OS will also need to reset the access bit periodically to track the recently accessed

pages. However, every time the OS resets the access bit, it also needs to make sure that the next access to that page will set the access bit. In Midgard, if the cache block required for the memory access is present in the cache hierarchy, the memory-side translation will not be required and thus the access bit will not get set. Therefore, to ensure the exact same semantics for access bits as in the systems today, the OS needs to flush the corresponding page from the cache hierarchy as well, thus guaranteeing that the next access will lead to a cache hierarchy miss and result in memory-side translation which will consequently set the access bit.

Flushing the pages out of the cache hierarchy can have significant performance overheads because of the cache miss, especially if the page is being frequently used. Moreover, as the cache hierarchy gets larger in capacity, the flush operations themselves get increasingly expensive. Previous papers [47, 131] make the observation that as the memory capacity in the system increases, the overall fraction of data that is cold also increases. Therefore, in high-capacity memory servers today, it is easier to find cold pages to replace, and therefore we do not need to refresh the access bits for the hot pages frequently. Based on the above insight, we can conclude that the access bits are typically only reset for the cold pages which are anyways not actively being used, and thus are not present in the cache hierarchy.

We can further optimize the performance overheads incurred by the cache flushing operations caused when resetting the access bits. As we are typically resetting access bits for only cold pages that are not present in the cache hierarchy, searching the cache hierarchy for corresponding cache blocks is anyways not useful. Therefore, we propose that we can use filtering and prediction techniques to identify if a particular block is not present in the cache hierarchy, and thus skip the cache flush operation for the same. The same can be attained by using mechanisms such as a bloom filter [120], which can monitor the cache insertion and eviction operations to predict if a particular cache block is present in the cache hierarchy. A bloom filter is particularly suitable in such a case because it ensure no false negatives, thus ensuring no risk in case of a 'cache block not present' prediction. Overall, using a bloom filter can significantly reduce the overhead of the cache flush operations, and thus reduce the overheads of resetting the access bit in Midgard.

6.2.6 Dirty Bits

Dirty bits are used in modern processors to track which pages have been modified, where the obtained information can then be used to decide if the page has to be written back to the backing storage device to update the contained copy, when evicting the page from memory. Therefore, dirty bits are simply used to reduce the bandwidth consumed by writeback options when writing pages to the backing storage. Similar to access bits, dirty bits are also tracked per page, and therefore are contained in every leaf page-table entry.

Figure 6.7 depicts the flowcharts describing the setting of dirty bits in traditional systems and Midgard. In case of traditional systems, the dirty bit is set by the MMU on the first store operation to a particular page. Every subsequent store operation then requires checking if the dirty bit is already set, and sets the dirty bit if it is not already set. Always reading the dirty bit from the page table entries would consume significant bandwidth from the cache hierarchy. Therefore, the dirty bit is cached along with the translation information in the TLBs to provide fast checks for every new store operation, and when the first store operation takes place, the dirty bit is set in both the TLBs and the page table entries present in the cache hierarchy. While the MMU is responsible for setting the dirty bit, the OS is responsible for clearing it when it is evicting the page, or simply writing it back to the storage device for persistence reasons [45]. When the OS resets the dirty bit, it needs to writeback the dirty page to the backing storage and shootdown the corresponding stale TLB entry.

In case of Midgard, the dirty bit cannot be set or checked for each store operation as done in the traditional systems. In Midgard, the logic-side translation is performed at the VMA granularity, and therefore cannot represent dirty bits belonging to contained pages. Therefore, the dirty bits will be represented in the memory-side translation, similar to the access bits. In case of Midgard, the knowledge of a store operation to a particular page can only reach the memory-side translation when the corresponding dirty cache blocks are being evicted from the cache hierarchy. Therefore, when the first dirty block in a page is evicted and goes through memory-side translation for writeback to memory, the dirty bit will be updated in the page

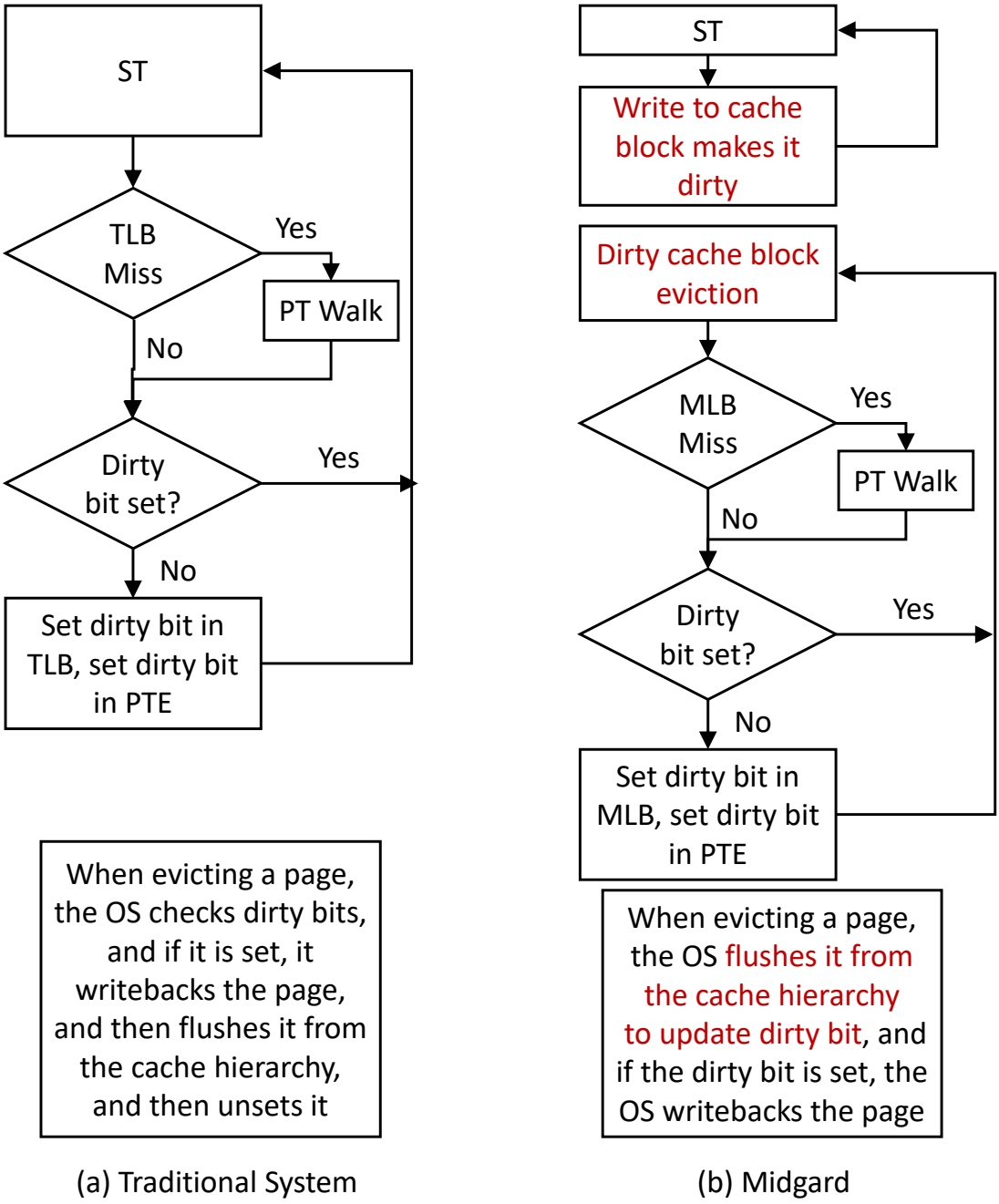


Figure 6.7: Flowcharts describing the setting of bits bits by an MMU in (a) Traditional System, and (b) Midgard.

table entry. Similar to traditional systems, the dirty bit can also be cached in the MLBs for faster lookup, which can benefit workloads with streaming writes. Finally, when the OS wants to check if a page is dirty or not, it will first need to flush the page from the cache hierarchy to ensure that the dirty bits present in the page table entry are updated. Fortunately, in case of a page eviction, such a flush in the cache hierarchy is anyways required, and thus does not pose any additional performance overhead.

