Modeling and Designing Secure Tightly-Coupled Accelerators in CPUs

By

David J. Schlais

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: 12/15/2020

The dissertation is approved by the following members of the Final Oral Committee:

Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Mark D. Hill, Professor Emeritus, Computer Sciences
Gurindar S. Sohi, Professor, Computer Sciences
Joshua San Miguel, Assistant Professor, Electrical and Computer Engineering

To my dad, John Schlais, for teaching me to be willing to take risks.

ACKNOWLEDGMENTS

First and foremost, I thank Jesus Christ, my Savior. Without Him I am nothing, and I thank Him for giving me passion to explore, learn, design, and engineer while I am here on this earth. I also want to thank my wife, Kim, for her unconditional love and support. I am forever grateful for all of her sacrifices she made so that I could pursue my dream of a Doctoral Degree. She also was always there to listen whenever I needed someone to talk to about the struggles in research; she has listened for hours and has learned more about caches, accelerators, branch prediction, and speculative execution than I'm sure she ever expected (or wanted) when marrying me. She has always been there by my side and demonstrated the meaning of true love. I also owe a huge debt of gratitude to my parents, John and Sue, for teaching me the importance of education, dedication, and hard work. Their support has been invaluable.

Second, I express my deepest gratitude to my PhD advisor, Prof. Mikko Lipasti. He is one of the smartest, most humble, and kindest people I have ever met. Without his guidance, I would have made exponentially less meaningful contributions, and this dissertation would not have been possible. He has taught me how to find the "interesting" questions among my flurry of often vague thoughts and focused my direction when I started going down unproductive or aimless paths. I also want to thank him for his deep care for his students as individuals and to let students pursue their own passions, dreams, and personal/career goals. His willingness to follow his students' passions rather than pushing his passions onto his students made my, as well as my entire lab's, experience much more enjoyable. That has also taught me how to explore and lead my own research paths rather than just

knowing how to follow someone else's. Overall, he has already taught me so much, and he is someone I look up to as a high role model, both as a researcher, and as a person. I am honored to have had him as an advisor and to have worked so closely with for the past 6 years.

Next, I wanted to thank all of the University of Wisconsin-Madison professors in the ECE and CS department, particularly my PhD Committee members. Having such an elite group of Computer Architects to observe and learn from for so many years has shown and taught me more than I could have ever imagined going into graduate school. I am honored to have taken computer architecture courses from Professor Emeritus David Wood and Professor Emeritus Mark Hill, having entered graduate school at the perfect time to be able to gain their wisdom and experience prior to their retirement. Having these computer architects as teachers and attending talks with them is an invaluable experience that I will always cherish. I'm grateful to have Professor Emeritus Mark Hill now on my defense committee and I thank him for providing me insights and feedback on this research. Likewise, I am extremely honored to have Professor Guri Sohi on my committee with all of his experience and expertise. I thank him for meeting up with me, for his thought-provoking questions, guidance, and advice to help direct me towards related work that I could apply concepts from in my dissertation. I am also extremely grateful to have Prof. Josh San Miguel on my committee. His ability to grasp complex concepts and ask deep questions on a wide assorting of computer engineering topics is remarkable. Many of his questions over the past couple years, even prior to being on my committee, have led me to explore important details to improve several of my papers' and dissertation's content. The number of awards, papers, patents, and honors that belong to those I have

attended talks with, had meaningful conversations with, had as instructors, and now have as committee members at the University of Wisconsin is humbling. I'm so grateful to have had the opportunity to have learned in this environment.

I also have to thank all of my friends and colleagues, particularly in Prof. Mikko Lipasti's PHARM research lab. Having friends that understand the difficulties associated with pursuing PhD research and could be there alongside me has been a great emotional support. Additionally, our conversations often led to very stimulating intellectual conversations, and have made me a better researcher. I am very grateful for those that joined this path before me that I could look up to and reach out to for support and guidance, including Dr. David Palframan, Dr. Dibakar Gope, Dr. Rohit Shukla, Dr. Michael Mishkin, Sooraj Puthoor, and Dr. Gokul Ravi. A special thanks goes to Dr. Gope, who not only pulled me into his project while a PhD student, but also provided his detailed PHP analysis, which I was able to use as the basis for evaluating the performance overheads in Section 6.7. I also value our discussions regarding the security invariants proposed.

I'm also extremely grateful for all the students that joined soon after me, who I've been able to get to know and have meaningful conversations with, including Heng Zhuo, Kyle Daruwalla, Ravi Raju, Chien-Fu Chen, Carly Schulz, and Soroosh Khoram. I am very blessed to have been able to have these friendships and get to know each and every one of them. I cannot wait to see all of the amazing things that these smart and determined people will accomplish. A special thanks goes to Heng Zhuo, who has been both a great friend and has been a great research collaborator. He helped create the gem5 implementation for setting "NT" and "NL" instructions used to compare our analytical model to. He also added the new μops (predicate loads and M$ checks) within gem5 needed to test the NoRF *matrix*

cache implementation. I also value all the amazing graduate students I've met over the years in the CS and ECE departments. Each of them has been a vital part in my graduate studies to improve my skills to think critically, see things from new perspectives, conduct research, communicate novel ideas, collaborate, and share ideas.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Hardware accelerators are commonly deployed to increase performance and efficiency over general-purpose compute. These accelerators have been increasingly used to offload work from CPUs. This dissertation evaluates a class of accelerators called tightly-coupled accelerators (TCAs) which target fine-grained tasks. TCAs present new challenges in their modeling, implementation, and verification compared to loosely-coupled accelerators. This dissertation focuses on several of these new challenges and proposes solutions to gain better understanding of TCAs and increase the feasibility of deployment.

We first look at ways to mathematically model the performance of TCAs under different execution models. Sending TCA instructions through the CPU pipeline can impact the available concurrency between the TCA and CPU depending on the supporting hardware. It is important for architects to be able to quickly estimate and reason through the design-space and estimated impacts of an accelerator before going through the full process of simulation and implementation. With reasonable error, our model allows computer architects to estimate the performance of these accelerators orders of magnitude faster compared to detailed simulators.

Our next contribution evaluates the different design considerations for transferring data between TCAs and the CPU. We specifically focus on the differences of passing data between the CPU and TCA through register files and the memory hierarchy. We propose data transfer that completely bypasses the CPU's register file for significant energy savings for the evaluated workloads.

We lastly introduce the concept of having a reconfigurable TCA within a CPU and

discuss timing-based and speculative-based security challenges that emerge. We show that TCAs can create a new attack surface to both amplify existing known attacks, as well as create a new side-channel for large-scale attacks. We evaluate these potential vulnerabilities and propose invariants that, when enforced, close these vulnerabilities. We also demonstrate the ability to create useful TCAs while enforcing these secure invariants with reasonable overheads in performance, area, and power.

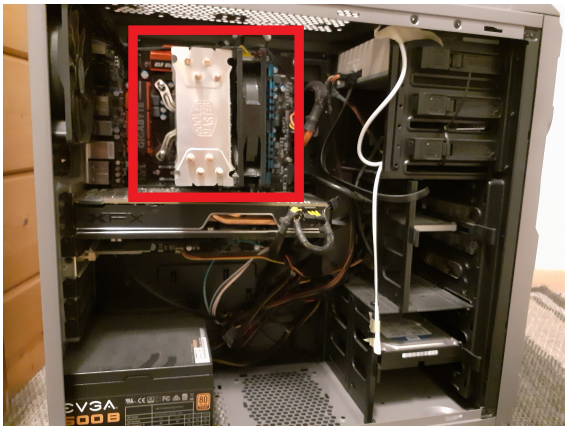
1 EXPLAINING MY PHD RESEARCH TO THE GENERAL PUBLIC

This section is written as part of the Wisconsin Initiative for Science Literacy (WISL) program to explain PhD Research to the general public. Most of the work contained further in this defense, as well as the majority of PhD research in the scientific community, is focused towards an audience with a specific technical background. However, many of my friends, family, as well as others that are excited to learn more about what I have been doing for the past six years should be able to see the work I've conducted without having to go through all the jargon, technical details, and background of numerous Computer Engineering classes. Additionally, the scientific community should strive to excite future young minds to pursue the fields and see the excitement that research allows without digging through hundreds of pages of technical detail. I am excited to be a part of this initiative to dedicate a chapter of my research to explain things through the lens of the everyday person not studying Computer Engineering or Computer Architecture.

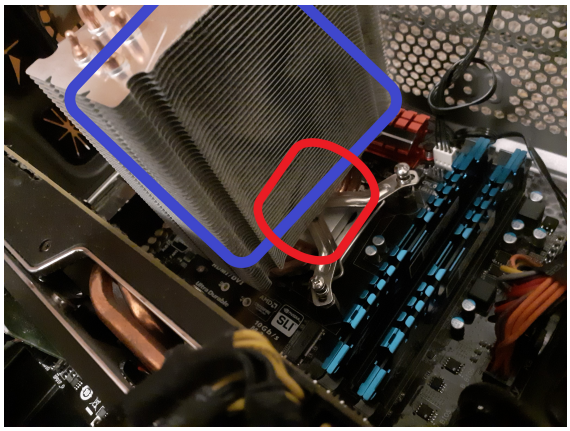
1.1 Background

What do you think of when you hear the word "computer?" You probably think of a big rectangular box next to a desk, laptop, tablet, or maybe a large company datacenter/server. These are used in our everyday life to solve complicated problems, store our information, or entertain us through video games or streaming entertainment. However, most of the things we think that the whole laptop, desktop, smartphone, etc. are doing are in fact done within the processor. You may have heard the term CPU before, which stands for Central

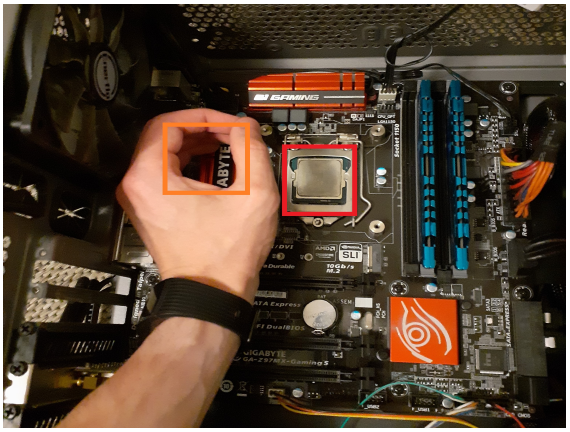
(a) Desktop – The actual desktop computer is shown in the red box. The monitors on the desk only show you what the computer is working on.



(b) Inside a desktop computer. The processor sits under the big radiator shown in the red box.



(c) A different angle of the CPU cooling radiator (blue square), which sits on top of the processor (red square). The radiator keeps the processor from overheating.



(d) The processor is only visible after removing the radiator. The size of the the computer's brains, or processor (red square), compared to the size of my hand (orange square).

Figure 1.1: Photos of my desktop computer, zooming in from the desktop housing (a), to inside the desktop housing (b), all the way to the actual processor (d). Computer architects are responsible for designing the circuitry within the processor shown in photo (d).

Processing Unit. Whenever you do something on your computer, the CPU is almost always the part responsible for making your commands happen. The CPU determines when and how your "enemy" or "opponent" moves in your video games, determines what to do next based on your mouse movements, solves difficult math problems, searches for files on your computer, determines what to display on your screen when you surf the web, and much more. The CPU can be thought of as the "brains" of a computer. For simplicity, whenever I use the terms "CPU," "processor," and "computer," think of the brain. It may be surprising to learn that this brain is only about the size of the square created by your finger outline if you touch your index finger to your thumb on the same hand (See photos in Figure 1.1)! My field of research is in "computer architecture," which is responsible for designing and upgrading this processor.

Computers operate by running "computer programs," also known as "software." Software tells the processor what operations (also called instructions) to perform, and in what order. But what instructions are available for the software programmer to use? This is where "hardware" comes in, which is the focus of computer architecture. Hardware is a group of electrical circuits that performs those instructions, such as addition, subtraction, multiplication, comparisons, reading data, and writing data. A few examples are things like "add these two numbers together," "move the number from location A to location B," or "tell me if number C is less than number D." That's it. You might be asking yourself right about now "but how can a computer solve so many complicated problems so quickly if all it can do is simple operations?" The answer is because it can do these operations extremely quickly. A modern processor can compute a few billion of these operations every second. So, although the operations are simple, by breaking down big problems into a series of

these small operations, the processor can still solve the problems much faster than we can. The processor has no idea what it is doing – it will blindly do the operations it is told to do. That is why one philosophical view is that computers are not very smart, they're just fast at what they do. Once the hardware has a circuit built to perform an instruction, the programmer can group sequences of these instructions together to perform complex tasks.

As a computer architect, I work on both designing individual circuits to perform operations, and linking existing circuits together in new ways to be better than prior designs. I organize the circuits and components to detail how a new processor should be built, just like building architects design the details of a new building's blueprint. Computer architects attempt to make the processor faster, more powerful in its computation ability, use less energy, and modify anything to optimize it for both today's and tomorrow's needs. Computer architects also have to be aware of and involved with the software community to understand what requirements and what building blocks they wish they had in future processors.

1.2 Motivation Behind Accelerators

Processors are extremely powerful because they are extremely general-purpose. By this, I mean that there is an infinite number of computer programs that a processor will be capable of running. Software can be updated, and the processor will still be able to run it. You can create a new program, and the processor will be able to run it. It is only up to the developer's imagination of how to combine these operations together, the computer will be able to run it. It is flexible enough to run any valid computer program across any domain,

whether video games, company software, warehouse management, medical record storage, etc. However, this flexibility comes at some cost. By having only basic building blocks, sometimes you need very long sequences of building blocks to perform a task. If that task is performed often enough, the CPU is simply repeating the same blocks over and over again. However, if the computer knew how to perform the task (rather than only the small building blocks), it could perform that task through a single command, rather than by a sequence of many commands. This is where "accelerators" come into play. An accelerator is a circuit that is specialized to do a single task extremely well. The benefit of accelerators is that they are fast and efficient. Their downside is that they only know how to perform that one task. If the same task needed to be even slightly adjusted or modified, the accelerator cannot perform that task at all, and must go back to using basic instructions. This is why accelerators are typically only built when it is fairly certain that (a) this task is used often enough to justify adding it (otherwise it will just sit doing nothing most of the time), and (b) that the task will not need to be updated or changed in the near future.

When accelerating large tasks involving lots of work (such as taking the video captured by your smartphone camera and converting it into a movie file), accelerators are built far away from the processor core (the "heart" or center of the processor), and are called loosely-coupled accelerators. However, when accelerating fine-grained tasks (something like renaming a computer file) where the time it takes to perform operations cannot afford to move data far away from the processor, these need to be tightly-coupled accelerators. Being tightly-coupled makes data movement faster and should be justified only if used frequently, as it takes the place of other circuits that could be close to the core. My work focuses on tightly-coupled accelerators, as it is a new and emerging type of accelerator that

has the potential to provide significant speedups for many emerging computer needs, such as machine learning, artificial intelligence, and server optimization.

1.3 Contribution 1 – Performance Modeling of Tightly-Coupled Accelerators

I started working on tightly-coupled accelerators because of their potential future impact. There is a large amount of excitement in the computer field around artificial intelligence (AI) and machine learning (ML). These are software techniques to try to predict information about either the here-and-now or the future based on information seen in the past. AI and ML work by showing thousands or millions of examples to a computer, telling the computer what the example is, and having the computer try to find similarities in each of the examples. Eventually, the computer can correctly understand the details or "features" that made that example what it was. For example, a computer could be shown several different CAT scan images of cancerous and non-cancerous regions. After seeing enough examples that doctors have determined cancerous and non-cancerous, the computer will determine similarities between the cancerous images, and similarities between the non-cancerous images. The idea is that when the computer is presented with a new CAT scan, it can say whether it looks more like the cancerous or non-cancerous images it has seen in the past. Websites like Amazon, Netflix, or Youtube can also use AI and ML to predict what movies to recommend to you based on what you have stated you like and don't like.

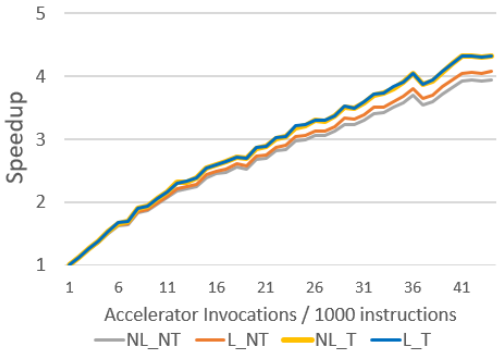
All of these AI and ML tasks can be calculated by the computer through a series of

matrix-matrix multiplications. Simply put, the computer performs mathematical operations to understand similarities and dissimilarities. Since many different AI and ML tasks all perform these mathematical operations, it makes sense to build an accelerator to perform this task. However, as computer architects, we need to decide how much faster this accelerator would run compared to the CPU as it runs today. This task is more difficult than it may originally seem. Processors have become more complex over the years. Computer architects have optimized processors to run instructions out-of-order as well as predict what the computer program will attempt to do next. The computer processor can perform multiple computer program instructions simultaneously before it even knows if it should perform those instructions. An example might be predicting the value of a complex mathematical formula before the final calculation is finished. The reason the processor does this is because correct predictions make the program run faster. In the end, any mistakes the processor made in guessing (called incorrect execution) will always be undone, ensuring the program still ran correctly.

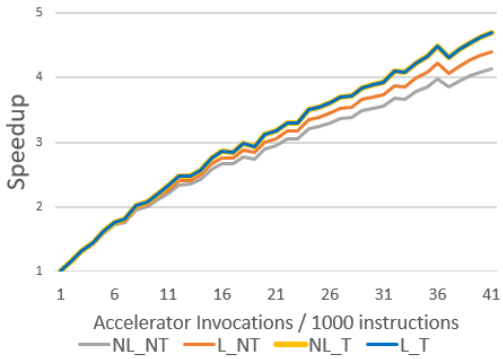
However, having a complex processor brings challenges for computer architects to integrate additional accelerators into the processor. A complex way for an architect to address this issue would be to design an accelerator that can always operate no matter what else is going on around it, and be able to recover from any mistakes made by the processor in guessing where it should go next. A less complex way for an architect to address this issue would be to design an accelerator that waits for the processor to be certain that the task should be performed, and only resume its guessing work after the accelerator has finished its task. Having this 'wait if unsure' policy means that the accelerator never has to correct mistakes, but waiting will slow down the processor from useful work it could have

been doing while the accelerator ran.

The architect could design multiple complex and simple accelerator versions, see how they do, and decide which one to use in the next processor. However, designing accelerators that will not be used takes time and will waste several months of engineering work. Additionally, to actually build processors/accelerators costs millions of dollars per different design. Throwing out designs after building them would waste lots of money. For this reason, computer architects use simulators. These simulators can be thought of as a computer program that is acting like a computer. So really, it's like running a computer within your computer. The reason this is useful is that you can make modifications to your simulated computer without having to build or pay for a new computer. The downside is that you have to tell the program every detail about your simulated computer, how the accelerator interacts with the other components around it, and it runs thousands of times slower than the computer itself. Also, the architect needs to know enough about the



(a) Simulator-based estimation of tightly-coupled accelerator speedup.



(b) Our mathematical model estimation of tightly-coupled accelerator speedup.

Figure 1.2: Estimated performance speedup for a tightly-coupled accelerator. Our model is slightly different than the simulator results, but the overall trend and insights are maintained. The detailed simulator took over 100 hours, and our mathematical formulas took under 5 minutes.

computer simulator to be able to make the modifications required to even test their new designs. This can take hundreds of hours to complete.

My contribution is to create mathematical formulas to estimate the impact of different designs on the execution time of any specific program. Figure 1.2 shows the output from our mathematical formula as compared to the detailed simulator. Instead of building the hardware for millions of dollars, or spending dozens of hours modifying a simulator, an architect can use the formulas that I've created in order to get a good estimation of the overall speedup of various designs in the matter of seconds or minutes. Although these mathematical formulas aren't as exact as the detailed simulator, by having a good estimate, the designer can be confident which design is worthwhile to try to model in the simulator or to build without wasting time focusing on designs that would have been quickly thrown out through simple mathematical estimations. This allows designers to potentially save millions of dollars and hundreds of engineering hours, allowing for a much more rapid time to deploy these accelerators into real-world models!

1.4 Contribution 2 – Removing Unnecessary Data Movement in Tightly-Coupled Accelerators

So far, I've been talking about processors performing calculations, such as doing addition, multiplication, or matrix-matrix multiplication operations. However, in order to perform these operations, the data that is being operated on (also called operands), needs to be supplied to the units doing the computation. This can be thought of as a blender making a

smoothie – the blender will mix all the ingredients, but you have to bring the ingredients to the blender for a smoothie to be possible. All data in a computer is stored through memory. Memory is simply the storage of numbers. Typically, in order to perform operations in a processor, data is transferred from memory into something called a Register File (RF). Anytime an operation is performed, the operands must come from the RF, and if the data is not currently in the register file, then it must be brought from memory into the RF. The RF can only hold a certain number of operands, so if there is not enough space, then something needs to be removed from the RF (and back into memory if it needs to be saved) before a new value can be brought back in. The RF gains the most usefulness when a value is used many times in the RF before it is removed. This eliminates the need to keep transferring the value to and from memory.

A second analogy could be someone operating a carnival dart game, where small and large prizes can be won. All prizes are originally kept in the back room, which in our analogy is memory. The worker brings in a basket from home, which is the RF in our analogy. Contestants are more likely to win small prizes, so the worker fills their basket with two boxes, one for each small prize – pencils and rulers. In reality, there might be dozens of items in the RF, but for simplicity, we'll say there are only two. When a contestant wins a small prize, the worker simply gives the small prize to the winner from the basket without having to go into the back room. However, when a contestant wins a large prize, like a stuffed animal, the worker goes to the back room to get the prize. If the worker is required to give all prizes from the basket, which can only fit two boxes, they might remove the ruler box to replace it with the stuffed animal box. However, as more small prizes are won later on, the worker will have to go to the back room again to replace the stuffed

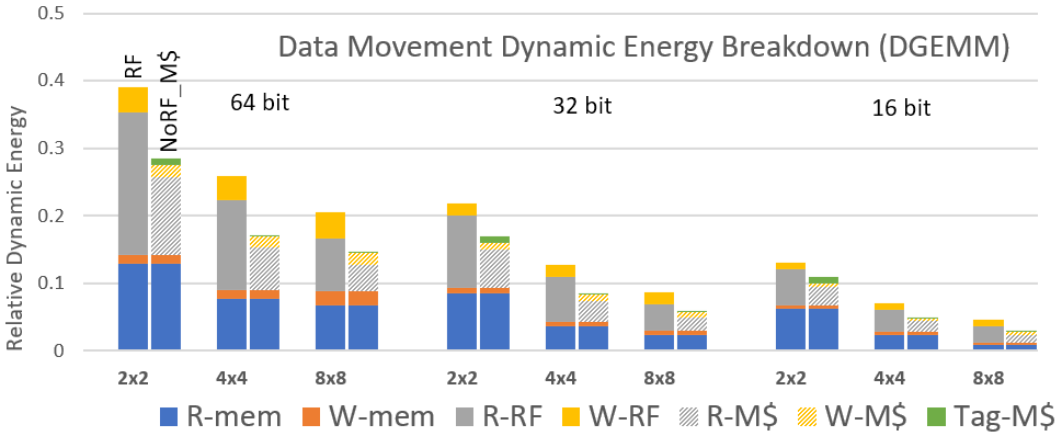


Figure 1.3: Energy consumption moving data in tightly-coupled accelerators using the old method of the Basket/RF (left bar per pair) compared to using our proposed method (right bar per pair). Lower is better. Overall height is decreased for every pair, showing a reduction in energy consumption, which is good.

animal box with the ruler box. Here, the requirement to pass prizes from the basket forced the worker to remove an item they knew they would likely use again soon.

This is how operand passing has been done for decades – forcing all operands (prizes) to be passed to the processor through the register file (basket). However, we make the argument for many of the proposed accelerators for today, such as matrix-matrix multiplication, there is not enough reuse to warrant using the basket, or register file. Alternatively, the accelerator/worker could use a method to simply take a large prize from the back room, never touching the basket. However, the downside of bypassing the basket comes when contestants win large prizes consecutively. In that case, bringing the entire stuffed animal box from the back room could have spared the worker the extra trips. Our analysis shows that there is not as much reuse in these designs, and that significant energy can be saved in the processor by eliminating the cost to fill and replace items in the basket, even if the backroom is used more often. Figure 1.3 shows the reduction in energy consumption of our method (denoted NoRF_M$) compared to the RF method for various tightly-coupled

accelerators. This creates a big incentive to move away from the long-lived tradition of using the RF as the only means to operate on data. We also created a clever technique to prevent multiple trips to the large prize box without using the basket if consecutive large prize winners are expected. Using our technique instead of the basket operates at a fraction of the energy consumption as the old technique.

1.5 Contribution 3 – Security within the Tightly-Coupled Accelerator

It is common to hear on the news about security breaches or hackers that have been allowed to see information from computers that they did not have permission to see. Security is always a cat-and-mouse game where hackers are trying to find new techniques to extract information, while researchers are trying to come up with ways to close loopholes before they can be exploited. In 2018, a new classification of hardware security vulnerabilities was discovered. The two large exploits were called Spectre and Meltdown. These attacks took advantage of a process computer architects had been using to increase performance for decades called speculative execution. Speculative execution means that the computer is predicting what it should do next before knowing for sure that it should be done. It's a strange process to think about, because programs are written in-order, or sequentially. The issue arises when the processor hits a "fork in the road," and it will take a while before it learns which path to take. It could simply wait at the fork until it knows for sure, or it could pick one of the two paths, and later be told whether or not it was correct. When it

predicts correctly, the program will run faster. In the past, if it was incorrect, all operations it had done were thought to be erased, and the processor would simply go back and go down the correct path.

Here's an analogy: let's say you are doing a scavenger hunt on a team in the woods with many forks in the road. Each fork has a security guard and a hard puzzle to solve which tells you which direction to go. Your team members communicate via cell phone which path to take, and you're looking for specific items. When you hit a fork in the road, you call your team, tell them the puzzle, and they start to solve it. To save time, you guess which path is the correct one, and start searching for items. You get a call back at some point to determine if you went down the correct path or not. If you went down the correct path, you continue on your way. If you did not, you need to backtrack to the fork, at which point the security guard will take away any items that you found on the wrong path. The vulnerability comes when researchers found a way to slingshot a **single** item (1 Byte, or 8 bits of a secret) from the incorrect path into the correct path. So even when passing the security guard, the searcher did not have any items on them, and the security guard let them go. They could then later pick the item up after changing direction into the new path. A cheater may be able to exploit this to cheat during the scavenger hunt. Figure 1.4 shows the attack. The math problem is the fork in the road, and the scavenger went down the 'YES' path, slingshot the sensitive data (such as a password letter) from the 'YES' to 'NO' path, and later went back to the NO path once the math problem was completely solved. Since the searcher no longer has the value in their pockets after using the slingshot, the security guard does not catch the cheater.

Now let's say we want to create an arbitrary accelerator into the processor design. You

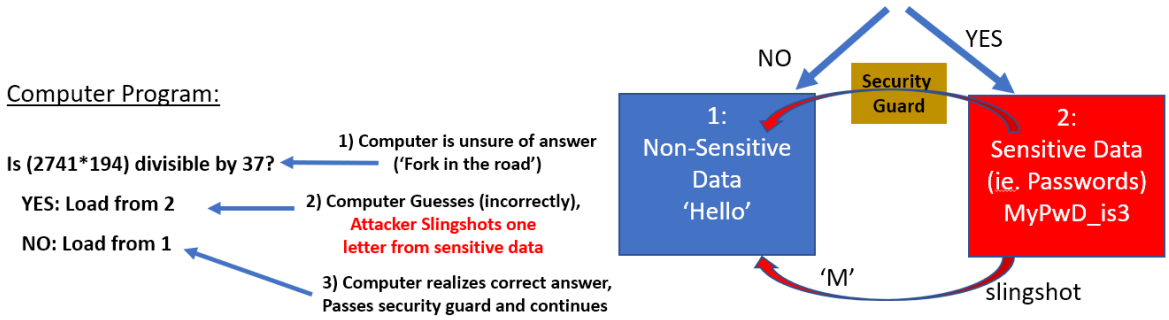


Figure 1.4: Example of Spectre-like attack. Even though the sensitive data is supposed to be discarded once passing the security guard, the attacker found a way to 'slingshot' information from the sensitive data. Since the attacker doesn't physically hold any values, the security guard lets them through. Doing this multiple times eventually tells the attacker 'MyPwD_is3'

might want to do this in order to make a frequently-used computer program run faster. Building this new accelerator is as if instead of you wearing a standard uniform with only a single pocket for the security guard to check, you're allowed to wear any uniform you want, which may have secret compartments. If the security guard only knows how to check a single type of pocket, the cheater could wear a hat, for instance, and stick hidden items in their hat without being detected by the guard. This is even worse than the slingshot case, because instead of a single item being stolen, you could stash as many items as possible for each wrong path. The benefit, however, to having an arbitrary accelerator would be huge in the computer architecture community, as you could accelerate virtually anything you want, making programs much faster. This impact would be even larger if the accelerator could be switched in and out. Although it may sound like fantasy to have circuits and physical parts that can change within a computer, there's a technology called Field-Programmable-Gate-Arrays (FPGA) that has the capability to implement a desired circuit, and can later be reconfigured to physically implement a different circuit. If this

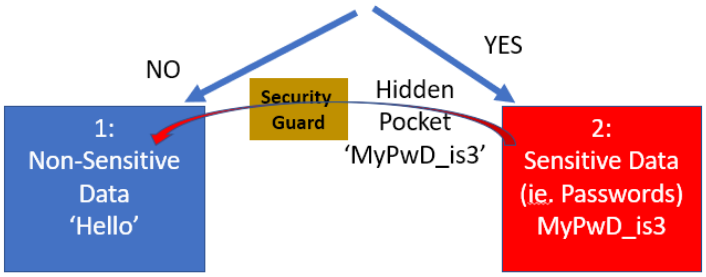


Figure 1.5: Same attack as Figure 1.4, but this time the attacker has a hidden pocket, containing the entire password 'MyPwD_is3' – Our security rules prevent the searcher from having pockets that it doesn't know about, preventing this much faster and more detrimental attack.

technology could be combined into the processor as a tightly-coupled accelerator, then the same FPGA hardware could implement an infinite number of accelerators, where the programmer can choose which accelerator to use for their specific program.

The risk of having any arbitrary accelerator is clear in the analogy of being able to stash lots of secret information without the security guard's detection. Our work solves this problem by making rules that the accelerator designer must use in order to run that accelerator. For example, in our analogy, an outfit might only be allowed zipper pockets or open pockets, but not be allowed stitched-shut pockets or hats. The security guard then knows exactly what to look for. In the analogy this might be silly, but from the hardware perspective, certain circuit components that are allowed to hold information long-term (called flip-flops) can be required to be connected to a wire from the CPU that clears all data from wrong paths before it can be stolen. If the designer does not follow the rule, then the circuit cannot be loaded. If the rule is followed using the trusted, safe, building blocks we provide, we can make certain guarantees about safety.

This set of rules can be extremely powerful. If a processor company such as Intel or

AMD wants to design accelerators, by following our rules, they can make certain guarantees about the security of their accelerators. Alternatively, this allows for the possibility of even allowing the everyday programmer to make whatever accelerator they want. Traditionally, hardware is designed up front and not changeable once it is shipped. This forces programmers to use the existing tools to make their designs, but having the possibility to make their own hardware could lead to significant benefits in everyday computing. By having the security check rules in place, companies like Intel or AMD may be able to have reconfigurable hardware to accelerate many tasks without risking someone with malicious intent using that hardware to obtain secret information.

1.6 Concluding Remarks and Future Work

The focus of my work is tightly-coupled accelerators, which are specialized circuits to significantly speed up programs within a general-purpose processor. I first created a series of mathematical formulas to estimate the benefits of various accelerators without the need for many hours of detailed simulation or millions of dollars to test. Second, I demonstrated how data movement in these tightly-coupled accelerators should not follow the same rules that computer architects have been using for decades to move data, and that our method can save significant energy without compromising performance. I lastly created a framework to allow arbitrary tightly-coupled accelerators to be built in a way that will guarantee that speculative execution attacks are not more detrimental.

In the future, I hope to see this work used by processor companies that will allow both the general processor instructions (such as addition, subtraction, multiplication,

comparisons, etc.), as well as a reconfigurable accelerator instruction that the programmer can use for anything they would like. This would be a huge move forward in the history of processors, as a general-purpose processor could be specialized in millions of ways without losing its existing generality or having to incorporate millions of accelerators at design-time. That means that a processor built in 2025, for example, could be configured to specialize for a program invented in 2027! I hope this work begins the journey of a new era of accelerators and computing.

2 INTRODUCTION

General-purpose computing systems have changed greatly over the past several decades. Virtually all computer abstraction layers have had significant improvements. From the hardware device physics level, advancements in transistor technology have made significant frequency increases and higher yield, which has led to larger (more transistors per die) and more complex processor designs. Similarly, various programming languages and improved compilers have allowed high level algorithmic design to be easier to program. With each passing technology generation, there have been new algorithms desired to be employed on these systems which further push compute and memory limits.

2.1 Execute Units and Accelerators

Microarchitecture and Instruction Set Architecture (ISA) abstraction layers are no exception to the vast changes that have occurred. Some of the most notable changes in microarchitecture have been the move towards out-of-order execution, longer pipelines, multi-threading, superscalar designs, RISC-like internals, and various prediction mechanisms. In fact, with the breakdown of Dennard scaling, there have been increased pressures in recent years within these abstraction layers to have dramatic increases to keep up with expected performance improvements. This is because the breakdown creates less benefit from the device physics levels, causing researchers to have increased focus on other abstraction layers to maintain performance improvements.

When defining an instruction set architecture (ISA) for processors, there is often a balance between creating more complicated, specialized instructions within a Complex

Instruction Set Computer (CISC) and simpler instructions within a Reduced Instruction Set Computer (RISC). Complex instructions can reduce number of instructions at the expense of increased hardware complexity and reduced instructions per cycle. On the other hand, RISC instructions allow optimized instructions for simplicity and parallelism in pipelined designs at the cost of an increased number of instructions.

Commonly repeated software routines can evolve over time from software functions to designated hardware units. For example, floating point operations were previously calculated using floating point software emulation. When workloads increasingly used floating point operations, architects tightly-integrated specific hardware units to replace the slower software functions. A more recent example is how the increase of matrix-matrix multiplication on GPUs has led architects to create designated 4×4 matrix-matrix multiplication instructions and dedicated hardware in the NVIDIA Volta architecture's 'tensor cores.' Jia et al. [47] analyze how closely these tensor cores approach optimal theoretical throughput for matrix multiplication. Similarly, SIMD extensions (e.g., [45]) can be viewed as the same class of specialized hardware, where previously software-defined loops of operations are replaced with designated wide registers and operators.

On a larger scale, when entire algorithms or programs are commonly used, computer architects often design specialized circuits to perform the given task in hardware, called accelerators, rather than through software routines. Computer architects introduced hardware accelerators to deliver expected performance improvements without violating power budgets. These accelerators improve energy efficiency over a processor's general-purpose pipeline and increase performance by trading off software flexibility for performance. Certain programs and algorithms can greatly benefit from these accelerators, providing

much-desired improvements in the era of decelerated transistor-level improvements.

2.1.1 Tightly-Coupled Accelerators (TCAs)

When accelerating fine-grained tasks, it becomes important to tightly-couple them to the core. This is because accelerators that are loosely-coupled have longer invocation delays, and generally do not execute speculatively. Since fine-grained tasks have shorter execution times, all delays have an increased relative penalty compared to their coarse-grained counterparts. These invocation delays of loosely-coupled accelerators can significantly diminish, or even eliminate, the benefit of accelerating a fine-grained function. TCAs are a new class of accelerators that require a new type of evaluation from modeling, implementation, and security perspectives.

We define a tightly-coupled accelerator based on the following three requirements: First, the TCA is invoked via a designated instruction in the ISA. Second, we say TCA instructions have the same in-order commit semantics as other instructions. Lastly, we say that operands to the TCA are passed through either register files or the core's coherent cache hierarchy. Outside of these requirements, we define TCAs as broadly as possible, allowing them to incorporate control flow (branching and looping), multiple and unpredictable memory accesses, and arbitrary computational latency.

