Integrating Computing Systems from the Gates up: Breaking the Clock Abstraction

By

Gokul Subramanian Ravi

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

2020

Date of final oral examination: 8/7/2020

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

Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Yu Hen Hu, Professor, Electrical and Computer Engineering
Joshua San Miguel, Assistant Professor, Electrical and Computer Engineering
Gurindar Sohi, Professor, Computer Sciences
Pradip Bose, Distinguished RSM & Manager, IBM Research

To Amma and Appa, for their wisdom, love and sacrifice.

ACKNOWLEDGMENTS

At the end of this long and eventful journey, I realize that my learnings from grad school go far beyond the technical aspects. Among many things, I've learned that even the most arduous of times can become enjoyable when you have the right support system around you - mentors, colleagues, friends and family. For this, there is a long list of important individuals to whom I am thankful and indebted to.

First and foremost, I thank my parents whose support and guidance extends far beyond this PhD. Everyone credits students like me for the sacrifices we make in order to pursue our career dreams. At least in my case, I believe that my sacrifices are negligible compared to those made by my parents (and in general, parents of all students - especially international ones). I hope that this PhD and my other career pursuits will, in some small way, help in filling the void of my absence in their day-to-day lives. I salute my sister, Aruna, for leading the way, enabling me to follow in her footsteps to becoming a fellow Dr. Ravi. It seems not so long ago that Aruna and Antriksh supported me in my very first footsteps in the US and in Wisconsin. Thank you for your support from the very beginning.

Next, and most important to my career, I express my sincere gratitude to my PhD advisor Prof. Mikko Lipasti. It is truly an honor to be mentored and supported by Mikko. His technical acumen is second to none - his ability to analyze both macroscopic and microscopic details of our endeavors with surgical precision is exhilarating to experience first-hand. Perhaps even more integral to his mentoring is the utmost intellectual freedom he allows his students. I thank him for enabling and molding my development in the most relaxed and pressure-free environment possible. He is also an extremely affable person

and a pleasure to work with. I am grateful to have learned from the very best! In the future, if I have the opportunity to be a mentor, I hope I am able to imbibe in Mikko's technical as well as non-technical qualities.

Next, I would like to thank my other PhD Committee members for their guidance and advice throughout the course of my PhD. I'm lucky to have been able to closely interact with Prof. Joshua San Miguel over the last few years - in our approximate computing collaborative projects and beyond. Josh is a brilliant researcher and, at the same time, a fun person to interact with. Observing his work has taught me a lot about how to establish myself as a reputed early-career independent researcher - hopefully I can follow in some of his footsteps! I am extremely thankful to Dr. Pradip Bose for agreeing to be a part of my PhD Committee. His diligence and enthusiasm towards my research has been heartening to me and has significantly helped shaped my research and development. It's been a pleasure to collaborate with him on ongoing projects as well. I'm honored to have Prof. Gurindar Sohi, one of the highly decorated computer architects, on my PhD Committee. I cherish my interactions with him and thank him for his guidance and advice. I also thank Prof. Yu Hen Hu for serving on my committee. My first course in computer architecture (ECE 552) was taught by Prof. Hu. His enthusiasm has clearly rubbed off on me! I also enjoyed serving as his TA for the same course, which was a fruitful experience.

I'm highly indebted to Prof. Nam Sung Kim for advising and supporting me during my first year at Wisconsin as a Master's student. It was a pleasure to learn from him - both technical aspects as well as his work ethic. Moreover, for international students like me, financial concerns play a big part in deciding which university to choose for pursuing grad school. I was fortunate that Nam agreed to support me as an RA even before I came to

Wisconsin, and that made my transition considerably smoother. I also received considerable guidance and support from Hao Wang who took me under his wing when I worked with Nam - it was always fun to work with Hao and I thank him for his patience and daily support.

I also appreciate other collaborators for their guidance and contributions to the research I've pursued over the course of my PhD. I thank Prof. Tushar Krishna for the TNT project - I'm proud that this project, which started of as an informal chat with Tushar at ASPLOS 2018, blossomed into what it is today. I thank Dr. Ramon Bertran for MicroGrad and ongoing projects - it's been exciting to work with Ramon who is super diligent, helpful and has developed a wealth of open source infrastructure. I thank Dr. Bradford Beckmann and Sooraj Puthoor for an exciting internship at AMD Research - it was a great experience for me to take a break from my UW research and work on cool ideas with high potential for product impact. I'm grateful to Prof. Hyeran Jeon and Prof. Murali Annavaram for the opportunity to contribute towards their research on GPU Register File Virtualization. I also enjoyed working with Rahul Singh on promising approximate computing research. I look forward to continued collaborations with all of them in the future!