6.2.7 Memory-side Permissions

Memory access permissions are an important component in the virtual memory implementation and originate from the both the application and the OS. The application along with the toolchain provides permissions required for semantic correctness of the program at a VMA granularity. For example, the code VMA is supposed to have only read and execute permissions. In contrast, the OS provides permissions that are required for memory management optimizations. For example, swapping out a page renders the permissions invalid, while optimizations such as copy-on-write and Kernel Same-page Merging require marking a page as read-only even if the application requires the page with read-write permissions. Traditional systems coalesce the permissions obtained from both the application and the OS, and represent the resulting permissions in the page table entries.

In Midgard, the permission checks have to be split between the logic-side and memory-side translations. As explained in section 5.1, the logic-side translation represents the permissions specified by the applications at a VMA granularity. However, the permissions specified by the OS do also need to be represented for the correct implementation of memory management optimizations. As the OS specifies permissions at a page granularity for capacity management, these permissions are best represented in memory-side translation, similar to the access and dirty bits. However, unlike the access and dirty bits, the memory-side permissions need to be checked for every memory access while ensuring high performance.

While the logic-side permissions are present in the VLBs, as explained in subsection 5.2.1, we propose that the memory-side de permissions are present along with the cache blocks as part

of the tags, which is similar to the implementation of memory permissions in Virtual Cache Hierarchies [23, 24]. Therefore, the cache block tags will now contain the Midgard address, the memory-side permissions replicated for each cache block, and the bits associated with the cache coherence protocol. For each memory operation, the logic-side permissions are checked at the core, while the memory-side permissions present in the tag are checked when the required cache block is brought and accessed from the L1 cache. An alternate implementation for memory-side permissions is also possible by restricting the coherence states, e.g., if a page is not writable, then the corresponding cache blocks should never attain the exclusive or modified state as in MESI protocol, thus requiring the memory-side permission checks at the directories. However, such approaches require interaction with the cache coherence protocols which are an intricately complex system on their own, and therefore such approaches are less likely to be adopted. Finally, in cases where the memory-side permission check determines that the required memory operation cannot take place, then an exception is raised as explained in 4, requiring the OS to step in. For example, if the OS has swapped out a page, then the memory-side translation for the corresponding page will fail at the MMU, triggering a page fault exception. Similarly, in case of copy-on-write optimizations, the required cache block might be present in the cache hierarchy but only with read-only permissions, thus causing any write operation to fail and trigger an exception.

7 Evaluation

This section presents our evaluation methodology, followed by an evaluation of Midgard's opportunity for future-proofing virtual memory by exposing the already-exiting VMA abstraction for address translation in hardware. First, we characterize the VMA count and usage present in modern workloads to show that VMA-based translation can indeed be fast. Then, we present Midgard's performance sensitivity to cache hierarchy capacity compared to both traditional TLB hierarchies and huge pages. We finally present an evaluation of hardware support required to enhance Midgard's performance when the aggregate cache hierarchy capacity is limited.

7.1 Methodology

We implement Midgard on top of QFlex [89], a family of full-system instrumentation tools built on top of QEMU. Table 7.1 shows the detailed parameters used for evaluation. We model a server containing 16 ARM Cortex-A76 [129] cores operating at 2GHz clock frequency, where each core has a 64KB L1 cache and 1MB LLC tile, along with an aggregate 256GB of memory. Each core has a 48-entry L1 TLB and a 1024-entry L2 TLB that can hold 4KB or 2MB pages translations. For Midgard, we conservatively model an L1 VLB with the same capacity as the traditional L1 TLB along with a 16-entry L2 VLB. The cores are arranged in a 4x4 mesh architecture with four memory controllers at the mesh corners.

Core	16× ARM Cortex-A76 [129]
Traditional TLB	L1(I,D): 48 entries, fully associative, 1 cycle Shared L2: 1024 entries, 4-way, 3 cycles
L1	64KB 4-way L1(I,D), 64-byte blocks, 4 cycles (tag+data)
LLC	Base 1MB/tile, 16-way, 30 cycles, non-inclusive
Memory	256GB capacity (16GB per core) 4 memory controllers at mesh corners
Midgard	VLB: L1(I,D): 48 entries, fully associative, 1 cycle L2 (VMA-based VLB): 16 VMA entries, 3 cycles

Table 7.1: System parameters for simulation on QFlex [89].

As Midgard directly relies on the cache hierarchy for address translation, its performance is susceptible to the cache hierarchy capacity and latency. We evaluate cache hierarchies ranging from MB-scale SRAM LLC to GB-scale DRAM caches and use AMAT to estimate the overall address translation overhead in various systems. To approximate the impact of latency across a wide range of cache hierarchy capacities, we assume three ranges of cache hierarchy configurations modeled based on AMD's Zen3 Rome systems [128]: 1) a single chiplet system with an LLC scaling from 16MB to 64MB and latencies increasing linearly from 30 to 40 cycles, 2) a multi-chiplet system with an aggregate LLC capacity ranging from 64MB to 256MB for up to four chiplets with remote chiplets providing a 50-cycle remote LLC access latency backing up the 64MB local LLC, and 3) and a single chiplet system with a 64MB LLC backed by a DRAM cache [113] using HBM with capacities varying from 512MB to 16GB with an 80-cycle access latency. Our baseline Midgard system directly relies on Midgard page-table walks for performing memory-side translations. We also evaluate Midgard with optional support for memory-side translation to filter requests for Midgard page-table walk for systems with conservative cache hierarchy capacities. We use Average Memory Access Time (AMAT) as a metric to compare the impact of address translation on memory access time. We use full-system trace-driven simulation models to extract miss rates for cache and TLB hierarchy components, assume constant (average) latency based on LLC configuration (as described above) at various hierarchy levels, and measure memory-level parallelism [27] in benchmarks to account for latency overlap.

	Dataset Size (GB)				Thread Count					
	0.2	0.5	1	2	4	8	12	16	24	32
BFS SSSP	51	51	52	52	52	60	68	76	84	108

Table 7.2: VMA count against dataset size and thread count.

To evaluate the full potential of Midgard, we use graph processing workloads including the GAP benchmark suite [17] and Graph500 [83] as they provide highly irregular access patterns and have a high reliance on the address translation performance. The GAP benchmark suite contains six different graph algorithm benchmarks: Breadth-First Search (BFS), Betweenness Centrality (BC), PageRank (PR), Single-Source Shortest Path (SSSP), Connected Components (CC), and Triangle Counting (TC). We evaluate two graph types for these algorithms: uniform-random (Uni) and Kronecker (Kron). Graph500 is a single benchmark with behavior similar to BFS in the GAP benchmark suite. The Kronecker graph type uses the Graph500 specifications in all benchmarks. All graphs evaluated contain ~200GB worth of graph vertices each for 16 cores [16].

7.2 Logic-side Translation

VMA Usage Characterization: We begin by confirming that the number of unique VMAs needed for large-scale real-world workloads, which directly dictates the number of VMA entries required by the L2 VLB, is much lower than the number of unique pages. To evaluate how the VMA count scales with the dataset size and the number of threads, we pick BFS and SSSP from the GAP benchmark suite as they exhibit the worst-case performance with page-based translation. Table 7.2 depicts the change in the number of VMAs used by the benchmark as we vary the dataset size from 0.2GB to 2GB. Over this range, the VMA count only increases by one, possibly from the change in algorithm going from malloc to mmap for allocating large spaces. The VMA count plateaus when scaling the dataset from 2GB to the total size of 200GB ($\sim 2^{25}$ pages) because larger datasets use larger VMAs without affecting their count. Table 7.2 also shows the required number of VMAs while increasing the number of

Benchmark	TLB MPKI in existing systems		Logic-side translation				
			Hit rate with L2 VLB capacity (%)				Required VLB
	Uni	Kron	1	2	4	8	
BFS	22.6	28.9	53.91	79.46	96.89	99.51	16
BC	0.50	0.50	54.49	83.39	97.26	99.99	8
PR	70.8	68.0	45.03	75.71	98.73	99.97	8
SSSP	74.1	70.1	39.52	79.08	89.88	99.93	8
CC	22.5	18.3	60.95	93.68	97.51	100	8
TC	61.6	0.20	76.92	98.87	99.99	100	4
Graph500	-	27.1	55.76	77.76	95.21	99.34	16

Table 7.3: Analysis of the logic-side translation: for each workload, the table depicts the TLB MPKI in existing systems, the hit rate for different L2 VLB capacities, and the overall required VLB capacity.

threads in our benchmarks using the full 200GB dataset. The table shows that each additional thread adds two VMAs comprising a private stack and an adjoining guard page. Because these VMAs are private per thread, their addition does not imply an increase in the working set of the number of L2 VLB entries for active threads.