2.1.2 Accelerator Design Process

There are several steps required to take the idea of an accelerator and turn it into a physical product. The first obvious step involves deciding what accelerator to build, and determining

whether or not it makes sense to create the given accelerator. Since accelerators are typically fixed-function or have limited reprogrammability, the designer should have some level of confidence that the targeted function or task will still be useful by the time the accelerator is built and verified in silicon. Since the entire process often takes many months or years, there is little benefit designing an accelerator that will be obsolete by the time is is created. After having the confidence of future demand and usefulness, the designer should have some expectation about what the accelerator will accomplish.

Performance is one of the primary metrics for determining an accelerator's usefulness. However, in order to know the performance benefits an accelerator will provide, analytical models or detailed simulation is required. Currently, analytical models for accelerators have focused on loosely-coupled accelerators and often have assumptions that are not applicable for TCAs. Alternatively, it can take a significant amount of engineering time to correctly integrate a new piece of TCA hardware into a detailed simulator. For this reason, in this work, we focus on creating an analytical model to quickly estimate the performance for TCAs. We expand the model to allow estimation for 4 types of execution modes with various possible TCA implementations.

After using our analytical model, the designer has reasonable confidence in the expected TCA gains. Once the designer has enough confidence to move forward, the architect is required to make microarchitectural design decisions on how the accelerator should be integrated into the host system (in our case, a CPU). Typically, these design decisions are not obvious, as there are often tradeoffs with each possible design. One of these design tradeoffs is regarding data movement and operand passing. As one potential option, architects could use conventional wisdom along with further critical thinking to determine

design specifications. Alternatively, they could take a more qualitative approach through detailed tools and simulators to gain a deeper understanding of the tradeoffs in the context of TCAs. In this work, we perform this detailed approach to evaluate these tradeoffs. We provide evidence that conventional wisdom of using a register file for operand delivery is not the optimal method for our evaluated TCAs.

After a design has been implemented, it is important to go through the verification process. Verification involves many potential aspects ranging from functional correctness, environmental testing, security testing, stress testing, and system-level integration testing to name a few. Any issues that arise from verification typically require another iteration of design, or the attempt to try to patch the known issues. However, it is obviously optimal to catch verification issues early (preferably in the design stage) to prevent extra delays and expenses that come from fixing issues late in the creation process. Security is no exception to this phenomenon; creating secure hardware during the design stage can prevent many financial and technical challenges later down the road. In this work, we propose secure hardware primitives and invariants for TCA designs. In our view, it is more beneficial to have design rules and invariants during implementation rather than patching security vulnerabilities after it is deemed to be exploitable.

2.2 Modeling TCAs

The performance of an application utilizing a TCA is dependent on the concurrency between the TCA and the CPU. Concurrency with the core requires the accelerator instruction to execute simultaneously with leading (L) and/or trailing (T) instructions. Allowing this

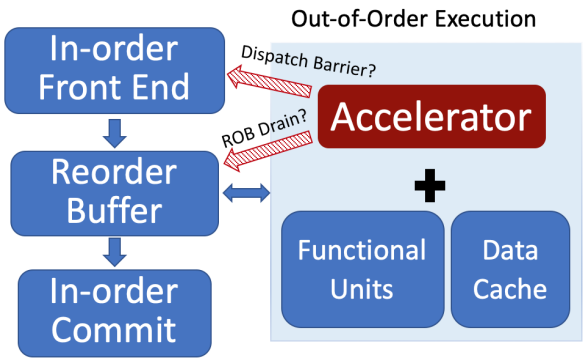


Figure 2.1: TCA integration with an OoO core. ROB drains eliminate leading (L) concurrency. Dispatch barriers eliminate trailing (T) concurrency.

concurrency requires hardware support within the TCA and in CPU integration. For example, concurrency with L instructions requires capability to checkpoint and revert state whenever a leading instruction is a mispredicted branch or causes an exception. Concurrency with T instructions requires dependency checks and data forwarding in case they are dependent on the TCA's output(s). This concurrency also requires support for renaming logic to eliminate false data dependencies.

It is possible to eliminate these hardware requirements by eliminating this concurrent execution. Delaying the issue of a TCA instruction until it is at the head of the reorder buffer (ROB) will prevent TCA execution until all L instructions have committed. Placing a dispatch barrier after the TCA instruction is dispatched will prevent T instructions from dispatching until the TCA instruction has executed (see Figure 2.1).

However, both of these solutions come at some performance cost. Since the architect can either support or **not** (N) support L and/or T concurrency, there are 4 possible modes of execution: NL_NT, NL_T, L_NT, and L_T. A table showing each TCA mode's concurrency with respect to CPU instructions is shown in Figure 2.2. Before deciding which of these modes to design the TCA for, it is important to be able to know the performance impact for

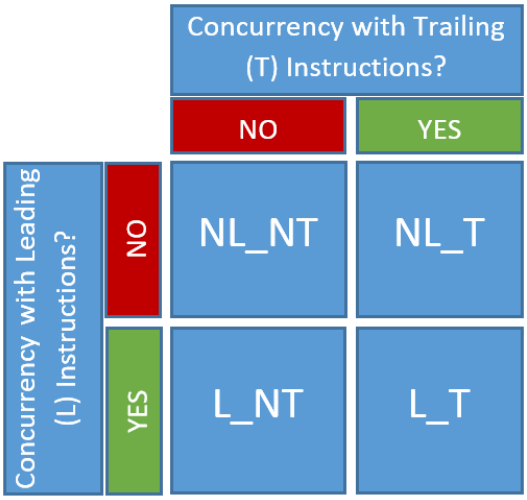


Figure 2.2: TCA modes of execution with various support for concurrency with leading and trailing CPU instructions.

the given application, accelerator, and architecture.

Currently, there are no existing tools besides detailed simulators that allow an architect to estimate these performance impacts. This work provides an analytical model with simple parameters to estimate performance differences orders of magnitude faster than integrating a TCA into a detailed simulator. Our model allows the architect to also do high-level analysis by sweeping through model parameters to come away with important intuition regarding TCA execution across various architectures. Details surrounding the model's design and considerations are provided in Section 4.3.

Figure 2.3 shows an example sweep that can be performed using our analytical model. The model shows that coarse-grained functions that are accelerated (typically implemented as loosely-coupled accelerators) have small performance differences in the 4 execution modes. However, fine-grained accelerators (typically implemented as tightly-coupled accelerators) demonstrate much larger performance differences between the execution modes. Our model demonstrates that the more fine-grained a function is, the more crucial

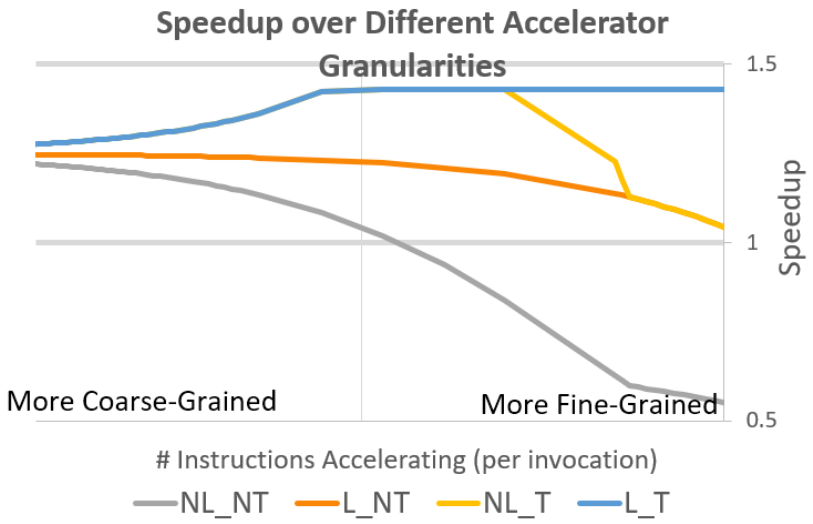


Figure 2.3: Example analytical model sweep over various TCA granularities. More coarse-grained functions (left) have small performance differences in OoO execution modes. Larger performance differences occur for fine-grained functions (right).

it is to support full concurrency with the core (L_T mode).

Since our analytical model also takes the processor's architecture into account, the model also provides insights into policy differences when integrating TCAs into different architectures. The model shows an increase in both NL and NT performance penalties when a TCA executes in more aggressive OoO cores (larger ROBs and/or higher IPC). The delays associated with ROB drains and dispatch barriers increase with these larger cores, increasing the relative penalties associated with limited TCA concurrent execution. Our model both helps architects to make informed choices about specific accelerators and provides insights into the large design space of integrating various accelerators into different architectures. More detailed discussion about conclusions generated by our analytical model is provided in Section 4.7.

(a) TCA integrated with RF

(b) TCA integrated with NoRF

Figure 2.4: Dataflow for TCA that uses RF vs. NoRF. Requiring the use of an RF to deliver TCA operands can add extra data movement steps.

2.3 Operand Passing within TCAs

It is typically common for operands of ALU instructions to be passed through the general-purpose register file (RF). Register files allow reuse between sequences of instructions within the program's dataflow graph. This helps to reduce the number of memory accesses required to perform these long sequences of instructions. However, this buffering of data does not come for free, since it requires extra reads and writes to write the data to this intermediate structure (see Figure 2.4a).

An alternative to passing data through the RF would be to use memory-based instructions, eliminating the need for the RF (see Figure 2.4b). Using this method, which requires **No RF** (NoRF), TCA instructions completely eliminate these extra reads and writes at the cost of an increased number of memory accesses. At first glance, it is unclear which design is optimal for TCAs.

Despite most ALU instructions using the RF, there are differences between typical ALU

instructions and TCA instructions that may lead to different optimal outcomes for operand passing. First, as we have already stated, RFs are designed for reuse between multiple instructions within an application's dataflow graph. TCAs are inherently designed to target specific dataflow graphs and captures all data reuse internally. By capturing this reuse, this reduces the amount of reuse left for the RF to capture between the TCA and its surrounding instructions. Despite having reduced RF reuse, the price to insert and remove data from the RF stays the same. This weakens the argument to use the RF as the primary structure to deliver operands for TCA instructions.

Additionally, TCAs can often have specialized operands (e.g., matrices) that general-purpose CPU RFs were not inherently designed for. Using general-purpose structures for specialized operands creates inefficiencies. We note that when reuse is possible between TCA instructions, it is more efficient to internally cache the data within the TCA in specialized local memory (which we denote as a *matrix cache* in the case of our example DGEMM TCA as detailed later in Section 5.4). A more complete analysis between the differences of RF- and NoRF-based TCAs are described in Chapter 5.

2.4 Security Defenses within TCAs

When a TCA is allowed to execute in full OoO mode (L_T), it interacts with speculative data and has fine-grained interaction with CPU structures such as the LSQ and memory hierarchy. Without careful consideration, this creates the possibility for the TCA to become a new side-channel security vulnerability. It also has the potential to amplify vulnerabilities in existing side-channels used in timing- and speculative- based attacks.

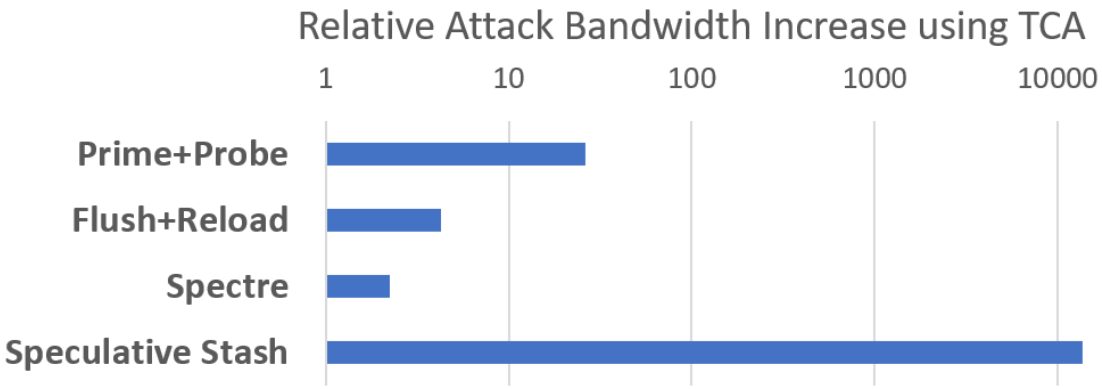


Figure 2.5: Potential attack bandwidth increases using unsecure reconfigurable TCAs.

Figure 2.5 shows that existing timing-based attacks such as Prime+Probe and Flush+Reload can be amplified through the use of TCAs. This can create significant speedups for Spectre attacks that use Prime+Probe or Flush+Reload for data extraction by amplifying the timing side-channel present in cache hierarchies. We describe in Section 6.3.1 that even seemingly benign TCAs can parallelize timing-based attacks in ways not possible in existing CPU hardware. We also present a vulnerability, *speculative stashing*, capable of turning TCAs into a new side-channel. Speculative stashing occurs when a TCA mistakenly or maliciously lets speculative data of a flushed instruction persist beyond a flush signal. This has the potential to leak speculative data at bandwidth rates orders of magnitude higher than Spectre attacks.

By detecting these potentially devastating attacks, we began looking for invariants that would prevent these attacks from occurring.

Speculative-based attack defenses: Speculative-based attacks are possible when speculative data impacts the microarchitectural state of the TCA or CPU. In order to prevent these attacks, we note that it is required to prevent speculative state from being visible to both older (leading) and younger (trailing) instructions. To prevent data from being passed

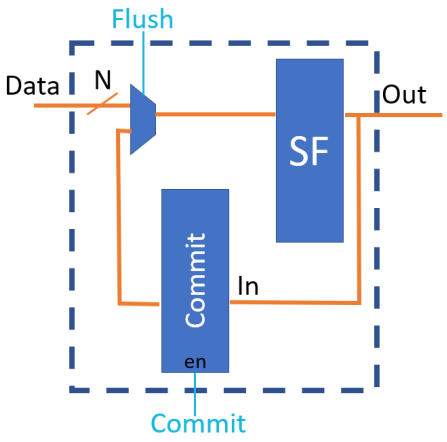


Figure 2.6: A simple example of a FF that tracks speculative and committed state.

to younger instructions, we require the TCA design to use only secure flip flops (FFs) which keeps track of both speculative, as well as committed state (see Figure 2.6 for a simple example of a reverting FF design). Any flush signal reverts all secure FFs to the last correct value, preventing future instructions from seeing the speculative changes from a flushed instruction's execution. A design rule checker (DRC) can enforce that only secure FFs attached to the CPU flush/commit signals exist within the design. To prevent older instructions from seeing the speculative data, the DRC can prevent data from flowing in the reverse direction through the use of a pipelined TCA framework. Although this is an overly-simplistic example, we provide further details in mitigating speculative attacks for realistic TCA integration in Section 6.4.2.

Timing-based attack defenses: Timing-based attacks detect differences in execution time to gain information about the processor's state. For example, memory-based attacks can detect the presence of values in a cache based on the time it takes for its result to return. Clock-gating has been used for decades for energy-efficiency purposes. However, we determine that this energy-efficiency technique has the capability to be a security defense

against timing-based attacks. We make the observation that flops are unable to modify state while their clock is disabled. That means that if a circuit contains only clock-gated flops and single-cycle combinational logic, the output of that circuit cannot be dependent on the duration the clock was disabled. If clock-gating is enforced when accessing a hardware structure that is vulnerable to timing-based attacks, the circuit will not be able to detect the timing variation of that hardware structure. Section 6.4.1 describes these timing defenses in further detail. One of the benefits of using a clock-gating defense is that it is extremely general and can be applied to any timing-vulnerable structure (it is not specific to memory-based timing attacks).

2.5 Dissertation Contributions

This dissertation makes the following contributions:

- **Defining and describing speculative operational execution modes for TCAs** — We are the first to define and give detailed description for 4 different execution modes of tightly-coupled accelerators with various levels of speculative execution in relation to its neighboring non-acceleratable instructions.

- **Designing an analytical performance model for these TCA modes** — After defining each of these modes of execution, we create a first-order analytical model for quickly estimating the performance of the 4 execution modes using easily available parameters.

- **Validating our analytical performance model** — A model is only as useful as its ac-

curacy to various applications. We test various accelerator designs in both real-world and synthetic applications in a cycle-level simulator to compare our mathematical model to detailed simulation.

- **Describing takeaways and implications generated by the model** — Our validated model helps us to explore a large design space of possible accelerators and come away with high-level understanding and interesting conclusions regarding TCA design.

- **Designing multiple DGEMM TCAs** — We design and test a CPU dense general matrix-multiplication (DGEMM) TCA for various submatrix sizes (2×2, 4×4, and 8×8) roughly similar to NVIDIA's GPU Tensor Cores. We believe we are the first to show detailed evaluation between a μop (micro-op) memory-based and register file-based TCA CPU implementation.

- **Evaluating performance, energy, and programmability tradeoffs for memory-based vs. RF-based DGEMM TCAs** — We give detailed analysis of performance differences through a cycle-level simulator and energy analysis through RTL design and existing memory estimation tools. We also detail differences in programming for each design.

- **Proposing a fully reprogrammable TCAs** — We believe we are the first to propose having fully reprogrammable TCAs as complex functional units within existing out-of-order CPU architectures.

- **Evaluating two classes of security vulnerabilities with TCAs** — We describe two security challenges that arise: timing-based and speculative-based attacks.

- **Providing defense mechanisms against these TCA vulnerabilities** — We propose using clock-gating as a way to prevent hardware from monitoring time when latency can leak information. We propose limiting flops during the RTL design stage to modules where flush and commit signals can be verified during compile time to guarantee speculative recovery.

- **Designing various TCAs using secure invariants** — We build 4 different TCA designs using the security defenses proposed in the form of design invariants. This demonstrates that the security invariants do not hinder the feasibility of creating practical TCAs.

- **Evaluating overheads of TCA invariants** — We evaluate performance, power, and area overheads for each of our secure TCA designs.

2.6 Dissertation Organization

The overall organization of the dissertation is broken up into the following components:

- Chapter 1 from earlier in this document was included as a way to describe the contents of this dissertation's contributions in a way that is understandable to the general public.

- Chapter 2 provides an introduction to give context behind hardware accelerators and to describe the contributions and contents of this dissertation.

- Chapter 3 goes into the background surrounding each of the dissertation's contributions, as well as describing the existing related work.

- Chapter 4 describes various modes of concurrent execution possible for TCAs and provides details in the design and validation of our novel mathematical model to estimate TCA performance in each mode. This chapter also uses the model to provide insights and general understanding for different TCAs integrated into various architectures.

- Chapter 5 describes NoRF, our TCA execution model that passes operands through the memory hierarchy instead of the traditional means of a register file (RF). NoRF is numerically evaluated for area and power tradeoffs and verbally evaluated for programmability tradeoffs.

- Chapter 6 introduces the concept of a reprogrammable TCA and focuses on some of the security challenges present with that concept. It provides invariants to close timing- and speculative-based attacks currently present with the concept. Area, power, and performance tradeoffs are evaluated, and 4 example accelerators following those invariants are described and shown.

- Chapter 7 gives closing remarks and describes ideas and examples for future directions to continue this line of research.

2.7 Chapter Summary

In this chapter, we introduced some of the key concepts and ideas presented in this dissertation. We defined tightly-coupled accelerators and explained the steps required in proposing, evaluating, and implementing them. We then looked at TCA performance

modeling, and described how the analytical model presented in Chapter 4 helps reason through the complex design-space of implementing TCAs with various levels of concurrent execution in different architectures. We also described the tradeoffs between passing TCA operands through the existing general-purpose RF and memory hierarchy. We determined that several of the benefits of using the RF are hindered when applied to TCAs. We also looked at the potential timing-based and speculative-based security vulnerabilities caused by TCAs. We described the potential for modified FFs to force reversion upon a flush signal to help prevent speculative-based attacks. We also described how clock-gating, which is typically used for energy efficiency, can be used as a defense against timing-based attacks. This chapter also formally listed each of the contributions made in this dissertation and described the organization of this document.

3 BACKGROUND AND RELATED WORK

Before we can describe the new contributions provided in this dissertation, it is important to understand the context surrounding our proposals. One of the ways context can be provided is through exploring the related work over the past several years to see the progression of accelerator design and computer architecture. It is extremely valuable to be able to take elements of existing ideas and incorporate them to new contexts and domains. Machine learning is a perfect example how algorithmic ideas from several decades ago can now be modified, implemented, and mapped to today's current hardware. It is also important to show related work in order to point out the current limitations and research fronts yet to explore. This dissertation, as with all research, is simply placing a new series of novel ideas among the vast collection of ideas in a way to help turn ideas into reality.

Another way to provide context is to detail the computer architecture concepts surrounding TCAs and their integration into CPUs. This chapter provides both related work and architectural concepts as a way to present a more complete picture surrounding our contributions.

3.1 Hardware Accelerators and Reconfigurable Architectures

Prior work has identified key principles that drive the increasing popularity of specialized hardware accelerators. Nowatzki et al.[61] explains in detail the benefits that come from accelerators, as well as the characteristics of tasks that can greatly benefit from acceleration. The speedups of accelerators come from (1) concurrency specialization, (2) computa-

tion specialization, (3) communication specialization, (4) data reuse specialization, and (5) coordination specialization. Although their discussion focused on coarse-grained accelerators, the same principles are relevant for fine-grained acceleration.

Initial hardware accelerators (including GPUs and their related computation uses [60]) often targeted very large tasks, such as video encoding [44], graphics, image processing [7], encryption [41], physics modelling [4], etc. Then, loosely-coupled *sea of accelerators* [9] became a model to accelerate several different domains, which is now a widely adopted approach in commercial products, such as mobile phones. To maintain high performance gains for common workloads, there has been a trend towards domain-specific and even application-specific architectures/accelerators [13] [75]. Particularly in modern Systems on Chip (SoCs), it is common to see many of the coarse-grained tasks offloaded to hardware accelerators.

Naturally, as designers search for additional acceleration opportunities, they necessarily consider increasingly fine-grained software tasks. For this reason, we see tightly-coupled accelerator (TCA) proposals for heap management, hash maps, string functions, and other fine-grained tasks [49] [29]. This general trend has moved towards accelerating smaller algorithms and groups of instructions, and is increasingly moving into the realm of specialized functional units, similar to complex instructions in a CISC ISA. However, some of the differentiators of TCAs over generic CISC instructions include longer dataflow/dependency graphs for a sequence of operations, more memory operations per invocation, and/or a more complex control flow, resulting in a dynamic number of operations that may only be known at runtime. For example, a DGEMM TCA performs multiple memory operations and significant number of intermediate values passed internally for further calculation.

String accelerators might issue a dynamic number of memory references until it reaches a termination character. These are operations that add an extra level of complexity over typical CISC-like instructions.

Several workloads have been shown to be difficult to obtain speedup without accelerating fine-grained tasks [29]. TCAs are an emerging class of accelerators that show promise for increased performance in various domains run on CPU, such as machine learning, artificial intelligence, and server workloads [62, 49, 28]. Using a sea of accelerators or multiple accelerators can increase the code coverage at the cost of additional hardware resources. It may be impractical or impossible to incorporate dozens or hundreds of these TCAs close to the core without moving critical structures such as the L1 caches further away, but there is potentially large speedups available across different domain workloads if multiple TCAs can exist in the same core. Ideally, we would like TCAs to be specific to each application, as well as tightly-coupled for the most benefit.

Additionally, having TCAs that are invoked via an ISA instruction should likely be handled similar to the other instructions in a pipeline, namely speculatively with in-order commit. This also produces challenges for tracking data dependencies between TCA and non-TCA instructions, since TCAs have the potential to operate on more data than a typical instruction. Parakram [37] uses a software support approach to support scheduling between acceleratable tasks by using just-in-time (JIT) dynamic objects to signal dataflow. These objects and functions passed to an execution manager responsible for keeping track of dependencies and sending computation to available resources. This approach can work well, particularly for operations in which any JIT and invocation overheads do not consist a significant portion of the accelerator execution time. For this reason, this method works

primarily well for larger acceleratable tasks.

Outside of this method, treating TCA execution the same as other functional units provides potential hardware implementations which support various levels of speculative execution. Full OoO execution will require the most hardware support for speculative recovery and data forwarding, while fully sequential execution requires the smallest hardware support at the cost of CPU and TCA stalls and limitations in instruction level parallelism (ILP). However, it may be possible to try to extract higher ILP through larger effective window sizes. Continual flow pipelines [79] provide a look-ahead ability to create effectively large window sizes while keeping the register file and scheduler small. It provides the capability to move forward on long latency instructions and do bulk commits upon their completion by checkpointing state and making the scheduler and register file non-blocking to these instructions. Multiple stores in progress can use techniques similar to Speculative Versioning Caches [27]. These techniques can allow the CPU to allow runahead execution while a longer latency TCA instruction executes. These techniques are inherently more useful when TCAs are long latency operations. In the TCA designs we incorporate, we evaluate TCA instructions with latencies typically less than 10 cycles.

Reconfigurable execution architectures have been popular through the use of coarse-grained reconfigurable architectures, or CGRAs [59, 83]. Running program flows of CPU applications through CGRAs, or having CGRAs as part of the processor pipeline such as DySER [32] allows efficient execution that makes it possible for in-order execution CGRAs to outperform their OoO execution CPU counterparts. CGRAs are effective at accelerating tasks such as algorithms with streaming inputs or long sequences of repeating dataflow graphs. Tightly-coupled accelerators, alternatively, focus more on fine-grained tasks in

both regular and irregular workloads. Chimaera [93] provides an example of incorporating flexible tightly-coupled acceleration for in-order processors provided that the operation has a single register output and limited register input size.

Reconfigurable logic has also been proposed as an addition to the standard OoO processor pipeline to collect data on retired instructions and provide ability to modify branch predictions or insert instructions from reconfigurable logic into the core [55]. The natural progression of this line of work is to allow further levels of reconfigurability to interact with the OoO core, such as execution units, which we evaluate in this dissertation.

3.2 Performance Modeling

Performance models are an important tool for computer architects to use. The "Iron Law"[76] of processor performance breaks down the execution time of a program into three key components – cycles/second (frequency), instructions/cycle (IPC), and number of instructions. The latter two are often the components targeted in today's proposed microarchitectural changes. Accelerators reduce the number of instructions in the program, and TCAs attempt to execute concurrently with the CPU to not decrease IPC.

Amdahl's law [3] can be used for evaluating multicore systems either with homogeneous (symmetric) or heterogeneous compute [40]. In practice, the law can be applied to many situations, including the performance increases possible when accelerating a portion of an algorithm. These tools are invaluable to create quick estimations to determine how worthwhile it is to solve a given research problem. For example, the law may quickly determine that it is more beneficial to make small performance improvements for a large

portion of an algorithm rather than making substantial performance improvements for a small portion of an algorithm. This first order analysis can save researchers months and years of research only to see incremental improvements.

These models are extremely useful, but show limited scope in detail. More advance models can provide this extra level of detail, often at the cost of increased complexity and requiring additional input parameters. The reason simple models like Amdahl's law save so much time is because one of the most common alternatives is to use full detailed simulators such as gem5 [6]. These simulators are extremely beneficial to architects by being able to test designs without requiring the time, money, effort, and verification required to physically create custom silicon. They also allow for physically infeasible designs (such as infinite caches, perfect branch prediction, etc.) to be tested for limit studies. However, since these simulators estimate hardware through software, they run many orders of magnitude slower than on a native machine. This is because a single simulated instruction often requires thousands of native instructions to update the simulator's internal data structures for many hardware modules in each pipeline stage, update their related performance counters, and update internal CPU state.

So far, on one extreme we've discussed simple models that are fast but limited in detail. On the other extreme we've discussed cycle-level simulators that provide much more detail but are slow. Many other analytical and statistical models lie somewhere in-between. For example, Eeckhout et al. [16] demonstrates that by knowing the degree of reuse between register instances, lifetime of register instances, and other register traffic information within a program, programs can be effectively characterized and later used to quickly estimate sensitivity to microarchitectural variations without the use of detailed

simulation. Obtaining these values may involve in-house or special evaluation tools to collect these statistics about a given workload.

Likewise, collecting events that can be captured from performance counters such as branch misprediction rates, cache miss rates, as well as program knowledge of functional unit distribution and load independence (for estimating non-dependent overlapping cache misses) can be used to quickly estimate the design-space of how performance and hardware area will be impacted by microarchitectural changes [50].

Analytical models are also extremely useful in order to have fundamental understanding about bottlenecks in a system, and where architects should focus their attention. CPI stacks break down what structures instructions spend their cycles, which provide insights regarding the largest opportunities for performance increases. These can be difficult to capture in out of order processors, particularly due to the challenges of overlapping execution which often hides long-latency instructions [19].

Prior accelerators often focused on coarse-grained tasks, where pipeline drains and fills created negligible impact on overall speedup. For this reason, prior work that models accelerator speedup such as LogCA [2] naturally ignored these pipeline drains and fills. LogCA also assumed that the CPU was idle during accelerator execution, which similarly targets coarse-grained accelerators. While these assumptions were valid for loosely-coupled accelerators, the emergence of TCAs require a new look into accelerator modeling as described in Chapter 4.

The Gables Model for SoC accelerators [39] focuses mostly on hardware resource utilization, and estimates an accelerator's overall speedup given the architecture's configuration, memory bandwidth available to the accelerator, and computational complexity of the work

done by the accelerator per byte of data. This is a useful step to estimate an accelerator's speedup and could be used in early design stages or even in conjunction with our analytical model.

Models have also been used to estimate performance of programs through a couple key parameters of the application and the processor it runs on [18]. These parameters include branch mispredictions, cache misses, processor issue width, and window size. One of the main benefits of this sort of analysis allows architects to have fundamental understanding how changing these parameters will impact the program performance. For example, by changing the cache miss rate, the architect can estimate how much benefit comes from a larger or more efficient cache. Similarly, changes to a superscalar processor's width, branch predictor, pipeline stages, or ROB size can quickly be tested without the overheads of detailed simulation. These tools are vital to have fundamental understanding to determine estimated impacts without the need of slow detailed simulators.

One of the key contributions, in our opinion, this prior work contributes is the realization that while out-of-order (OoO) execution is extremely complex and hard to model due to the concurrent execution of both useful and non-useful instructions, the front-end of

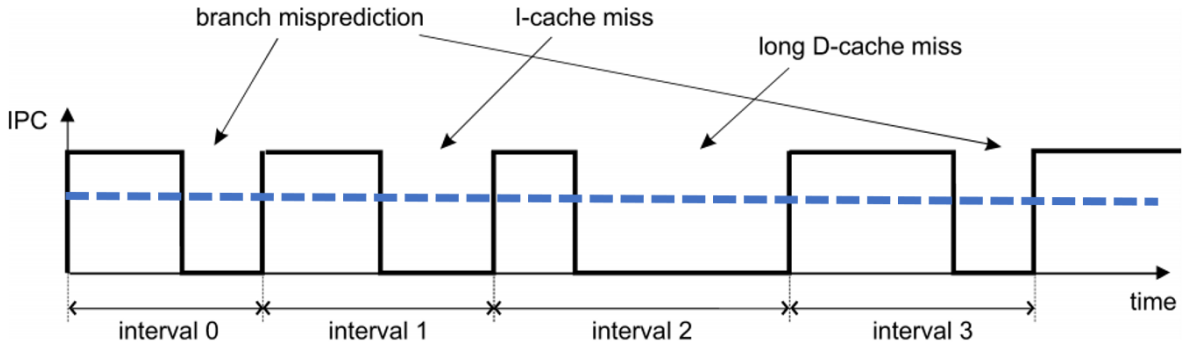


Figure 3.1: Mechanistic modeling from Eyerman et al. [18] showing front-end useful IPC dispatch width, overlayed with a blue dashed line showing average dispatch IPC.

the machine operates in-order and has a simpler view of effective IPC. In other words, the front-end will either be dispatching all correct-path instructions, or all incorrect-path (non-useful) instructions apart from the boundary conditions of a mispredicted branch. This allows an opportunity to evaluate an OoO machine much easier by looking at the effective performance of the front-end. Figure 3.1 graphs an example of the front-end's effective IPC over time. It becomes easy to see that it is very easy to integrate these graphs to determine duration of the program and penalties due to various processor events (branch misprediction, I-cache miss, and D-cache miss shown as examples). We recognized that our TCA could be incorporated into this model in order to estimate its performance in various modes of execution.

By building upon this model, we expanded its scope beyond superscalar processor performance to also include tightly-coupled accelerator performance modeling. If IPC is not known at design time, the architect can use Eyerman et al.'s model [18] to estimate performance and average IPC. However, we note that for most programs, IPC of a target

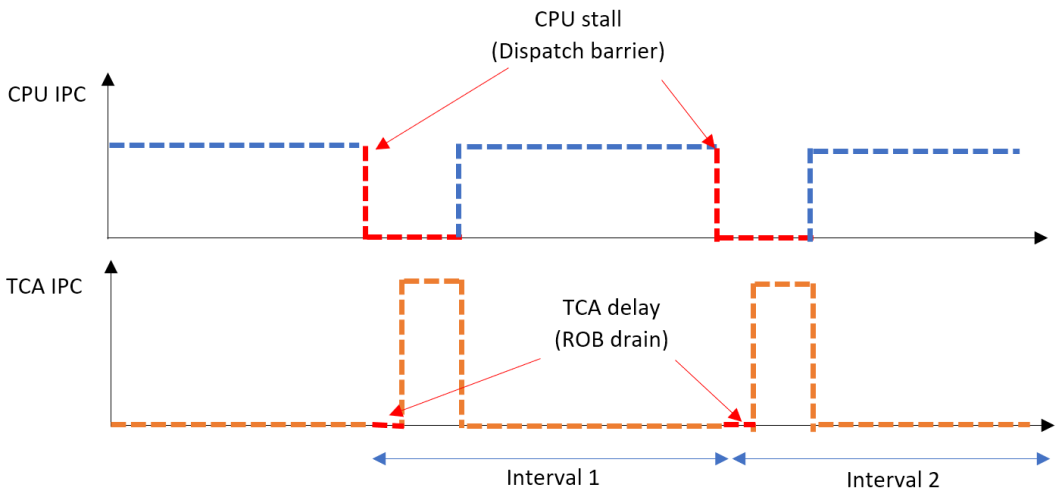


Figure 3.2: Adapted mechanistic model to show TCA and CPU effective IPC for front-end dispatch. CPU and/or TCA stalls can increase duration of stalled dispatch.

application is already known at accelerator design-time, creating an even simpler baseline to use our model. This can be captured either by running the algorithm on existing hardware or as a last resort running the target application in a simulator. Either way, in our model, we use the average IPC as the effective average IPC dispatch rate for the CPU. We note that the TCA stalls due to non-speculative execution and the CPU stalls due to dispatch barriers. TCA instructions can be effectively modeled as a new event added on top of the existing analytical model (see Figure 3.2). By performing similar analysis, we can estimate performance impacts of various TCA microarchitectural implementations without the need for detailed simulators.

3.3 Register Files vs. Memory Operations

ALU instructions perform computation through the use of operands. Operands can be delivered through two primary avenues: the memory hierarchy and through register files. Each has their own design tradeoffs. Register files are small structures that sit close to the core through its own "addressing." The register file consists of logical and physical registers, where logical registers are architecturally visible. Compilers are responsible for keeping track of variables and data kept in the architecturally visible register file. Physical registers provide the capability for successive instructions accessing the same logical registers to proceed without introducing false data dependencies. Register files offer fast access with deterministic execution bandwidth and latency. However, they are limited in size, requiring register spills and fills to be satisfied through memory when capacity conflicts occur. When reuse is possible, the register file can make many accesses to data without having to load

and store values to memory multiple times.

Alternatively, data can be passed directly from memory, bypassing the register file completely. This is typically done in loosely-coupled accelerators, since they do not have access to the processor's register file. Even if they had access, it is common for loosely-coupled accelerators to have their own internal scratchpads or buffers to hold intermediate results or reused values.

As a general rule, ALU instructions are typically executed through the use of the register file for various reasons. First, ALUs for RISC machines perform basic operations, from logical operations such as and, or, xor, etc. to basic algebraic computation such as compare, addition, subtraction, etc. Dataflow graphs for most algorithms will string multiple of these operations together to provide meaningful computation, allowing for multiple reuses of a variable prior to memory writeback. Since register files gain the most benefit with reused values and eliminating memory references, this is a natural conclusion.

In a similar concept to trying to reduce memory operands, certain architectures apply this technique one step further. The TRIPS processor [73] was designed as a way to explicitly pass operands directly between instructions without excessive reads and writes to the register file. This attempts to capture dataflow graph execution internally without adding contention to the register file for intermediate results. The E2 Explicit Data Graph Execution (EDGE) architecture [68] also demonstrates this phenomenon.