Next, I thank all members of the PHARM Research group for making our lab a home away from home. Back when I was a new student in the group, I looked up to my seniors - David Palframan, Dibakar Gope, Rohit Shukla and Michael Mishkin, and they were always willing to help and guide me throughout their time at Wisconsin and even beyond. I also thank my "juniors" in the lab - David Schlais, Heng Zhuo, Kyle Daruwalla, Carly Schulz, Ravi Raju, Chien-Fu Chen and Soroosh Khoram. I learn an incredible amount from all of them everyday, both technically and otherwise, and I am excited to see them grow as

researchers. Outside of my group, I also thank Swapnil Haria, the Architecture Reading Group regulars and other ECE/CS graduate students (both past and present) for thought provoking discussions. I will always cherish the memories of being a part of this wonderful, incredibly supportive and intelligent group of researchers.

"I get by with a little help from my friends". Over the course of grad school, I've been fortunate to be surrounded by circles of supportive friends. At the very beginning of my journey here, I'm thankful to Sahil Bajwa, Dinesh Natarajan, Prakash Natarajan and Kumara Guru for being by my side as we all acclimatized to this new Western world. Next, I thank Lokesh Jindal, Urmish Thakker, Keshav Mathur for bringing back some of the undergrad flavor to grad school. Finally, I thank Aaditya Chandrasekhar, Shaswath Sekar and Suchita Pati for making the final leg of this journey purely memorable. Thank you for making grad life so much more fun - it would hardly be worth it if not for all the good memories! There were so many more friends along the way, I cherish all the good times!

Finally, I thank the UW ECE Department's IT support, especially David Hanke. Scrambling to run 100s of jobs, hours before a paper deadline, is part and parcel of every computing PhD student's life and I'm grateful for the seamless infrastructure support.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In classical computing, there is tremendous unexplored opportunity to pursue system integration from the gates up. This would enable the upper layers of computer system abstractions to conveniently exploit the lower hardware-level circuit/device characteristics. With this view, this dissertation targets bottom-up integration in classical computers with specific focus on the system's clock. Our research proposals break the computer's abstraction of the clock by understanding the clocking system's structural, functional and variational characteristics; and leveraging them via optimizations across multiple layers of the system stack. Apart from exploiting clock characteristics, these proposals also present other architecture and system-level enhancements. Overall, these contributions have enabled considerable performance and energy efficiency benefits across a variety of computing substrates, while targeting multiple application domains. These proposals are summarized below:

CHARSTAR performs resource allocation and reconfiguration in a tiled architecture, cognizant to the power consumed by each node in the clock distribution system. CHARSTAR is further optimized for balanced spatio-temporal reconfiguration and enables efficient joint control of resource and voltage/frequency scaling. The increased complexity in resource-management is handled by utilizing a control mechanism driven by a lightweight offline trained neural predictor.

SlackTrap proposes aggressive timing speculation that caters to the average clock cycle slack across a sequence of operations rather than the slack of the most critical operation. This is achieved atop a "transparent" pipeline, a design that allows asynchronous multi-cycle

execution of computation sequences.

REDSOC aggressively recycles clock cycle data slack to the maximum extent possible. It identifies the data slack for each computation based on operation characteristics. It then attempts to cut out (or recycle) the data slack from a producer operation by starting the execution of dependent consumer operations at the exact instant of completion of the producer operation. Further, REDSOC optimizes the scheduling logic of out-of-order cores to support this aggressive operation execution mechanism. Recycling data slack over operation sequences enables their acceleration.

SHASTA is a cross-layer solution for building optimal General Purpose Approximation Systems (GPAS). SHASTA's hardware approximation techniques enable fine grained spatio-temporal diversity in approximation. Its compute approximation component is implemented by interpreting each computation's tolerated approximation as clock cycle slack and recycling it in the manner of REDSOC. Further, it proposes fine-grained memory approximation, novel hardware cognizant approximation tuning developed atop a gradient descent algorithm and system-wide synergic approximation.

TNT exploits the NOC facet that the clock utilization during NOC wire traversal is dependent on the structural characteristics of the system. Closer the connected routers are, lower is the clock utilization on that hop. Further, the utilization considerably varies with the physical heterogeneity in the NOC. TNT attempts end-to-end NOC traversal in a single pass, traversing multiple hops per cycle (as allowed by wiring capability). To our knowledge, TNT is closest to potentially achieving ideal wire delay network-latency. This end-to-end traversal is achieved by performing only neighbor-neighbor control interactions, allowing TNT to be built in a modular manner.