Finally, Table 7.3 depicts the required L2 VLB size for benchmarks. The table presents the power-of-two VLB size needed for each benchmark to achieve a 99.5% hit rate. In these benchmarks, >90% accesses are to four VMAs, including the code, stack, heap, and a memory-mapped VMA storing the graph dataset. TC is the only benchmark that achieves the required hit rate with four VLB entries. All other benchmarks require more than four entries but achieve the hit rate with eight, with BFS and Graph500 being the only benchmarks that require more than eight entries. These results corroborate prior findings [80, 135] showing that ~10 VLB entries are sufficient even for modern server workloads. We, therefore, conservatively over-provision the L2 VLB with 16 entries in our evaluation.

7.3 Memory-side Translation

7.3.1 Memory-side Translation without MLBs

Address Translation Overhead: Besides supporting VMA translations directly in hardware, a key opportunity that Midgard exploits is that a well-provisioned cache hierarchy filters the majority of the memory accesses, requiring memory-side translation for only a minuscule fraction of the memory requests. Much like in-cache address translation [132], a baseline Midgard system uses page-table walks to perform the memory-side translation for memory requests that are not filtered by the cache hierarchy. In contrast, a traditional TLB-based system typically requires provisioning more resources (e.g., TLB hierarchies, MMU caches) to extend address translation reach with an increased aggregate cache hierarchy capacity. Figure 7.1 compares the overall address translation overhead as a fraction of AMAT between Midgard and TLB-based systems. We plot the geometric mean of the address translation overhead across all benchmarks. We also vary the cache hierarchy configurations (as described in 7.1) in steps to reflect aggregate capacity in recent products such as Intel Kabylake [130], AMD Zen3 Rome [128], and Intel Knights Landing [113].

Figure 7.1 shows that address translation overhead in traditional 4KB-page systems running graph workloads with large datasets even for minimally sized 16MB LLCs is relatively high at around 17%. As shown in Table 7.3, the TLB misses per thousand instructions (MPKI) in 4KB-page systems is overall relatively high in our graph benchmarks (except BC and TC with Kron graphs). These miss rates are also much higher than those in desktop workloads [65] or scaled down (e.g., 5GB) server workloads [66]. The figure shows that Midgard achieves only 5% higher overall address translation overhead as compared to traditional 4KB-page TLB-based systems for a minimally sized LLC while virtually eliminating the silicon provisioned for per-core 1K-entry L2 TLBs (i.e., ~16KB SRAM), obviating the need for MMU caches and hardware support for memory-side translation and page walks.

As the dataset sizes for server workloads increase, modern servers are now featuring increasingly larger aggregate cache hierarchy capacities [113, 128, 130]. With an increase in aggregate

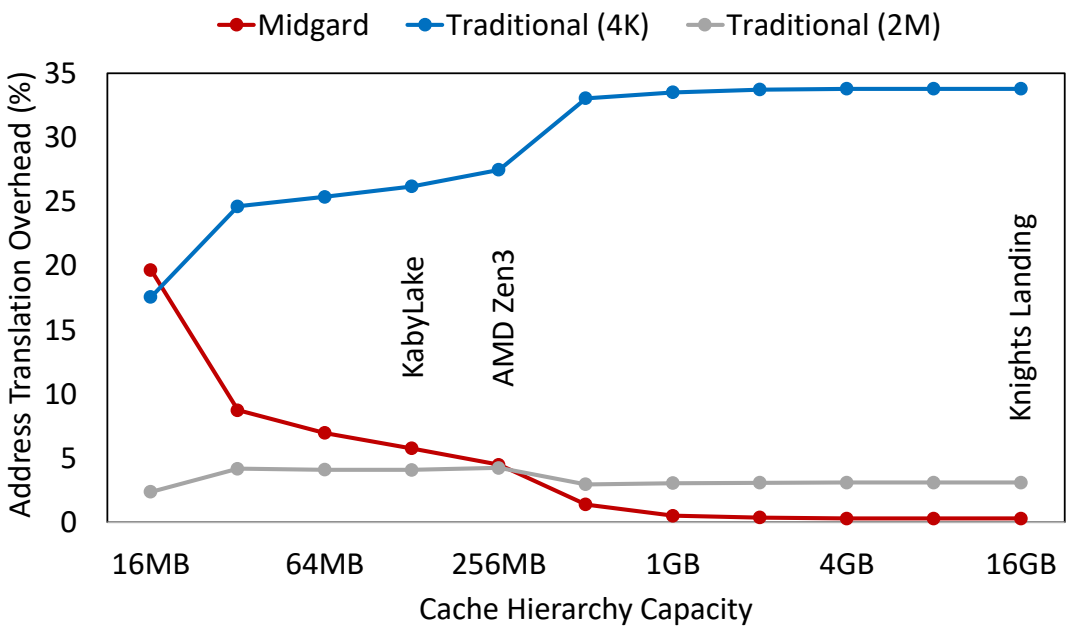


Figure 7.1: The graph shows the address translation overhead (%) as part of the overall Average Memory Access Time (AMAT) for varying cache hierarchy capacity. The X-axis shows the overall cache hierarchy capacity along with various products such as Intel Kabylake [130], AMD Zen3 [128], and Intel Knights Landing [113].

cache capacity, the relative TLB reach in traditional systems decreases while the average time to fetch data decreases due to higher cache hit rates. Unsurprisingly, the figure shows that the address translation overhead for traditional 4KB-page TLB-based systems exhibit an overall increase, thereby justifying the continued increase in TLB-entry counts in modern cores. The figure also shows that our workloads exhibit secondary and tertiary working set capacities at 32MB and 512MB where the traditional 4KB-page system's address translation overhead increases because of limited TLB reach to 25% and 33% of AMAT, respectively.

In contrast, Midgard's address translation overhead drops dramatically at both secondary and tertiary working set transitions in the graph thanks to the corresponding fraction of memory requests filtered in the cache hierarchy. Table 7.4 also shows the amount of memory-side traffic filtered by 32MB and 512MB LLCs for the two working sets. The table shows that 32MB already filters over 90% of the memory-side traffic in most benchmarks by serving data directly in the Midgard namespace in the cache hierarchy. With a 512MB LLC, all benchmarks have over 99% of traffic filtered with benchmarks using the Kron graph, (virtually) eliminating all translation traffic due to enhanced locality. As a result, with Midgard, the resulting address translation overhead (Figure 7.1) drops to below 10% at 32MB and under 2% at 512MB of LLC.

Next, we compare average page-table walk latency between a 4KB-page TLB-based system and Midgard. Because Midgard fetches the leaf page-table entries from the cache hierarchy during a page walk in the common case, on average, it requires only 1.2 accesses per walk to an LLC tile which is (~30 cycles) away (Table 7.4). In contrast, TLB-based systems require four lookups per walk. While these lookups are performed in the cache hierarchy, they typically miss in L1, requiring one or more LLC accesses. As such, Midgard achieves up to 40% reduction in the walk latency compared to TLB-based systems. BC stands as the outlier with the high locality in the four lookups in L1, resulting in a lower TLB-based average page walk latency than Midgard.