Similarly, the MANIC architecture [25] is motivated by the excessive writes of intermediate results and dead registers back into the register file. Their work describes the large energy savings possible by direct operand passing rather than through the register file. However, the MANIC architecture assumes that the ISA specifies vector operands using

register identifiers, hence requiring them to be loaded to and stored from entries in the RF. In other words, although MANIC reduces the loads and stores from intermediate results, all data from memory (even if they have short liveness in the RF) are still required to be loaded and stored at some point into the register file, and does not evaluate energy costs of completely bypassing the register file.

By the very nature of accelerators, the maximum amount of computation that can be done in the dataflow of an algorithm will be performed in a single hardware operation. For this reason, all internal data passing often exists within the accelerator itself. Reuse only becomes possible between accelerator invocations or when the accelerator is producing/consuming an operand for a neighboring instruction. As more of the algorithm is calculated internally within TCAs, the amount of reuse in outside structures (namely the RF) is limited. When no reuse is available, the RF acts as an inefficient buffer to load and store data that will immediately be spilled to memory.

General-purpose CPUs are designed to provide satisfactory performance across a wide range of domains and applications. Accelerators are often used to try to increase performance and/or energy efficiency for specific functions and applications, trading off generality for efficiency. However, delivering operands to the accelerator using existing CPU structures (such as a general-purpose register file) incurs multiple potential inefficiencies, and can limit the benefits of the specialization provided by the accelerator. Since these inefficiencies occur at every accelerator invocation, these penalties can add up when accelerators are frequently invoked. This is especially the case when data is frequently transferred between the CPU and accelerator, which is most likely to occur during fine-grained acceleration.

As an example, NVIDIA's Volta GPU design has incorporated specialized 4×4 matrix-matrix multiplication units and instructions into the hardware and ISA. The programmer can perform larger matrix-matrix multiplication by tiling the larger matrix into these 4×4 blocks, and issuing a long sequence of 4×4 matrix multiplication operations. This requires frequent passing of data between the accelerator execution units and the compute engine's storage (here, a GPU register file).

Although some of the specifics about the hardware have not been released, a few papers have attempted to reverse-engineer these GPUs to give estimations into the architecture of specific GPUs that support these instructions [47] [46]. These works provide analysis that predicts the data movement of submatrices into specialized registers that are then passed to the tensor cores. These complex functional units were designed to integrate well into the existing GPU architectures, which have very large register files and datapaths that integrate easily with register files. One of the questions which has yet to be explored prior to this dissertation is applying a similar functional unit to CPU architectures and evaluating whether or not a register file-based implementation makes sense with today's CPUs.

Operand	Year	Bits	RF	Memory
Burrows Scientific Processor	1982	48 (17 banks)	No	Yes
X86 SSE	2000	128	Yes	No
ARM Neon SIMD	2008	128	Yes	No
AVX-256	2011	256	Yes	No
AVX-512	2013	512	Yes	No
TensorCore 16FP	2017	256 (4x4x16)	Yes	No
TensorCore 64FP	2020	512 (4x2x64)	Yes	No
TCA:RF	N/A	512	Yes	No
TCA: NoRF	N/A	512	No	Yes

Figure 3.3: Evolution of adding wide vector operations.

We point out in Figure 3.3 that all recent specialized operand CPU instructions use a register file implementation to pass operands to wide ALUs. The most recent design we found in which we denote true memory-based computation without the use of register files is from 1982 with the Burrough's Scientific Processor [54]. SSE [70] and AVX [57] use wide registers in Intel CPUs to get additional ILP through the use of operating on multiple data through the use of a single instruction (SIMD). Similarly, ARM Neon has vector registers [71] for their SIMD operations. Despite requiring more specialized registers, these designs followed the common trend of using RFs for more specialized ALU computation. Although using wider registers, RISC computation is performed, allowing for possible reuse within these register files. As described above, for TCA instructions which are more CISC-like, reuse may be limited outside the TCA instruction. We present quantitative and qualitative evidence to claim it is often beneficial to remove the RF from the datapath for TCAs, which we denote as NoRF, for "No RF." This dissertation attempts to shift the focus for TCAs away from a long-lived tradition of using RFs for operand passing within the CPU. We focus on a DGEMM TCA for our primary example.

There have been works that look at DGEMM acceleration for various applications. Google's TPU [48] is a large-scale DNN accelerator for CNN/RNN inference that uses a systolic two-dimensional matrix-matrix multiplication array that operates on matrix multiplication orders of magnitude larger than 8×8 matrices. DaDianNao [10] is a customized chip that uses both custom compute and storage to gain large speedups over general-purpose CPUs and GPUs. EyeRiss [8] is a reconfigurable accelerator aimed at speeding up CNN inference computation (matrix multiplication) through optimized data movement through a row stationary processing dataflow on a spatial architecture. All of

these accelerators seek speedup through custom hardware that reduces the generality of a general-purpose CPU.

Compared to these accelerators, which often perform acceleration on large tasks, we look at how the architecture should change when performing more fine-grained tasks. Instead of offloading large matrices to the off-chip accelerators, the large matrix can be broken up into a series of smaller matrix operations that can be computed on-chip. This can add flexibility of operating on either small and large matrices efficiently, while also allowing parallelism by having multiple CPU cores computing smaller matrix operations of a large matrix concurrently. Both coarse- and fine-grained matrix operations can be beneficial for different types of use case applications.

Matrices with a significant number of zero elements contain significant resource overheads in memory storage, memory accesses, and computations when using the DGEMM algorithm. For higher efficiency and performance in computing operations for these matrices, it is common to perform a separate sparse matrix-multiply algorithm (SpGEMM). Sparse matrices are typically expressed in compressed sparse row (CSR) or similar format. This is used in order to only store data elements that are non-zero to save space. To do this, the format stores column information for each non-zero element, as well as a second vector used to specify how many non-zero elements are contained in each row. This greatly increases the efficiency of multiplication units (not multiplying zero values), at the cost of increased control flow complexity and irregular memory accesses. Most works of sparse matrix multiplication acceleration look at optimizing an algorithm of sparse matrix traversal [33] and/or sparse matrix encoding [22]. MatRaptor [80] is a SpGEMM near-memory accelerator gaining efficiency both through accelerator design and a novel

C^2SR data storage format. SAVE [26] is a vector engine that detects sparsity and skips unnecessary operations to improve performance and efficiency. In this work, we simply analyze whether or not our TCA (which is designed primarily for DGEMM) can also gain benefits in sparse applications. Our only algorithm change to the traditional CSR format is to turn non-zero elements into non-zero blocks of size 2×2, 4×4, or 8×8 depending on the TCA designed, creating a blocked CSR (bCSR) format that we traverse.

Regardless of application, both RF and NoRF designs are different in how they handle exceptions, interrupts, and context switches. RF designs are required to save state when servicing an interrupt or other means in which a context switch is required. The existing scalar and vector register files already perform this operation, while creating a specialized TCA register file will add extra state that must be saved and restored on context switches. Although relatively rare events, adding extra state for context switches can create undesired performance and energy overheads. Memory-based designs can handle interrupts and context switches like all other instructions by waiting for all in-flight instructions to commit without creating extra state to be saved/restored for correctness. This is because all data is accessed in a memory address space rather than a software-managed direct mapping register file address space.

TCA instructions that issue multiple μops per invocation may require special handling for exceptions in both RF and memory-based TCAs. The most common exception case for these TCAs are page faults. Since their occurrence should be rare, exceptions can be handled either through a software fallback routine (as proposed by Gope et al. [29]) or through TCA instruction replay as long as forward progress can be guaranteed.

3.4 Hardware Security

Having multiple workloads that can benefit from TCAs at an application and domain level pushes the demand for reconfigurability within the core. Recent work has proposed placing reconfigurable logic to interact with the innermost core pipeline [55]. By placing reconfigurable logic, the processor is able to have updates post-fabrication to test and develop new designs, optimize for certain applications, include custom branch predictors, as well as many other use cases. We build upon this idea to propose integrating reconfigurable logic within the execute stage of the processor pipeline. This reconfigurable logic can act as a TCA, where the logic can be optimized to execute commonly used tasks for the specific program. This addresses the two cases of the ideal TCA. 1) The TCA can be specific to each application, as the accelerator can be reconfigured for each application, leading to high utilization. 2) The TCA will be tightly-coupled, since all TCAs are loaded onto the same FGPA-style fabric and will be equally close to the core. Since unused TCAs can be replaced on the reconfigurable fabric, a new TCA does not have to compete for area with dozens or hundreds of other TCA hardware designs.

However, before implementing these accelerators in commercial products, the security implications must be evaluated. Placing arbitrary TCAs into the execute stage poses unique security challenges, since these accelerators sit so close to the core, which operates primarily on speculative instructions and data. In this dissertation, we will evaluate two classes security vulnerabilities that could be exploited through malicious or buggy TCAs. These attacks are timing-based attacks and speculative data extraction. We then evaluate the

fundamental root causes of these attacks and propose hardware mechanisms from the ground-up that are invulnerable to those attacks. One of the interesting discoveries that comes from creating rules for arbitrary hardware is the ability to apply those rules to existing proposed accelerators. We found that seemingly benign accelerators do not automatically follow all of these rules. By looking how the rules are broken, we can more easily detect potential vulnerabilities that we were likely not able to catch without our design rules. We then show how these trusted hardware mechanisms can be grouped together to create useful TCAs. We lastly demonstrate the set of rules that, when passed, demonstrate that the TCA is not vulnerable to these attacks.

Hardware security became an increasingly discussed topic after Spectre [51] and Meltdown [56] vulnerabilities were broadcasted to academic and industry researchers and engineers. These attacks were so impactful on the computer architecture community primarily because these attacks didn't exploit buggy hardware, but rather exploited fundamental hardware features that has been in virtually every processor made in the past several decades. Speculative execution has been one of the fundamental features that has allowed tremendous performance increases in processor microarchitectures. These new classes of attacks demonstrated that these features could potentially leave hints of wrong path execution (and potentially sensitive information) in virtually all hardware structures within the processor. While it's extremely easy to come up with a fix at an extremely large performance cost (preventing all speculative execution, eliminating a large portion of microarchitectural advancements in recent decades), researchers have taken the challenges of trying to close, or at least limit, these vulnerabilities without such a high performance penalty.

One such proposal is speculative taint tracking (STT) [94], a way to keep track of speculative state and to not forward speculative data to any instruction that can cause an implicit or explicit covert channel. MuonTrap [1] attempts to keep all speculative changes caused by a speculative memory access to be limited to a flushable non-inclusive filter cache at the L0 level. This filter cache keeps track of speculative changes and only updates the L1 and other CPU structures upon instruction commit.

The Spectre and Meltdown speculative-based vulnerabilities most commonly use timing-based attacks to extract information about flushed speculative instructions. Caches are the most commonly targeted for timing-based attacks. Prime + Probe, [67] Flush + Reload, [92] Flush + Flush, [35] and Evict + Reload [34] demonstrate ways to use the cache as a timing side-channel that can be successfully executed across many different processor microarchitectures. CEASER [69] provides a defense against Prime + Probe based attacks through encrypting cache accesses into seemingly random sets. SHARP [91] proposes modifying the clflush instruction to be restricted to privileged mode or user mode only with page write permissions as a defense against Flush + Reload attacks. These works highlight problems and potential solutions for cache-based timing-based attacks, but are often specific to caches.

Timing-based attacks can exist through a wide variety of microarchitectural structures [23]. Introducing noise into timing operations or reducing fine-grained timing precision have also been proposed to reduce the effectiveness of these attacks [43, 84]. If the CPU incorporates these solutions, having reconfigurable hardware could potentially bypass these security measures by using the reconfigurable hardware to capture cycle-accurate timing and/or use multiple attack iterations to predict sensitive information.

Security and safe/correct memory requests of reprogrammable and third-party accelerators is also not a new topic. The Xilinx ZYNQ [90] SoC allows for a fully reconfigurable FPGA to communicate with an ARM core through an Accelerator Coherency Port (ACP). Similarly, CAPI is an interface for accelerators such as FPGAs and reprogrammable logic to make coherent memory requests with the CPU [81]. Having these accelerators interact close to the core using CAPI has been shown to be a viable option for accelerating arithmetic operations [24]. Border Control provides an interface between physical memory and the accelerator to only allow memory accesses to memory locations the accelerator has permission to [65, 63]. These works focus on memory requests from loosely-coupled accelerators, whereas this dissertation focuses on security issues of memory requests from TCAs. Unique security challenges arise in TCAs because they interact closely with speculative data and have fine-grained interaction with existing CPU hardware, which we will describe in detail.

3.5 Chapter Summary

This chapter explores key context surrounding hardware accelerators, performance modeling, TCA operand delivery, and TCA security. Both related work and architectural concepts were presented to show a more complete picture surrounding our contributions. We built upon existing analytical performance models to apply them within the context of TCAs. We presented prior techniques to avoid data movement of intermediate results and showed how we apply those discussions to our proposals regarding TCA operand delivery. We discussed why TCA characteristics may require a different solution than conventional wisdom

for operand delivery within the CPU execute stage. We also presented recent hardware security vulnerabilities and showed that those vulnerabilities can also be applicable to TCAs.

4 MODELING TCA PERFORMANCE

As we describe in Section 2.1.2, the first steps in turning accelerators from design to silicon involves deciding the function or algorithm to accelerate and to estimate the expected impact. There are limitless numbers of functions that could be accelerated, and it is up to the architect to determine which accelerator(s) are best to target at any given time. However, without supporting tools, the architect has limited options. As a first option, they could try to reason through this selection process without formal analysis, potentially choosing a poor algorithm to accelerate. As a second option, the architect could spend large amounts of time to implement the accelerator in simulators. In this chapter, we provide an alternate option by designing an analytical model that can be used as an effective tool for estimating TCA performance. Since TCAs are a new and emerging class of accelerators, we believe we are the first to provide such a tool. Our analytical model can provide fast and easy estimates of TCA performance without the extensive work required to integrate into processor simulators.

4.1 Chapter Overview

As proposed accelerators target finer-grained chunks of computation and data movement, it becomes increasingly important to couple them tightly with the processor, avoiding long invocation delays.

However, the large implementation design space of these Tightly-Coupled Accelerators makes it difficult to balance tradeoffs between hardware complexity and accelerator

performance.

Previous performance models for accelerators focused on the penalties associated with loosely-coupled accelerators, which abstracted away many of the fine-grained interactions with complex out-of-order structures and program behaviors that have large impacts on TCA performance.

In this chapter, we introduce an analytical model that studies TCA behavior when interacting with the core, in the context of both high and low memory bandwidth applications supporting various levels of speculative and out of order (OoO) execution.

Our analytical model reduces the turnaround time in early design stages when estimating performance gains over detailed simulation with tolerable error. We also discuss potential design choices that can impede the benefits that come with TCAs, and illuminate differences with traditional accelerators.

This chapter encompasses the work as published in ISPASS 2020 [74] but provides additional details, analysis, and discussion that was not presented in the original publication.

This chapter addresses these principles, design choices, and tradeoffs in the context of

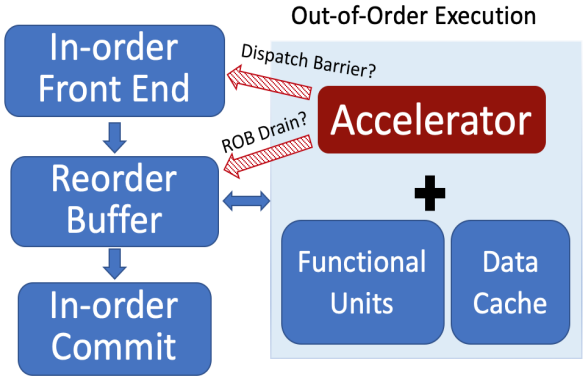


Figure 4.1: High-level system diagram of a TCA integrated into an OoO core. Our studies evaluate how requiring dispatch barriers or reorder buffer (ROB) drains can have significant performance impacts.

TCAs. As we defined in Chapter 2, a TCA is a hardware block that replaces a software function/routine, is invoked via a dedicated ISA instruction, reserves an entry in the reorder buffer (ROB), has in-order commit semantics, and integrates with the processor's register file and/or coherent memory hierarchy. Figure 4.1 shows a high-level diagram of an example TCA integrated into an OoO core.

4.2 Motivation

Prior TCA proposals often provide little discussion of how these accelerators should be integrated into cores, leaving a large design space to explore, with large differences in power, area, and even performance. For example, the single decision on whether or not the accelerator is allowed to execute speculatively has tradeoffs in performance, (by allowing concurrency and ILP around the TCA instruction), area, and power. The performance increase provided by speculative execution requires dedicated hardware in charge of rollback on misspeculation, as well as register and memory dependency resolution mechanisms.

One can imagine that when accelerating small chunks of code (fine-grained acceleration), failing to support speculation by forcing the processor to drain its ROB and/or stall dispatch every time the accelerator is invoked can significantly impact performance. The analytical model proposed in this dissertation captures this expected result, as shown in Figure 4.2. We insert parameters into the model based on an ARM A72 processor, assume 30% of code acceleratable, with an accelerator speedup of 3x. Markers for the granularity of H.264 [44], Google's TPU [48], GreenDroid [31], speech recognition using STTNI [77], heap management [29][49], regular expression [29], string function [29], and hash map

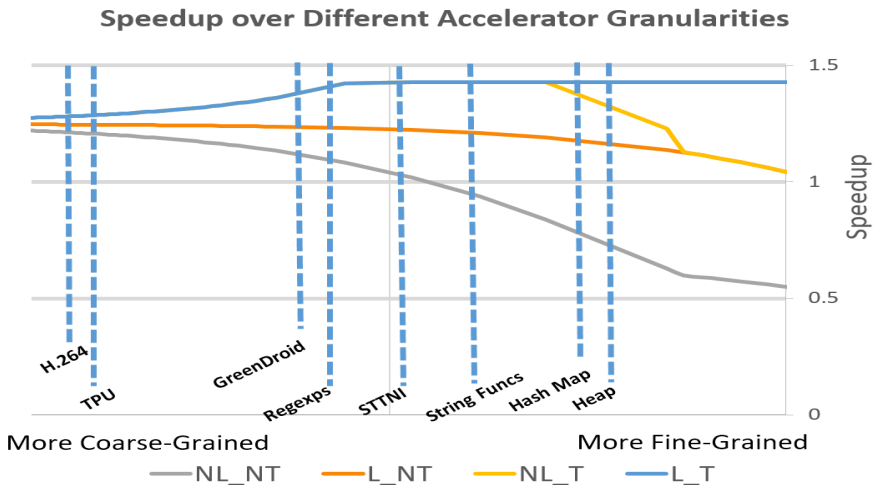


Figure 4.2: High-level analysis of program speedup for accelerators invoked at different granularities using the novel analytical model proposed in this dissertation.

[29] acceleration are estimated for points of reference.

Note that the TCA mode choice of full (L_T), partial (L_NT and NT_L), or no (NL_NT) support for concurrency between the accelerator and program execution has a larger impact on program speedup in **fine-grained** accelerators. For very **coarse-grained** accelerators, OoO support is less important for speedup, since accelerator latency is dominated by the execution time instead of the latencies associated with initialization and cleanup. For moderate granularity, full OoO support delivers better speedup than coarse accelerators, since the ROB experiences fewer stalls due to the still reasonably long latency of the accelerator, exposing more ILP for non-accelerated instructions. For fine-grained scenarios, inadequate OoO support (NL_NT) can lead to slowdowns. This result motivates development of analytical models that provide insight into these effects. Creating a high level model (discussed in detail in Section 4.3) allows the designer to numerically estimate these impacts and make informed design estimations as a first step prior to the laborious development of a detailed simulator.

Understanding the tradeoffs between each implementation will be an important step for choosing the optimal implementation based on the type of accelerator, processor architecture, program behavior, and desired metrics. However, to our knowledge, no prior research has numerically evaluated these tradeoffs, leaving architects to either try to intuitively predict the optimal implementation or spend a large amount of time to design and test each potential implementation. We create an analytical model to quantify these impacts. After validating the model over various workloads and accelerators, we discuss overarching trends in TCA design, and summarize key conclusions.

4.3 Analytical Model

When designing a TCA, the architect must decide whether or not the accelerator is allowed to reorder its execution with leading (L) and/or trailing (T) instructions. If the accelerator is allowed to execute concurrently with L instructions, this means that the accelerator is executing speculatively, since L may contain unresolved branches or instructions that will raise an exception. If the accelerator is allowed to execute concurrently with T instructions, trailing instructions must know whether or not they are dependent on the TCA's output(s) through dependency resolution hardware.

The result of allowing concurrency with L and T instructions is improved performance, but the downside is the complications of designing hardware to ensure program correctness. Papers have discussed the performance benefits that can come from tightly-coupled accelerators [29] [49], and it is often assumed that these accelerators allow full out of order execution with respect to the rest of the program.

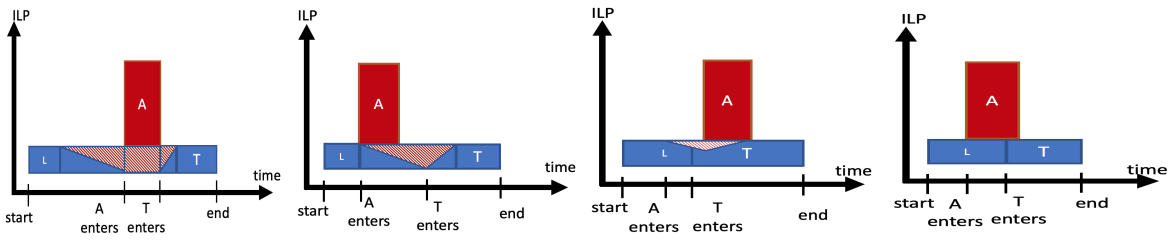


Figure 4.3: Example of effective ILP in the execute stage for a core with a TCA operating in four different modes. This figure shows an interval with leading (L) instructions, trailing (T) instructions, and a single accelerator (A) instruction. The striped sections show reduced ILP in the core caused by TCA mode.

Specifically, speedup is usually estimated by replacing the time spent within an acceleratable region with the accelerator execution time, which assumes the core can maintain its OoO execution rate around the accelerator. In this section, we describe the hardware design changes needed to operate in different modes of OoO execution, and the equations our analytical model uses to predict performance when operating in each mode.

As described in Section 3.2, one of the key insights of the work from Eyerman et al. [18] is that OoO performance modeling can be simplified by looking at the throughput of useful instructions dispatched in the (*in-order*) front-end of the processor. The front-end effective IPC resembles a square-wave, whereas trying to model OoO execution IPC results in more complex behavior and is extremely application dependent (Figure 4.3 shows a toy example in effective IPC from the OoO execute stage). Since the TCAs described in this chapter are invoked by inserting an instruction through the normal pipeline of an OoO core, one can view accelerator invocations as a new category of 'special events' that occur during program execution. Depending on the implementation of the TCA, it may act like a branch misprediction (zero useful dispatches until the accelerator instruction commits), typical long-latency instruction, or new type of event whose penalty is related

to the window drain and/or execution time of the TCA.

Our analytical model estimates overall performance changes using interval analysis of either an entire program or region of interest. For the rest of the chapter, we discuss this in the context of analyzing an entire program, but the process is identical for a specific region of interest. Interval analysis allows overall program execution to be estimated by the execution of the average interval. Invocation frequency (ν) is calculated by dividing the number of accelerator invocations by the number of total instructions. This variable represents the rate of acceleratable tasks in the baseline program. Regardless how these invocations are distributed throughout execution, the model predicts speedup assuming an even distribution of accelerator invocations. The model also assumes that that IPC of the core for non-acceleratable instructions is equal to the average program IPC before acceleration, and speculative accelerator instructions can begin execution as soon as they are dispatched. Note that this assumption may be overly optimistic for workloads with many dependencies between the acceleratable and non-acceleratable instructions. The average dispatch rate is also assumed to be equal to the IPC while the core is not stalled, and 0 when the core is stalled. The model also estimates the average non-acceleratable work to be evenly distributed across accelerator invocations.

Table 4.1 shows the parameters needed to use our analytical model. The red parameters (% acceleratable code and invocation frequency) denote software-specific parameters. These values are determined solely by the application or section of code the architect wishes to accelerate. The blue parameters (ROB size, issue width, and commit latency) denote hardware-specific parameters for the given target architecture the accelerator will be integrated with. The green IPC value denotes that it is a parameter that is dependent both

variable	name
a	% acceleratable code
v	invocation frequency
IPC	Instrctions / cycle
A	Acceleration factor
s_{ROB}	size of ROB
w_{issue}	issue width
t_{commit}	commit latency

Table 4.1: Analytical model parameters with software-specific parameters (red), hardware-specific parameters (blue), and hardware & software dependent parameters (green).

on the software algorithm and the hardware it is executed on. Since the acceleration factor is relative to the program's IPC, it could also be considered dependent on both the hardware and software. However, it's also possible to just use the proposed accelerator execution latency into our analytical model which may not be dependent on either the software algorithm or the hardware it is deployed on. This model allows for either acceleration factor or accelerator latency as an input parameter.

When integrating an accelerator into an OoO core, the simplest hardware design eliminates concurrent execution of instructions with the accelerator, while fully supporting OoO execution will lead to the most complicated design. Partially allowing OoO execution will fall somewhere in between.

The baseline assumes a full software implementation with no invocations to the accelerator. The time can be estimated by the number of instructions in the interval divided by the average program IPC, as shown in Equation (4.1).

$$t_{baseline} = \frac{\#inst}{IPC} = \frac{1}{v * IPC} \tag{4.1}$$

Graphically, Figure 4.4 shows this baseline case from the analytical model's point of

view. The CPU dispatches a constant rate of IPC instructions per cycle for both acceleratable and non-acceleratable instructions.

Figure 4.4: Frontend dispatch rate in the baseline case (acceleratable and non-acceleratable instructions are all dispatched to the core).

Accelerator execution time can either be an explicitly provided latency inserted by the architect, or estimated by the number of 'accelerated_instructions' divided by the accelerator effective IPC (A*IPC). This comes out to be:

$$t_{accl} = \frac{\#accl\_inst}{A * IPC} = \frac{a}{v * A * IPC}. \tag{4.2}$$

Similarly, we can estimate the execution time of non-accelerated instructions in the core by taking the total number of non-accelerated instructions and dividing it by the program's average IPC as shown in Equation (4.3).

$$t_{non\_accl} = \frac{1 - a}{v * IPC} \tag{4.3}$$

The rest of this section describes penalties specific to the four different TCA implementations and how they are added into our analytical model.

4.3.1 Non-Leading & Non-Trailing (NL_NT)

We call this mode Non-Leading & Non-Trailing (NL_NT), since this mode does **not** (N) allow TCA execution to overlap with leading (L) instructions or trailing (T) instructions. Specifically, a NL_NT TCA must wait until all leading instructions commit, and all trailing

instructions must stall in the front-end until the TCA commits. This simplifies the TCA hardware design, in that the designer knows the processor will never squash an in-flight TCA instruction. It does not have to checkpoint internal states, since it will never have to roll back. Similarly, no dependency checking hardware is required, since the accelerator is never executing simultaneously with other instructions.

For performance modeling purposes, as soon as the accelerator is dispatched into the issue queue, the dispatch stage will drop to zero useful instructions dispatched per cycle until the ROB window drains. From the front-end perspective, this causes a penalty for 'window_drain' (t_{drain}) time, plus the pipeline backend time to commit instructions (t_{commit}). Drain time is based on the latency of the critical path in a given ROB window size. Work from Eyerman et al. [18] shows a power-law relation between window size and critical path length for SPEC2006 benchmarks. Our model allows the user to manually adjust the window drain time if it is explicitly known for the target program, but by default will estimate t_{drain} by knowing the program IPC and using the power-law relationship to estimate the average critical path length for the given ROB size. If t_{non_accl} is smaller than t_{drain}, denoting shorter instruction execution in the core than the estimated drain time, t_{non_accl} is used instead.

After the window has drained, the accelerator begins execution, but the dispatch rate of trailing (T) instructions will still remain zero until the accelerator executes (t_{accl} cycles) and commits (for a second t_{commit} penalty).

This second barrier further simplifies the TCA hardware design, in that the designer does not need to implement memory or register renaming for accelerator outputs to eliminate false dependencies in OoO execution of younger instructions. It also does not require

any data forwarding or dependency checks with respect to the trailing (T) instructions. Graphically, from the front-end perspective, although the acceleratable code is offloaded from the CPU to the TCA, there is no concurrent execution between the TCA and CPU. The NL requirement causes the drain penalty, and the NT requirement causes a stall in CPU execution (Figure 4.5).

Figure 4.5: Frontend dispatch rate in the NL_NT case. Red denotes dispatch rates drop to 0 for either the CPU (top) or TCA (bottom) for stall (S) or draining (D) conditions. Commit penalties are considered part of the drain and stall penalties for graphical purposes.

Adding these stall penalties of length t_{accl} and t_{commit} with the drain penalties of t_{drain} and t_{commit} and results in a total NL_NT execution time of:

$$t_{NL\_NT} = t_{non\_accl} + t_{accl} + t_{drain} + 2 * t_{commit}. \tag{4.4}$$

4.3.2 Leading & Non-Trailing (L_NT)

We call this mode Leading & Non-Trailing (L_NT), since this mode allows TCA execution to overlap with leading instructions, but no trailing instructions can be dispatched until the TCA commits. Designing in this mode means that due to the very nature of speculative execution, it is possible that the processor squashes the TCA instruction during or after execution due to a branch misprediction or another similar event. The accelerator architect must guarantee any state changed by the accelerator will be reverted to its previous state if squashed.

Because trailing instructions cannot execute concurrently with the TCA, the front-end stalls, dispatching no useful instructions until the TCA commits. From the front-end perspective, no useful instructions are dispatched for $t_{accl} + t_{commit}$ cycles (Figure 4.6). Adding this penalty to the baseline gives us the total execution of the (L_NT) mode shown in Equation (4.5).

$$t_{L\_NT} = t_{non\_accl} + t_{accl} + t_{commit} \tag{4.5}$$

Figure 4.6: Frontend dispatch rate in the L_NT case. Although the TCA begins execution speculatively (no drain penalty), the CPU dispatch stalls until TCA completes execution.

4.3.3 Non-Leading & Trailing (NL_T)

We call this mode Non-Leading & Trailing (NL_T), since this mode does not allow TCA execution to overlap with leading instructions but does allow the trailing instructions to be dispatched immediately after the TCA instruction, even as it waits for older instructions to drain. Hence, non-dependent trailing instructions can execute before the TCA finishes, or even begins execution. This requires the architect to guarantee dependence checks for registers and memory locations that the TCA will modify. This dependency check can be done through modifications of existing structures such as the load/store queue (LSQ) and expanded register renaming and forwarding logic.

Our model predicts that the front-end can continue dispatching useful instructions until the ROB (potentially) fills up during accelerator execution (see Figure 4.7a). In the

previous NT cases, this condition would never occur since no instructions were added to the ROB after the TCA. However, we have to account for this potential condition, particularly for longer latency TCAs.

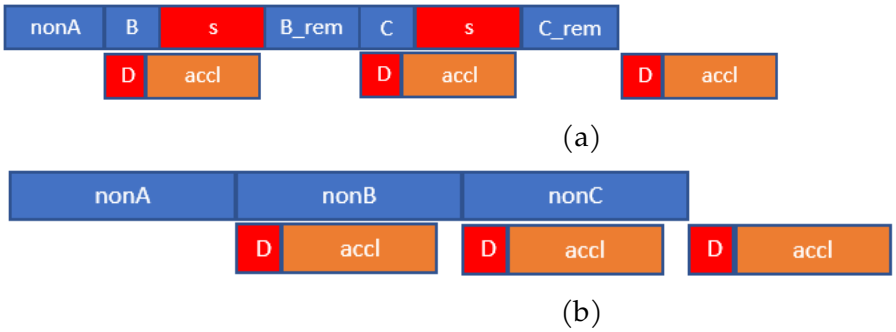


Figure 4.7: (a) NL_T case where TCA execution time is long enough to fill ROB, causing CPU front-end stall. (b) Typical NL_T case where TCA execution does not cause ROB full stall conditions. The TCA delays initial execution, but otherwise concurrently executes with the CPU.

Once TCA execution begins, the front-end can issue w_{issue} instructions per cycle until the ROB (potentially) becomes full. The accelerator will have a delayed execution start of t_{drain} while the ROB drains. If $t_{accl} + t_{drain} + t_{commit}$ is greater than the ROB fill time ($t_{ROB\_fill} = \frac{s_{ROB}}{w_{issue}}$), then the front-end will stall for every cycle after that until the TCA instruction commits. The time that the core will stall in this NL mode can then be estimated using Equation (4.6). If the ROB full condition is never met, our formula will count the stall penalty as 0, as dictated by the MAX function.

$$t_{NL\_ROB\_full} = MAX(\ 0, \quad t_{drain} + t_{accl} + t_{commit} - t_{ROB\_fill}) \tag{4.6}$$

The total execution in the NL_T mode will be the greater of the accelerator's (delayed) execution time or the core execution time with its penalties.

$$t_{NL\_T} = MAX(\ t_{non\_accl} + t_{NL\_ROB\_full},\ \ t_{accl} + t_{drain} + t_{commit}) \tag{4.7}$$

4.3.4 Leading & Trailing (L_T)

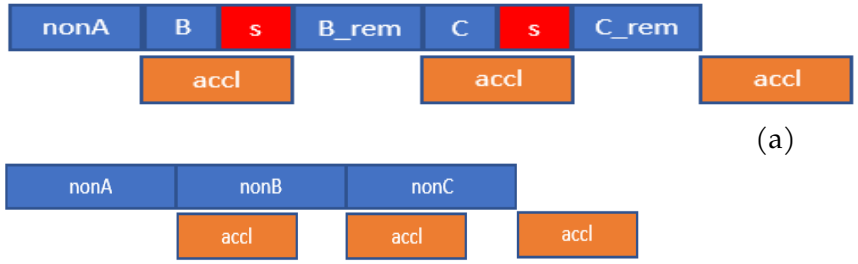


Figure 4.8: (a) L_T case where TCA execution time is long enough to fill ROB, causing CPU front-end stall. (b) Typical L_T case where TCA execution does not cause ROB full stall conditions. The TCA has full concurrency with the CPU.

We call this mode Leading & Trailing (L_T), as it allows both the TCA to execute speculatively, as well as trailing instructions to be dispatched while the TCA is executing. This mode will have the best performance, since under normal operation, the dispatch stage can continually dispatch useful instructions. In order to prevent any dispatch stalls, the core/TCA must be able to roll back on misspeculation, and the control logic must enforce register and memory dependences correctly between the TCA and both L and T instructions. From the front-end perspective, there is a continuous dispatch of instructions and there are no penalties unless the accelerator execution time is long enough to fill the entire ROB. However, unlike the case in NL_T, the accelerator begins execution immediately, and is not delayed for t_{drain} cycles (see Figure 4.8a) . Therefore, the ROB full time is slightly different:

$$t_{ROB\_full} = MAX(\ 0,\quad t_{accl} - t_{ROB\_fill}). \tag{4.8}$$

In this mode, the CPU's traditional execute units will have the highest utilization while the TCA is executing. The total execution time becomes:

$$t_{L\_T} = MAX(\ t_{non\_accl} + t_{ROB\_full},\quad t_{accl}). \tag{4.9}$$

4.3.5 Bringing It All Together

Analytical models that include variables and terms that are easily understood can be helpful from an intuitive standpoint. They can provide important general understanding by enabling limit studies and analysis of theoretical upper/lower bounds. They also can help provide early-stage design guidance and performance estimates without running detailed simulations. The key factors identified above regarding support for overlapping TCA execution with leading and/or trailing instructions can dramatically affect the overall speedup, but speedup is also highly dependent on the characteristics of the individual accelerator. We use the analytical model to show all estimated speedups for all four models.

The model attempts to capture the duration of front-end stalls dispatching into the ROB based on architecture, accelerator, and program properties for each of the four levels of speculation. By incorporating these TCA overheads in a mechanistic program model of the CPU front-end similar to [18], we use interval analysis to estimate program speedup. On average, the CPU issues roughly IPC useful instructions per cycle, but can drop to 0 when the TCA forces a ROB drain or dispatch barrier penalty as described in Sections

4.3.1 through 4.3.4 and summarized below. The overall speedup can then be estimated by comparing the execution time and penalties in all four accelerator modes to the baseline implementation.