1 INTRODUCTION

Classical computing systems have advanced by leaps and bounds over many decades. Be it improvements in performance or throughput, power reduction or energy efficiency - these advances have come from innovations at different *abstraction layers* of computing - device physics, circuits, micro-architecture, architecture, compilers, operating systems, programming and application algorithms. While architecture innovations alone saw 20-50% processor performance growth per year from the 1980s to the 2000s, returns have been less favorable in the current decade falling to a 3% performance growth per year. As we fell away from the device trends set by Moore's law and beyond the physical capabilities of Dennard scaling, improving the performance and energy efficiency of computer systems has become ever more challenging. Nevertheless, with the volume of data roughly doubling every two years, there is still an ever-growing need for building fast and more power efficient computer systems.

1.1 Integrating Computing Systems from the Gates up

Despite these challenges, or rather because of them, we look forward to a new golden age of computer architecture. Among many important takeaways from the 2018 Turing Lecture [84], one which stood out was: *The need for vertical integration for a new renaissance in computer architecture.* Historically, long-established and deep-set layers of abstractions, across both hardware and software, allowed designers to cope with the complexity of designing sophisticated computer systems. Unfortunately, these abstractions have limited cross-stack innovation within these systems and this lack of vertical integration has left

significant untapped opportunity on the table. Vertical integration is key to building better computing systems - fundamentally it seeks to achieve better exchange of information between different layers of the system stack, so that each layer can be designed more efficiently.

In classical computing, there is tremendous unexplored opportunity to pursue integration from the gates up. This would enable upper layers of computer system abstractions to conveniently exploit the lower hardware-level circuit/device characteristics. With this view, this dissertation targets bottom-up integration in classical computers with specific focus on the computer system's clock. Our contributions *break the computer's abstraction of the clock* by understanding the clocking system's structural, functional and variational characteristics; and leveraging them via optimizations across multiple layers of the system stack. By doing so, they have enabled considerable performance and energy efficiency benefits across a variety of computing substrates, while targeting multiple application domains.

1.2 Breaking the Clock Abstraction

Computer engineers are familiar with Douglas Clark / Joel Emer's Iron Law of Processor Performance, which states that the time taken to execute a program is a product of the code size, the CPI (clock cycles per instruction) and the cycle time. The Iron Law is a classic example of the use of abstraction layers in computing systems: the compiler designers focus on the code size, the processor designers focus on the CPI and the chip designers focus on the cycle time. There are two important takeaways from Iron Law in the context of this dissertation:

1. The clock cycle is one of the most fundamental constructs in synchronous computing.
2. System architects have abstracted away the complex clocking system into a simple clock period.

The clock is fundamental to synchronous computing because it influences the entire system stack - from circuits and micro-architecture to the OS and applications. However, the clocking system is still somewhat of a black box across various layers of computing and this abstraction limits the capability of synchronous systems. Breaking the clock abstraction has implications across the following categories:

1. **Functional:** The relationship between clock cycle utilization and any functional unit, its computations and the operands involved.
2. **Structural:** The association between hardware structure and the clock. This can be further sub-divided as: a) the relationship between clock cycle utilization, the topology of the system, and the distance between system nodes and b) the power consumed by the clocking system and its unequal distribution across the clock hierarchy.
3. **Variational:** The effect of variation on the clock cycle utilization.

The contributions proposed in this dissertation understand and exploit these implications, to optimize diverse parts of the computer system stack and benefit multiple domains of applications. These proposals are classified based on the above taxonomy, in Table.1.1.

Contribution	Functional	Structural	Variational	Substrate	Domain
CHARSTAR(Sec. 2.1, Ch. 5)	✗	✓	✗	Spatial	GP
SlackTrap (Sec. 2.2, Ch. 6)	✓	✗	✓	Spatial	GP
REDSOC (Sec. 2.3, Ch. 7)	✓	✗	✗	OOO	GP/LP/ML
SHASTA (Sec. 2.4, Ch. 8)	✓	✗	✗	OOO	Approx.
TNT (Sec. 2.5, Ch. 9)	✗	✓	✓	NOC	Graph

Table 1.1: Categorizing the dissertation research contributions

1.2.1 Exploiting the Functional Implications

The functional implication tells us that clock cycle utilization in a functional unit is dependent on the computation being performed on the functional unit and the operands involved in the computation. A typical example of this is the single-cycle ALU [181] (Arithmetic and Logic Unit) wherein, arithmetic full-precision operations utilize almost the entire clock cycle while logical or low-precision operations do not (resulting in clock cycle slack). Further, approximate computations often allow even shorter computation times resulting in even greater clock cycle slack. The fundamental challenge in exploiting or recycling clock cycle slack is that traditional classical computing systems are synchronous, operating at the granularity of clock cycles. Moreover, the fine-grained variation in clock cycle slack from one operation to the next (with a range of almost 50% of the clock cycle), implies that harnessing this slack is non-intuitive.