Comparison with Huge Pages: To evaluate future-proofing virtual memory with Midgard, we also compare Midgard's performance against an optimistic lower bound for address translation overhead using huge pages. Huge pages provide translation at a larger page granularity (e.g., 2MB or 1GB), thereby enhancing TLB reach and reducing the overall address transla-

Benchmark	Traffic filtered by cache hierarchy (%)				Average page-table walk cycles			
	Uni		Kron		Uni		Kron	
	32MB	512MB	32MB	512MB	Existing	Midgard	Existing	Midgard
BFS	95	99	95	100	51	31	30	30
BC	100	100	100	100	20	35	20	35
PR	85	100	89	100	45	30	42	30
SSSP	87	98	90	100	47	31	38	30
CC	98	100	97	100	39	34	44	31
TC	80	92	100	100	48	30	48	30
Graph500	-	-	96	100	-	-	32	30

Table 7.4: Analysis of the memory-side translation: for each workload, the table depicts the % of memory operations filtered by the cache hierarchy and the average page-table walk cycles in memory-side translation.

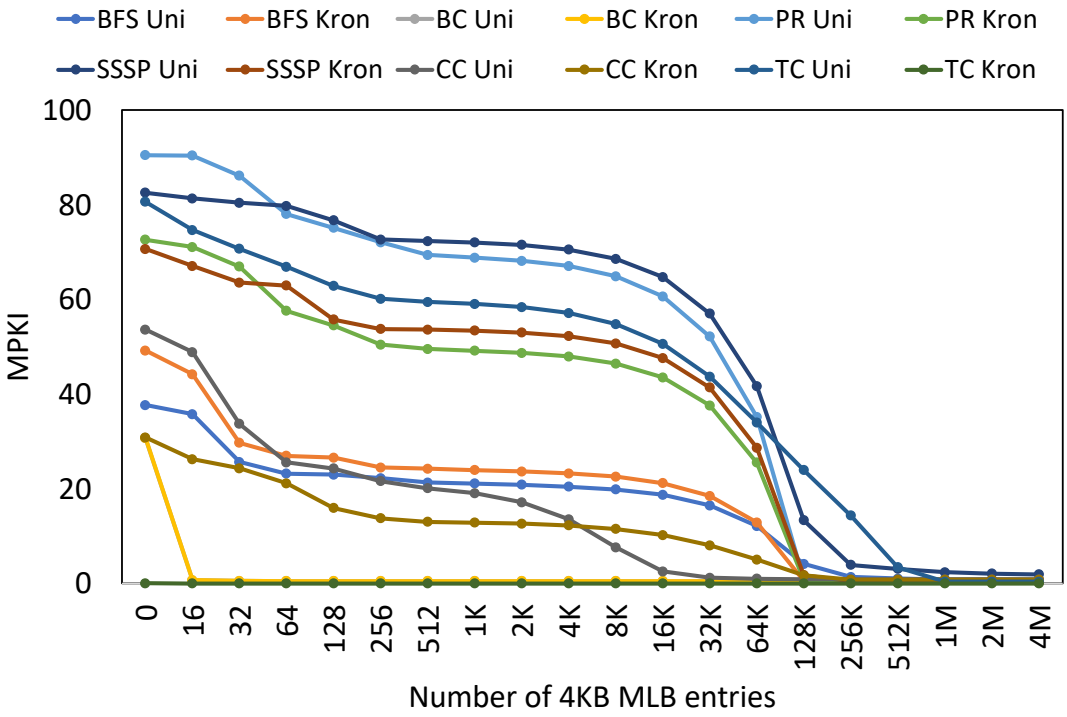
tion overhead. Prior work [85, 93, 135] indicates that creating and maintaining huge pages throughout program execution requires costly memory defragmentation and frequent TLB shootdowns [9, 10]. Huge pages may also inadvertently cause a performance bottleneck, e.g., when migrating pages in a NUMA system [91]. We optimistically assume zero-cost memory defragmentation and TLB shootdown to evaluate a lower bound, thus allowing ideal 2MB pages for address translation that do not require migration. We also assume the same number of L1 and L2 2MB TLB entries per core as the 4KB-page system.

Figure 7.1 also depicts a comparison of address translation overhead between Midgard and ideal 2MB-page TLB-based systems. Not surprisingly, a 2MB-page system dramatically outperforms both TLB-based 4KB-page and Midgard systems for a minimally sized 16MB LLC because of the 500x increase in TLB reach. Much like 4KB-page systems, the TLB-based huge page system also exhibits an increase in address translation overhead with an increase in aggregate LLC capacity with a near doubling in overall address translation overhead from 16MB to 32MB cache capacity. In contrast to 4KB-page systems, which exhibit a drastic drop in TLB reach from 256MB to 512MB LLC capacity with the tertiary data working set, huge pages allow for a 32GB overall TLB reach (i.e., 16 cores with 1K-entry L2 TLB) showing little sensitivity to the tertiary working set fitting. Instead, because for the DRAM cache configurations, we assume a transition from multiple chiplets to a single chiplet of 64MB backed up by a slower

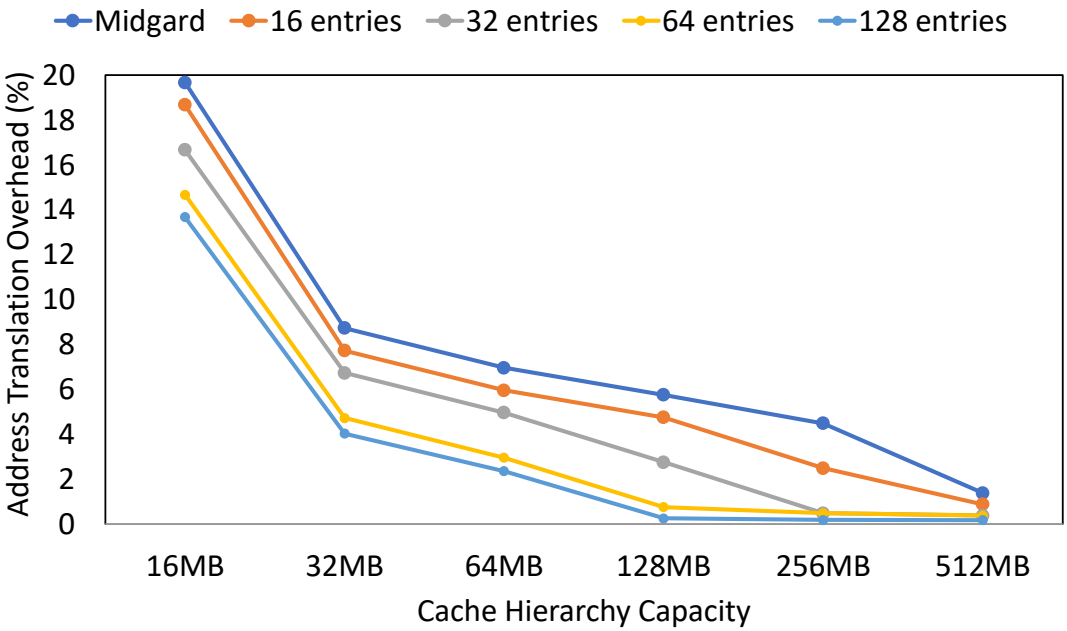
512MB DRAM cache with a higher access latency, a larger fraction of overall AMAT goes to the slower DRAM cache accesses. As such, the address translation overhead drops a bit for DRAM caches but increases again with aggregate cache capacity. In comparison to huge pages, Midgard's performance continuously improves with cache hierarchy capacity until address translation overhead is virtually eliminated. Midgard reaches within 2x of huge pages' performance with 32MB cache capacity, breaks even at 256MB, which is the aggregate SRAM-based LLC capacity in AMD's Zen3 Rome [128] products, and drops below 1% beyond 1GB of cache capacity. While Midgard is compatible with and can benefit from huge pages, Midgard does not require huge page support to provide adequate performance as traditional systems do. Overall, Midgard provides high-performance address translation for large memory servers without relying on larger page sizes.