- Reorder buffer drain (NL modes): The modes requiring the ROB drain will have an accelerator dispatch stall time equal to the window drain time. The window drain time can be explicitly entered into the formula or estimated based on ROB window size and program behavior as described in [18]. However, if T instructions are allowed to issue OoO, the core front-end does not see a drain penalty until after the ROB is full of T instructions before the accelerator commits.

- Dispatch stall (NT modes): The duration of the stall for modes that do not allow trailing instructions to execute concurrently with the accelerator is associated with the duration of the accelerator execution time. Early in the design cycle, this accelerator latency can be estimated, or it can be exact if the accelerator design is already well defined. Until the accelerator is done dispatching, executing, and committing, no further instructions can be dispatched.

4.4 Methodology

To validate our analytical model, we use the cycle-level simulator gem5 [6]. We test our model over a synthetic microbenchmark to measure error over many different accelerator/workload scenarios, followed by experiments with heap management and matrix multiplication microbenchmarks to test real-world proposed accelerators that have both

low (heap accelerator) and high (matrix accelerator) memory bandwidth requirements.

We replace acceleratable code of the compiled benchmarks with a special accelerator instruction. The NL drain and NT dispatch penalties are modeled by setting flags in gem5 for the accelerator, either to be a non-speculative instruction or to serialize the pipeline and not allow younger instructions to be dispatched until commit.[1] Accelerator instructions issue all memory requests needed required to execute, assuming ability to issue contiguous loads for sizes up to 64B (same width as an AVX-512 register). An accelerator instruction is not considered committed until all memory and compute μops of the accelerator have committed.

Our synthetic heap microbenchmarks do not contain dependencies between the acceleratable and non-acceleratable code, and the matrix-multiplication workload only has dependencies from memory loads/stores. As long as the processor has a large enough window, we assume in the analytical model that the core can continue executing useful instructions while dependencies are resolved (unless the ROB becomes full). The accelerator is assumed to have its own compute resources and does not need to arbitrate for functional units in the core. However, all memory requests required by the accelerator pass through arbitration for shared access to the core's LSQ and memory hierarchy. Priority is granted based on age (program order).

We first test our model with a sweep of microbenchmarks which varies over many different invocation frequencies and percentage of acceleratable code to validate over many different scenarios. The accelerator instructions (or acceleratable code in the baseline case) are randomly distributed within the program to see how our model performs while

[1] Heng Zhuo implemented these instructions with serialization flags within gem5.

violating our assumption of uniform TCA distribution. Each point in Figure 4.9 represents a separate workload instance with a differing number of acceleratable instructions replaced by accelerator invocations. This provides us several different workload scenarios to compare our analytical model against. After gaining confidence in the model over a variety of scenarios and accelerator designs, we then focus on both heap management and matrix-matrix multiplication accelerators to test more realistic TCAs.

Malloc and free calls in the TCMalloc library were evaluated to take about 39 and 20 cycles (69 and 37 x86 μops), respectively [28]. The proposed heap manager accelerator has single-cycle latency, which we assume in our model as well. Our heap microbenchmarks randomly perform malloc and free calls throughout the benchmark for the desired percentage of acceleratable code under the constraint that the heap accelerator will always have a pointer to return on malloc calls and always have an available entry for free calls (the common case).

As described in Chapter 3, matrix-matrix multiplication accelerators are already incorporated into NVIDIA's Tesla and Turing GPUs through their tensor cores. It is likely they load matrices into the large GPU register files through wide memory loads and perform computation through register file operands.

Our tested CPU matrix multiplication TCAs are modeled similarly but with a few differences. Instead of operating through the register file, our implementation of matrix multiplication operates through memory loads and stores, since current CPUs do not have dedicated 4×4 matrix registers. This allows us to simulate matrix multiplication acceleration without making significant ISA and microarchitectural changes. By making the requests through memory rather than registers, we can easily adjust our instruction

to be able to accelerate 2×2, 4×4, and 8×8 matrix multiplication sizes with minor architectural changes in the core apart from the TCA itself. This allows us to test our model over various accelerator designs.

However, one downside to this approach is that Volta's implementation of register-based acceleration will allow easier reuse of the output matrix for accumulation, while our implementation of outputs require redundant loads and stores through memory. Additionally, our implementation has separate load instructions for each matrix row, as opposed to the register-based separate submatrix storage which can be loaded in a single instruction. However, Volta's registers appear to be filled through multiple loads prior to invoking the matrix multiply [11], incurring a similar load latency to ours.

We created a software harness to calculate a 512×512 double-precision matrix-matrix multiplication. A naive implementation would do the entire multiplication in one triply-nested loop. However, since the two input matrices and output matrix have a total memory footprint of 6MB, much larger than the L1 D-cache (typically on the order of 32kB), there would be significant thrashing in the L1 cache that would limit temporal reuse. To optimize the algorithm to a L1 D-cache of 32kB, we calculate the 512×512 double-precision floating point matrix multiplication through 32×32 submatrix tiling. The two input and output submatrices now only require 24kB, and all accesses to elements of these matrices should hit in the L1 D-cache after the first access. We perform our multiplication using an output-stationary algorithm. Our performance results are based on the time it takes to calculate the full 512×512 output matrix.

Within the 32×32 matrix multiplication operations, we implemented an element-wise software kernel (our baseline), as well as accelerators that can directly multiply-accumulate

2×2, 4×4, and 8×8 submatrices. These accelerators all request memory addresses for the cache lines that they will need to access to do their computation. Within each of the accelerators, we also model the 4 different TCA implementations (NL_NT, NL_T, L_NT, and L_T).

4.5 Model Validation/Verification

4.5.1 Adaptive Microbenchmark

We first validate our model through an adaptive microbenchmark that varies program and accelerator parameters. As we increase the number of accelerator instructions, we are increasing both the invocation frequency and the percent of acceleratable code. By randomly placing the accelerator instructions within the baseline microbenchmark, the compiler creates slightly different optimizations, slightly varying the program's base IPC

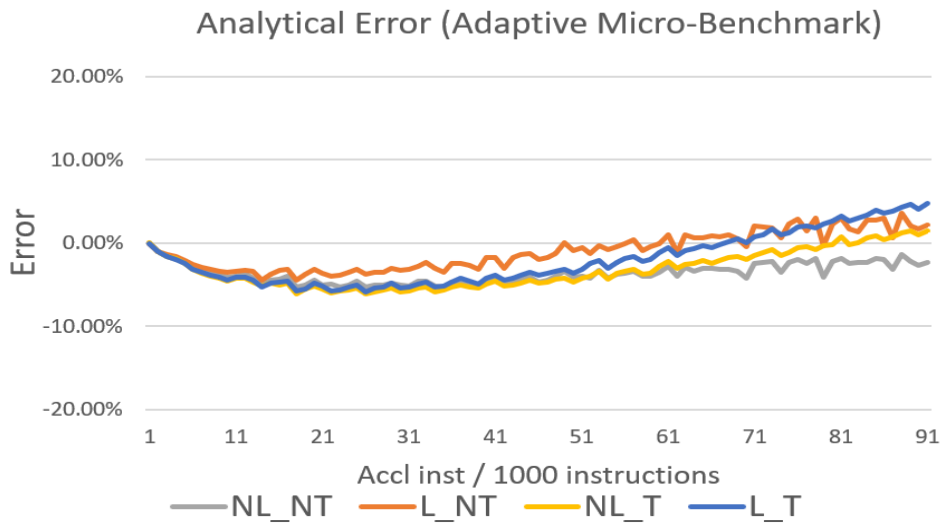
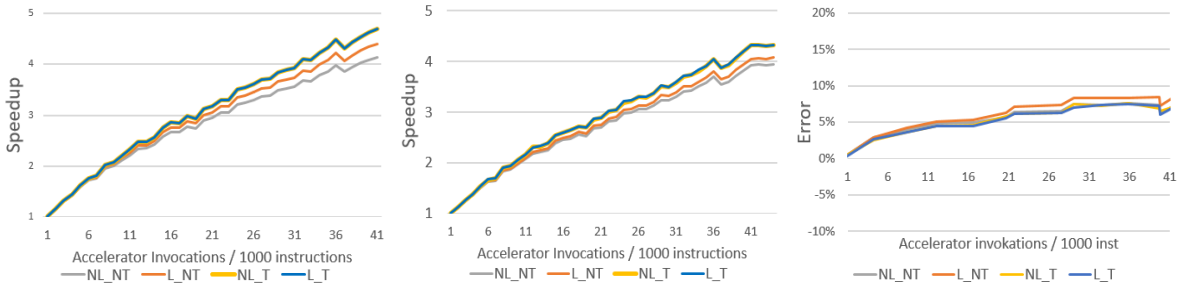


Figure 4.9: Error of our analytical model speedup prediction compared to gem5 simulation of a synthetic microbenchmark while varying the number of accelerator instructions (increasing invocation frequency and % acceleratable code).

in each resulting microbenchmark. This also helps compare the simulated versus analytical speedup when the benchmark does not match the model's first-order assumption of evenly distributed accelerator instructions. We end up testing over a wide variety of program behaviors with both low and high frequency of accelerator invocation and determine that our analytical model typically has less than 5% error (see Figure 4.9), giving us confidence to test real-world applications as well.

4.5.2 Heap Manager TCA

We then applied this same method to current accelerator proposals, starting with the heap management accelerator. This accelerator contains hardware tables to store a subset of the free lists tracked by the the TCMalloc library, and provides accelerated (single-cycle) calls to both malloc and free. We built a microbenchmark that allocated from one of 4 different class sizes (0-32B, 33-64B, 65-96B, or 97-128B). The baseline executes the TCMalloc operations by invoking software library calls to the heap manager. For the accelerator, since



(a) Analytical model speedup (b) Gem5 simulation speedup (c) Error of analytical model

Figure 4.10: Error of our analytical model in predicting accelerator speedup of heap microbenchmarks with different frequency of malloc/free function calls, each with 4 different TCA implementations. The NL_T line closely follows L_T. The highest error occurs at high invocation frequency, but still remains within a 10% error for a vast sweep of TCA invocation frequencies.

the overwhelming majority of requests hit in the accelerator, we assume that the accelerator will never have to fall back to the software subroutine. By inserting accelerator instructions in place of these function calls, we see increasing speedup as the frequency of malloc and free requests increase. Although our model has slightly higher error at higher invocation frequencies (up to 8.5% as seen in Figure 4.10), we can still observe the same general trends of the effect that each of the 4 TCA modes has on overall program execution. This shows that our model still correctly predicts overarching trends for this workload.

4.5.3 Matrix-Matrix Multiplication TCAs

Since many machine learning and artificial intelligence applications involve matrix-matrix multiplication, we decided to further test our analytical model with different hardware accelerators with different size matrix-matrix operations.

As we can see from Figure 4.11, our analytical model captures the general trends for all 4 TCA implementations. However, we see that the analytical model is slightly pessimistic

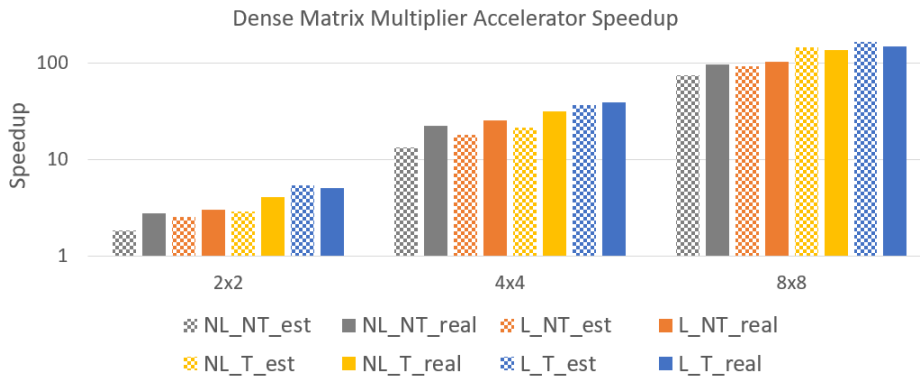


Figure 4.11: Speedup of a 512 × 512 dense matrix multiplication (blocked in 32 × 32 submatrix tiles) through various TCAs relative to software element-wise multiplication. TCA speedup was calculated through both gem5 simulation ('real') and our analytical model formulas ('est'). Note the same general trends of the measured vs estimated speedup of the 4 different TCA implementations for the 2 × 2, 4 × 4, and 8 × 8 DGEMM accelerators.

for the non-L_T modes. Our error reaches as high as 44%. Although such errors seem rather high, they are being amplified by the very large speedups in these scenarios, and our model still accurately predicts the right relative trends, enabling valuable insights to accelerator design without the need for highly accurate absolute speedups. We also realize that the highest errors come from NL modes due to high estimations for drain penalties. The power law estimates higher critical path length than the DGEMM algorithm incorporates. Manually adjusting critical path length helps significantly reduce this error (down to an average absolute error of 14%). However, we show the results including the higher error since this algorithm analysis may not be available to a researcher in early-stage analysis.

As we consider the overall execution time of the 3 different accelerator designs and 4 different TCA implementations, note that there is a larger absolute difference in execution time between the 4 different modes of the 2×2 accelerator than the 4×4 and 8×8 accelerators. This is because the overall program speedup is much smaller in the 2×2 TCA case, increasing the absolute penalty. By the time we get to the 4×4 and 8×8 submatrix multiply-accumulate cases, the larger speedup from the accelerator amortizes the cost of the drain and barrier penalties.

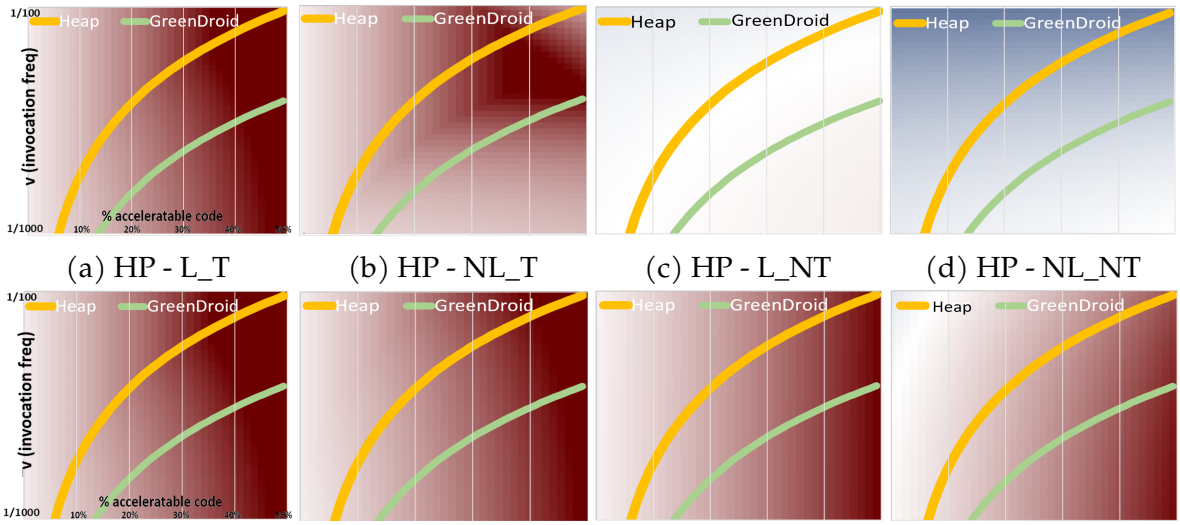
4.6 Analytical Model Use Case

We want to use our analytical model to gain valuable insights into both existing and previously proposed fine-grained accelerators. In this section, we apply our model to the heap manager accelerator and to GreenDroid functions. GreenDroid [31] proposes

mapping common functions in Mobile SOC workloads to TCAs with shared access to the L1 D-cache. This provides a use case of relatively fine-grained acceleration (hundreds of instructions) where we can use our analytical model to gain further insights.

The GreenDroid work shows the percentage of dynamic code execution that is run inside the given function, as well as the static number of instructions in that accelerator invocation. However, the percentage of dynamic code execution increases both when (a) the function contains loops, increasing the number of dynamic instructions for each invocation, and (b) when the function is called more frequently. If the function has no loops and executes straight through, the function will have the largest invocation frequency. However, if the functions iterate many times per invocation, the accelerator will be called less often. In this analysis, we consider only the 9 functions described in the GreenDroid literature[31] and assume straight-through execution of the functions for the highest invocation frequency case.

We apply our first-order analytical model varying 2 parameters (invocation frequency and percent acceleratable code) to create a 2D heatmap shown in Figure 4.12. Red locations represent program speedup, blue areas represent program slowdown, and darker shades represent higher magnitudes of speedup/slowdown. We test over each of the 4 modes, as well as using processor characteristics from both a high-performance core (1.8 IPC, 256 entry ROB, 4-issue) and low-performance core (0.5 IPC, 64 entry ROB, 2-issue). We use the number of instructions in software heap management routines and estimate GreenDroid functions based on available documentation to map where they lie on the 2D map. Each point on the curve represents the invocation frequency required to achieve a certain percent of acceleratable code. For example, a fixed-function accelerator will have to be invoked



(a) HP - L_T (b) HP - NL_T (c) HP - L_NT (d) HP - NL_NT

(e) LP - L_T (f) LP - NL_T (g) LP - L_NT (h) LP - NL_NT

Figure 4.12: Heatmap of speedups (red) and slowdown (blue) when sweeping over percent acceleratable code and invocation frequency (logarithmic scale). We map the locations heap manager and GreenDroid accelerators would fall over workloads with different % acceleratable code. Figures (a)-(d) show a high-performance (HP) OoO core, and (e)-(h) show a low-performance (LP) OoO core. The 4 columns from left to right are the TCAs operating in modes L_T, NL_T, L_NT, and NL_NT.

more frequently to obtain greater coverage (percent acceleratable code). Since GreenDroid is motivated by energy efficiency rather than performance, we assume a much lower acceleration factor of 1.5× compared to the high-performance core.

The curve for the heap accelerator is shown for reference of a more fine-grained accelerator than the GreenDroid functions. Using the analysis from our model, we make the following observations:

4.6.1 High Performance Core vs. Low Performance Core

The comparison between the first row and second row of Figure 4.12 tells us that high-performance cores are more sensitive to different modes of TCA. The first reason is that fixed-function accelerators will have higher relative acceleration factors on low-performance

cores due to the slower baseline execution time. Additionally, penalties arising from not allowing OoO execution are relatively higher on a high-performance core due to the larger ROB size and drain penalties. Thus, architects should pay close attention to OoO integration when designing TCAs for high-performance cores. For low-performance cores, the impact on OoO integration is less severe.

4.6.2 Fine-grained vs. Less Fine-grained

Fine-grained accelerators need to be careful about slowdown, especially for NT modes. Although both accelerators are fine-grained compared to conventional accelerators, GreenDroid is less fine-grained than the heap manager. Consequently, for GreenDroid acceleration, the TCA implementation mode is slightly less important, as the plots never cross into the slowdown range for these input parameters into our model. However, the heap manager is fine-grained enough that if it only achieved an acceleration factor of 1.5$\times$ in a HP core, it experiences slowdown when operating in the NT modes.

4.6.3 Limits of the Model

Because it is a first-order analytical model, many details are omitted and these have the potential to mislead a designer. For example, from comparing L_T and NL_T mode, one may draw conclusion to not implement the L_T design, since it leads to a more complicated hardware design with little performance gain. Here, the penalties from non-speculative execution of TCA instruction are visually similar, since the core can usually cover the overheads with other OoO execution. This may be because of our assumption made in

Section 4.4 that IPC remains constant when the core is not stalled, regardless of dependencies. When there are many dependencies between the core and accelerator, the accelerator has a longer latency before it can begin execution, and core IPC will likely drop until the dependencies are satisfied when the accelerator completes execution. This could create a difference in practical use depending on the actual accelerator and workload. When using malloc/free, explicit dependencies between instructions could lead to noticeable slowdown in the absence of speculation support for TCA invocations, e.g., when younger instructions wait for a malloc pointer. This is a shortcoming of our simple, fast, and easy first-order model, and represents a very typical tradeoff between a model's level of abstraction and its ability to capture every aspect of execution. Our model *generally* makes a reasonable prediction, but it is important to understand scenarios where it may fall short.

4.7 Discussion

Accelerators typically have two primary functions: to increase performance and/or increase energy efficiency over the general-purpose core. For the case of the primarily energy efficiency-motivated accelerators, the overall speedup may not seem important. However, even in this case, our model can detect the areas in which non-L_T modes start creating program slowdowns. It is important for these accelerator designers to make sure these points are avoided for both performance and energy reasons. Program slowdown requires the core to run longer, increasing the amount of static energy consumed by the core, eroding the energy gains created by the accelerator.

For accelerators motivated by speedup, our analytical model quantifies costs associated

with partial or full elimination of OoO execution. From this model, we can see that the higher the invocation frequency of the accelerator and the smaller the region of acceleratable code, the larger the relative penalty that arises from not supporting full OoO execution.

4.7.1 TCAs in Various Processors

We also learn from our model that high-performance cores have the largest discrepancies between the 4 different modes. This is exacerbated for several reasons. First, the drain penalties are larger because ROB size is bigger. Second, the pipeline depth is longer in high-performance cores, further increasing the drain penalties. Lastly, barrier penalties are relatively higher because a fixed-latency accelerator has less relative acceleration in a high-performance core than for a low-performance core, making an effectively longer barrier penalty. Similarly, the overall speedup factor is higher for low-performance cores. This means that designing accelerators for low-performance cores, perhaps for energy purposes, designers may decide to forgo accelerator complexity of L_T implementation with a relatively small impact on performance. This reduction of complexity will also further decrease power consumption of the accelerator.

4.7.2 Maximizing Concurrency

Our model also helps us come to an interesting conclusion about a new form of concurrency that arises from supporting full OoO execution (L_T mode) for TCAs. We can see from Figure 4.13 that a TCA with an acceleration factor of $A = 2$ can give the program a speedup of 3. OoO execution allows concurrency between the core and accelerator executing at

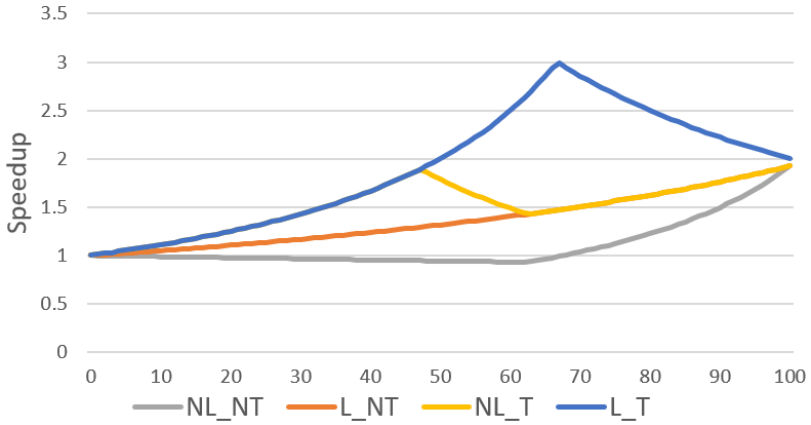


Figure 4.13: Analytical model predicted speedup for a TCA of 100 instructions with a speedup factor of 2. ROB size is 352. Note that the maximum speedup does not occur at 100% acceleratable code, since concurrency exists between the core and TCA in OoO modes.

the same time. This means that the maximum obtainable speedup when introducing a TCA is not just A, but actually $A + 1$. The peak overall speedup of 3 occurs when 67% of code is acceleratable. This is because for an accelerator with $A = 2$, work is evenly distributed between the TCA and the core when the accelerator has $2\times$ more work to execute than the core. For $A = 5$, the peak occurs when $5\times$ more work is offloaded to the accelerator, or $\frac{5}{6}$ of the workload. Beyond this point, offloading more work to the accelerator diminishes performance, as the core becomes underutilized. Our model also shows maximum concurrency cannot be obtained when introducing dispatch stalls, non-speculative accelerator execution, and/or when the ROB fills up. As a side note, even in these cases, our model can still estimate where the highest possible concurrency is reached for the given input parameters. We also see interesting behavior in the NL_T mode, where a speedup local maximum is reached at a lower % acceleratable code, since the concurrency is maximized when the TCA latency plus delayed start (t_{drain}) is equal to core execution time. After this point, the TCA execution time plus delay determines overall speedup.

At first, offloading more work to the TCA slows down execution due to decreased CPU utilization, but eventually the TCA stall time decreases when enough code is removed from the CPU pipeline to reduce drain time (t_{drain} approaches 0 as the program approaches 100% acceleratable code).

4.7.3 TCA Granularity

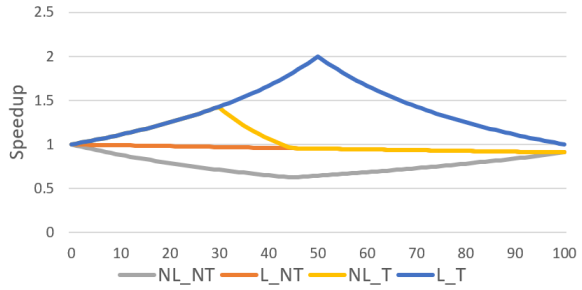
For the following comments on discussion, we insert the following values into our analytical model unless otherwise stated:

variable	value
IPC	1.5
s_{ROB}	256
w_{issue}	4 inst/cycle
t_{commit}	1 cycle

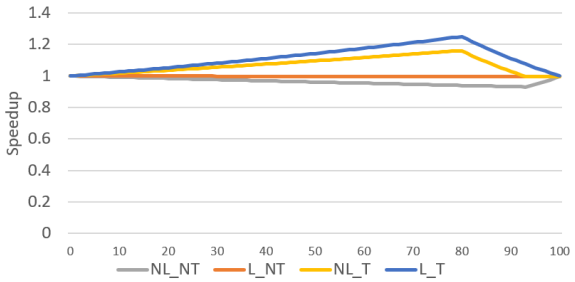
Table 4.2: Analytical model parameter values used in generating graphs for further discussion.

We expand this analysis to different acceleratable functions and different TCA designs. Figure 4.14 shows relatively coarse- and fine-grained functions being accelerated by a TCA with relatively low acceleration factors ($1\times$ and $2\times$). When comparing Figure 4.14a and Figure 4.14b, both of which have $1\times$ TCA speedups, we see some interesting differences.

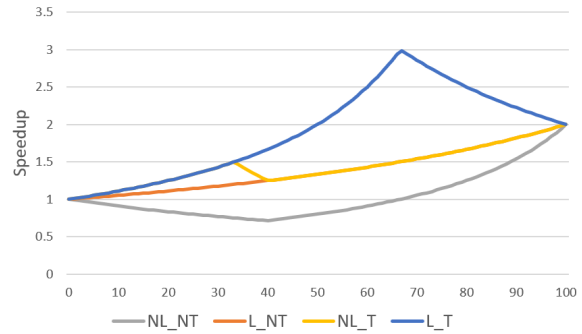
First, for the coarse-grained function TCA (Figure 4.14b) does not achieve a speedup at $2\times$ or its peak value at 50% acceleratable code, as might be expected. This is because having long latency TCAs create ROB stalls. If the ROB fills up in 64 cycles, a 400 instruction TCA executing at 1.5 IPC may only allow 24% utilization of the CPU. However, since the TCA is able to achieve practically 100% utilization, it is worthwhile to offload more work to the TCA until eventually the core only has enough instructions to do during the ROB fill time.



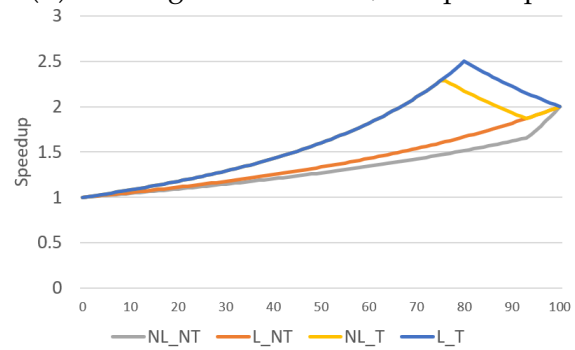
(a) fine-grained 20 inst, 1× speedup



(b) coarse-grained 400 inst, 1× speedup



(c) fine-grained 20 inst, 2× speedup



(d) coarse-grained 400 inst, 2× speedup

Figure 4.14: Speedup graphs for 4 modes of execution for accelerators targeting fine-grain 20 instructions – (a) and (c)), and coarse-grain (400 instructions – (b) and (d)) functions with small speedups (1× and 2×).

After this point, offloading more work to the TCA will decrease CPU utilization further. The fine-grained function (4.14a) does not see this trend because the ROB will never fill up due to TCA execution.

We also see that the NL case case has higher penalties in the fine-grained (4.14a) function. This makes sense, because the overall latency of delaying execution has a higher impact for low latency TCA instructions. The maximum concurrency comes at a much lower % of acceleratable code, due to the fixed cost latency that has to be included as part of the concurrency equation in TCA execution time.

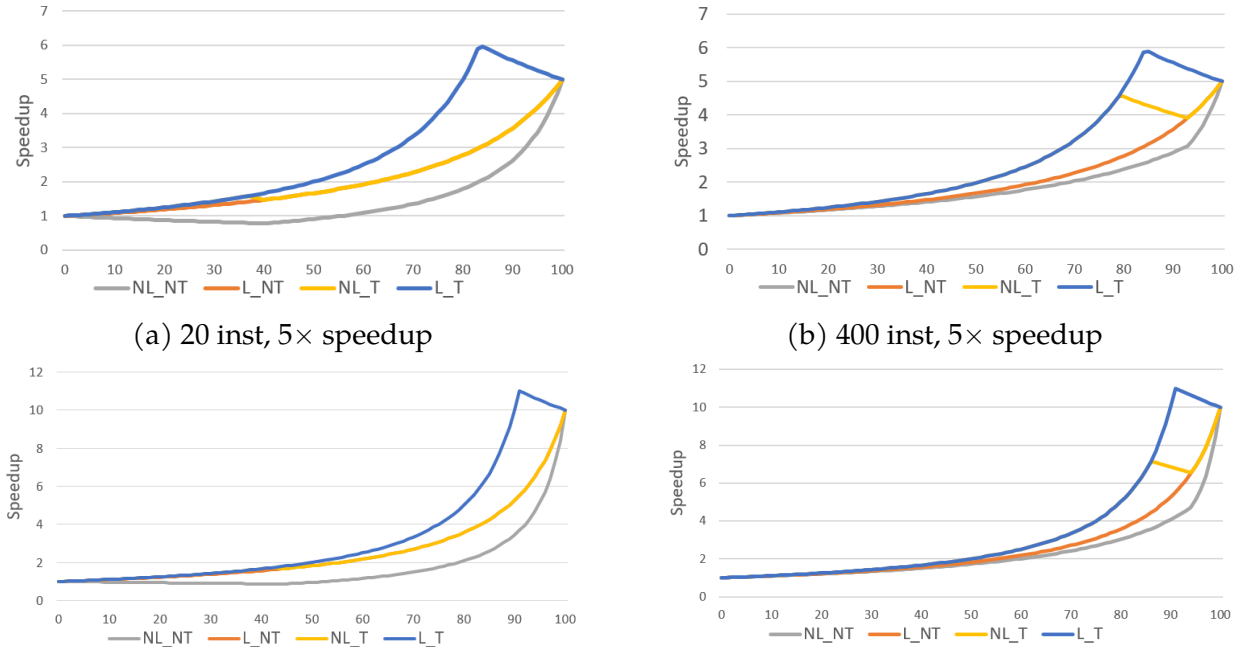
It is interesting to note that the NL_NT mode within the TCA can show significant performance slowdowns over the baseline. Using the TCA in this way will cause the CPU

to have large portions of time where there is no TCA or CPU execution happening. Since the TCA does not provide benefits over the CPU, it is impossible in the 1× scenario (4.14a and 4.14b) for NL_NT to ever provide speedup.

Moving across to the 2× speedup accelerators (4.14c and 4.14d), we see similar trends. The coarse-grained function (4.14d) still does not reach maximum concurrency due to the long-latency TCA instruction. However, there is overall higher utilization compared to the 1× TCA since the stall caused by the full ROB condition is reduced significantly. We also see that the NL_NT mode is able to achieve speedup, since the TCA speedup factor is eventually enough to make up for stalls caused by delayed execution. However, there is still a significant gap between this mode and L_T with full OoO execution. We also see the fine-grain accelerator (4.14c) still sees slowdown in the NL_NT mode and is much more impacted than the coarse-grain function equivalent.

We learn from the analytical model that coarse-grained accelerators, such as the 8×8 DGEMM TCA, are less sensitive to the execution mode. More fine-grained accelerators, such as 2×2 DGEMM and heap manager TCAs, have increased performance sensitivity and motivate tightly-coupled OoO execution.

Figure 4.15 shows the same graphs but for TCAs with larger speedups. We note that with 5× and 10× speedup, the TCA execution time is short enough in both cases to eliminate all CPU ROB stalls due to its completion prior to the ROB full condition. Overall, the higher speedups also have less potential to cause application slowdown. However, we note that even with high TCA speedup, it is still possible for non-OoO execution to cause substantially different results. For example, for the fine-grained function with a 10× speedup TCA (4.15c), our model predicts no speedup for NL_NT at 50% acceleratable



(a) 20 inst, 5× speedup

(b) 400 inst, 5× speedup

(c) 20 inst, 10× speedup

(d) 400 inst, 10× speedup

Figure 4.15: Speedup graphs for 4 modes of execution for accelerators targeting fine-grain (20 instructions – (a) and (c)), and coarse-grain (400 instructions – (b) and (d)) functions with larger speedups (5× and 10×).

code due to the number of invocation penalties, while the L_T mode gains close to 2× speedup as predicted by Amdahl's law.

4.8 Chapter Summary

In this chapter, we describe and validate a model that estimates TCA performance with various levels of OoO execution. The model shows that for increasingly fine-grained accelerators, allowing OoO execution around the accelerator can have significant impacts on overall performance.

Through this model, we also evaluate likely candidates for TCAs, and calculate different speedups based on implementation. Even with very different workloads, ranging from

high memory and low memory applications, as well as high invocation frequency, the analytical model seems to show reasonable error and allow a designer to make general conclusions about specific accelerators. Through this, we show robustness in our analytical model.

Our analytical model allows us to do large sweeps of different design spaces to come up with various conclusions. We determine that high-performance cores are more susceptible to slowdowns or lower than expected speedups when not operating with fully OoO TCA instructions. We also determine that more fine-grained tasks see larger performance degradation from non-OoO execution. This model allows for a quick way to estimate impacts of TCA designs over various algorithms, processor architectures, and TCA designs.

This analytical model helps direct the search for likely candidates of TCA acceleration in multiple ways. First, an architect that already has an idea for an accelerator can quickly determine the estimated effectiveness of the TCA. Second, our model also helps the designer determine the estimated penalties associated with various implementations for them to determine specifications desired for the design phase. Third, our model helps give high-level intuition about different classes of functions for various architectures. This can help direct the search of an architect to classify functions in their target application that could see the most benefit from a TCA.

5 NORF: AN ARGUMENT AGAINST REGISTER FILES FOR TCAS

After the architect determines which algorithm they wish to accelerate, the next step is to move onto the implementation phase. Chapter 4 helps in determining the desired level of speculative and out-of-order execution. Beyond that is an important decision regarding the microarchitecture and datapath within the TCA. The datapath for passing operands to and from the TCA can come from existing register files, specialized register files, and/or the memory hierarchy. This chapter analyzes the tradeoffs in operand passing and describes why it is not particularly beneficial to use register files as used by existing compute operations.

5.1 Chapter Overview

General-purpose processors typically supply operands from scalar and/or vector registers whose size is determined at design time. However, accelerators often perform computations on data structures that may not be a natural fit for these registers. To best utilize such an accelerator, the designer can either use existing general-purpose registers at some efficiency cost, incur the cost of designing a new register file (RF) tailored for the accelerator, or eschew a RF entirely, making the unit access its operands directly from the memory hierarchy. We evaluate these alternatives in the context of matrix-matrix multiplication accelerators attached to a high-performance out-of-order processor, and show that the energy cost of optimal RF access can exceed the savings from eliminating memory accesses, while any reuse captured by the RF can more efficiently be captured by a small *matrix cache*

embedded directly inside the tightly-coupled accelerator. We argue that most TCAs should be designed with a NoRF (no register file) implementation, using a matrix-matrix TCA as an example. NoRF can reduce dynamic energy consumption of data movement by up to 61% for algorithms with little reuse (block SpGEMM and DGEMV). For algorithms with high reuse (DGEMM), we propose using NoRF with a matrix cache, which reduces dynamic energy consumption by up to 39%. The NoRF implementation can also simplify programmability. We also discuss and evaluate microarchitectural alternatives for efficiently supporting direct memory operations issued by a tightly-coupled accelerator within an out-of-order processor core.