7.3.2 Memory-side Translation with MLBs

While we expect future server systems to continue integrating larger cache hierarchies, our memory-access intensive workloads exhibit a non-negligible degree of sensitivity to address translation overhead (i.e., > 5%) with aggregate cache hierarchy capacities of less than 256MB. In this subsection, we evaluate hardware support for memory-side translation using MLBs. While MLBs (like TLBs) complicate overall system design, the higher complexity may be justified by the improvements in address translation overhead for a given range of cache hierarchy capacity. We first analyze the required aggregate MLB size (i.e., the total number of MLB entries across the four memory controllers) for the GAP benchmarks for a minimally sized LLC at 16MB. Figure 7.2a illustrates the MPKI (i.e., the number of memory-side translations per kilo instruction requiring a page walk) as a function of the log scale of MLB size. The figure shows that while the MLB size requirements across the benchmarks largely vary, there are approximately two windows of sizes that exhibit memory-side translation working sets. The primary working set, on average, appears to be roughly around 64 aggregate MLB entries, with many benchmarks exhibiting a step function in MPKI. The latter would provision only four MLB entries per thread, indicating that the memory-side translations result from spatial



(a) The MPKI distribution for varying MLB capacity with a 16MB cache hierarchy capacity.



(b) The address translation overhead for varying cache hierarchy capacity.

Figure 7.2: Graph (a) shows the overall MPKI distribution for varying MLB capacity, while graph (b) shows the address translation overhead as part of the Average Memory Access Time (AMAT) while varying the cache hierarchy capacity.

streams to 4KB page frames in memory. Beyond the first window, the second and final working set of memory-side translations is around 128K MLB entries, which is prohibitive, suggesting that practical MLB designs require only a few entries per memory controller.

Figure 7.2b illustrates the address translation overhead for cache hierarchies of up to 512MB while varying the number of aggregate MLB entries from 0 to 128 averaged over all the GAP benchmarks. Midgard in the figure refers to the baseline system without an MLB. The figure corroborates the results that, on average, 64 MLB entries are the proper sweet spot for 16MB LLC. Comparing the figure with Figure 7.1, we see that for a 16MB LLC, Midgard can break even in address translation overhead with traditional 4KB-page systems with only 32 overall MLB entries (i.e., eight entries per memory controller). In contrast, practical provisioning for MLB will not help Midgard break even with an ideal huge page system for a minimally sized LLC, assuming that the memory-side translation in Midgard does not use huge pages itself. The figure also shows that with only 32 and 64 MLB entries, Midgard can virtually eliminate address translation overhead in systems with 256MB and 128MB aggregate LLC, respectively. Moreover, with 64 MLB entries, Midgard outperforms huge pages for LLC capacity equal to or greater than 32MB. Finally, for an LLC capacity of 512MB or larger, there is minimal benefit from hardware support for memory-side translation.

8 Future Work and Conclusion

The Midgard address space allows for the reduction of the address translation overheads by providing fast translation at a VMA granularity for the common case cache hierarchy accesses while requiring slow translation at page granularity only if the required cache block is not present in the cache hierarchy and has to be fetched from memory. While the Midgard address space requires the redesign of various OS mechanisms to retain compatibility, it also creates interesting challenges about various system aspects such as how virtualized systems will work if the Midgard address space is included in the guest and the host OS, and how Midgard can work with innovating heterogenous memory hierarchies along with memory pooling technologies. This chapter will briefly describe the future research directions with Midgard along with a thesis conclusion.

8.1 Path to Practical Adoption

While our simulator-driven evaluation in chapter 7 shows the potential benefits of Midgard, there is still a long way to go towards the commercial adoption of Midgard in datacenter-scale computing systems because significant changes are required on both the hardware and the OS side. In this thesis, we mainly focused on the traditional processors and applications, but there are many other cases which require compatibility and can benefit from Midgard.

On the hardware side, Midgard introduces fast logic-side translation, and a comparatively heavy-weight memory-side translation. We need to provide logic-side translation for cores, accelerators, and even IO devices, as explained in subsection 5.2.1 and subsection 5.2.2. All such logic-side translation implementations need verification in order to ensure that the hardware is correctly designed, and does not give way to modern security vulnerabilities. Because logic-side translation does not require maintaining a lot of logic or silicon state, we expect that the verification can be easily performed in comparison to the traditional TLB-based translation mechanisms. In contrast, the memory-side translation has to be adopted to be able to work with multi-socket or multi-chiplet designs, where the responsibility of performing the translation has to be co-located with either the cache controllers, or the memory controllers. The memory-side translation also needs to be adopted to the emerging horizontally or vertically-tiered memory hierarchies [47] built using heterogeneous memory hierarchies. In such cases, the memory-side translation might have various constraints, such as aggressive caching requirements in case of extremely slow backing memory devices, but with built-in latency slack opportunities.

On the OS side, Midgard requires redesigning the support for VM which is a core feature of modern OSes. Such fundamental changes might touch various subsystems, such as page-replacement design and algorithms, optimizations such as copy-on-write, support for file systems and file caches, support for huge pages and memory compaction in conjunction with Midgard. In particular, because Midgard requires the cache hierarchy to be indexed using the Midgard address space, we need to ensure that there are no homonyms or synonyms present in the address space for full compatibility with the traditional cache coherence protocols. However, in modern OS implementations, there are various practices that can create such a problem, e.g., the direct mapping of the physical address space in Linux, or the implementation of the file cache as a collection of random pages which can then generate synonyms with the file VMAs/MMAs being used by applications. Therefore, the OS designers have to rethink such subsystems that interact with VM to ensure compatibility with Midgard. While Midgard requires significant changes on both the hardware and OS front, we expect that it will provide new opportunities for better performance and simpler design using reduced silicon

requirements, thus paving the way forward for post-Moore datacenter servers.

8.2 Virtualization

With the increasing number of users of datacenter servers, datacenter operators aim to co-locate multiple users on the same physical server to increase the overall server utilization and benefit from a lower cost per user. Virtualization [22] is an important technique to support such multi-tenancy in modern cloud deployments. Virtualization provides the illusion of isolation to each virtual machine operated by an individual user, while hosting multiple virtual machines on the same physical server. The memory isolation provided in virtualization is fundamentally based on VM. Therefore, as the memory capacity in servers increases and the traditional VM implementation results in significant performance degradation, virtualized deployments result in a significantly higher performance degradation compared to native deployments [39, 40, 80].

The introduction of Midgard promises to reduce the performance overheads of VM in a native deployment, but it is unclear if similar benefits could be realized in a virtualized deployment as well. The benefits of Midgard rely on delaying the page-based translation to physical addresses, and invoking it only when accessing the physical memory. The same optimization should reap benefits in a virtualized deployment as well where page-table walks for translation to physical addresses (both guest and host) incurs a higher performance overhead because of the two-dimensional nature of the page-table walk. However, in order to benefit from the above optimizations, we require both the guest and host OS to be Midgard compliant because the underlying processor architecture will be Midgard compliant.

Assuming a hypothetical system where both the guest and host OS support Midgard, we can adopt the current two-dimensional page-table walk logic to work for the logic-side translation. Both the guest and host can specify their own logic-side translations, and the hardware is responsible for performing a two-dimensional VMA-table walk, where it first converts the guest virtual addresses to guest Midgard addresses, and then consequently converts the

guest Midgard addresses to host Midgard addresses. Both these translations can happen at a VMA granularity, and therefore be fast in nature, while the resulting host Midgard address can be used to index the cache hierarchy as in native systems. Even when performing a two-dimensional translation with VMAs, we expect the logic-side translation to have high performance because there are only a limited number of VMAs to consider from both the guest and host, and therefore the resulting translations can be effectively cached in the microarchitecture. Overall, the virtualized logic-side translation will perform the permission check for both the guest and host logic-side translation, and translate the original guest virtual address to host Midgard address.

However, it is a bit trickier to design the memory-side translation in a virtualized deployment. Let us assume that the guest OS is operating in a full-virtualization mode where it does not know that it is being virtualized, and the host OS also supports in maintaining this illusion. In this case, the two-dimensional memory-side translation needs to pass both the guest and host memory-side translations (which are equivalent to typical page table walks) before accessing physical memory. While it seems possible to design a standard two-dimensional page table walk, there are a few complications that need to be taken care of. For example, as we are using the host Midgard address to index the cache hierarchy, the two-dimensional memory-side translation is required to convert the host Midgard address to host physical address (which is equivalent to the host memory-side translation itself). However, we first need to check the guest memory-side permissions before performing the physical memory access, which can be only done by using the guest Midgard access and the guest page-table walk in the the guest memory-side translation. Therefore, we need to track the guest Midgard address corresponding to every host Midgard address which is not trivial in hardware.