Section 5.2 explains for our decision to use DGEMM as our example TCA. We explain design considerations when integrating operand movement both through RF (Section 5.3) and NoRF (Section 5.4) implementations. We believe we are the first to optimize design choices for both TCA implementations in the context of limiting data movement, maximizing reuse, and easing programmability. We then implement the DGEMM TCA using our design optimizations in both a cycle-level simulator and through synthesized power estimations (Section 5.6).

5.2 Motivation

Matrix-matrix multiplication ($A \times B = C$) is an increasingly popular workload for multiple reasons, but particularly due to many machine learning (ML) and artificial intelligence (AI) techniques that rely on it, including convolutional neural networks (NNs), recurrent NNs, or other similar algorithms. This makes matrix multiplication operations a prime candidate

(a) TCA integrated with RF

(b) TCA integrated with NoRF

Figure 5.1: Dataflow for TCA that uses RF vs. NoRF. The first letter denotes either a register (R) or memory (M) action that is being performed. The second letter denotes either a read (R) or write (W) to that structure. Without reuse in the register file, the RF requires many more steps of data movement external to the tightly coupled accelerator (TCA).

for creating a TCA. Depending on the application, these matrix multiplication operations may be performed on either sparse or dense data. In the case of sparse data, the programmer can decide whether to do dense operations on the entire matrix (at the inefficiency cost of performing many unnecessary multiplications with one or both operands being 0), or a sparse algorithm (at the cost of more complicated control-flow).

In both cases, when matrix multiplication operations are performed, either the data or pointers to the data must be provided to the TCA. If the CPU sends the data to the accelerator, it could either send it through the general-purpose register file, through a specialized register file for the specific data structure (e.g., 4×4 element matrix), or through memory-to-memory operations.

The tradeoffs between these possible solutions are not obvious. At a high level, register files help reduce memory accesses (steps 1, 2, 5, and 6 in Figure 5.1(a)) when the compiler finds reuse of operands in the program dataflow. On the other hand, direct memory access

simplifies the data movement by completely eliminating the intermediate reads and writes to a register file but issues more memory requests. Moreover, other tradeoffs such as programmability, hardware complexity, and memory coherency complications need to be understood before designing these TCAs. This chapter provides insights to the difference between these options of operand delivery.

It is also important to note that TCAs by their very nature are designed to capture as much of the data flow graph execution of an algorithm internally. Increasing internal data forwarding within the TCA may provide smaller opportunities to have significant data reuse externally with surrounding instructions. Limited reuse discourages the requirement to occupy register files prior to TCA invocation. This chapter evaluates both modes of operation to determine these impacts.

We also look at the differences between 2×2, 4×4, and 8×8 matrix multiplication operations, since DGEMM partial product multiplications grows at the rate of $O(n^3)$, while the number of element loads/stores from memory only grows at the rate of $O(n^2)$. For a TCA with a fixed number of floating multiply accumulate (FMAC) units, the rate of loads/stores required to balance throughput of compute with data movement changes.

To add another dimension of variability, researchers have investigated performing these matrix operations at various quantization values. Deniil et al. [14] showed how only a few values of weights need to be learned in deep learning applications, and surrounding values will be similar, leading the charge towards quantization and shared exponent data formats for neural networks. Quantizing convolutional and fully connected layers in CNNs can drastically reduce memory consumption and execution time with minimal increases in output error [88]. Blocked floating point format [21] shares 5-bit exponents across

128 elements while maintaining high accuracy. Reducing the number of bits for either weights or inputs can increase effective memory throughput as well as reduce computation complexity. Our work analyzes whether or not the differences in bit widths changes the optimal data transfer technique for tightly-coupled accelerators.

5.3 Register File GEMM

The natural solution for a new functional unit or TCA that is tightly integrated into the core delivers operands through the register file. Existing functional units are already integrated through the register file, so we evaluate this method as the baseline. When there is temporal reuse of data elements, register files can help reduce the number of requests sent to the memory hierarchy. However, when little or no reuse is possible, it simply becomes an intermediate step of reads and writes in the data movement process.

Whether having a traditional or specialized register file as explained in the following subsections, we assume a few common similarities. First, when reading/writing elements to/from the RF, we send all memory requests through the load-store queue (LSQ). In the front-end, all addresses are virtual, but like all memory requests, each load μop is translated through the TLB. We implement all register loads from the RF/TCA as multiple existing vector load μops that already support translation. Second, to give the best-case scenario of the RF for reducing overheads, we assume the TCA can effectively eliminate reads and writes of intermediate data by passing all intermediate data values internally. In our example of DGEMM, although a 4×4 matrix multiplication performs 64 multiply-accumulate (MAC) operations, only the final 16 elements (or 4 vectors) of the calculated

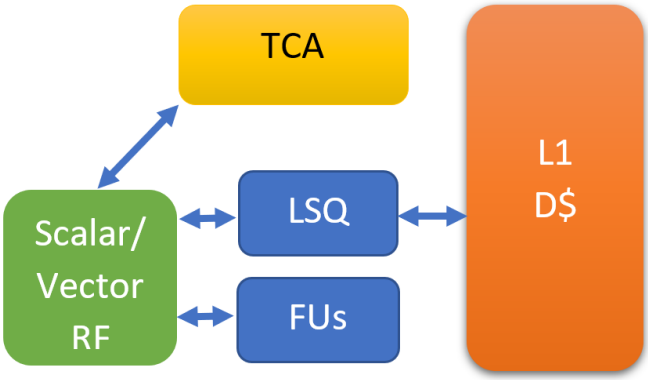


Figure 5.2: Integrating TCA with traditional register file.

matrix output are written to the RF, with all intermediate values internally passed through the TCA datapath. Lastly, if modifying shared data in the TCA , we assume the programmer must explicitly provide synchronization for the time between when the data is loaded from memory, operated on, and finally stored back to memory. This is typically done through the use of locks or semaphores surrounding the critical section, creating additional programming complexity and performance overheads.

5.3.1 Traditional Register File

The first option we consider is trying to use the existing register file as the central point for operand delivery (Figure 5.2). This has the benefit of using existing hardware and requiring little redesign of existing core structures.

However, using the general-purpose register file has several potential inefficiencies. First, scalar registers are only 64 bits wide, only allowing a single double-precision floating point value per register. In order to store enough elements to hold the A, B, and C matrices for a single 4×4 matrix-matrix multiplication operation, at least 48 such registers are required (16 per operand). This also requires many read and write requests to and from

the register file just to pre-load the matrices into the register file. It is easy to see that the register file cannot hold enough data to get any substantial (if any) reuse with the matrices brought in, as the registers will spill back to memory to make room for future iterations. This problem only gets worse with 8×8 matrices, as 192 (64*3) registers are required.

Vector registers (e.g., SSE or AVX) could be used instead of the general-purpose register file to try to eliminate some of the problems listed above. First, these registers are separate from the general-purpose registers, which contain information such as stack pointers, temporary variables, and loop induction variables. Second, but more importantly, the increased width can load multiple elements per register. SSE-128, AVX-256, and AVX-512 registers can hold an entire row of a 2×2, 4×4, and 8×8 double-precision matrix, respectively. This reduces the required registers of a single invocation from 12, 48, and 192 registers down to 6, 12, and 24 registers, respectively. However, in order to fit 32 logical matrices, 64, 128, or 256 vector registers are required, respectively, which is impractical in current designs.

When encoding an ISA instruction for matrix multiplication, it is not feasible to encode 24 registers into the TCA instruction. Additionally, since registers are renamed dynamically to allow out-of-order execution, there is little that can be done to simplify the process of decoding and reading multiple registers to load a single matrix into the TCA. This puts tremendous stress on the decode stage of the processor.

Each of these general-purpose register file solutions exhibit the problem that although each matrix is logically a single data structure, it is physically distributed across multiple registers. This creates inefficiencies when loading and storing matrices to and from the TCA. Similarly, when multiple registers are used, it limits the amount of reuse that is possible

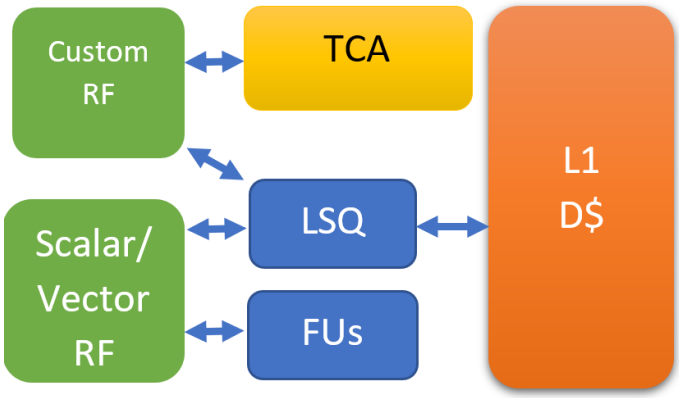


Figure 5.3: Integrating TCA with separate, specialized register file.

before all logical registers are used, inefficiently thrashing values back and forth between the register file and the cache hierarchy. For these reasons, we conclude that even after completely eliminating reads and writes of intermediate results, it is impractical to use existing register files to pass data to and from a matrix multiplication TCA. Instead, in order to create the best-case, optimistic baseline, we propose a specialized register file, tailored for the dimensions of the TCA matrix accelerator. Throughout the rest of the chapter, we consider any RF design to be specifically tailored to the dimensions optimal for the TCA's operands, and analyze the costs and benefits of such a design.

5.3.2 Specialized Register File

Figures 5.2 and 5.3 show the differences between using a traditional versus a specialized register file. The specialized register file adds a separate path for data to be loaded from memory. This extra register file, however, allows the accelerator to have access to an entire matrix operand through a single register file location.

When creating specialized registers for the exact matrix size used by the TCA, fewer registers are needed to be passed between the register file and TCA. This simplifies the

complexity of passing data to and from the TCA. Note that it is still possible to have each logical register access split across multiple cycles to provide the same functionality, datapath width, and interface as the vector register files. However, specialized registers can then guarantee spatial locality for vectors belonging to the same matrix, as well as fewer ISA bits for register encoding.

Additionally, by creating a specialized register file, the architect can choose an efficient number of registers for the amount of reuse desired, as well as bit-width for the registers that is optimal for the accelerator. Since the register file is separate, it also does not cause capacity conflicts within the general-purpose or vector register file. This comes at the area and power costs of adding a new register file to the datapath.

5.3.3 Register File Reuse

When performing matrix multiplication, it is common to try to maximize performance through matrix tiling [78] [30]. One of the most common forms of tiling is for maximizing reuse within the D$, L2, or LLC. Because the latency of memory operations significantly changes depending on what cache (if any) the data resides in, software algorithms change the ordering of the partial product calculations to maximize reuse in the fast data accesses (such as L1 hits), and reduce the number of requests made to higher latency structures (such as DRAM). This same idea for reuse can be applied at the register file level. Since register file hits will both provide fast data access as well as reduce D$ bandwidth contention, we want to maximize the number of floating-point multiply-accumulate (FMAC) operations that can be performed with the data residing in the register file, and minimize the number

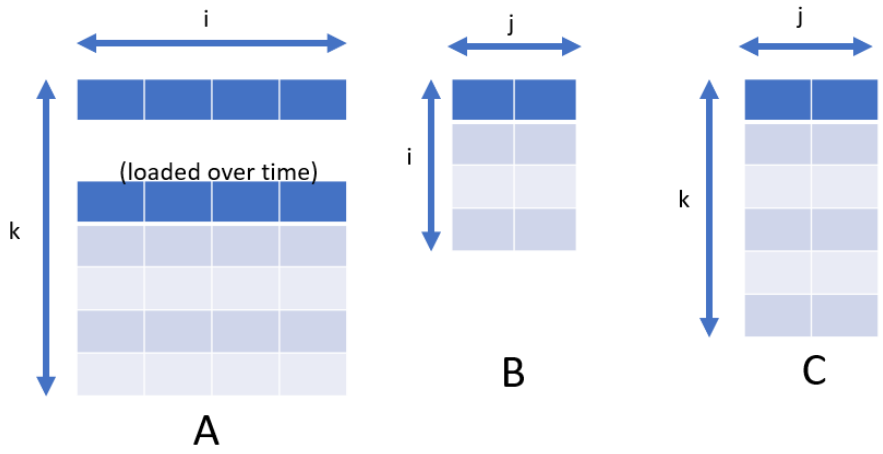


Figure 5.4: Number and dimensions of registers while keeping C stationary. Number or registers for A, B, and C are i, $i * j$, and $j * k$, respectively. Registers for A are loaded temporally (only one needed in the RF at any given time) to apply to all elements of B. For our C-stationary algorithm, the C registers will be replaced the least frequently.

of register file fills and spills to the D$.

We create an output-stationary DGEMM algorithm with a triply-nested loop for the matrix, tiled into a triply-nested loop for L1 cache tiling, and lastly a third triply-nested loop for RF tiling within the cache.

We mathematically analyze different dimension sizes for input matrices A and B and output matrix C in the register file that will minimize the number of reads/writes to the register file per FMAC that can be performed with the data provided.

Figure 5.4 visually shows how modifying i, j, and k changes the dimensions of A, B, and C matrices stored in the RF. Table 5.1 shows the number of reads from memory, as well as the number of FMAC operations that can be performed when storing different rectangular chunks of A, B, and C in a RF-based TCA. We look to minimize the number of reads (from memory) per FMAC to maximize reuse, while having the constraints of the A, B, and C chunks being able to fit in the number of registers available.

Matrix	# Registers	# Elements	# Vector Reads	# FMACS	Reads/FMAC
A	i	$n^2 * i$	$n * i$	$n^3 * i * j$	$1/(n^2 * j)$
B	$i * j$	$n^2 * i * j$	$n * i * j$	$n^3 * i * j * k$	$1/(n^2 * k)$
C	$j * k$	$n^2 * j * k$	$n * j * k$	$n^3 * i * j * k * (size/(i * n))$	$1/(n * size)$

Table 5.1: Equations used to calculate the number of memory reads per FMAC. To get the most reuse within the register file, this value should be minimized. Size denotes the number of blocks that fit within the L1 cache, where C reuse is maximized when more of A and B fit in the cache.

Timeloop [66] demonstrates empirically how energy/FMAC drastically changes based on traversal path, and we numerically find the optimal reuse for our size RF. Through an exhaustive search of all values of i, j, and k, we find that for 32 registers, we can minimize loads/FMAC with i=1, j=5, k=5. Or in other words, A occupies 1 register, B occupies 5 registers, and C occupies 25 registers. The programmer can decide whether or not to keep the size of j and k limited to powers of 2 (to not require padding for matrices with dimensions in powers of 2). Either way, these formulas can provide insights to the optimal reuse within the register file for both energy efficiency (fewest reads / FMAC), as well as likely for performance (requiring fewer reads to slower memory). The formula can also be used to find the optimal {i,j,k} values for 16, 64, and 128 register implementations being {1,3,4}, {1,7,8}, and {1,9,13}, respectively.

5.3.4 Programmability When Using an RF-based TCA

Listing 5.1 shows how the programmer could code a dense matrix multiplication algorithm using the knowledge of optimal register placement. The Matrix Load (MLD) instructions are assumed to be ISA instructions that give a starting address and load the appropriate 16 elements of the 4×4 submatrix.

```
//Code after Cache Tiling
//Register Tiling
for (kk=k; kk<k+Cache; j+=Reg_k){
  for (jj=j; jj<j+Cache; i+=Reg_j) {
    //Aquire lock for C chunk
    Lock_set4x4(c+kk*n+jj);
    //Matrix load (MLD) c into 16 reg
    r0 = MLD(c+kk*n+jj,n); //c0
    ... // MLD c1 through c14
    r15 = MLD(c+(kk+3)*n+jj+3,n);//c15
    for (ii=i; ii<Cache; ii+=Reg_i) {
      //MLD b into 4 registers
      r16 = MLD(b+ii*n+jj,n);//b0
      ... // MLD b1 through b3
      //MLD a into 1 register
      r20 = MLD(a+kk*m+ii,m);// a0
      //apply accl inst of a to all b's
      4x4DMM(r20,r16,r0,m,n); //a0,b0,c0
      4x4DMM(r20,r17,r1,m,n); //a0,b1,c1
      4x4DMM(r20,r18,r2,m,n); //a0,b2,c2
      4x4DMM(r20,r19,r3,m,n); //a0,b3,c3
      r20 = MLD(a+(kk+1)*m+ii,m); //new a1
      4x4DMM(r20,r16,r4,m,n); //a0,b0,c4
      4x4DMM(r20,r17,r5,m,n); //a0,b1,c5
      4x4DMM(r20,r18,r6,m,n); //a0,b2,c6
      4x4DMM(r20,r19,r7,m,n); //a0,b3,c7
      r20 = MLD(a+(kk+2)*m+ii,m); //new a1
      4x4DMM(r20,r16,r8,m,n); //a0,b0,c8
      4x4DMM(r20,r17,r9,m,n); //a0,b1,c9
      4x4DMM(r20,r18,r10,m,n); //a0,b2,c10
      4x4DMM(r20,r19,r11,m,n); //a0,b3,c11
      r20 = MLD(a+(kk+3)*m+ii,m); //new a1
      4x4DMM(r20,r16,r12,m,n); //a0,b0,c12
      4x4DMM(r20,r17,r13,m,n); //a0,b1,c13
      4x4DMM(r20,r18,r14,m,n); //a0,b2,c14
      4x4DMM(r20,r19,r15,m,n); //a0,b3,c15
    }
    //Release lock for C chunk
    Lock_reset4x4(c+kk*n+jj);
  }
}
```

Listing 5.1: Example pseudocode to perform 4×4 dense matrix multiplication while maximizing reuse within the register file.

The typical way to program matrix multiplication is to use element indexing for A, B, and C. Since the reuse of registers is split across many different levels of nested loops, and dynamically recalculating the index of A, B, and C, it is likely difficult for the compiler to

unroll these loops and to know that reuse is possible. One can manually store addresses to registers, and only reload the registers once it is known that it is required. This puts the burden on the programmer to explicitly list the register reuse by being aware of both the algorithm and the hardware. From the code, it is easy to see that this is a burdensome task for the programmer to perform. This also requires the programmer to have considerable knowledge about the hardware to have efficient code execution, breaking away the abstraction intended by the ISA.

We also see that explicit synchronization through the use of locks is required for the program to be aware whether or not a given 4×4 region of the C matrix can be safely loaded and modified in the RF.

5.4 Memory-based, NoRF GEMM

Section 5.3 explains in detail the implications of data movement when it comes to a TCA that interacts with a register file. Through diagrams and intuition, it becomes unclear as to whether or not the register file is actually providing benefit to the TCA for data movement. On one hand, with a large enough, specialized register file, data reuse is possible, which reduces the number of memory requests needed to the D$. On the other hand, with the amount of reuse being limited, frequent reads and writes to an intermediate register file seem potentially unnecessary. Having a register file as an intermediate changes a datapath of the L1D$ to the TCA from 2 total reads/writes to 6 total reads/writes (refer back to Figure 5.1). If a register file element is read and immediately removed, it seems beneficial to completely bypass the register file. This implementation of using no register

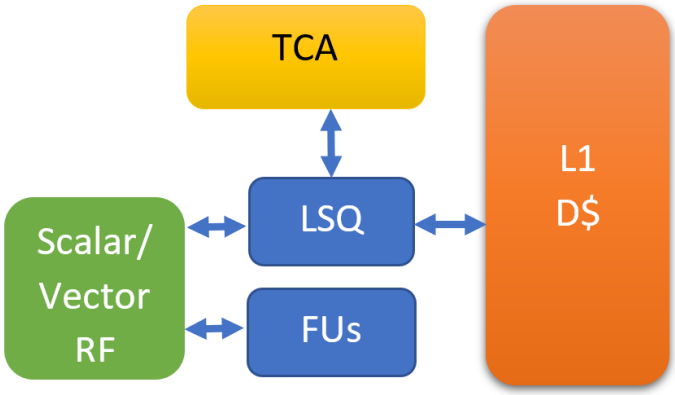


Figure 5.5: Integrating TCA that performs memory-to-memory data movement.

file can be called a **NoRF implementation** (Figure 5.5). We explore both the practicality of building a NoRF TCA that interacts with memory cache hierarchy, and the difference in data movement cost compared to RF-based implementations.

5.4.1 ISA Changes

Instead of loading values to a register file, the TCA can directly request the values from memory whenever that input matrix is required. The NoRF design has some similarities to its RF loading counterpart. First, these memory operations can be issued based on a base pointer and the matrix row size. For double-precision floating point, 2 16-byte loads can fill 2×2 matrices, 4 32-byte loads can fill 4×4 matrices, and 8 64-byte loads can fill 8×8 matrices. Second, since the Intel Sunny Cove architecture supports 64-byte loads in the LSQ, we assume the TCA can simply insert these wide loads into the load/store queue (LSQ). This allows requests to be made through the coherent cache hierarchy, and also resolves out-of-order address resolution for memory requests. Lastly, the decoded TCA instruction can be decoded into multiple load and store micro-ops (μops) that enter the

LSQ pipeline.

To perform matrix multiplication through memory (whether a 2×2, 4×4, 8×8, or any other size), at least 5 operands are required. The first 3 are associated with the pointers of the start locations of the A, B, and C submatrices being multiplied. Since the elements in the same row of a matrix are contiguous and captured in the wide loads, elements within the same row enter the TCA without additional operands. However, for elements in subsequent rows, the addresses are dependent on the size of the entire matrix. Due to matrix multiplication being of dimensions $i \times j$, $j \times k$, and $i \times k$, the dimensions j and k are required for calculating memory addresses of adjacent row elements.

Now that we know what 5 operands are required, it is up to the architect to decide how to implement these instructions. If there are enough bits in the ISA, the designer could attempt to fit all 5 operands within the same instruction. This could be done either through immediate values, or likely by passing registers that contain these values. Note that even if using a RF for argument passing, this is just for holding pointers and immediate values, not for the data itself. NoRF designs enforce that data from memory loads and stores never enter (are buffered in) the RF.

Alternatively, the accelerator could get its parameters through a sequence of instructions that push operands to the TCA. In other words, each TCA invocation instruction could be prefaced by TCA push instructions, passing the parameters to the accelerator over the course of multiple cycles. A combination could also be implemented, such as configuring the accelerator ahead of time with push instructions for the j and k dimensions for the A, B, and C matrix widths, which are updated either at extremely coarse granularity or never at all in most realistic applications, and only passing the A, B, and C parameters

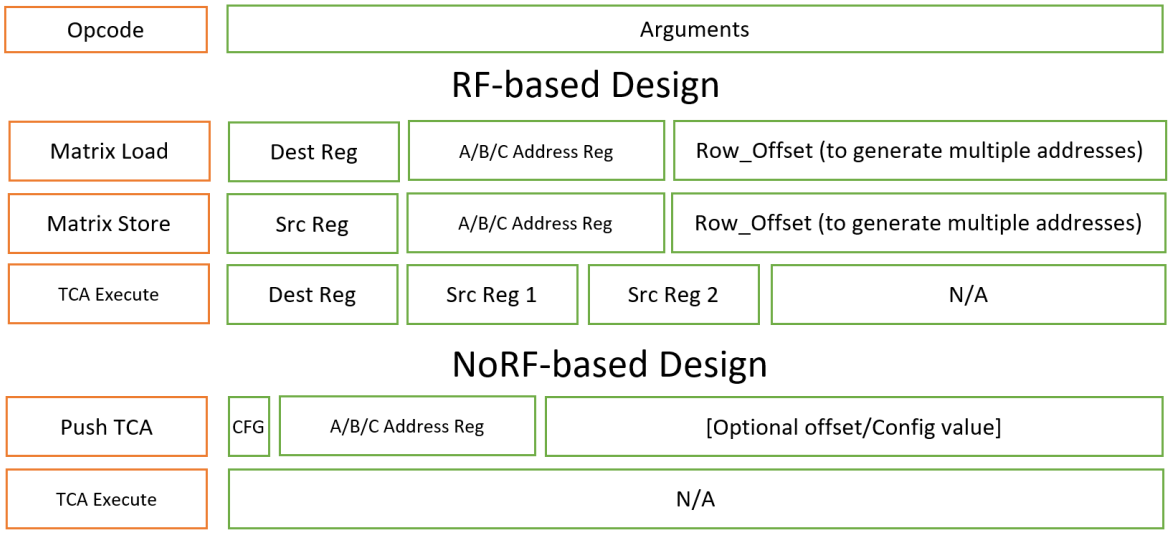


Figure 5.6: ISA additions for NoRF-based and RF-based TCAs. Note, Dest/Src Reg used in RF-based design are specialized, matrix registers. For NoRF-based design, designer can choose to fit all 5 operands within the same instruction.

via TCA invocation. This option is more versatile to other TCAs, as argument passing ISA bits does not have to be TCA specific, and each TCA could have any arbitrary number of preceding TCA push instructions in order to execute. Differences between RF and NoRF ISA additions are shown in Figure 5.6.[1]

Since data movement is done directly through memory μops, there is a much simpler path for data movement to/from the TCA. Since there is no intermediate step, the accelerator can directly produce outputs and directly consume inputs to and from memory, respectively. Also note that there are minimal hardware changes, as the specialized register file is no longer needed, and the memory requests are made through existing hardware directly to the TCA. In our example of using the LSQ to process memory requests, the TCA loads values when the LSQ entries allocated for the TCA become valid.

[1]Heng Zhuo helped in the organization of this figure.

5.4.2 Data Reuse using a Matrix Cache (M$)

A potential downside of eliminating the register file is losing the ability to reuse operands. However, if the TCA is allowed to buffer loaded operands, reuse is possible in memory-to-memory operations in the same way that reuse can be done through the register file. At first it may seem like buffering data causes similar overheads to the RF, but by having a known datapath and access patterns (i.e., having different banks for A, B, and C matrix elements), this implementation eliminates the multi-ported nature of the RF.

There are two potential ways to capture data reuse within TCA internal buffers. (1) The TCA could be explicitly programmed to perform certain reuse. Similar to a scratchpad with explicit data movement, or hints for reuse via specialized invocations, the TCA could reuse data it has recently seen in the internal buffer. This is very similar to manually filling the registers for optimal reuse in the register file software implementation (Listing 5.1). As stated before, this is not desired, since we do not want to burden the programmer or compiler with handling these cases.

(2) The TCA M$ could use a tag-match based approach. Similar to all other caches, these data buffers store tags to designate the address associated with each element in the internal buffer. Since the accelerator knows what elements represent the A, B, and C matrices on invocation from the ISA instruction, the accelerator could have an internal cache to store elements expected to be used again. Through our analysis, we determine optimal reuse can be gained using a single element of A, vector of B, and matrix of C (all of which can be cached, both data and tag). Whenever the accelerator is invoked with the 3 matrix block addresses, it checks its internal cache to see if the addresses provided match

the tags of its internal cache. Since this structure behaves like an internal cache for storing submatrices, we call this the TCA's *matrix cache*, or M$.

If the A, B, and C operands are pushed to the TCA through separate instructions, the TCA can check the hit status of the internal cache for each instruction, and the actual invocation can then conditionally issue only the loads that missed. If the accelerator is invoked via a single instruction, the hit/miss status is unknown during the decode stage, requiring the TCA to either issue conditional loads at decode (predicated on M$ hits or misses), or notify the LSQ during execution based on which elements hit in the cache. The M$ can use virtual tags and ASID to reduce latency/TLB bandwidth determining hits, and simply keep track of physical tags after translation of loads for coherency purposes. A hit would tell the accelerator that the memory request for that element is not required. We incorporate reuse in our analysis of NoRF TCAs through the use of method (2).

With a matrix cache, the DGEMM TCA can get as much reuse from the data as possible without having an intermediate register file. Although the "caching" of the TCA behaves similar to the register file, it should always be more efficient than a general-purpose register file since it avoids multi-ported structures by using separate banks. It is important to note that any reuse of a register file holding n elements can also be captured by a NoRF TCA that can cache n elements. It is important to note that the M$ is completely optional, and the TCA can function with and without it. This also means that it could be dynamically enabled and disabled depending on expected reuse within the given workload/application and still have correct execution. This is different than RF designs, which always pass data through the RF, where disabling the RF is not an option for most ALU instructions.

5.4.3 Coherency and Write-through vs. Write-back

During design time of a NoRF implementation, the TCA could be designed as a write-through or a write-back accelerator. For the write-through case, all updates to the output vector or matrix are stored to memory immediately to make sure that the memory hierarchy contains the most up to date copy of C. The synchronization can be implemented through hardware by requesting an exclusive copy to C on the loads to C, and writing back to memory on commit. In the NoRF implementation of the TCA, write-through could remove the programmer's need for explicit synchronization if memory and TCA values are updated together. If TCA coherency is desired, the TCA could potentially be designed to enforce atomicity through existing hardware techniques (e.g., hardware transactional memory [38]).

For the write-back TCA, all changes to the output matrix or vector are lazily updated to memory. In other words, multiple invocations of the TCA may update the same output matrix/vector before the store requests are made to update the memory hierarchy. This has performance benefits of fewer writes to memory, but at the cost of program complexity in multithreaded execution to add synchronization to prevent non-atomic updates to partial products. If the TCA has internal caching (described in detail in Section 5.4.2), coherency can be maintained by snooping coherence messages.

Both RF and NoRF implementations have implications for memory consistency and coherency. Typically, elements in a register file are not guaranteed to be coherent with their corresponding memory location. This puts the synchronization burden on the programmer as previously described. Likewise, the memory-based TCA could also leave the burden on

the programmer, or alternatively have hardware mechanisms to support coherency within internal caching. This form of caching would require the same coherency mechanisms as the current D$ (such as snooping of invalidations) to guarantee coherency and consistency. Any other process requesting read or write access to the memory location will send a coherence request, and the TCA can send the data with its ack response when it invalidates its own copy. Maintaining inclusion in the lower-level caches will streamline this process, as is the case for conventional caches. The matrix cache needs to support coherence messages based on physical addresses, either by keeping track of physical tags or relying on a reverse translation mechanism. While we leave validation of coherency for future work, NoRF-based designs appear to have extra potential for coherency that does not exist for RF designs.

5.4.4 Programmability

By putting the ISA changes in place in addition to an internal matrix cache, we can see that functionally the NoRF implementations behave similarly to the RF implementation. However, using a NoRF design allows the TCA to explicitly monitor the reuse of submatrices rather than requiring the programmer to explicitly exploit temporal reuse through register-based operations. In our example, no triply-nested loop unrolling is required to determine what TCA operations should be mapped to identical matrix load operations. The NoRF design dynamically determines reuse within the M$, providing a useful abstraction for programmers to gain all reuse possible without programming burden. Listing 5.2 shows how much simpler the code is to modify to different algorithms or architectures, to

understand, and to write.

```
//Code after Cache Tiling
//Register Tiling
for(kk=k; kk<k+Cache; kk+=RegK){
  for(jj=j; jj<j+Cache; jj+=RegJ){
    for(ii=i; ii<i+Cache; ii+=RegI){
      for(iii=ii; iii<ii+RegI; iii++){
        for(kkk=kk; kkk<kk+RegK; kkk++){
          for(jjj=jj; jjj<jj+RegJ; jjj++){
              4x4DMM(a+kkk*m+iii,b+iii*n+jjj,c+kkk*n+jjj,m,n);
}   }   }   }   }   }
```

Listing 5.2: Example program to perform 4×4 dense matrix multiplication with memory-based TCA.

Through this example, we can see that programmability can be greatly simplified by having a NoRF TCA.

5.4.5 Tradeoff Summary

Table 5.2 shows a summary of the M$ design compared to the RF alternative. We also show scratchpads as a point of reference, showing our M$ design addresses issues that also exist for scratchpad-based reuse designs. NoRF designs request operands through memory addresses, where the M$ detects hits dynamically through tag-based matching. Since RFs and scratchpads are general-purpose structures, they are not optimally banked or sized for the specific TCA, consuming additional energy per data access. Since the

	RF	Scratchpad	NoRF M$
Addressing	Direct	Local Addressing (Base + Offset)	Memory Addressing (Tag-based)
Generality	General	General	Specialized
Implementation	Multi-Ported	Multi-ported or Banked	Single-ported banks
Reuse Detection	Static	Static	Dynamic
Management	SW	SW	HW
Coherency Support	No	No*	Yes

Table 5.2: Differences in the NoRF M$ compared to RF or Scratchpad alternatives. *Stash [53] provides a scratchpad variant that supports coherency.

TCA's functionality is specific to DGEMM, generality occurs overheads without being able to utilize the benefits of a general-purpose structure. The M$ dynamically detects reuse through hardware, which removes the complexity of managing reuse through software. NoRF designs also use memory-based accesses which is more suited to support coherency.

5.5 Methodology

We evaluate the differences in data transfer and energy costs for register file and NoRF implementations of 64-bit, 32-bit, and 16-bit DGEMM algorithms. We first explicitly code a software implementation of dense matrix multiplication tiled both at the L1 D$ level, as well as register file level within the L1 D$. We can then count the number of accelerator operands required to pass to and from the accelerator, and number of operands that are passed between the register file and memory hierarchy. We evaluate DGEMM using 512×512 matrices. Due to the irregular access pattern and low probability for reuse within most SpGEMM matrices, we assume no reuse in our block SpGEMM energy estimation. We evaluate GEMV using a 8192×8192 matrix multiplied by a 8192 wide vector.

We use CACTI [82] to estimate the energy associated for L1 cache access, at 40nm with a design focus on low power. We can adjust the matrix multiplication algorithm to tile with different register dimensions and see the energy costs associated with changing these dimensions.

The simple NoRF implementation requires more memory accesses, since there is no data reuse, but completely eliminates the read and write costs to the intermediate register file. We use a write-through TCA with an internal write-buffer to coalesce writes before

parameter	value
fetch/decode width	6-way
dispatch/issue width	10-way
L1 D$ size	48kB
L1 D$ set associative	12-way
L1 D$ Load Ports	2/cycle
Prefetcher	Stride
L2 size	512kB
ROB size	352

Table 5.3: Notable gem5 parameters, simluating Sunny Cove-like core[85][12].

sending to the L1.

The NoRF with M$ implementation has the same cache capacity as the specialized RF, but has 3 banks for A, B, and C matrices sized optimally for reuse. These specialized RFs are implemented in SystemVerilog. We also create the M$ tag matching and data forwarding in SystemVerilog, and use Design Compiler to estimate the power of these hardware elements targeting a TSMC 40nm low-power standard cell library. We assume proper clock gating to only access register flops being accessed on a given cycle. We use the same methodology to create a 2 read port, 1 write port RF for our RF analysis.

In order to test execution performance of the TCA design with a cycle-level simulator, we use gem5 [6]. We configure gem5 to mimic Intel's Sunny Cove [85] architecture (Table 5.3). For the performance baseline, we generate a software-based implementation with element-wise traversal, blocked at the L1 level. We then use Eigen [36] as an existing software library designed to exploit SIMD and vector registers to try to utilize the system's vector registers for extra parallelism. We then incorporate our proposed 2×2, 4×4, and 8×8 DGEMM TCA instructions to see the additional performance that can be gained over vector register execution. To do this, we designate new instructions within gem5 to invoke

the TCA. We create both RF and NoRF-M$ instructions and implementations.

The RF implementation has specialized load and store instructions, decoded into multiple load or store μops based on the starting address. An execute μop then performs the 2×2, 4×4, or 8×8 MAC operations for the operand registers with average throughput of 8 FMACs/cycle. The load and store μops are issued into the LSQ, while the execute TCA μop is issued to a separate TCA functional unit.

The NoRF-M$ implementation is similar, except that only a single specialized instruction is necessary. First, we create a specialized μop that takes the A, B, and C addresses and checks for the presence in the M$, filling 3 predicate registers (1 bit each). The instruction is also decoded into predicated loads for the A, B, and C vector loads required for computation. If the predicate for A, B, or C designate a hit, the respective loads are immediately marked as executed. On a M$ miss, the load executes and populates the M$ tag status and data fields when the data returns. After the specialized execute instruction is complete, the stores are written to the M$.[2]

We manually insert the specialized TCA instruction as well as specialized RF load/store instructions as shown for Listing 5.1. We manually insert the NoRF-based TCA instructions for the NoRF-based algorithms as shown for Listing 5.2.

5.6 Results

We first confirm that substantial performance benefit can be gained from implementing large DGEMM through tiled, frequently invoked small DGEMM TCA instructions. Although the

[2]Heng Zhuo created the matrix cache check μops within gem5, assuming a LRU replacement policy. He also played a significant role in the testing and debugging of these gem5 TCA μops and instructions.