However, the above complication also encourages the idea of a para-virtualized system where the guest OS is aware that it is being virtualized, and can cooperate with the host OS to ease the microarchitecture and benefit the overall performance. The main insight behind such a para-virtualized system is that the guest does not have a direct control on physical memory, and it is only the capacity management performed by the host OS that controls the physical

memory. Therefore, if the guest OS can be made aware that it is being virtualized, then it can completely skip the capacity management (and guest memory-side translation), and let the host OS control the physical memory as usual. In such a system, the virtualized memory-side translation will be equivalent to the typical host memory-side translation, where the host Midgard address is translated to the host physical address. The guest OS in such a system will not have a physical address space of its own at all.

Overall, the introduction of Midgard opens up new possibilities to holistically redesign the virtualization subsystem by removing the redundant capacity management being performed by the guest. Optimizing the overall virtualization design will lead to performance benefits which will probably be even higher than those achieved by Midgard in a native deployment. In summation, the introduction of Midgard indicates a path to possibly eliminate the overhead of memory virtualization.

8.3 Memory Pooling

As the memory requirements of online services increases, system designers struggle to constantly increase the memory capacity in servers to meet the demand. With the end of Moore's law, adding more and more DRAM is no longer an option because DRAM has stopped scaling in capacity [47], and adding additional DRAM DIMMs is both prohibitively expensive and limited by the number of pins available in the processor. Therefore, system designers are looking at supporting heterogenous memory hierarchies which can contain a collection of various memory technology devices such as storage class memory [45, 56]. Another option that is being explored is memory pooling.

As a datacenter contains thousands of servers, there exists natural imbalance in the memory capacity utilization across the servers. Therefore, pooling memory across servers allows servers with higher memory capacity requirement to use the unutilized memory capacity from other servers with a slight performance overhead [86]. With the introduction of new technologies like RDMA and CXL [33, 99], such memory pooling designs are becoming increasingly feasible.

CXL in particular supports cache coherent memory pooling without requiring any additional software primitives.

However, when using such memory pooling technologies, VM is an important abstraction for transparently deploying applications without the applications explicitly controlling the data residing in local vs remote memory devices. In VM-compliant systems with memory pooling, the large memory capacity will quickly become a performance bottleneck. However, as explained in this thesis, Midgard can delay the translation to physical addresses even in case of pooled memory systems, and thus benefit the overall performance.

Another interesting possibility with Midgard is to share the Midgard address space among multiple servers, and use the individual OSes running on these servers to collaborate as in a distributed system. The presence of a shared Midgard address space allows easing coherent accesses and synchronization primitives among servers, thus leading to the relaxation of requirements from technologies such as CXL. Overall, the introduction of memory pooling in servers increases the overall pressure on VM because of the availability of large memory capacity, and Midgard can help in reducing the performance overheads in such deployments.

8.4 Conclusion

Despite decades of research on building complex TLB and MMU cache hardware and hardware/OS support for huge pages, address translation has remained a vexing performance problem for datacenter systems with increasing memory demands. As computer systems designers integrate cache hierarchies with higher capacity, the cost of address translation has continued to surge.

In this thesis, we realized and evaluated a proof-of-concept design of Midgard, an intermediate namespace for all data in the coherence domain and cache hierarchy, in order to reduce address translation overheads and future-proof the VM abstraction. Midgard decouples address translation requirements into core-side access control at VMA granularity and memory-side translation at page granularity for efficient capacity management. Midgard's decoupling

enables fast core-side virtual-to-Midgard address translation using leaner hardware support than traditional TLBs and filtering costlier Midgard-to-physical address translations to only situations when the cache hierarchy does not contain the required cache blocks. As processor vendors increase the cache hierarchy capacities to fit the primary, secondary, and tertiary working sets of modern workloads, Midgard to physical address translation becomes infrequent.

We used AMAT analysis to show that Midgard achieves only 5% higher address translation overhead than traditional TLB hierarchies for 4KB pages when using a 16MB aggregate LLC while requiring only small hardware caching structures (16 entries) instead of TLB hierarchies with 1000s of entries. Midgard also breaks even with traditional TLB hierarchies for 2MB pages when using a 256MB aggregate LLC. For cache hierarchies with higher capacity, Midgard's address translation overhead drops to near zero as secondary and tertiary data working sets fit in the LLC, while traditional TLBs suffer even higher degrees of address translation overhead.

This thesis described the first of several steps needed to demonstrate a fully working system with Midgard. We focused on a proof-of-concept software-modeled prototype of key architectural components. Overall, we showed that Midgard can utilize the increasing cache hierarchy capacity to directly reduce the address translation overheads, while providing a variety of opportunities to reduce performance overheads that exist at different levels of the hardware-software stack in modern datacenter platforms today.

Bibliography

[1] 7-cpu, “Apple M1 Specs,” https://www.7-cpu.com/cpu/Apple_M1.html, 2022.

[2] 7-cpu, “Intel Pentium 4 Specs,” https://www.7-cpu.com/cpu/P4-180.html, 2022.

[3] S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial.” *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[4] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. T. Blaauw, and R. Das, “Compute Caches.” in *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 481–492.

[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, “Fences in Weak Memory Models.” in *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV)*, 2010, pp. 258–272.

[6] Amazon, “Amazon EC2 Instance Types,” https://aws.amazon.com/ec2/instance-types/, 2023.

[7] Amazon, “Amazon EC2 On-Demand Pricing,” https://aws.amazon.com/ec2/pricing/on-demand/, 2023.

[8] AMD, “AMD Ryzen™ 7 5800X3D Gaming Processor,” https://www.amd.com/en/products/cpu/amd-ryzen-7-5800x3d, 2023.

[9] N. Amit, “Optimizing the TLB Shootdown Algorithm with Page Access Tracking.” in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 27–39.

[10] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!" in *Proceedings of the 2020 EuroSys Conference*, 2020, pp. 35:1–35:14.

[11] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a Scala embedded language." in *Proceedings of the Design Automation Conference (DAC) 2012*, 2012, pp. 1216–1225.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization." in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 164–177.

[13] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)." in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, 2010, pp. 48–59.

[14] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[15] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers." in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 237–248.

[16] S. Beamer, K. Asanovic, and D. A. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server." in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 56–65.

[17] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite." *CoRR*, vol. abs/1508.03619, 2015.

[18] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems." in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, 2008, pp. 26–35.

[19] A. Bhattacharjee and D. Lustig, *Architectural and Operating System Support for Virtual Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[20] C. Blundell, M. M. K. Martin, and T. F. Wenisch, "InvisiFence: performance-transparent memory ordering in conventional multiprocessors." in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 233–244.

[21] B. Boothe and A. G. Ranade, "Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors." in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 214–223.

[22] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[23] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors." *IEEE Micro*, vol. 17, no. 5, pp. 64–71, 1997.

[24] M. Cekleov and M. Dubois, "Virtual-address caches.2. Multiprocessor issues." *IEEE Micro*, vol. 17, no. 6, pp. 69–74, 1997.

[25] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency." in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 278–289.

[26] D. Chaiken and A. Agarwal, "Software-Extended Coherent Shared Memory: Performance and Cost." in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 314–324.

[27] Y. Chou, B. Fahs, and S. G. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism." in *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004, pp. 76–89.

[28] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory." in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 133–146.

[29] J. Corbet, "Ringing in a new asynchronous I/O API," 2019. [Online]. Available: https://lwn.net/Articles/776703/

[30] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes." in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017, pp. 435–448.

[31] Daisy Quaker, "Amazon Stats: Growth, sales, and more," 2023. [Online]. Available: https://sell.amazon.com/blog/amazon-stats

[32] David Brooks, "What's the future of technology scaling?" https://www.sigarch.org/whats-the-future-of-technology-scaling/, 2018.