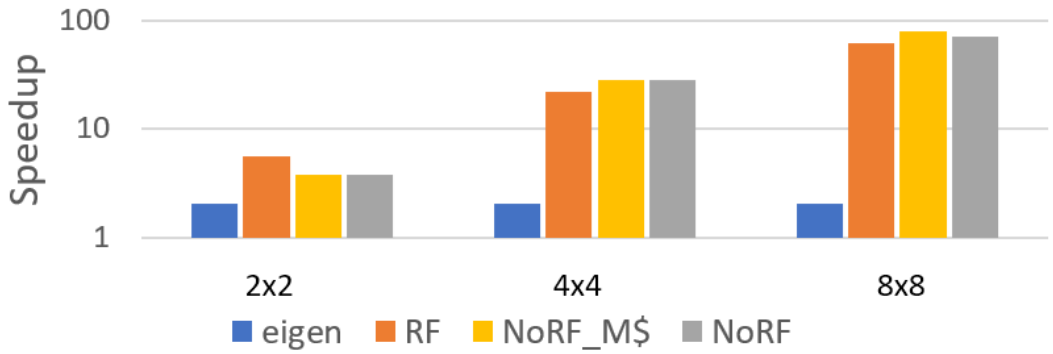


Figure 5.7: Performance speedups of Eigen, 2×2, 4×4, and 8×8 TCA accelerators normilized to element-wise execution. Eigen is a library that gains speedup by utilizing vector registers [36]. The TCA gains speedup through added MAC execute units, partial product forwarding, and wide loads from the L1D$.

Eigen algorithm has the benefit of not having to change the existing hardware or datapaths, Figure 5.7 shows that it has the lowest overall speedup. Speedup from Eigen is limited by the width of vector registers and the number of floating point execution units. As expected, custom TCAs which supply additional floating point MACs and custom logic to supply those MACs see additional speedup. As expected, we see significant speedups for the larger DGEMM TCAs, as the work done each invocation grows by a factor of $O(n^3)$, reducing control flow operation and utilizing larger vector loads.

Figure 5.7 also shows relatively little difference in performance between NoRF and RF implementations, as expected. The NoRF design issues more µops per execute instruction and has some degradation in the smallest TCA (2×2) due to extra front-end pressure. For larger TCAs, NoRF actually surpasses RF performance due to algorithmic differences spreading out the bursty loads from RF-tiling. However, the overall trend shows similar performance benefits for both RF and NoRF designs.

Sparse matrices can still utilize the TCAs proposed by utilizing a blocked-sparse format. Prior work looks at blocking non-zero elements within a sparse matrix [95], as well

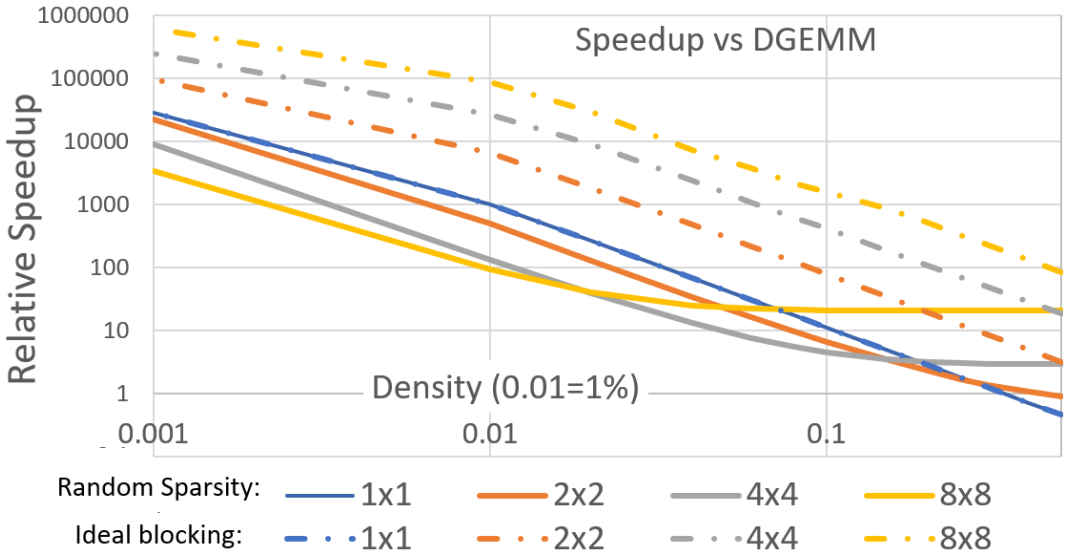


Figure 5.8: Performance speedup of doing block-sparse SpGEMM using 2×2, 4×4, and 8×8 blocks with respective TCAs. Dotted lines show an ideally blocked matrix, and solid lines shows a randomly distributed matrix.

as several matrices that naturally have local density along the diagonal. We see several matrices from the SuiteSparse Matrix collection [52] such as Gleich/Minnesota and DIMACS10/Luxembourg_osm that demonstrate high locality among the diagonal or clusters of dense elements within the larger sparse matrix.

We test both ideal case (where all non-zero elements are fully packed into TCA blocks), as well as a pessimistic, randomly distributed sparse matrix (no local density). Each is normalized to computing the full DGEMM algorithm for the entire matrix size, and we test over various levels of matrix sparsity. Figure 5.8 shows that for very sparse matrices, the highest speedups are achieved, since limited calculations are required. Interestingly enough, however, when non-ideal packing is involved, the 1x1 (traditional CSR algorithm) surpasses each of the TCA execution times at a specific crossover point for each TCA size. This is because if there is not enough local density, the TCAs have to perform overheads of extra loads and multiplications of zero values. Not surprisingly, however, ideally blocked

bCSR matrices achieve much higher performance as TCA size increases.

Taking a closer look at Figure 5.8 shows an interesting mirroring pattern between the ideal and randomized case. For dense matrices, it is unsurprising that the 8×8 TCA has the largest speedups. Both the randomized and ideal blocking cases show this to be the case, where the large speedups outweigh the low inefficiencies of having few 0 elements in the randomized case. For the sparse matrices, the ideal blocking also maintains this trend, since there are no inefficiencies of calculating any 0 elements. However, the randomized sparsity has the direct opposite effect, where 8 TCAs perform the worst due to long execution latency with little forward progress on useful calculation. The 4×4 TCA has the second slowest execution due to a medium length execution latency while having similar forward progress as each 8×8 TCA invocation. This generates a mirroring effect where dense matrix TCA performance order is opposite that of sparse matrix TCA performance, as well as a mirroring effect within the highest sparsity matrices from ideally-blocked sparsity

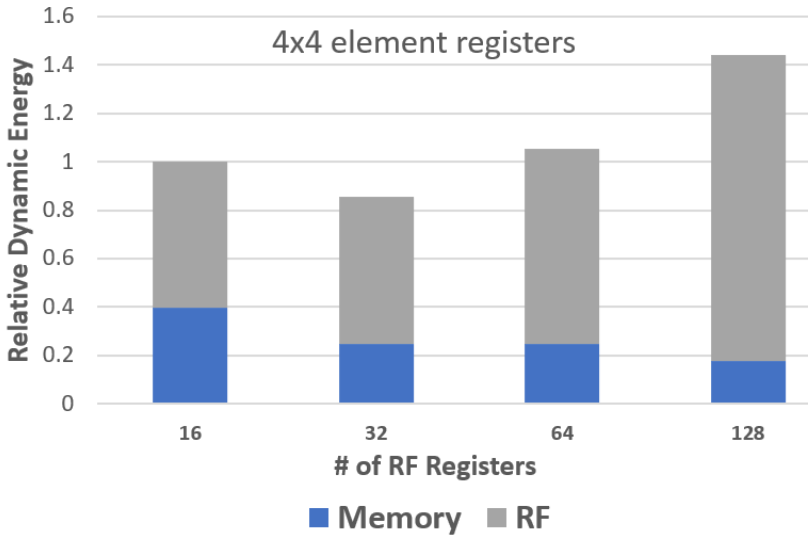


Figure 5.9: Normalized energy breakdown for different size specialized register files. Increasing the register file size reduces memory accesses at the cost of increased energy consumption per register file access.

compared to random-sparsity.

We next shift our focus to energy analysis. We first look at the optimal size for the register file to have the smallest energy cost. Larger register files increase the energy cost per access, and eventually that energy cost outweighs the energy saved by preventing memory accesses, as confirmed by Figure 5.9. Moreover, for the 4×4 elements of double-precision floating point units, 32 registers is the optimal size, which also holds for single and half-precision elements. To give the best case for RF designs, we choose this optimal point for comparison against NoRF implementations for the rest of our energy discussions.

Figure 5.10 shows that in DGEMM, even the unoptimized (non-caching) NoRF implementation can sometimes have roughly equal or lower energy consumption than the RF design, despite requiring more memory accesses (4×4 32-bit, and $4 \times 4/8 \times 8$ 16-bit). This is even after we give the RF an ideal sizing and algorithmic tiling. The optimized NoRF with M$ implementation reduces the memory accesses to the same amount as the optimal RF implementation, while having much smaller tag check and data movement penalties than RF reads and writes. Regardless of data type (64 bit, 32 bit, and 16 bit), the NoRF with

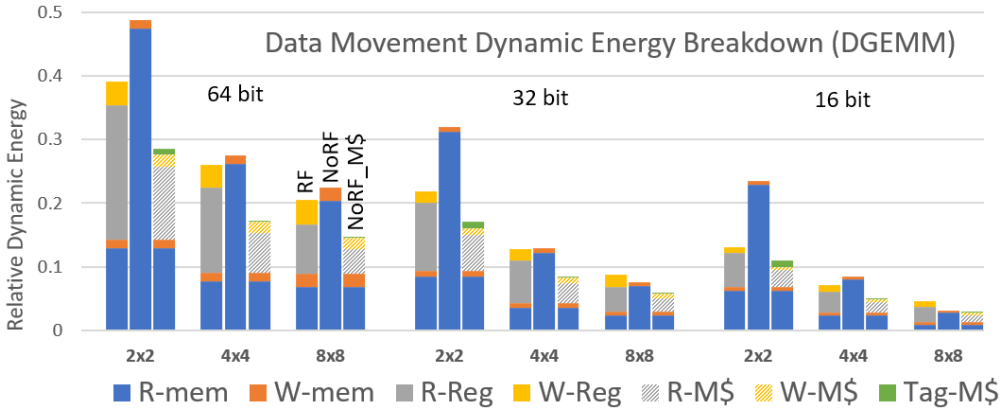


Figure 5.10: DGEMM energy breakdown of data transfer for RF, NoRF, and NoRF with M$ TCA implementations. The M$ has the same memory reference reduction and more efficient reads and writes over the RF design.

M$ implementation has a lower energy consumption than optimized RF implementations. Figure 5.10 demonstrates up to a 39% reduction in energy consumption of data movement by completely eliminating the register file, and instead implementing a specialized TCA datapath with a matrix cache. This energy gain is even after we optimistically reduce the likely required 3-read, 3-write register file (in order to get full data transfer between a Sunny Cove-like LSQ and the TCA) to 2-read, 1 write required for TCA throughput. This would only be possible with additional buffering during bursty memory access. Without this optimistic reduction, energy savings could be even higher.

It is also important to note that the cost of accessing the tags for the M$ does not change with data width size because the addresses do not decrease size, even when the data width does. This means that the relative overhead will be larger for smaller operands (such as 16-bit). The larger TCAs are invoked less often, so the absolute energy also decreases with larger TCAs. We note that in all TCA sizes and data widths, these fixed energy costs are small enough to still provide energy savings over RF-based designs.

By looking at dense matrix-matrix multiplication, we are evaluating an algorithm that

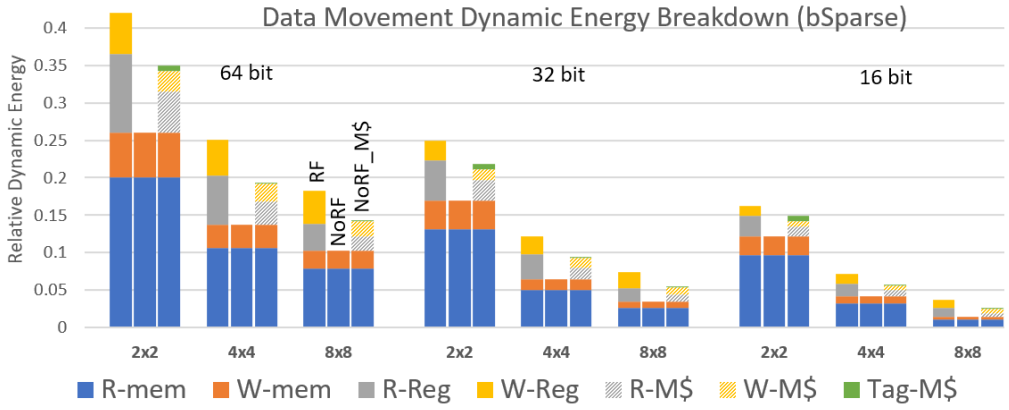
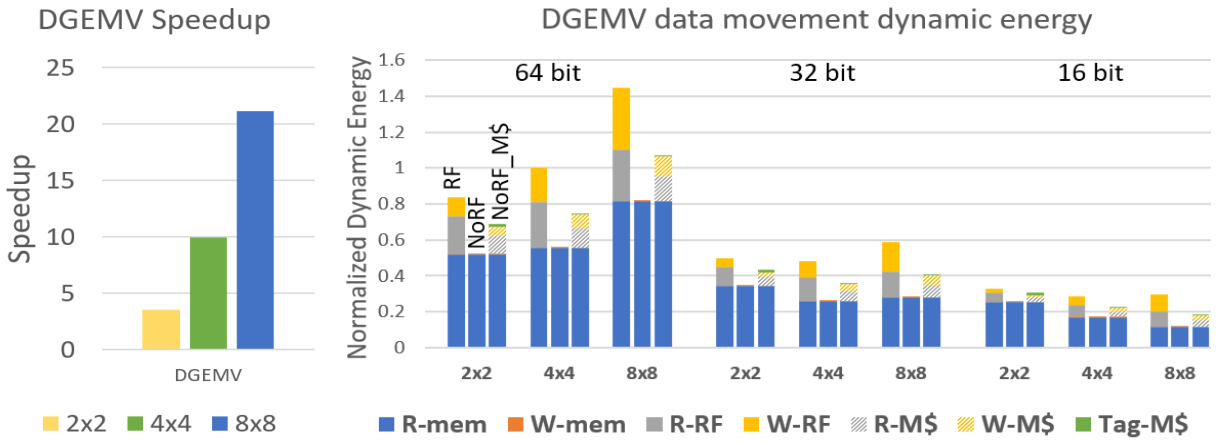


Figure 5.11: Block SpGEMM energy breakdown of data transfer for RF, NoRF, and NoRF with M$ TCA implementations. Since neither the RF nor M$ can capture reuse, NoRF with the M$ disabled demonstrates the lowest energy consumption.

inherently has the potential for lots of reuse. Without reuse, it just adds another step in the data movement process, and does not add extra functionality. Since this is the case, both sparse matrix multiplication and matrix-vector multiplication should see additional wins.

We expand our energy analysis over different sized data types and dense versus sparse algorithms. In blocked sparse matrix multiplication (SpGEMM), the output element partial products are calculated out of order. For this reason, the output matrices do not exhibit temporal reuse, and do not capture reuse either in the RF or NoRF implementations. This means disabling the M$ provides the best energy wins, up to 61% over the RF design (Figure 5.11). Note TCAs encoded for RF operands cannot disable the RF, consuming unnecessary energy in non-beneficial data movement.

We can also see from both Figure 5.10 and Figure 5.11 that the larger the TCA (8×8), the more inherent energy reduction there is in data movement. This is because the number of partial products grows $O(n^3)$. In other words, the more work the accelerator can do per access, the smaller the inefficiencies in data movement.



(a) DGEMV performance

(b) DGEMV TCA energy breakdown

Figure 5.12: DGEMV TCA performance and energy consumption.

Due to limited reuse, matrix-vector multiplication (DGEMV) also has similar energy savings as block SpGEMM in NoRF with and without M$ (Figure 5.12b). There are two main differences in the energy graph compared to block SpGEMM energy breakdown graph. The first is the reduction in memory writes due to algorithmic differences (writing to a vector rather than entire matrix). The second is the trend over various TCA sizes due to the computational efficiency for each load. Just as reuse captured within the RF increases RF efficiency, algorithmic efficiency of computation per memory access increases memory energy efficiency. GEMM has $O(n^3)$ computation operations, with GEMV only having $O(n^2)$. This means that GEMV wide loads will only reduce number of accesses to a given matrix by $O(n)$, whereas GEMM reduces number of accesses by $O(n^2)$. CACTI modeling estimates that doubling access width for larger bit widths increases energy per access by slightly more than $2\times$, increasing dynamic energy. However, for smaller bit widths, CACTI estimates slightly less than double the energy per access, showing energy efficiency increases. The main takeaway from Figure 5.12 is that the M$ does not capture reuse in the DGEMV algorithm, and maximum energy reduction occurs by using NoRF without the M$ enabled. The DGEMV geomean energy reduction for NoRF with the M$ is 21%, and NoRF without M$ has a geomean energy reduction of 41%.

5.7 Chapter Summary

In this chapter, we evaluated operand delivery options for tightly-coupled matrix-matrix accelerators attached to a high-performance out-of-order processor. We looked at ways to optimize RFs by reducing overheads and capturing the most possible reuse within the RF.

We also considered removing the RF from the datapath entirely by having the TCA access the memory hierarchy directly. We showed that using register file for TCA execution adds extra (and often unnecessary) steps in the data movement process and can be avoided in NoRF (no register file) implementations.

From the energy perspective, the cost of multi-ported RF access can exceed the savings from eliminating memory accesses; meanwhile, all reuses captured by the register file can be more efficiently captured by a small *matrix cache* (M$) embedded directly inside a NoRF TCA. We also demonstrated that programmability also favors NoRF implementations. This is because allowing the TCA to capture reuse dynamically simplifies programmer and compiler's burden to statically determine potential reuse. NoRF implementations supporting snooping or other coherence mechanisms also eliminate the complexity and performance degradation that comes with synchronization. This provides an inherent advantage for NoRF designs on practically all fronts, all the while achieving the performance of its more power-hungry RF counterpart.

6 BUILDING TIGHTLY-COUPLED ACCELERATORS SECURELY FROM THE GROUND UP

Regardless of the potential speedups and benefits that come with a TCA design, it will not become a commercially or practically viable addition to CPUs unless it can be verified for security. Security vulnerabilities can be found in either software or hardware bugs. Since TCAs are a new class of accelerators, it is important to consider the potential security vulnerabilities before deployment. In this chapter, we focus on recent known hardware vulnerabilities and evaluate them in the context of TCAs. We determine that if the designer is not careful, TCAs have the potential to amplify existing attacks or create new attack vectors. We also provide proposed solutions to close these vulnerabilities without significant changes or overheads to real TCA designs.

6.1 Chapter Overview

Fixed-function, tightly-coupled accelerators (TCAs) provide application- or domain-specific performance and energy improvements for high-performance, out-of-order processors. Despite careful design, it is likely that future TCAs will expose new security attack surfaces since they often operate in the realm of speculative execution as motivated by Chapter 4. Adding reconfigurability to TCAs broadens their scope, enables post-silicon enhancements, and improves utilization of the TCA substrate, but, when coupled with an aggressively speculative processor, raises previously unexplored serious security concerns. In this chapter, we study two classes of attacks—timing-based and speculation-based—that are

enabled and/or amplified via poor design or malicious reconfiguration of the TCA. To mitigate this issue, we propose a set of invariants for reconfigurable designs that effectively disable these attacks within the TCA, and describe how these invariants can be enforced as part of a trusted synthesis flow. The same invariants can also be applied to the design of fixed-function TCAs, mitigating security risk. We show that incorporating these mitigations in a family of TCAs incurs negligible area and power overheads (<0.5%) with acceptable performance degradation (<2.5%).

6.2 Motivation

Proposed accelerators have a fixed design that show large performance benefits, but with slight modifications or errors in design could also significantly speedup malicious workloads such as Spectre- [51] and Meltdown-based attacks [56]. For example, these attacks use either prime and probe (PP) [67] or flush and reload (FR) [92] algorithms to use the cache as a side-channel. Currently, FR and PP based attacks serialize their accesses with barriers to get accurate timing information on cache accesses. The serialization is currently used to get accurate memory reference timing despite out-of-order execution and miss-under-miss latency hiding. Any mechanism that allows multiple timers or relative information of multiple accesses could greatly speed up these attacks.

A malicious, buggy, or even correctly functioning TCAs could speed up these attacks. As an example, a matrix-matrix multiplication (DGEMM) TCA, as benign as it may seem, might be used by an attacker to expedite the memory loading process for the PP and FR attacks. If the TCA implicitly or explicitly encodes state regarding the timing or ordering

of its memory operations (whether malicious or not), an attacker may be able to parallelize PP and FR based secret extraction. This may be done through detecting ordering of partial product calculations, explicit timers for load instructions, or a debugging register holding information on cache misses.

An even more dangerous attack vector on reconfigurable TCAs is the ability for it to hold on to speculative data. CPUs currently go through vigorous security testing to prevent leakage of information. However, allowing reconfigurable hardware without that level of validation could lead to catastrophic bandwidth of speculative data leakage. For example, if a Spectre-like attack used a DGEMM TCA to access the secret information, multiple 64B cache lines of data is loaded per invocation. If the TCA fails to clear its internal registers, the TCA now holds orders of magnitude more secret information than the original Spectre-based attack without the need to even use PP or FR to extract the information.

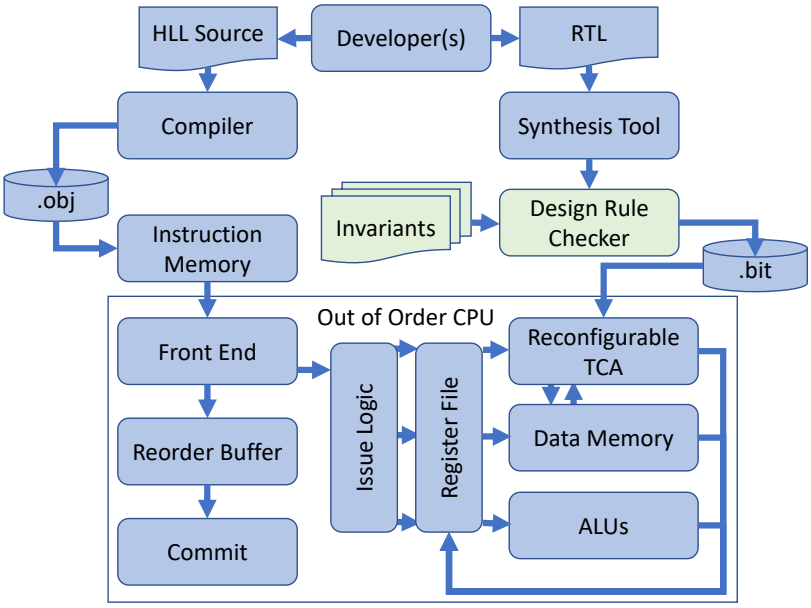


Figure 6.1: Overview of a reconfigurable TCA design and its operation. The reconfigurable bitstream is validated by the design rule checker that enforces security invariants before being loaded onto the substrate within the CPU.

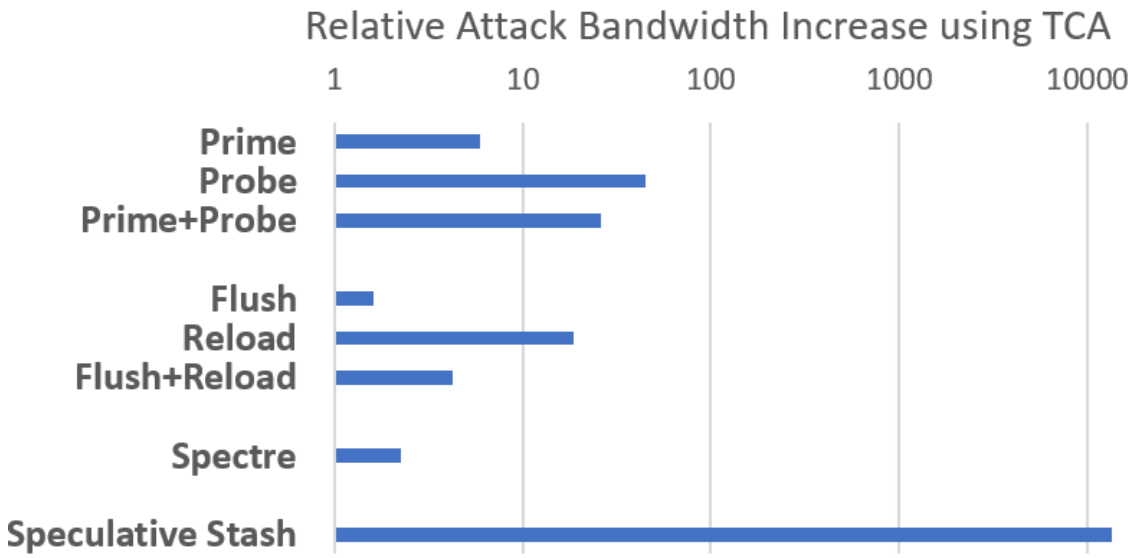


Figure 6.2: Peak attack bandwidth increases using unsecure reconfigurable TCAs.

However, as discussed in Section 3.4, reconfigurability has the potential to provide large benefits for many different types of current and future workloads. For this reason, we consider a system like the one in Figure 6.1[1] and propose security invariants for a Design Rule Checker (DRC) to enforce. This design allows RTL code to be mapped and integrated into the CPU to be invoked via ISA instructions within a compiled binary.

A summary of these attacks can be shown in Figure 6.2. From this figure, it is absolutely clear that TCAs without any limitations on reconfiguration has the potential to increase bandwidth of security vulnerabilities by large amounts. This section also showed how attackers could even use non-maliciously-designed TCAs to increase attack bandwidth, motivating a need to prevent these side-channels from existing in the first place.

In this work, we are the first to evaluate timing-based and speculative-based attacks on reconfigurable tightly-coupled accelerators. We present a new class of speculative attacks called *speculative stashing* that can cause orders of magnitude of attack bandwidth over

[1]Mikko Lipasti helped in the creation of this figure.

existing Spectre-based attacks. We provide solutions at the gate and accelerator level to close these potential exploits, and evaluate accelerators built using these safety mechanisms.

6.3 TCA Attack Model

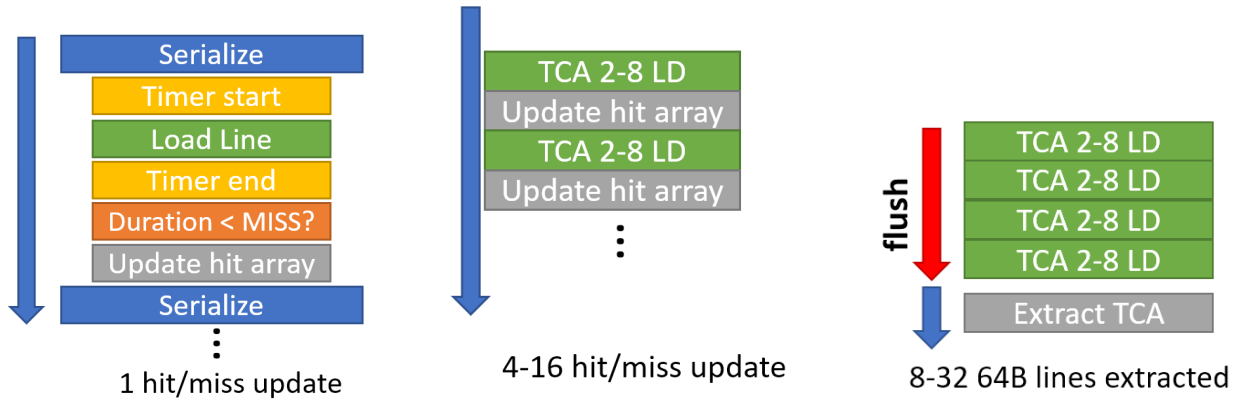
In this section, we look at a set of attacks that could be employed against reconfigurable TCAs. Spectre and Meltdown variants further exposed the awareness of the architecture community of any hardware state that is modified based on speculative operations. Hardware, (TCAs included), that are exposed to speculative data are inherently a possible side-channel that could be exploited by malicious programs. In this section, we expose some of the potential risks that come by allowing TCAs to access speculative data. We show proof of concept attacks that could take advantage of vulnerable accelerators and have attacks with orders of magnitude more bandwidth than existing Spectre-based attacks. Although this may not be an exhaustive list of potential attacks used against a reconfigurable TCA, we focused on the root causes to speculative- and timing-based hardware attacks. Currently, most Spectre variants work through leaking of speculative data, and extracting the data through timing-based side channel attacks.

The first vulnerability we address is the ability for timing to be tracked concurrently for multiple memory accesses, allowing parallelization of timing-based attacks. The second vulnerability is for memory accesses to communicate with one another to determine execution ordering, also enabling concurrent timing-based memory attacks. The third vulnerability is that there is no guarantee of speculative information being discarded upon squashing of an instruction.

6.3.1 Timing-based Attacks

Prime + Probe (PP) [67] and Flush + Reload (FR) [92] attacks use the cache hierarchy as a potential side channel to leak information about the execution of a victim program. The latency of memory references is dependent on the contents of the L1, L2, and L3 caches. Since these structures are shared between processes within a core and other cores within the processor, the cached contents of a program's execution are dependent on the other processes running concurrently. An attacker can exploit this fact to learn information about other processes running despite the desire for completely isolated processes. For example, by using a Flush + Reload attack, the attacker can flush shared library functions from its cache, and access it later. If the later memory reference returns quickly, denoting a cache hit, the attacker knows that the victim process(es) have accessed this function. It is not too hard to imagine ways for an attacker to infer memory patterns about supposedly "isolated" processes. An attacker can also learn information from speculative instructions based on how cache contents were speculatively modified, which is the basis for Spectre and Meltdown attack data extraction.

The timing of cache accesses is measured sequentially, where each load is separated with serializing instructions. Serialization is required since out-of-order execution would overlap consecutive memory references, making it difficult or impossible to differentiate the latency of each individual load. By serializing the out-of-order core for each load, an accurate timing measurement is possible, enabling the attacker to determine hits and misses reliably (see Figure 6.3a). However, the lack of concurrency decreases the bandwidth of the attack.



(a) Traditional Probe/Reload (b) Probe/Reload with TCA (c) Speculative Stashing

Figure 6.3: Program flow for (a) traditional probe/reload extraction. (b) shows how the TCA can speed up the attack bandwidth by removing serialization and timing comparison logic. (c) shows speculative stashing, a method of loading entire cache lines in speculative state, and extracting during correct path execution after TCA fails to clear speculative data.

TCA vulnerabilities to timing-based attacks

Although timing-based cache side-channel extraction is still possible through most of modern day's CPUs, this does not mean that accelerator designers do not have to build a defense against these attacks. If the attack's effectiveness could be significantly increased through the use of a TCA, this could prevent adoption of a design from emerging in commercial products. However, with the TCA, if a timer is attached to every load within the TCA (in the most insecure case), or even if information is leaked about relative timing of order of completion, the attacker can learn information about the hit/miss status of multiple loads concurrently, allowing the attack to be parallelized. We note that a malicious accelerator designed to speed up cache-based timing attacks may not appear much different from a useful TCA design. We consider a general DGEMM TCA as an example. If an 8×8 DGEMM TCA load instruction invokes 8 vector loads per invocation, the TCA may have 8 concurrent loads executing simultaneously. If the same accelerator has a timer attached to

each load, it is possible to have 8 loads concurrently tracking timing information (Figure 6.3b). Further attack bandwidth gains come from removing the need for serialization and reduced control flow, creating dramatic speedup over a serialized PP or FR attack.

Even if timers do not exist for each load, information may inadvertently be leaked about the order of execution in multiple accesses. For a PP based attack, for example, all accesses are expected to be cache hits. If any of the multiple accesses are a miss, this will delay the execution. However, unless the attacker can specify which load(s) missed, the attack must be repeated to determine the exact value that missed. This is the same impact of doing a timing-based attack with multiple loads per serializing instruction to determine whether any of the accesses resulted in a cache miss.

If the TCA contains any state that can explicitly or implicitly determine the ordering of memory access execution (such as the order in which partial products were produced), it may be vulnerable to a timing-based attack, because a single cache miss can be detected from a group of parallel memory accesses. If an attacker can extract this information based on TCA state (or CPU state, for that matter), this would equally allow parallelization of timing information gained per invocation, speeding up PP and FR attack bandwidths.

6.3.2 Speculative Stashing TCA Attacks

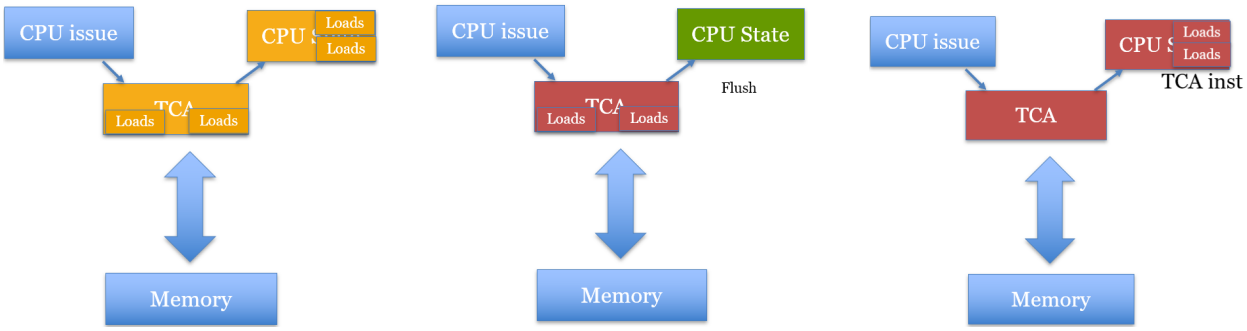
Discovery of the Spectre and Meltdown hardware security vulnerabilities has placed a new focus within the architecture community on secure speculative execution. Ever since the introduction of OoO processors, large verification efforts have been in place for commercial processors to guarantee architectural state such as registers and memory would

not let speculative data persist after an instruction was squashed. Despite architectural state erasing all traces of speculative information, the new vulnerabilities showed that microarchitectural state such as cache contents could be used as a side-channel to leak information.

TCA vulnerabilities to Speculative Stashing

So far, we have discussed the ability for TCAs to amplify existing side-channel extraction attacks. However, a new class of attacks that could be even more dangerous than an amplification of a PP- and FR-based attack is what we coin as *speculative stashing*. As we just discussed, Spectre and Meltdown use the cache as a side channel to detect information read during speculative information through indirection. However, if the state of the TCA is modified during speculative execution and not reverted back to previous states, the values can be explicitly read from the TCA without the need of FR or PP to extract the information. This means that the TCA has the potential to not only increase bandwidth of existing side channels, but also become a new side channel for attackers to exploit.

For example, TCA state that is modified during the course of execution (see Figure 6.4a) now holds information about the speculatively read (secret) value. A buggy TCA with incorrect recovery could allow future iterations to detect the secret value, or a malicious TCA could purposely store multiple secret values in flip-flops (FFs) that are not flushed on recovery (see Figure 6.4b). Speculative stashing is the ability for the TCA to hide potentially large amounts of speculative data that can be recovered after resuming the correct execution path. Although contents of the TCA might seem like microarchitectural state, reconfigurable TCAs allowing arbitrary execution could easily be designed to output



(a) TCA speculative load (b) TCA instruction flushed (c) Future TCA instruction

Figure 6.4: Speculative stashing attack summary. Green denotes proper (safe) committed state, yellow denotes speculative in-flight state, red denotes persisting or committed flushed speculative state. Speculative loads (a) are issued by a TCA instruction. Despite the CPU reverting state on the flush of a TCA instruction (b), the data persists within the TCA's internal flops. A later TCA instruction (c) reads the persisting values from the TCA and persists in the CPU after it commits.

internal microarchitectural state to future instructions (see Figure 6.4c).

Modifying state during speculative execution is not a security issue until the state is not correctly recovered during the squashing of the instruction. For the entire CPU pipeline, verification is done to ensure proper squashing of data in speculative paths. However, unless the same level of verification is done with the TCA in relation to all the interactions with the core, it becomes difficult to have confidence that the TCA will not retain any speculative information as a new side channel. This becomes even more difficult to enforce when the TCA can be reconfigurable. In the case of a buggy TCA, it is possible for a side channel to not be detected during verification, and later causing security issues within the TCA. In the case of a malicious TCA, the TCA need not even bother to revert when the instruction is squashed.

The impacts of the attack can be abundantly clear with a malicious TCA design toy example that has two modes of operation: speculative loading, and content dumping (Figure 6.3c). During the speculative loading instruction, the TCA can store speculative

loads into its internal FFs. On instruction squashing, the CPU restores all architectural state, but a malicious designer allowed to create arbitrary circuits could simply ignore the flush signal. Later, during correct path execution, a content dumping TCA instruction is allowed to issue, and could write results into CPU architectural state such as registers or memory, leaking all flushed speculative data. This enables data leak rates close to the TCA's memory bandwidth.

6.4 TCA Security Defenses

To address each of the security issues presented above, we present a set of design properties of TCAs that would defend against each attack. In order to enforce these properties, we provide a set of security invariants that have implications to maintain those properties. For timing-based attacks, we assume that a fully synchronous design without multi-cycle paths or combinational feedback loops cannot maintain timing information without an active clock signal. We enforce rules about when clocks should be gated and how data is allowed to transfer to eliminate timing data from being captured during specified events (such as memory accesses). For speculative based attacks, we assert that speculative values should not persist beyond its instruction being flushed. If all FFs within the design are reset upon an instruction flush, are not forwarded to older instructions, no combinational feedback loops exist, and all younger instructions are replayed, data and calculations made within a speculative path will not persist after the flush. We denote *lanes* to be circuits of independent work within the TCA, and assume a pipelined designs to pass data from one lane to another. We describe these defenses in detail below.

6.4.1 Eliminating Timing-Based Attacks in TCAs

As the name suggests, timing-based attacks use the passage of time to infer and leak information to an attacker. This requires a means for measuring the passage of time, which can be something as simple as a synchronous up-counter. In a synchronous design, state changes only occur on clock transitions, which can be inhibited via clock gating. Hence, we can prevent the TCA circuitry from measuring time by inhibiting its clock. Although clock-gating is typically used for energy reduction [89, 5, 17], we use it as a means to eliminate timing monitoring within the TCA. We call each independent path/item of work a "lane" of execution, where each lane has its own clock-gated domain.

Assumption 1: A TCA must be a fully-synchronous design that meets minimum clock period requirements and has no multi-cycle combinational paths or combinational loops.

Implication – All combinational logic completes in less than 1 clock cycle and holds constant until FF values are changed. By remaining constant, the combinational logic is not able to monitor the passage of time regardless of the number of clock cycles the FF values are held.

Invariant T1: The state of a FF cannot be modified while its clock is inactive.

Implication – Timing cannot be monitored by the flop while its clock is inactive. The FF has no way to update or change state to be dependent on the duration the clock value is inactive. Tied with assumption 1, if the input to the FF (other FFs or combinational logic) also cannot monitor time, the state of the FF cannot record timing information while the clock is inactive.

Invariant T2: A TCA design can have one or more lanes. Within each lane, all FFs are

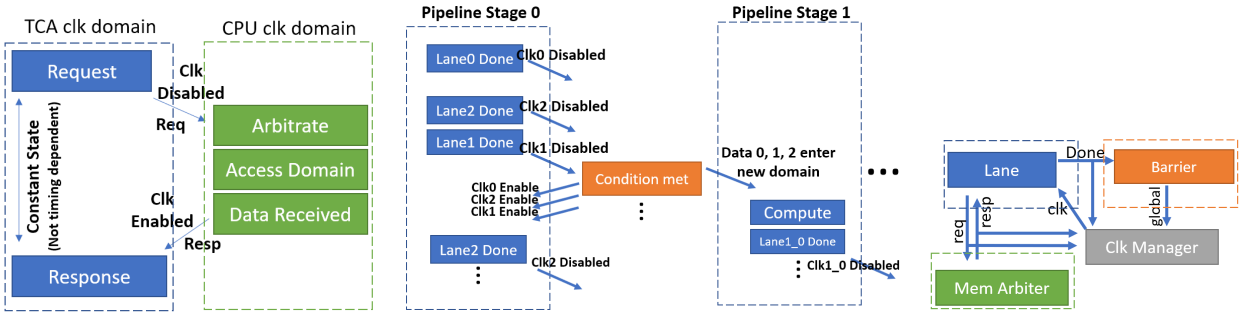
subject to the same gated clock signal.

Implication – Everything within a lane is oblivious to the passage of time whenever the clock is inactive.

Invariant T3: When two or more lanes synchronize (e.g., exchange data), they must gate their clock until synchronization completes.

Implication – No timing information is leaked when crossing lanes, or when crossing to or from the TCA to the CPU core or memory system.

Discussion. A simple TCA may consist of a single lane. Subject to Invariants T1-T2, it cannot measure time, so it is not capable of differentiating cache hits from cache misses, or estimating contention for any external shared resource, since it must gate its clock whenever it crosses a lane boundary to interact with the memory system. A more complex TCA may incorporate multiple lanes with their own separately gated clocks to enable concurrency, but Invariant T3 guarantees that no lane is able to measure the timing of any other lane. So, by induction, no timing information can be collected anywhere in the TCA. The following discussion elaborates on these arguments.



(a) Memory requests (b) Example pipeline data passing (c) Clock Manager

Figure 6.5: Data passing between lanes for (a) memory requests, and (b) pipeline stages (collecting data from previous 3-lane pipeline stage) through time. (c) incorporates these invariants into a trusted TCA clock manager module.

For this discussion, memory is the main timing mechanism that is of concern, so any memory accesses must go through a memory block that disables the clock of the requester (Figure 6.5a). Clock-gated lanes have clock signals that are connected to a trusted clock manager. When a request to memory is made, the clock manager will disable the lane's clock until the result returns. The DRC can enforce that the memory module clock gate signal is tied to FFs in the lane. Note that although we focus mainly on integrating with CPU memory in this example, this method can be utilized when crossing other CPU structures (Register Files, flags, internal signals, etc.) or TCA structures (shared ALUs, state, memory, etc.). Unlike many cache-based timing defenses, using this framework defends against timing-based attacks for all potential shared resources, and is not memory or cache specific.

Lane-level clock-gating. Naively, to enforce **Invariant T2**, the entire TCA could be designed as a single lane that is clock-gated during memory operations, eliminating the notion of time for the entire TCA. However, one of the main performance benefits that accelerators provide is multiple aspects of concurrency [61], and eliminating all concurrency during memory operations could eliminate most of the benefits from the TCA to begin with. However, we create optimizations to allow completely independent work to continue forward progress. The potential issue of having a new clock-gated lane is that it is able to keep track of time even while a different lane has a gated clock. However, by being completely independent, no data signals from one parallel lane can feed another lane. By enforcing that no lane contains inputs from any other parallel lane (e.g., a different gated domain), timing information can never be conveyed from other lanes. The DRC can enforce this by having each lane listed as a separate module, with no input ports coming from any other lane.

Pipeline-level clock-gating. However, it is often desired for parallel computation to eventually be aggregated, such as through a reduction stage. Passing information from one lane into another would break **Invariant T3**, potentially leaking timing information. This could happen if a lane had an up counter waiting for a different lane's memory request to finish, and latching the result once the memory output changes. Similarly, a counter could be added at the end of each lane to detect the number of clock cycles that have elapsed between completion and entering the next pipeline stage. We address **Invariant T3** by requiring all data to go through a synchronization barrier that clock-gates the lane before data can cross the boundary (Figure 6.5b). In effect, each lane is basically forfeiting its clock before it has an opportunity to see another lane's contents or even know if it has completed. This prevents one lane from updating FFs (and monitoring time) while parallel lanes finish executing. In the end, no parallel lanes will be able to infer any timing information about its execution. Here, a memory request made by a lane will clock-gate the entire lane, but not any other lane, since they are guaranteed to be independent. Neither the lane making the memory request nor any other parallel lane can determine the execution time of a memory request or detect hardware contention. The DRC ensures that a lane's clock signal is connected to clock manager (Figure 6.5c), and all FFs within the lane are clocked by that same signal.

6.4.2 Defense of Speculative Stashing in TCAs

One of the possible designs to prevent speculative stashing would be to delay execution of TCA instructions until it becomes non-speculative (i.e., the head instruction of the reorder

buffer, ROB). However, this can cause performance penalties by delaying TCA execution [74] as also described in Chapter 4. If the TCA designer wishes to avoid these penalties, an alternate solution would require guarantees for speculative data to not persist beyond the squashing of a TCA instruction.

Assumption 1: The CPU correctly does not modify memory or register values in a way that they persist after a flush. This is already a requirement for correct execution for all OoO processors, typically through register renaming and memory writes delayed until commit. After TCA instructions are flushed, younger instructions are replayed.

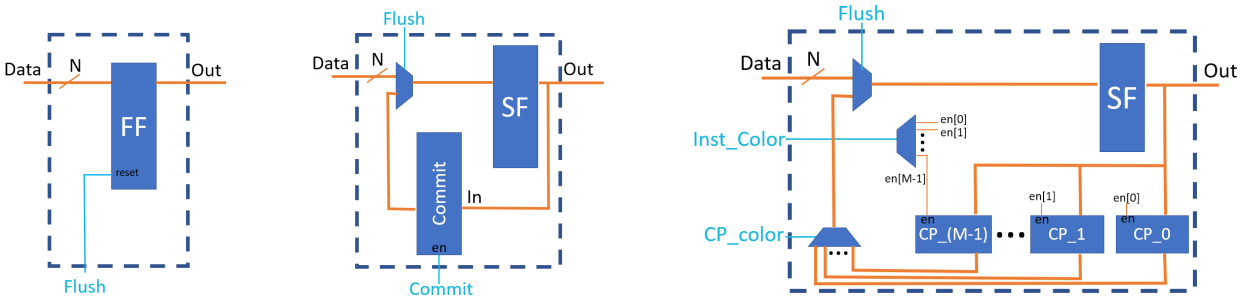
Invariant S1: TCA State changed by a speculative instruction must be reverted when the instruction is flushed.

Implication – Speculative changes from a flushed TCA instruction will not persist for *future* instructions that read TCA state.

Invariant S2: TCA state committed by an older instruction cannot reflect any changes from a flushed younger instruction (information cannot flow in the reverse direction).

Implication – TCA instructions that are squashed will not be able to share their state with any *previous* instructions. This prevents TCA instructions that will eventually commit from containing information dependent on flushed TCA instructions.

It may seem like flushing all FFs of the design upon misspeculation would prevent leakage of speculative information (Figure 6.6a). However, there are multiple problems with this approach. First, the clearing of data actually breaks **Invariant S1**, as the clearing of data allowed the speculative instruction to change (by clearing) the output of the FF. This leaks 1 bit of information, showing that the instruction was flushed. Second, requiring all FFs from a TCA to be cleared on squashed instructions would make it impossible to



(a) Flushable FF (unsecure)

(b) Stateful FF (single invocation)

(c) Stateful FF (multiple invocations)

Figure 6.6: Flushable flops (a) clear speculative data but do not revert state, and is therefore unsecure. Trusted flip-flop designs for various levels of TCA execution desired include: (b) preserving state but only allows new TCA invocations after commit, and (c) preserving state with multiple in-flight TCA instructions.

implement various accelerator designs. This is because some algorithms require state (from state machines, buffers, queues, etc.) to persist between invocations. For example, heap manager TCAs gain speedup through the use of internal LIFO structures holding memory pointers. Heap manager TCAs that are always consistent with memory's depiction of the heap may suffer significant performance penalties by flushing every misspeculation, while TCAs that perform lazy updates for added performance benefits could cause correctness problems if all LIFO queues are cleared before they are written back to memory. In order to allow these designs to exist, we propose trusted FF designs that enforce correct reversion on flushes to allow state to persist in TCA designs.

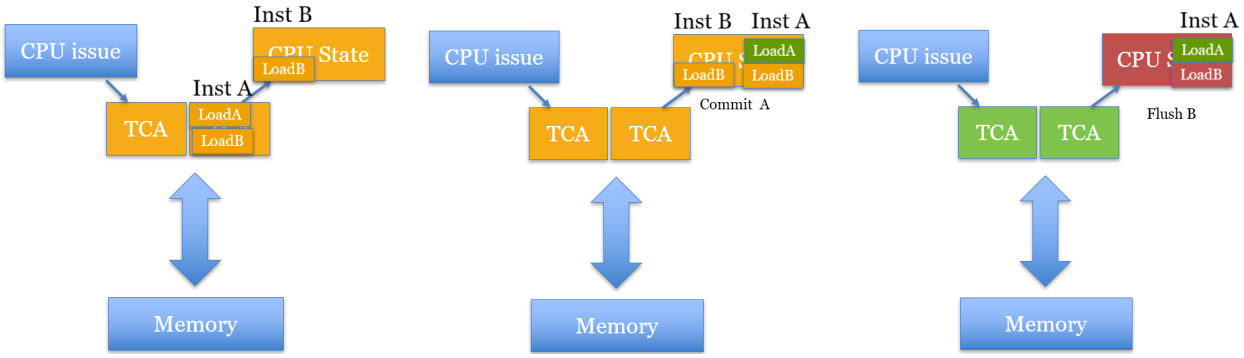
FF-level reversion: Data can be stored through the use of FFs and latches. Existing SystemVerilog syntax uses 'always_ff' blocks to denote the use of FFs and can detect combinational loops and inferred latches. To start the discussion, we start with a simple case of a TCA allowing a single in-flight instruction. If instead of using 'always_ff' blocks the TCA designer used a trusted FF module, all FF within the design will always be reverted

to the last committed state during instruction squashes (Figure 6.6b). The output of the FF is upgraded to the committed FF state upon receiving the commit signal from the CPU. This enforces **Invariant S1**, making it so the TCA's FFs cannot persist speculative data beyond the flush signal.

Note that these FFs are still connected to the flush signal, but instead of being completely cleared, the FF reverts to the last committed value. Each of these FFs will latch the FF output at completion of the pipeline stage and be loaded into the FF's committed state once the CPU designates the instruction as committed. This requires the Flush manager to also become a Commit manager, to signal to pipeline stages once instructions have committed.

Trusted FFs can also be expanded to allow multiple in-flight speculative TCA instructions to execute simultaneously at additional hardware cost of storing values for the FF at every instruction's instance (Figure 6.6c). If using a pipelined TCA design, the Flush Manager can keep track of instruction pipeline location and issue flush signals to the appropriate pipeline stages. Both of these stateful FF designs follow **Invariant S1**.

In order to enforce **Invariant S1**, the Design Rule Checker (DRC) can verify that all FFs within each lane use the trusted FF modules instead of general 'always_ff' blocks and combinations feedback loops/latches. Note that synthesis tools already issue warnings for combinational feedback loops and combinational latches, which could simply be changed into DRC errors. Secondly, the DRC can also confirm that all FFs in TCA "lanes" of execution are connected to the lane's flush and commit signals. Each lane's commit and flush signals are connected to the trusted Flush/Commit Manager block, which is directly integrated with the CPU's squash and commit signals. By passing all of these DRC checks, each TCA FF is guaranteed to be a trusted FF that will automatically be reset to the last committed



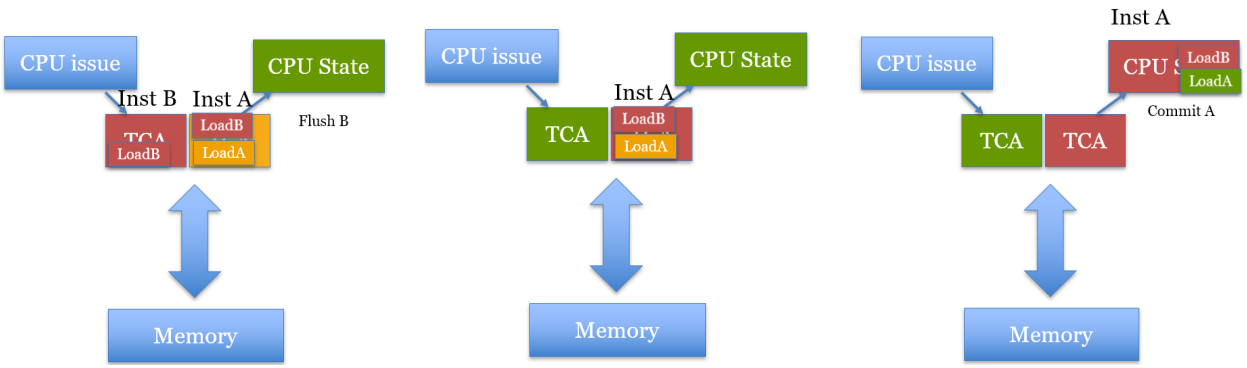
(a) Inst B left trail for A (b) TCA Inst A commits (c) TCA Inst B flushes

Figure 6.7: Out-of-order TCA issue speculative attack. Younger instruction B issues first and leaves data in TCA pipeline that instruction A can see (a). Instruction A can commit once it becomes the head of the ROB (b), committing state from the load from instruction B, which eventually flushes (c). Invariant S2 can be enforced by preventing the commit of A (b) until instruction B is non-flushable.

state upon CPU flushes. Without these rules, an arbitrary TCA design could maliciously or accidentally leave FFs modified from speculative data, potentially leaking speculative-based information.

Enforcement of **Invariant S2** prevents speculative state form flowing backwards with respect to program order. Backward flow can occur if TCA instructions are issued out of order, and a younger instruction changes TCA state which is then read by an older instruction. If a mispredicted branch separates the two TCA instructions, only the younger is flushed and the older can illegally commit speculative state generated by the flushed instruction (see Figure 6.7). Even if in-order issue is enforced, a malicious forward bypass path in the TCA can forward data from a younger TCA instruction to an older one, leading to the same illegal result (see Figure 6.8). However, dataflow analysis by the DRC can guarantee that information only flows in a forward direction.

The simplest solution for the out-of-order scenario is to segregate all FFs and memory in the TCA by instruction instance, and prevent any communication across them. The



(a) Inst B forwards to Inst A (b) Flop reversion after flush (c) TCA Inst A commits

Figure 6.8: In-order TCA issue speculative attack. Younger instruction B forwards data to instruction A before it is flushed (a). Even after TCA and CPU revert after flush of instruction B to follow Invariant S1 (b), instruction A commit allows speculative data to persist in the CPU (c) without following Invariant S2.

DRC can enforce this with straightforward dataflow analysis. This approach requires duplicating pipeline latches as well as any memory structures in the TCA, which may be prohibitive, and may also reduce performance in cases such as the heap manager, where fast access to shared queues is critical.

Alternatively, we can adapt the concept of a store vulnerability window (SVW) [72] to TCA instructions. Each TCA instruction is assigned a sequence identifier based on program order, and the TCA tracks the youngest TCA that has issued at any point in time. Any older TCA instructions inherit that youngest TCA timestamp, and delay their commit until that younger TCA is also ready to commit. If the younger TCA instead flushes, then all older TCA instructions are forced to replay after the flush. This ensures that any state conveyed to them by the younger TCA instruction has been restored, hence ensuring that **Invariant S2** holds. Deadlock is not a concern, as TCA identifiers are allocated at dispatch, nor is livelock, as replays are given priority over new TCA instructions.

In our evaluation, we enforced in-order issue to the TCA and relied on the DRC to

ensure that data only flows in the forward direction, hence enforcing **Invariant S2**. In these designs, all FFs are only accessed in a single pipeline stage, so information cannot flow from one stage to another without going through the pipeline stage barrier. This is enforced in the DRC by checking that lane inputs are always from the output of the previous pipeline stage barrier.

6.4.3 Summary of TCA Attacks and Defenses

TCAs have the capability not only to speed up desired workloads, but also have the potential to increase bandwidth of both existing and a new class of security vulnerabilities. However, these vulnerabilities have fundamental causes in timing measurement and persistence of speculative data.

For a circuit to perform a timing-based attack, its output or state must be dependent on the duration of the victim's timing-dependent transaction. Within the lane(s) accessing the victim, all notion of time is lost when interacting with victim structures (e.g., memory) through the use of a clock-gating handshake protocol. This makes the attack condition false within the lane, since all flops within the lane are independent to the duration of the transaction. For external lanes not accessing the victim, the notion of time is lost through the use of barriers that disable the lane's clock prior to seeing any values or signals from other lanes. This means the lane will freeze all output and internal flops prior to seeing any other lane's signals. This makes all external lanes independent to the timing of the transaction from any other lane. By construction, if both internal and external lanes to the transaction are timing independent to the transaction, then the attack condition is false for

all lanes. These can be checked through the use of trusted clock-gating modules and a DRC that checks that the input and output ports are connected to the expected signals/ports. Any lane avoiding or incorrectly connecting the clock-gated signals will fail the DRC.

Speculative stashing is a new class of attack the allows microarchitectural state from the TCA to stash large amounts of data during speculative execution in a new side channel.

For a speculative-based attack to work, speculative transient state from a flushed instruction is promoted into enduring (e.g., committed) state. The flushed instruction could either pass this information to younger or older instructions. In the in-order issue case, our defense prevents younger instructions from committing these changes by restoring all flops and replaying all in-flight younger instructions when a TCA instruction is flushed. Our defense prevents older instructions from committing this state by ensuring only forward-direction dataflow through a DRC and pipeline-based design. This makes the attack condition false for in-order issue TCAs.

In the OoO case, our defense prevents older instructions from committing state by delaying commit until all in-flight TCA instructions are non-flushable. Our defense prevents younger instructions from committing state by restoring all flops and replaying all younger instructions when a TCA instruction is flushed. This makes the attack condition false for OoO issue TCAs.

The DRC can enforce the use of reverting flops by eliminating the use of "always_ff" blocks and instead requiring trusted flop modules that checkpoint and revert state on TCA flush signals from the CPU. Failing to use these flops or incorrectly connecting these flops to the Flush/Commit manager will fail the DRC.

By enforcing these rules, there is now no timing of individual accesses, no timing that

is dependent on any other operations, no timing of how long work has been waiting, and no timing learned about the ordering of parallel memory references. There is also no persistent speculative data upon a TCA instruction flush.

6.5 Evaluation of Sample Accelerators

After imposing these invariants on TCA design, we want to demonstrate that this does not impose significant design constraints during the implementation phase of building TCAs. This section shows block diagrams and describes TCA implementations for the four TCAs we designed using our secure primitives. The flush/commit manager captures the instruction ID of incoming TCA instructions and keeps track of instruction location as dictated by the global barrier advancement signal. Similar to register renaming, each instruction has its own TCA instruction ID. Flush signals for a given instruction sent by the CPU allow the flush manager to send the previous TCA instruction ID to the appropriate pipeline stages with the flush signal. Each pipeline stage will then revert their flops to the correct previous TCA state.

The memory interface arbitrates for memory instructions to be serviced by the CPU. Allocating spaces in the LSQ in the middle of TCA would provide additional challenges, one of which being deadlock scenarios for TCAs with an arbitrary number of memory requests. Alternatively, the CPU can handle requests separately provided that there are no memory conflicts. To address this detection of memory conflicts, we propose using a bloom filter between CPU memory requests and TCA memory requests. Upon TCA dispatch, the bloom filter for the TCA instruction is reset. Any TCA memory access sets

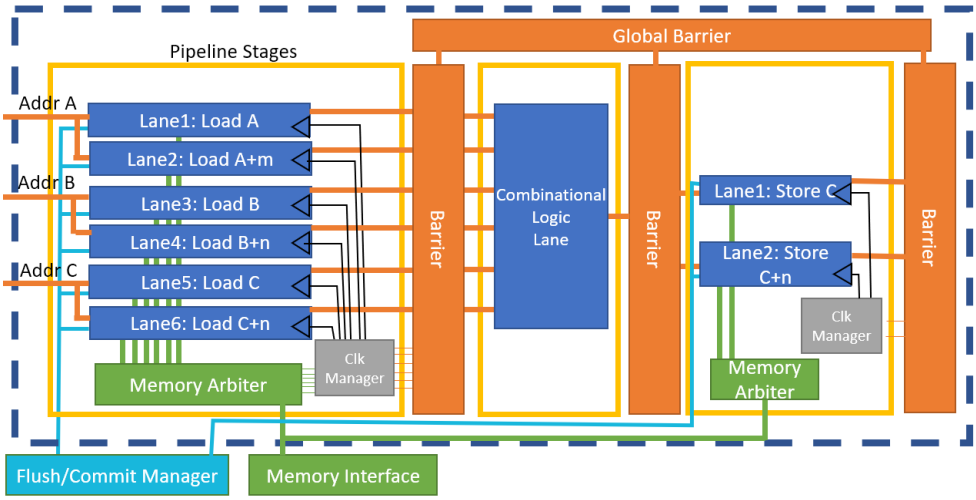


Figure 6.9: Block diagram of 2 × 2 DGEMM TCA using secure primitives.

the appropriate bloom filter entry. All future memory requests that see a set bloom filter entry stalls execution until TCA commit. TCA requests can also check entries set by CPU instructions to issue replay requests for younger instructions that have already executed. Given the regular memory access patterns, it is easy to create bloom filter mappings to avoid any false-positive matches between the TCA and CPU of the TCAs we evaluate.

DGEMM TCA Design – First, we consider the DGEMM accelerator in Figure 6.9. Here we show a 2 × 2 example for simplicity, but the 4 × 4 and 8 × 8 TCA follows the same process. The first section of work that needs to be completed is to load all the elements of the C += A × B submatrices. In order to load a 2 × 2 submatrix, 2 loads need to be issued to the memory subsystem, for a total of 6 loads. The address generation for each load is independent of all other loads, so we can place all independent work into separate lanes, for a total of 6 lanes during pipeline stage 1. Each lane will have its own clock-gated domain, each of which will have its clock disabled immediately when issuing its memory request to the arbiter (even if the arbiter will process its request in later cycles). Each lane has a state machine (using stateful FFs) to control the memory request and lane completion

signals. After the loads are complete, values from multiple lanes are required to calculate DGEMM partial products.

Since using values from different lanes crosses lane boundaries, all lanes must pass through the synchronization barrier (which can be thought of as a pipeline stage), forfeiting each lane's clock until all lanes have signaled their completion. In the second pipeline stage, the partial products are calculated. Since no clock-gating applies (this stage is purely combinational logic), there is no added benefit to adding multiple lanes, as full concurrency can be achieved within this lane, never requiring its clock to be gated. For 4×4 and 8×8 designs, loads and partial product execution can be distributed across multiple pipeline stages to generate a systolic array style execution. The barrier is added to go into pipeline stage 3 to create two new lanes. This stage takes values produced from pipeline stage 2, and issues store requests through the memory arbiter. These are also placed in separate lanes to prevent the first store to disable the clock and serialize the second store.

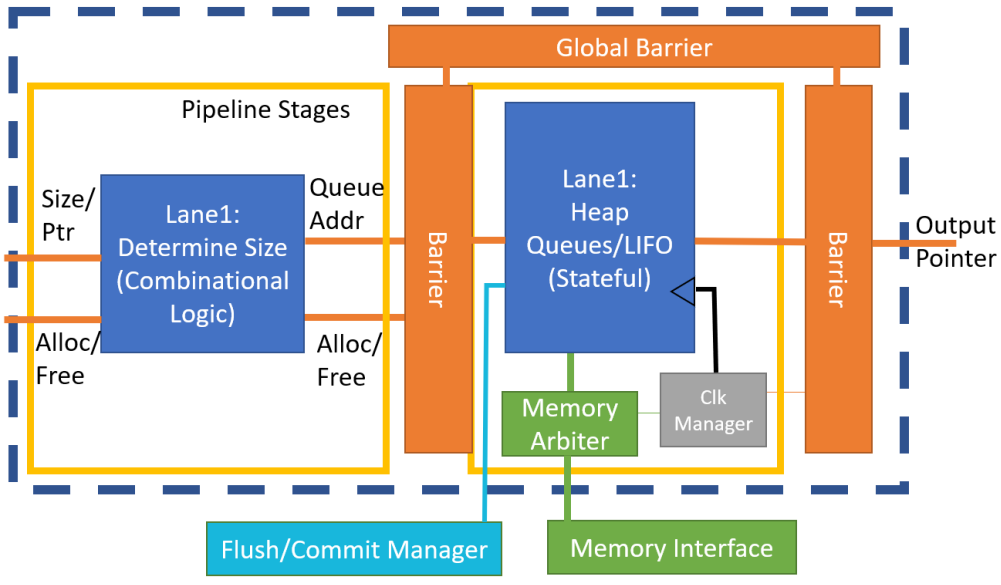


Figure 6.10: Block diagram of Heap Manager TCA using secure primitives.

Heap Manager TCA Design – We use the heap manager accelerator proposal detailed in PHP accelerator proposals [28] as a starting point for our TCA design in Figure 6.10. The first pipeline stage uses combinational logic to determine the correct queue to access.

The second pipeline stage of the heap manager TCA uses the address generated from the first stage as an index into a Last-In First-Out (LIFO) queue. Using a LIFO queue requires having stateful elements, which could naively be implemented using an array of separate trusted FFs for every LIFO bit. We denote that along with trusted FFs, additional primitives can also be added for commonly used stateful structures, such as queues, scratchpads, local caches, register files, etc. Structures besides just FFs can also follow speculative invariants S1 and S2, maintaining the security properties, giving designers more efficient tools to work with. Having a wide range of trusted primitives can increase efficiency of only updating FFs on commit impacted by flushes rather than the entire structure. We built a trusted LIFO queue to speculatively push and pop elements into the queue, and correctly update state on flush and commit signals as an example of an additional trusted primitive. The other three TCAs evaluated did not require any additional trusted primitives.

When a free pointer list within the LIFO queue is empty or drops below a certain threshold, Gope et al. [29] recommends having a prefetcher to fill the queue. Since outstanding requests may share this structure, naturally, the lane will be clock-gated to prevent the possibility of timing leakage across lanes.

Hash table TCA Design – We use the hash table accelerator proposal detailed by Gope [28] as a starting point for our TCA design in Figure 6.11. The first pipeline stage takes a pointer to the key value to access the hash table GET or SET request. This requires a load from memory to extract the key string that will be used to access the hash table.

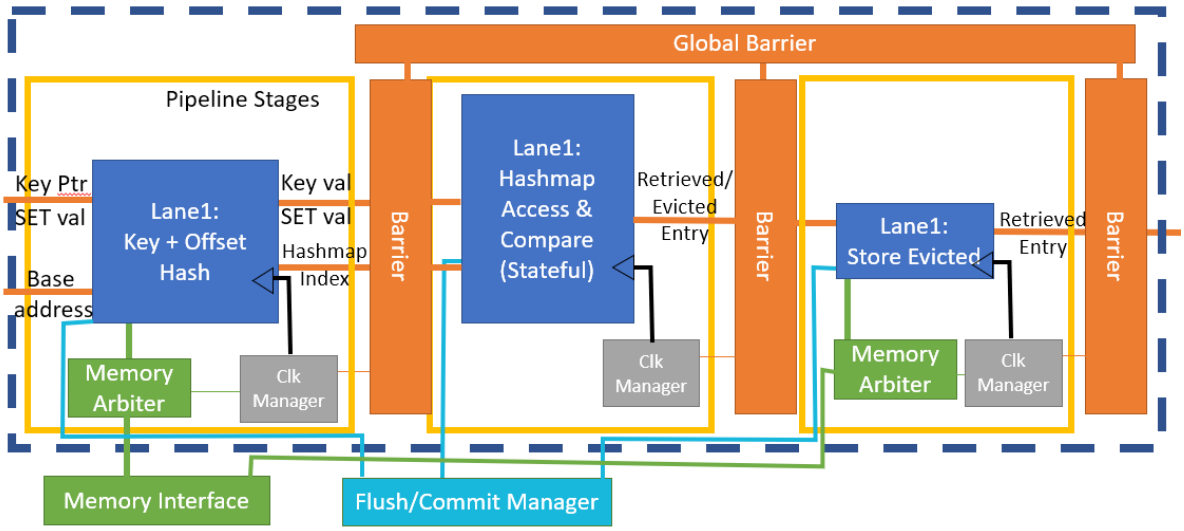


Figure 6.11: Block diagram of Hash Table TCA using secure primitives.

Crossing the memory boundary requires its clock to be gated by the clock manager. Upon receiving the string value, the hash computation is performed. Note that since the hashing is dependent on the string load, this cannot be placed into a separate lane within the same pipeline stage. Although it could be placed in a separate pipeline stage, we chose to place it within the same lane as the load. This means the clock will be gated during the load, but since it is a sequential operation after the load and low latency, this does not have a large impact on TCA throughput to keep it within the same stage.

The second pipeline stage performs the hash table lookup based on the index calculated in the prior pipeline stage. The entries in that index are compared to the returned key value from the prior pipeline stage on GET requests. Matches update the LRU bits and are simply sent to the last stage and immediately set as "done" to pass the result to the CPU. As the prior work proposes, misses fall back to software fetching. For SET requests, the index is calculated through the same process as GET requests. The key and value are written to any empty hash table entry for the hashed index. If all entries are full, the evicted value

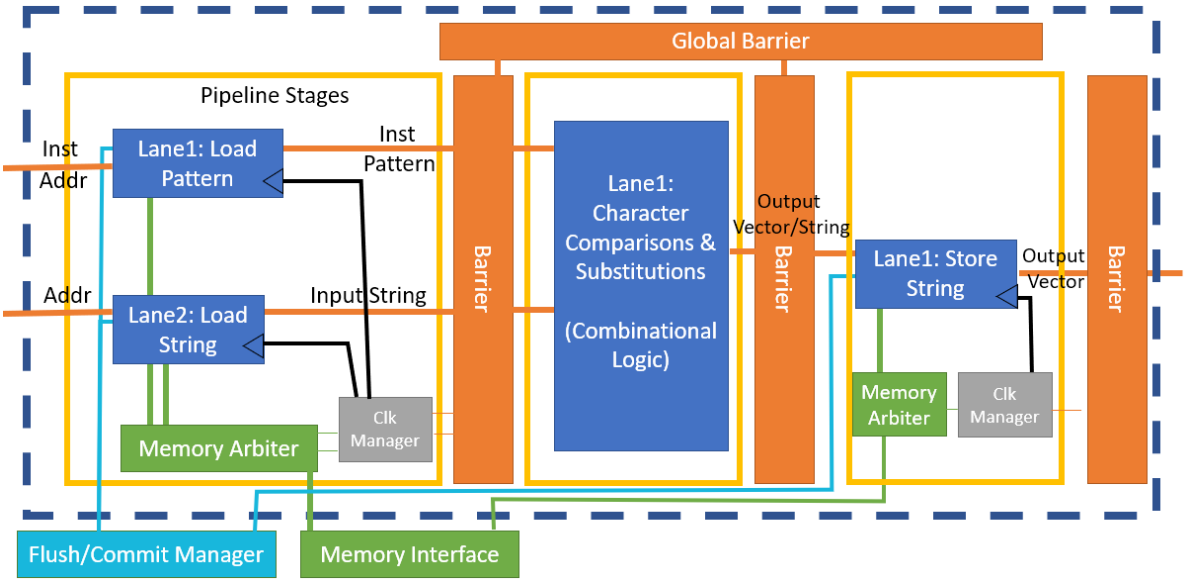


Figure 6.12: Block diagram of String Manipulation TCA using secure primitives.

(LRU replacement) is passed to the next pipeline stage to be written back to the software hash table data structure based on the hash table base pointer.

String Manipulation TCA Design – We create a string manipulation TCA to provide functionality to both string operations as well as supporting regular expression (regex) operations as shown in Figure 6.12. The first pipeline lanes are responsible for loading the input and/or pattern strings for the given TCA string operation. These can be independent operations, so they are placed in separate lanes within the same pipeline stage. The instruction operation (ie. compare, replace, priority encode, etc.) is simply forwarded through one of these lanes since it will be needed in the second pipeline stage.

The second pipeline stage is responsible for taking the input and pattern strings and performing comparisons and replacements depending on the string operation designated by the ISA instruction. The string accelerator supports character matching, character substitution, character range detection (e.g., uppercase or lowercase detection), and priority

encoding matching. Although supported SSE string operations provide speedup for certain applications such as string matching [20], this hardware accelerator provides additional speedup by being able to perform operations on multiple characters concurrently beyond just matching. For example, the accelerator can perform multiple character substitutions in a single invocation rather than through sequential processing.

6.6 Methodology

We begin by evaluating the consequences of having a dense matrix-matrix multiplication (DGEMM) TCA that can detect timing of its memory accesses on PP and FR attacks. The TCA has no difference from a normal DGEMM TCA instruction besides the ability to detect and output a bitmask vector of cache hits and misses in its memory accesses to compute $C \mathrel{+}= A \times B$. The attack could model both a buggy TCA where the ordering can implicitly determine a cache hit bitmask or a malicious TCA where the sole TCA output explicitly returns the bitmask.