[33] Debendra Sharma and Robert Blankenship and Daniel Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," 2023. [Online]. Available: https://arxiv.org/abs/2306.11227

[34] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory Access Buffering in Multiprocessors." in *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*, 1986, pp. 434–442.

[35] B. Falsafi and D. A. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA." in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 229–240.

[36] 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 17th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII), 2012, pp. 37–48.

[37] S. Flur and L. Maranget, "RISC-V architecture concurrency model litmus tests," 2022. [Online]. Available: https://github.com/litmus-tests/litmus-tests-riscv

[38] D. Fujiki, X. Wang, A. Subramaniyan, and R. Das, *In-/Near-Memory Computing*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2021.

[39] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks." in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 178–189.

[40] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging." in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 707–718.

[41] C. Gniady and B. Falsafi, "Speculative Sequential Consistency with Little Custom Storage." in *IEEE PACT*, 2002, pp. 179–188.

[42] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP=RC?" in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 162–171.

[43] J. R. Goodman, "Coherency for Multiprocessor Virtual Address Caches." in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, 1987, pp. 72–81.

[44] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting Virtual Memory with Midgard." in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 512–525.

[45] S. Gupta, A. Daglis, and B. Falsafi, "Distributed Logless Atomic Durability with Persistent Memory." in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 466–478.

[46] S. Gupta, Y. Li, Q. Kang, A. Bhattacharjee, B. Falsafi, Y. Oh, and M. Payer, "Imprecise Store Exceptions." in *Proceedings of the 50th International Symposium on Computer Architecture (ISCA)*, 2023, pp. 52:1–52:15.

[47] S. Gupta, Y. Oh, L. Yan, M. Sutherland, A. Bhattacharjee, B. Falsafi, and P. Hsu, "AstriFlash A Flash-Based System for Online Services." in *Proceedings of the 29th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 81–93.

[48] F. Guvenilir and Y. N. Patt, "Tailored Page Sizes." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 900–912.

[49] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. Oliveira, J. Appavoo, V. Seshadri, and O. Mutlu, "The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1050–1063.

[50] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 637–650.

[51] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocesors." *ACM Trans. Comput. Syst.*, vol. 11, no. 4, pp. 300–318, 1993.

[52] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors." in *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996, pp. 260–270.

[53] S. Iacobovici, "A pipelined interface for high floating-point performance with precise exceptions." *IEEE Micro*, vol. 8, no. 3, pp. 77–87, 1988.

[54] Ian Cutress, "Intel: Sapphire Rapids With 64 GB of HBM2e, Ponte Vecchio with 408 MB L2 Cache," https://www.anandtech.com/show/17067/intel-sapphire-rapids-with-64-gb-of-hbm2e-ponte-vecchio-with-408-mb-l2-cache, 2022.

[55] Intel, "5-Level Paging and 5-Level EPT," https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2017.

[56] Intel, "3D XPoint Breakthrough Non-Volatile Memory," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html

[57] Intel, "Intel Software Guard Extensions," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html

[58] Intel Corporation, "Intel ©64 and IA-32 Architectures Software Developer Manuals," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[59] Jeff Barr, "EC2 High Memory Update - New 18 TB and 24 TB Instances," https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/, 2019.

[60] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache." in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 25–37.

[61] 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 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 404–415.

[62] S. L. P. Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow, "A Semantics for Imprecise Exceptions." in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, 1999, pp. 25–36.

[63] A. Kannan, N. D. E. Jerger, and G. H. Loh, "Enabling interposer-based disintegration of multi-core processors." in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 546–558.

[64] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Redundant memory mappings for fast access to large memories." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 66–78.

[65] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation." in *Proceedings of the 22nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 631–643.

[66] 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 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 1–12.

[67] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. H. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud." in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[68] H. Kasture and D. Sánchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications." in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 3–12.

[69] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS." in *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, 2018, pp. 107–127.

[70] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architectural Support for Single Address Space Operating Systems." in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 1992, pp. 175–186.

[71] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, 2020, pp. 991–1010.

[72] M. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 651–664.

[73] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens." in *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, 2016, pp. 705–721.

[74] H. A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-Defined Far Memory in Warehouse-Scale Computers." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 317–330.

[75] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems." *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[76] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sánchez, and N. Beckmann, "Livia: Data-Centric Computing Throughout the Memory Hierarchy." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020, pp. 417–433.

[77] D. Lustig, G. Sethi, A. Bhattacharjee, and M. Martonosi, "Transistency Models: Memory Ordering at the Hardware-OS Interface." *IEEE Micro*, vol. 37, no. 3, pp. 88–97, 2017.

[78] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface." in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*, 2016, pp. 233–247.

[79] S. Maass, M. K. Kumar, T. Kim, T. Krishna, and A. Bhattacharjee, "ECOTLB: Eventually Consistent TLBs." *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 27:1–27:24, 2020.

[80] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation." in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 1023–1036.

[81] Miguel Fersen, "Latency is Having a Huge Negative Impact on eCommerce Companies," 2023. [Online]. Available: https://www.globaldots.com/resources/blog/latency-is-having-a-huge-negative-impact-on-ecommerce-companies/

[82] T. C. Mowry and S. R. Ramkissoon, "Software-Controlled Multithreading Using Informing Memory Operations." in *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2000, pp. 121–132.

[83] R. C. Murphy, K. B. Wheeler, and B. W. Barrett, "Introducing the graph 500," http://www.richardmurphy.net/archive/cug-may2010.pdf, 2010.

[84] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[85] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, Transparent Operating System Support for Superpages." in *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[86] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 3–18.

[87] NVM Express, Inc., "NVM Express Specifications," 2022. [Online]. Available: https://nvmexpress.org/specifications/

[88] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs." in *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2015, pp. 210–222.

[89] Parallel Systems Architecture Lab (PARSA) EPFL, "Qflex," https://qflex.epfl.ch, 2022.

[90] M. Parasar, A. Bhattacharjee, and T. Krishna, "SEESAW: Using Superpages to Improve VIPT Caches." in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 193–206.

[91] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated Page: Supporting Fragmented Memory Allocation for Large Pages." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 913–925.

[92] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs." in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 258–269.

[93] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: can you have it both ways?" in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 1–12.

[94] 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 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 743–758.

[95] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-Memory Address Translation." in *Proceedings of the 26th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2017, pp. 303–317.

[96] J. Picorel, S. A. S. Kohroudi, Z. Yan, A. Bhattacharjee, B. Falsafi, and D. Jevdjic, "SPARTA: A Divide and Conquer Approach to Address Translation for Accelerators." *CoRR*, vol. abs/2001.07045, 2020.

[97] X. Qiu and M. Dubois, "Tolerating Late Memory Traps in ILP Processors." in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 76–87.

[98] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models." in *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, 1997, pp. 199–210.

[99] RDMA Consortium, "Architectural Specifications for RDMA over TCP/IP," 2009. [Online]. Available: http://www.rdmaconsortium.org/

[100] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory." in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 325–336.

[101] RISC-V International, "Specifications," 2022. [Online]. Available: https://riscv.org/technical/specifications/

[102] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin, "Specifying and dynamically verifying address translation-aware memory consistency." in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010, pp. 323–334.

[103] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices." *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.

[104] S Dixon, "Global social networks ranked by number of users 2023," 2023. [Online]. Available: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/

[105] S. Sardashti, A. Arelakis, P. Stenström, and D. A. Wood, *A Primer on Compression in the Memory Hierarchy*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2015.

[106] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain Access Control for Distributed Shared Memory." in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 297–306.

[107] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täk¯: a polymorphic cache hierarchy for general-purpose optimization of data movement." in *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, 2022, pp. 42–58.

[108] A. Seznec, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB." *IEEE Trans. Computers*, vol. 53, no. 7, pp. 924–927, 2004.

[109] SiFive, Inc., "SiFive TileLink Specification," 2017. [Online]. Available: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[110] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi, "End-to-end sequential consistency." in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 524–535.

[111] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors." in *Proceedings of the 12th International Symposium on Computer Architecture (ISCA)*, 1985, pp. 36–44.

[112] Snehanshu Shah, "Announcing the general availability of 6 and 12 TB VMs for SAP HANA instances on Google Cloud Platform," https://cloud.google.com/blog/products/sap-google-cloud/announcing-the-general-availability-of-6-and-12tb-vms-for-sap-hana-instances-on-gcp, 2019.

[113] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product." *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[114] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly." in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, 2015, pp. 297–310.

[115] Stack Overflow, "What Happens When Stack and Heap Collide," https://stackoverflow.com/questions/1334055/what-happens-when-stack-and-heap-collide, 2023.

[116] M. R. Swanson, L. Stoller, and J. B. Carter, "Increasing TLB Reach Using Superpages Backed by Shadow Memory." in *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, 1998, pp. 204–213.

[117] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support." in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 171–182.

[118] M. Talluri, S. I. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in Supporting Two Page Sizes." in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 415–424.

[119] I. Tanasic, I. Gelado, M. Jordà, E. Ayguadé, and N. Navarro, "Efficient exception handling support for GPUs." in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 109–122.

[120] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems." *IEEE Commun. Surv. Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[121] The Linux Kernel Archives, "Kernel Samepage Merging," 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html

[122] T. N. Theis and H.-S. P. Wong, "The End of Moore's Law: A New Beginning for Information Technology." *Comput. Sci. Eng.*, vol. 19, no. 2, pp. 41–50, 2017.

[123] Tom Warren, "Microsoft Bing hits 100 million active users in bid to grab share from Google," 2023. [Online]. Available: https://www.theverge.com/2023/3/9/23631912/microsoft-bing-100-million-daily-active-users-milestone

[124] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, 2021, pp. 559–572.

[125] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat Caches for Scale-Out Servers." *IEEE Micro*, vol. 37, no. 2, pp. 90–103, 2017.

[126] D. L. Weaver and T. Germond, "The SPARC Architecture Manual - Version 9," 1994. [Online]. Available: https://www.cs.utexas.edu/users/novak/sparcv9.pdf

[127] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors." in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 266–277.

[128] Wikichip, "AMD Zen3 Specs," https://en.wikichip.org/wiki/amd/microarchitectures/zen_3, 2022.

[129] Wikichip, "ARM Cortex A76," https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76, 2022.

[130] Wikichip, "Intel Kabylake Specs," https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake, 2022.

[131] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt, "Mechanisms for Cooperative Shared

Memory." in *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, 1993, pp. 156–167.

[132] D. A. Wood, S. J. Eggers, G. A. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson, "An In-Cache Address Translation Mechanism." in *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*, 1986, pp. 358–365.

[133] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li, J. Lin, T. Liu, Z. Liu, J. Tan, H. Wang, H. Wang, K. Wang, C. Zhang, F. Zhang, L. Zhang, Z. Zhang, Y. Zhao, Y. Zhou, Y. Zhou, J. Zou, Y. Cai, D. Huan, Z. Li, J. Zhao, Z. Chen, W. He, Q. Quan, X. Liu, S. Wang, K. Shi, N. Sun, and Y. Bao, "Towards Developing High Performance RISC-V Processors Using Agile Methodology." in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1178–1199.

[134] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 331–345.

[135] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Translation ranger: operating system support for contiguity-aware TLBs." in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 698–710.

[136] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering Translation Bandwidth with Virtual Caching." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 113–127.

[137] A. P. Zarandi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus Prime: Accelerating Data Transformation in Servers." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020, pp. 1203–1216.

[138] A. P. Zarandi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC Tax in Datacenters." in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 407–420.

[139] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: architectural and operating system support for reducing the impact of address translation." in *Proceedings of the 24th International Conference on Supercomputing (ICS)*, 2010, pp. 159–168.

Siddharth Gupta

Ph.D. Student, Computer Science, École Polytechnique Fédérale de Lausanne (EPFL)
Email: siddharth.gupta@epfl.ch Website: https://sites.google.com/view/siddharthgupta04

Research Interests

I am broadly interested in the field of systems and cross-stack research problems found in modern, large-scale datacenters. Currently, my research focuses on designing high-performance and cost-effective TB-scale memory hierarchies for datacenter servers.

Education

Doctor of Philosophy in Computer Science — **EPFL, Switzerland**
Thesis: Rebooting Virtual Memory with Midgard — 2017-2023
Thesis Advisor: Prof. Babak Falsafi and Prof. Abhishek Bhattacharjee

Bachelor of Engineering in Computer Science — **BITS Pilani, India**
Thesis: Multi-Threading Aware Resource Management for Online Services — 2013-2017
Thesis Advisor: Prof. Boris Grot

Peer-Reviewed Conference Publications

Imprecise Store Exceptions — **ISCA'23**
Handling exceptions generated by logic embedded in the cache/memory hierarchy
S. Gupta, Y. Li, Q. Kang, A. Bhattacharjee, B. Falsafi, Y. Oh, and M. Payer

SecureCells: A Secure Compartmentalized Architecture — **IEEE S&P'23**
Compartmentalization with ns-scale operations using hardware support
A. Bhattacharyya, F. Hofhammer, Y. Li, **S. Gupta**, A. Sanchez, B. Falsafi, and M. Payer

AstriFlash: A Flash-Based System for Online Services — **HPCA'23**
Integrating NAND flash in the memory hierarchy for online services
S. Gupta, Y. Oh, L. Yan, M. Sutherland, A. Bhattacharjee, B. Falsafi, and P. Hsu

Rebooting Virtual Memory with Midgard — **ISCA'21**
Introducing VMA-based intermediate address spaces to optimize address translation
S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer

The NeBuLa RPC-Optimized Architecture — **ISCA'20**
Optimized hardware design to support RPCs
M. Sutherland, **S. Gupta**, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis

Optimus Prime: Accelerating Data Transformation in Servers — **ASPLOS'20**
Accelerator to perform fast data-transformation operations in servers
A. Pourhabibi, **S. Gupta**, H. Kassir, M. Sutherland, Z. Tian, M. Drumond, B. Falsafi, and C. Koch

Distributed Logless Atomic Durability with Persistent Memory — **MICRO'19**
Maintaining crash consistency while avoiding logging
S. Gupta, A. Daglis, and B. Falsafi

Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores — **HPCA'19**
Colocating online services with batch workloads while maintaining QoS
A. Margaritov, **S. Gupta**, R. Gonzalez-Alberquilla, and B. Grot
Best Paper Award.

Awards and Recognition

Recipient of the Qualcomm Innovation Fellowship 2021
Best Paper Award: HPCA 2019
Invited Talks: MSR Cambridge PhD Workshop 2021, SPMA 2022/2020, NVMW 2020
Selected to attend the Heidelberg Laureate Forum (HLF) 2022

Work Experience

Research Intern at Huawei Research Center, Munich **Summer 2021**
Manager: Dr. Javier Picorel, Intelligent Cloud Technologies Laboratory

Research Intern at PARSA, EPFL **Summer 2017**
Advisor: Prof. Babak Falsafi, Summer@EPFL

Research Intern at School of Informatics, University of Edinburgh **Fall 2016**
Advisor: Prof. Boris Grot

Software Contributions

QFlex | I am one of the main architects of the QFlex project that targets quick, accurate, and flexible simulation of multi-node computer systems (https://qflex.epfl.ch). QFlex combines QEMU, a popular open source machine emulator, along with Flexus, a cycle accurate simulator that can be used to practically model full-blown modern server software stacks. QFlex allows bringing up ARM ISA-based unmodified OS and server software stacks on complex CPU and accelerator designs, and performing cycle-accurate timing simulations of the same to assess their benefits.

Skills (Languages and Tools)

C/C++ (>10 years of experience), Bash, Linux, Python, Git, Latex

Teaching Experience

Teaching Assistant at IC, EPFL

Fall 2022,21,20,19	Introduction to Multiprocessor Architecture	Prof. Babak Falsafi
Spring 2021,20	Advanced Computer Architecture	Prof. Paolo Ienne
Fall 2018	Computer Architecture	Dr. Mirjana Stojilovic
Spring 2018	Introduction to Operating Systems	Prof. Willy Zwaenepoel