We implemented this TCA through the gem5 simulator [6] where a TCA macro-instruction is decoded into multiple loading micro-ops (μops). We modeled the parameters of the test CPU with what is public about the Sunny Cove architecture [85]. Note that the execution of these speculative attacks is based on gem5's OoO CPU model, and we are not making claims about Sunny Cove's susceptibility to speculative based attacks. We implemented our Flush + Reload (FR) program based on the secret extraction section of code as shown in the original Spectre paper [51]. We likewise modified this program to create a Prime + Probe (PP) benchmark, and optimized the provided Spectre code to

reduce extra loops, branches, and waiting mechanisms provided that the attack could still successfully run on both on real hardware (Intel Xeon CPU E3-1225) and in the gem5 simulator. We manually replaced and inserted our TCA instructions into the assembly versions of these attacks and adjusted the control loops accordingly to have each program issue the same number of loads (regardless of DGEMM TCA size). These tests allow TCA designs that break **Invariant T1 & T3**, where timing registers have a non-gated clock while crossing into the CPU domain. These fail our DRC by not using the output of the clock manager for their lane's clock signal.

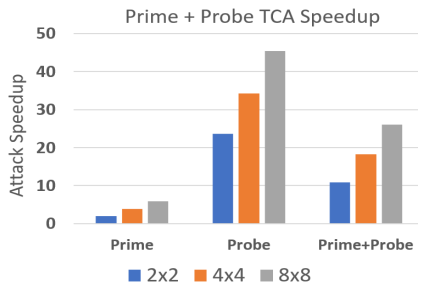
To test the results of speculative stashing, we modified the Spectre program to include multiple TCA instructions during the speculative attack portion of the code. We adjusted the number of instructions per attack iteration to see the difference in attack bandwidth in attempting to execute multiple instructions per misspeculated branch in the attack function. We inserted gem5 counters to determine how many of the speculative TCA load instructions finished execution. We assume any load that finished execution had time to be stashed by the TCA. This tests allows a TCA design that breaks **Invariant S1**, where speculative state persists after instruction flush. This fails our DRC by using FFs generated in an always_ff block, since there is no guarantee the flush signal will revert its state.

We examine the performance impact of our security hardware invariants through the use of previously proposed TCAs [29, 28] for three popular real-world PHP web applications – WordPress [87], Drupal [15], and MediaWiki [58] from the oss-performance suite [42]. WordPress is arguably the most popular blogging platform in use today supporting more than 60 million websites [86]. MediaWiki is used as the platform for Wikipedia and many other wikis, while Drupal powers at least 2.2% of all web sites including discussion forums,

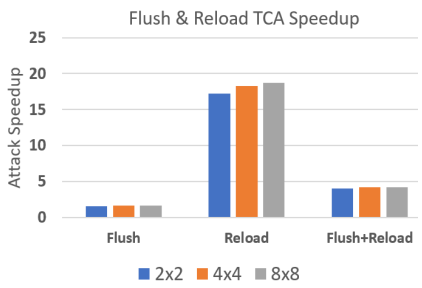
corporate sites, and personal blogs [29]. We evaluate the performance overhead using an in-house trace-driven execution simulator as used by Gope et al. [29]. We use data collected by this trace-driven execution to characterize the number of cycles spent in each accelerated function, as well as the number of invocations of each TCAs for these PHP applications.[2] This trace-driven simulator in turn assists to compare the overall execution cycle times of the previously proposed TCAs to our own, which incorporate all of the security properties described above. We also evaluate performance impacts of a secure 4×4 DGEMM TCA design, similar to that of 4×4 tensor cores within Nvidia GPUs. We performed DGEMM on 512×512 matrices, blocked to fit within the L1 D$, and tiled into 4×4 chunks for the TCA to perform operations on.

We estimate power and area overheads by designing each accelerator in RTL and using Design Compiler targeting a TSMC 40nm low power standard cell library. We assume that even the designs that do not use our secure primitives are properly designed and also have logic to revert state on misspeculation. This is because the only fair baseline is an accelerator that is also functionally correct. Our design forces clock gating, which could have additional power benefits over designs without using our secure invariants. However, we assume that the baseline will attempt to clock gate when possible as to not give our TCAs an unfair power advantage that is still possible without using our secure invariants.

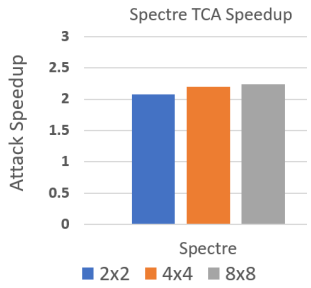
[2]Trace-driven simulator data was collected by Dibakar Gope for both a paper[29] and dissertation[28]. He provided this raw data to us and we modified accelerator latency of his collected data to show performance differences of secure TCAs for these 3 applications.



(a) Prime and Probe (PP)



(b) Flush and Reload (FR)



(c) Spectre Attack Algorithm using FR

Figure 6.13: Speedups (proportional to attack bandwidth gains) for (a) Prime and Probe (PP) and (b) Flush and Reload (FR) attacks when using 2×2, 4×4, and 8×8 DGEMM TCA instructions that leak timing information on memory references. (c) shows attack speedups for a Spectre example algorithm using DGEMM TCAs for FR extraction.

6.7 Results

We can see from Figure 6.13 that the PP and FR secret extraction process can be dramatically sped up with a malicious or leaky TCA. The speedup during the 'Prime' portion of the PP attack comes from reduced looping overheads and offloaded address generation. The 'Probe' portion gets this speedup in addition to the larger impact of detecting hits and misses in parallel rather than through serialized loads. The increase in throughput for larger DGEMM-like TCAs comes from a reduced number of control loops, instructions for address generation, and overall higher ratio of attack loads to control instructions.

Instead of streaming through cache hits, the FR attack streams through cache misses until the secret value is hit in the cache. For this reason, the 'Reload' portion is mostly determined by the number of MSHRs in the core. Regardless of TCA size, the parallelization of timing is dictated by the number of outstanding miss-under-miss requests allowed to be made in parallel.

Both attacks see most of the speedup coming from the extraction (probe and reload)

portions of the attack. A 5-25× attack speedup corresponds to a 5-25× bandwidth increase for secret information to be leaked. In a Spectre-based attack, extraction of the secret is only a portion of the execution time. When applying the FR techniques within the context of an optimized version of a Spectre based example [51], we show that significant speedup of the overall attack is possible. This shows that creating an accelerator for one type of application can also be susceptible to increasing bandwidth of known hardware attacks. Note that this speedup is not the same as speeding up an attack by using a faster or more efficient core, but rather gives a new ability (parallel focused timing) that parallelizes existing vulnerabilities.

This parallelization is not currently possible on existing hardware, but could easily be (accidentally or maliciously) created utilizing TCA hardware. Thus, when designing tightly-coupled accelerators, it is critical to keep security in mind to prevent designs from causing new or amplifying the severity of existing exploits.

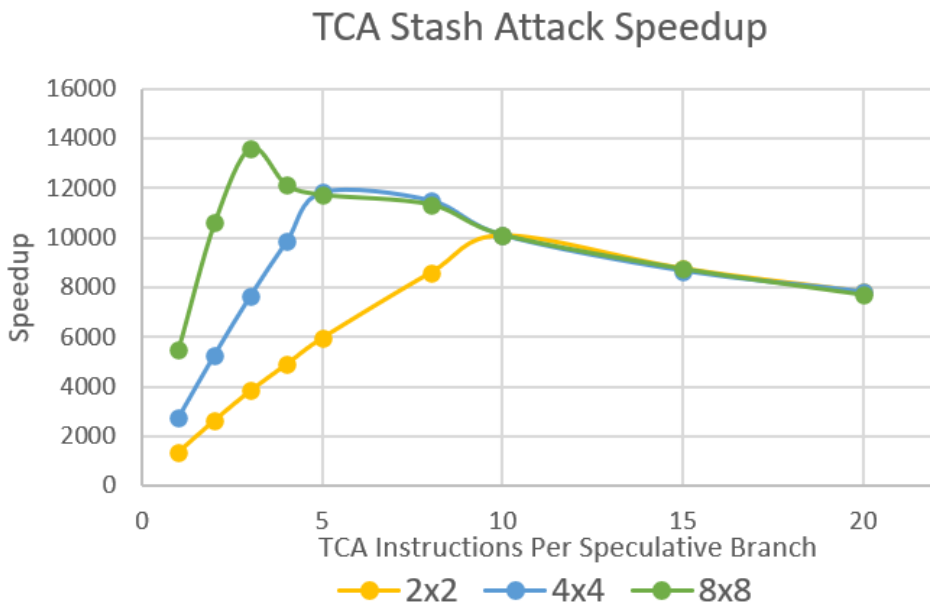


Figure 6.14: Attack speedups for DGEMM-like TCAs with speculative stashing.

To test the potential impacts of speculative stashing, we modify the Spectre-based program to issue multiple TCA instructions per attack (and training) iterations. We then measure how many loads were able to execute prior to branch resolution. Figure 6.14 shows that a speculative-stashing TCA can create 4 orders of magnitude speedup over the baseline Spectre attacks, and 3 orders of magnitude speedup beyond the TCA PP and FR amplification attacks. The fact that each stashed value holds 64B of information instead of just 1B creates a 64x increase in bandwidth over the Spectre baseline using PP or FR extraction. An additional 20x speedup multiple on top of the 64x speedup comes from the fact that multiple wide loads can be issued and executed before the branch is eventually resolved. In the time that a single byte of information was leaked during a misspeculated path, over 1kB of information can be stored in this stash attack. In this attack, the issued loads do not need a level of indirection, as the value of the load can be stored into internal TCA FFs, providing additional speedup. The remaining speedup comes from the fact that the stashed data can be directly read rather than having to go through the process of PP or FR to obtain the secret.

Figure 6.14 also shows that there is an optimal number of instructions to insert per attack iteration for optimal attack bandwidth. Increasing the number of instructions simultaneously increases the number of load instructions per attack iteration, but also the number of load instructions per training iteration. Once the number of instructions increases beyond the number of instructions that execute during attack iterations, the overall stash attack throughput decreases due to the increased overall execution time of the training iterations.

It is apparent that TCAs that allow these security vulnerabilities should be prevented

from being incorporated into commercial processors. However, prior to this work there was no existing framework to determine whether or not a TCA design was vulnerable to such attacks. Even seemingly useful and benign accelerators (such as the DGEMM example) might be able to be used to amplify speculative attacks. In this work we propose design rules that prevent speculative data from persisting and timing mechanisms from monitoring memory accesses. We demonstrate how to enforce these rules and prevent these accidentally or maliciously designed vulnerable TCAs from passing the design checks.

We look at some of the existing proposed tightly-coupled accelerators as described for PHP proposed accelerators [29] targeted for string manipulation, hashmap access, heap management, and regular expressions. We replicated these accelerators in RTL following our design rules. We then looked to see the performance impacts that these security features had compared to the prior work. Figure 6.15 shows that our security metrics impose a slight performance degradation as compared to the prior work. This degradation comes from

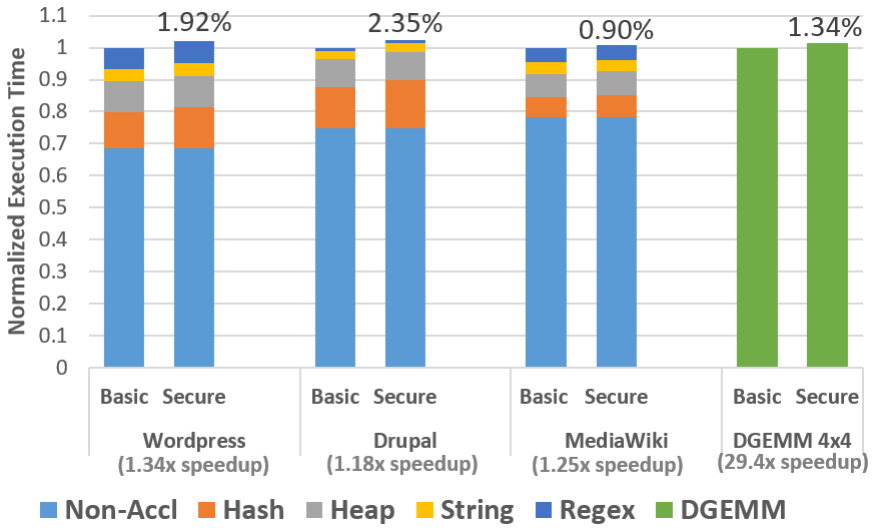


Figure 6.15: Performance breakdown of PHP server and DGEMM benchmarks with and without using our security requirements (Less than 2.5% performance overhead in all cases). Speedups relative to non-accelerated baseline shown in parenthesis.

some designs requiring extra pipeline stage(s) to prevent the potentially insecure passing of data within the same pipeline stage. Although it may be avoidable through more advanced clock-gating, we also conservatively impose an extra cycle delay between hand-off of a memory request and resuming of a clock-gated lane's execution. We also test a 512×512 DGEMM algorithm using 4×4 DGEMM TCA designs, and a TCA design following our security invariants. Figure 6.15 shows secure TCAs still gain most of the speedup potential from these workloads, suffering less than a 2.5% difference in the program's execution time.

We next look at the power and area overheads for each of these secure accelerators. Figure 6.16 shows there is less than 0.5% area and power overhead for these secure designs. For the heap and hashmap accelerators, most of the area and power exists within the hashmap table and heap manager pointer table. The string accelerator incurs most of its cost from a 64×64 comparison array for matching string logic, and the DGEMM accelerator with its FMAC units and submatrix FFs. The overheads are so low due to the fact that the

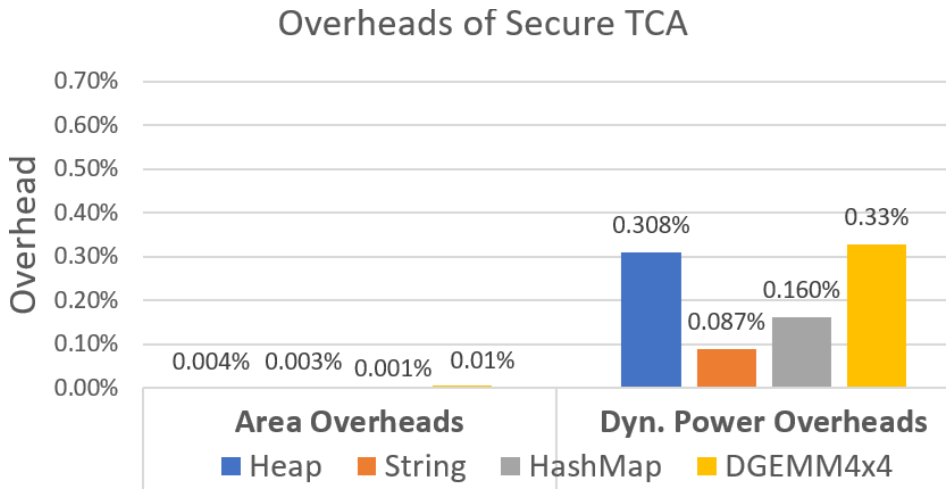


Figure 6.16: Area and power overheads for various TCA accelerators after redesigning to pass our security design rules (Less than 0.5% overhead in all cases).

clock gating logic requires only 2 bits of storage per lane, one for memory requests and the other for completion synchronization. The memory bit is set on any memory request and unset with memory arbiter responses, while the synchronization bit is set upon completion (the lane's done signal), and unset upon the global barrier advance signal. The clock output logic simply outputs the clock when both bits are unset, and synchronously held constant when either bit is set. Design Compiler confirms that these extra bits and gates have minuscule area and power overheads.

6.8 Chapter Summary

This chapter explores the potential security vulnerabilities that can come from TCAs. TCAs can be vulnerable both as a new side-channel, as well as amplifying existing side-channels to increase the bandwidth of known vulnerabilities. We explore the root causes for speculative and timing-based attacks to propose invariants from the ground up to prevent these attacks. These invariants can help designers of fixed-function TCAs gain insights into the security of their designs. They also allow the ability for reconfigurable TCA designs to have certain security guarantees. By imposing rules at design-time, it is possible to eliminate the possibility for speculative based attacks to further exploit TCAs over the existing CPU architectures.

Our analysis shows minimal power and area overheads (<0.5%) to enforce these invariants. These rules have slight impact on performance (<2.5%), but we argue that the security vulnerabilities they close (which have the potential of 25-10,000× increased attack bandwidth) make them worthwhile. We also make the case that the possibility for a

reconfigurable TCA may be possible through the use of hardware invariants enforced by a DRC to prevent various attacks. This builds the case to create accelerators from the ground up on trusted hardware.

Although our proposed solutions do not provide protection guarantees against all forms of possible hardware-based attacks, this research is an important step towards enabling reconfigurable TCAs. The potential benefits of a fully reconfigurable TCA could provide significant advancements in the microarchitecture and computer architecture community. Our research helps to show that it is possible to have design-time rules that can create more secure hardware without hindering the ability to design useful TCAs.

7 CONCLUDING REMARKS AND FUTURE DIRECTIONS

To go from idea to implementation requires many steps along the way, and building TCAs is no exception. First, it is important to know what functions(s) wish to be accelerated, and to evaluate the estimated speedups that can be gained. Our analytical model helps in this step of the process, as designers can quickly estimate application speedup with a limited number of parameters and limited knowledge about the accelerator's implementation.

Next, the implementation for the TCA design can bring new challenges due to the multiple possible ways to design its microarchitecture. Although it is not the only consideration, we specifically look at the details of operand delivery and data movement. Our analysis helps reason through the design considerations in the context of TCAs.

Finally, both during implementation and production, verification of the design is required. Security is one of these aspects of verification that needs to be addressed. Our work helps to create tests and checks for closing some of the possible security vulnerabilities of both new and existing speculative- and timing-based attacks.

7.1 The Project Selection

Throughout the entire process of researching each of the topics described in this dissertation's chapters, tough decisions were required regarding deciding which research topics to pursue and where to place the most attention. From starting with an ambitious goal to have completely reconfigurable compute integrated into the standard out-of-order processor, a long journey was required to even reach this point where we're just starting to scratch

the surface of all the potential avenues to explore. Each research project was specifically chosen to push the research front closer to this goal. In order to both motivate and explain the desire for reconfigurable compute, we built upon existing work that had proposed fine-grained acceleration that could be computed through tightly-coupled accelerators. We desired to show that there are benefits in integrating these functions into out-of-order cores by evaluating performance degradation that comes from non-concurrent execution with the core. Our analytical model helped to demonstrate the vast design-space of different types of accelerators, and that the more fine-grained tasks are, the more critical concurrent execution of fine-grained acceleration becomes.

We next wanted to evaluate what it would look like to have tightly-coupled accelerators integrated into the core, and realized that current ALUs usually use RF for operands and data passing. However, in the case of several accelerators we were focusing on (DGEMM, string manipulation, etc.), it seemed inefficient to have intermediate storage for elements not expected to have reuse. This required our NoRF work to evaluate and confirm our hypothesis that TCAs may not always operate best by using existing means of passing operands through the RF.

Lastly, we knew that reconfigurable hardware has its own complications for verification and security. We addressed some of the most recent and severe potential security vulnerabilities we could envision on the TCA to see if we could successfully prevent some of these attacks. This led to our work on mitigation for speculative- and timing-based attacks.

7.2 Future Work

We have started the process of designing reconfigurable TCAs within the execute stage of OoO cores. We currently have limited ourselves to simulations running a single TCA for a given application. There are many different avenues of direction to expand this work. Each of the proposed avenues regarding modelling, implementation, and security have future directions in which we envision exploring in future research.

7.2.1 TCA Modeling

Increased complexity for increased precision – From the analytical model perspective, we started the process of a first-order model that provides simple estimation. Modeling some of the more complex interactions between dependency within the instructions, or ways to add more program characteristics regarding concurrent CPU execution may provide better analysis than our simple assumptions of constant IPC. As with virtually all models, there is a tradeoff between model complexity and accuracy. The cycle-level simulator in which we compare against is also a model of sorts, where much higher complexity can provide more detailed results. Adding additional complexity such as instruction dependency metrics or more accurate critical path estimation to our first-order model could likely create a new estimation point that is more precise than our model while still being faster and less complex than a cycle-level simulator.

Confidence-based speculation –Since misspeculation can cause additional delay penalties on the core, and branch mispredictions are the most common cause of misspeculation,

partial TCA speculation (such as only speculating when on an instruction path with high-confidence outstanding branches) could also be a possible implementation for a design somewhere between the L and NL modes described in this work. Although the hardware for recovery is still required in case the predictor is wrong, this could potentially provide benefits in TCA designs where TCA recovery is on the critical path of branch mispredictions.

Models beyond performance –In this work, we primarily discussed the performance evaluation for TCAs. However, each of the implementations require different hardware, each with various area and power costs. In order to create a more complete evaluation, a pareto-optimal curve of design implementations could show the tradeoff between hardware costs, performance, and which (if any) design implementations fall outside of the curve and should not be considered.

7.2.2 NoRF and Microarchitectural Implementation

Increased focus on coherency – Through our research, it appears that coherency is more likely to be supported between the TCA and memory hierarchy in NoRF designs than in RF designs. However, coherency is extremely tricky to perfect with transition states and all corner cases. Formalized proofs are likely needed to correctly verify, which was beyond the scope of this work. Creating these proofs or implementing integration with prior work such as Crossing Guard [64] would be a value step forward in confirming our hypothesis.

Formal analysis among various workloads – This work created a case against RF operand passing through the use of counter-example. We verbally discussed ways that

NoRF could still create RF-like benefits without the downside of using it as a temporary buffer. This helps start the conversation of quantitatively determining optimal designs for specific accelerators. However, if it is possible to create a generic formula or design-space exploration for various acceleratable functions, this would help broaden the scope and visibility of our current analysis.

7.2.3 TCA Reconfigurability and Security

Granularity of Reconfiguration – Within a program, there might be regions of interest where different TCAs are desired. This might allow the TCA to be reconfigured mid-program to optimize the TCA for the duration of the program, rather than a particular region of interest. This also brings an interesting research question regarding the granularity of reconfiguration. Reconfiguration will have certain latency costs, so more detailed analysis is needed to determine the minimum number of cycles between regions where the TCA is allowed to reconfigure.

Reconfiguration among multiple threads – Multi-threaded cores may have different threads or applications desiring reconfigurable TCAs. This brings research questions regarding the partitioning and utilization of having multiple reconfigurable TCAs in the system. We also tested in systems where applications were allowed to run straight through without needing context switches. TCAs hold additional state not only within its internal state, but also through the TCA configuration itself. If reconfiguring the TCA has particularly long latency, this could now become the bottleneck during context switching. To avoid this cost, TCAs could simply not be swapped in and out during context switches

at the cost of limiting TCA reuse among multiple threads, and also having a maximum number of processes/threads that use a reconfigurable TCA. However, this would limit the abstraction currently provided by the OS being allowed to run many more processes and threads than available in hardware resources at any given time. If context switching becomes more costly, new cost functions may be required by the OS to find a new balance of performance and fairness among competing threads.

7.2.4 Comments on Future Work

These are just a few of the areas in which we have already considered next step evaluation and research. This certainly is not a comprehensive list of the new areas that can be explored regarding TCAs and reconfigurable TCAs. Many areas such as ML and AI are also rapidly evolving, creating new applications and algorithms that may provide new opportunities for TCA benefits. It becomes easy to see that the avenues of exploration are practically limitless already, with the likelihood to expand in the future.

7.3 Closing Remarks

As with most research, as more knowledge is gained in a topic, an even larger avenue of potential questions arises. We hope that providing a framework and starting point for reconfigurable TCAs will help start the era of specialized and/or reconfigurable compute being integrated into mainstream commercial processors. This avenue of research, particularly ones requiring integrating the details into simulators and RTL code, has highlighted many of the fine details that are required to make these operations functional, but also has

been an encouragement to see as theoretically possible. Research has shown me that it is a never-ending goal, but one that is always pushing the limits of new potentials. In exploring some of these new potentials, with continued research, I hope to see these goals morph into bigger and better ideas, and to see some of them become integrated into commercial processors.

BIBLIOGRAPHY

[1] Sam Ainsworth and Timothy M Jones. "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 132–144.

[2] Muhammad Shoaib Bin Altaf and David A Wood. "LogCA: a performance model for hardware accelerators". In: *IEEE Computer Architecture Letters* 14.2 (2015), pp. 132–135.

[3] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[4] Robert G Belleman, Jeroen Bédorf, and Simon F Portegies Zwart. "High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA". In: *New Astronomy* 13.2 (2008), pp. 103–112.

[5] Luca Benini et al. "Symbolic synthesis of clock-gating logic for power optimization of control-oriented synchronous networks". In: *Proceedings European Design and Test Conference. ED & TC 97*. IEEE. 1997, pp. 514–520.

[6] Nathan Binkert et al. "The gem5 simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[7] Bryan Catanzaro et al. "Efficient, high-quality image contour detection". In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE. 2009, pp. 2381–2388.

[8] Yu-Hsin Chen et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2016), pp. 127–138.

[9] Yu-Ting Chen et al. "Accelerator-rich cmps: From concept to real hardware". In: *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE. 2013, pp. 169–176.

[10] Yunji Chen et al. "Dadiannao: A machine-learning supercomputer". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2014, pp. 609–622.

[11] Jack Choquette, Olivier Giroux, and Denis Foley. "Volta: performance and programmability". In: *IEEE Micro* 38.2 (2018), pp. 42–52.

[12] Ian Cutress. *Intel's Architecture Day 2018: The Future of Core, Intel GPUs, 10nm, and Hybrid x86*. Last accessed 1 April 2020. 2018. URL: `https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86`.

[13] William J Dally, Yatish Turakhia, and Song Han. "Domain-specific hardware accelerators". In: *Communications of the ACM* 63.7 (2020), pp. 48–57.

[14] Misha Denil et al. "Predicting parameters in deep learning". In: *Advances in neural information processing systems*. 2013, pp. 2148–2156.

[15] "Drupal. https://www.drupal.org/". In:

[16] Lieven Eeckhout and Koenraad De Bosschere. "Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces". In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2001, pp. 25–34.

[17] Frank Emnett and Mark Biegel. "Power reduction through RTL clock gating". In: *SNUG, San Jose* (2000), pp. 1–11.

[18] Stijn Eyerman et al. "A mechanistic performance model for superscalar out-of-order processors". In: *ACM Transactions on Computer Systems (TOCS)* 27.2 (2009), p. 3.

[19] Stijn Eyerman et al. "A performance counter architecture for computing accurate CPI components". In: *ACM SIGPLAN Notices* 41.11 (2006), pp. 175–184.

[20] Simone Faro and M Oğuzhan Külekci. "Fast multiple string matching using streaming SIMD extensions technology". In: *International Symposium on String Processing and Information Retrieval*. Springer. 2012, pp. 217–228.

[21] Jeremy Fowers et al. "A configurable cloud-scale DNN processor for real-time AI". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press. 2018, pp. 1–14.

[22] Jeremy Fowers et al. "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication". In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2014, pp. 36–43.

[23] Qian Ge et al. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[24] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. "Accelerating arithmetic kernels with coherent attached FPGA coprocessors". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1072–1077.

[25] Graham Gobieski et al. "MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2019, pp. 670–684.

[26] Zhangxiaowen Gong et al. "SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 796–810.

[27] Sridhar Gopal et al. "Speculative versioning cache". In: *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. IEEE. 1998, pp. 195–205.

[28] Dibakar Gope. *Architectural Support for Scripting Languages*. The University of Wisconsin-Madison, 2017.

[29] Dibakar Gope, David J Schlais, and Mikko H Lipasti. "Architectural Support for Server-Side PHP Processing". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 507–520.

[30] Kazushige Goto and Robert A Geijn. "Anatomy of high-performance matrix multiplication". In: *ACM Transactions on Mathematical Software (TOMS)* 34.3 (2008), p. 12.

[31] Nathan Goulding-Hotta et al. “The greendroid mobile application processor: An architecture for silicon’s dark future”. In: *IEEE Micro* 31.2 (2011), pp. 86–95.

[32] Venkatraman Govindaraju et al. “Dyser: Unifying functionality and parallelism specialization for energy-efficient computing”. In: *IEEE Micro* 32.5 (2012), pp. 38–51.

[33] Felix Gremse et al. “GPU-accelerated sparse matrix-matrix multiplication by iterative row merging”. In: *SIAM Journal on Scientific Computing* 37.1 (2015), pp. C54–C71.

[34] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache template attacks: Automating attacks on inclusive last-level caches”. In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 897–912.

[35] Daniel Gruss et al. “Flush+ Flush: a fast and stealthy cache attack”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 279–299.

[36] Gael Guennebaud, Benoit Jacob, et al. “Eigen: a c++ linear algebra library”. In: *URL http://eigen. tuxfamily. org, Accessed* 22 (2014).

[37] Gagan Gupta and Gurindar S Sohi. “Semantically ordered parallel execution of multiprocessor programs”. PhD thesis. UNIVERSITY OF WISCONSIN–MADISON, 2015.

[38] Tim Harris, James Larus, and Ravi Rajwar. “Transactional Memory, 2nd edition”. In: *Synthesis Lectures on Computer Architecture* 5.1 (2010), pp. 1–263. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[39] Mark Hill and Vijay Janapa Reddi. “Gables: A Roofline Model for Mobile SoCs”. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 317–330.

[40] Mark D Hill and Michael R Marty. “Amdahl’s law in the multicore era”. In: *Computer* 41.7 (2008), pp. 33–38.

[41] Alireza Hodjat and Ingrid Verbauwhede. “Interfacing a high speed crypto accelerator to an embedded CPU”. In: *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004.* Vol. 1. IEEE. 2004, pp. 488–492.

[42] “https://github.com/hhvm/oss-performance”. In:

[43] Wei-Ming Hu. “Reducing timing channels with fuzzy time”. In: *Journal of computer security* 1.3-4 (1992), pp. 233–254.

[44] Yu-Wen Huang et al. “Analysis, fast algorithm, and VLSI architecture design for H. 264/AVC intra frame coder”. In: *IEEE Transactions on Circuits and systems for Video Technology* 15.3 (2005), pp. 378–401.

[45] Intel. *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*. Apr. 2019.

[46] Zhe Jia et al. “Dissecting the NVidia Turing T4 GPU via Microbenchmarking”. In: *arXiv preprint arXiv:1903.07486* (2019).

[47] Zhe Jia et al. “Dissecting the NVIDIA Volta GPU architecture via microbenchmarking”. In: *arXiv preprint arXiv:1804.06826* (2018).

[48] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *arXiv preprint arXiv:1704.04760* (2017).

[49] Svilen Kanev et al. “Mallacc: Accelerating Memory Allocation”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2017, pp. 33–45.

[50] Tejas S Karkhanis and James E Smith. “Automated design of application specific superscalar processors: an analytical approach”. In: *Proceedings of the 34th annual international symposium on Computer architecture*. 2007, pp. 402–411.

[51] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution”. In: *arXiv preprint arXiv:1801.01203* (2018).

[52] Scott P Kolodziej et al. “The Suitesparse matrix collection website interface”. In: *Journal of Open Source Software* 4.35 (2019), p. 1244.

[53] Rakesh Komuravelli et al. “Stash: Have your scratchpad and cache it too”. In: *ACM SIGARCH Computer Architecture News* 43.3S (2015), pp. 707–719.

[54] David J. Kuck and Richard A. Stokes. “The Burroughs scientific processor (BSP)”. In: *IEEE Transactions on Computers* 5 (1982), pp. 363–376.

[55] Chanchal Kumar et al. “Post-Silicon Microarchitecture”. In: *IEEE Computer Architecture Letters* 19.1 (2020), pp. 26–29.

[56] Moritz Lipp et al. “Meltdown: Reading kernel memory from user space”. In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 973–990.

[57] Chris Lomont. “Introduction to intel advanced vector extensions”. In: *Intel white paper* 23 (2011).

[58] “MediaWiki. https://www.mediawiki.org/wiki/MediaWiki”. In:

[59] Bingfeng Mei et al. “ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix”. In: *International Conference on Field Programmable Logic and Applications*. Springer. 2003, pp. 61–70.

[60] John Nickolls and William J Dally. “The GPU computing era”. In: *IEEE micro* 30.2 (2010), pp. 56–69.

[61] Tony Nowatzki et al. “Pushing the limits of accelerator efficiency while retaining programmability”. In: *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 27–39.

[62] Eriko Nurvitadhi et al. “Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–4.

[63] Lena E Olson. *Protecting host systems from imperfect hardware accelerators*. The University of Wisconsin-Madison, 2016.

[64] Lena E Olson, Mark D Hill, and David A Wood. “Crossing Guard: Mediating Host-Accelerator Coherence Interactions”. In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 163–176.

[65] Lena E Olson et al. "Border control: Sandboxing accelerators". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2015, pp. 470–481.

[66] Angshuman Parashar et al. "Timeloop: A systematic approach to dnn accelerator evaluation". In: *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2019, pp. 304–315.

[67] Colin Percival. *Cache missing for fun and profit*. 2005.

[68] Andrew Putnam, Aaron Smith, and Doug Burger. "Dynamic vectorization in the E2 dynamic multicore architecture". In: *ACM SIGARCH Computer Architecture News* 38.4 (2011), pp. 27–32.

[69] Moinuddin K Qureshi. "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 775–787.

[70] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. "Implementing streaming SIMD extensions on the Pentium III processor". In: *IEEE micro* 20.4 (2000), pp. 47–57.

[71] Venu Gopal Reddy. "Neon technology introduction". In: *ARM Corporation* 4.1 (2008).

[72] A. Roth. "Store vulnerability window (SVW): re-execution filtering for enhanced load optimization". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. 2005, pp. 458–468. DOI: 10.1109/ISCA.2005.48.

[73] Karthikeyan Sankaralingam et al. "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE. 2003, pp. 422–433.

[74] David J Schlais, Heng Zhuo, and Mikko H Lipasti. "Modeling Architectural Support for Tightly-Coupled Accelerators". In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2020, pp. 253–262.

[75] Yakun Sophia Shao et al. "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 97–108.

[76] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[77] Guangyu Shi, Min Li, and Mikko Lipasti. "Accelerating search and recognition workloads with sse 4.2 string and text processing instructions". In: *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 145–153.

[78] Jun Shirako et al. "Analytical bounds for optimal tile size selection". In: *International Conference on Compiler Construction*. Springer. 2012, pp. 101–121.

[79] Srikanth T Srinivasan et al. "Continual flow pipelines". In: *ACM SIGARCH Computer Architecture News* 32.5 (2004), pp. 107–119.

[80] Nitish Srivastava et al. “MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 766–780.

[81] Jeffrey Stuecheli et al. “CAPI: A coherent accelerator processor interface”. In: *IBM Journal of Research and Development* 59.1 (2015), pp. 7–1.

[82] Shyamkumar Thoziyoor et al. *CACTI 5.1*. Tech. rep. Technical Report HPL-2008-20, HP Labs, 2008.

[83] Timothy J Todman et al. “Reconfigurable computing: architectures and design methods”. In: *IEE Proceedings-Computers and Digital Techniques* 152.2 (2005), pp. 193–207.

[84] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. “Eliminating fine grained timers in Xen”. In: *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 2011, pp. 41–46.

[85] Wikichip. *Sunny Cove - Microarchitectures*. Last accessed 25 November 2019. 2019. URL: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove.

[86] “WordPress wikipedia. https://en.wikipedia.org/wiki/WordPress”. In:

[87] “WordPress. https://wordpress.com/”. In:

[88] Jiaxiang Wu et al. “Quantized convolutional neural networks for mobile devices”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4820–4828.

[89] Qing Wu, Massound Pedram, and Xunwei Wu. “Clock-gating and its application to low power design of sequential circuits”. In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47.3 (2000), pp. 415–420.

[90] UG585 Xilinx. *Zynq-7000 All Programmable SoC: Technical Reference Manual*. 2015.

[91] Mengjia Yan et al. “Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 347–360.

[92] Yuval Yarom and Katrina Falkner. “FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack”. In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[93] Zhi Alex Ye et al. “CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit”. In: *ACM SIGARCH Computer Architecture News* 28.2 (2000), pp. 225–235.

[94] Jiyong Yu et al. “Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 954–968.

[95] Yongfeng Zhang et al. “Localized matrix factorization for recommendation based on matrix block diagonal forms”. In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 1511–1520.