Our research contributions exploit this functional characteristic of the clock across spatial/tiled architectures (SlackTrap [183, 182]) and out-of-order cores (REDSOC [181] and SHASTA [185, 180]). While SlackTrap (Chapter 6) focuses on general purpose applications, REDSOC (Chapter 7) focuses on targeting a broad range of applications ranging from general purpose (GP) to low-precision (LP) applications and machine learning (ML) kernels. Further, SHASTA (Chapter 8), which builds upon the REDSOC proposal, targets

approximate computing.

The recycling of clock cycle slack is made possible via our proposals for *greedy dynamic transparent dataflow*. This mechanism attempts to cut out (or recycle) the slack from a producer operation by starting the execution of dependent consumer operations at the exact instant of completion of the producer operation. Slack recycling is performed over contiguous operations of any operation sequence. Importantly, it does not require adjacent operations to fit into single cycles or any rearrangement of operations. The resulting acceleration of the operation sequence results in application speedup when such sequences lie on the critical path of execution.

These proposals targeting transparent dataflow have potential for impact and adoption since they are built with a *plug-and-play* philosophy, maintaining synchronous design overall, while incorporating pseudo-asynchronous logic evaluation within just the compute path. The pseudo-asynchronous logic is strictly limited to the compute units and data bypass network - the rest of the synchronous data path is completely oblivious to its existence. On the whole, these proposals are able to improve both performance and energy efficiency of classical and emerging approximate computing systems by considerable margins while being suitable for real-world use.

1.2.2 Exploiting the Structural Implications

First, in the context of on-chip networks, the structural implication tells us that the clock utilization during NOC router-to-router link traversal is dependent on the structural characteristics of the system. The closer the connected routers are, the lower the clock cycle

utilization is on that hop. Further, this clock cycle utilization can vary across the NOC topology depending on unequal physical wire lengths / delays - due to asymmetric aspect ratios as well as sparsely distributed memory controllers and process variation across the NOC. The main limiter though, to achieving ideal wire delay latency in an NOC is the impact of modular design i.e. the quantization of network traversal into hops, introduced by the packet's interaction with routers at each node in the network. Hop quantization primarily increases packet latency because it under-utilizes the wiring's per-cycle traversal capability - traversing only a single hop in a cycle even if the wire is capable of multiple hops worth of traversal per cycle.

For a realistic network to be able to achieve near-ideal wire delay latency, it has to avoid hop quantization but without straying away from a modular and scalable design. With this in mind, our TNT proposal [179, 178] (Chapter 9) enables *greedy dynamic transparent dataflow* in the NOC. Overall, the proposed NOC design achieves end-to-end traversal in a single pass, traversing multiple hops per cycle (as allowed by wiring capability), while retaining modularity by performing only neighbor-neighbor control interactions. To our knowledge, TNT is closest to potentially achieving ideal wire delay network-latency.

Second, in the context of spatially distributed architectures, the structural implication tells us that the clock distribution system (CDS) is typically the largest circuit net, operating at the highest speed within the entire synchronous system, thus consuming significant power. Further, the power consumed by the CDS is unequally distributed across the system depending on the clock-tree structure/design. This implies that making CDS-aware resource management decisions can considerably impact system-wide execution efficiency.

With this in mind, we proposed CHARSTAR [184] (Chapter 5), in which resource

allocation and reconfiguration decisions are made cognizant to the power consumed by each node in the clock hierarchy, and entire branches of the clock tree are greedily shut down (i.e. gated) whenever possible. CHARSTAR is further optimized for balanced spatio-temporal reconfiguration and enables efficient joint control of resource and voltage/frequency scaling. The increased complexity in resource-management decisions, stemming from the impact of vertical integration i.e. knowledge of the clock hierarchy, is handled by utilizing a control mechanism driven by a lightweight offline trained neural predictor.

1.2.3 Exploiting the Variational Implications

Finally, the variational implication tells us that accounting for the impact of the system's variation characteristics on the clocking circuit can have a positive impact on execution efficiency. We focus on effects of process variation primarily, but also discuss voltage and temperature variation. Building designs which are both aggressive and robust at handling the effect of such external stimuli are becoming especially important with harder-to-build lower technology nodes and more expansive on-chip real-estate (eg. chiplet / wafer-scale designs). While SlackTrap [183, 182] addresses the impact of random chip-wide variation on compute, TNT [179, 178] tackles the impact of process variation in the NOC.

1.3 Thesis Organization

The rest of this dissertation is organized as follows:

- Chapter 2 introduces the five research proposals in this dissertation.

- Chapter 3 provides the background and motivations for all the proposals.
- Chapter 4 provides the various implementation and evaluation methodologies.
- Chapter 5 discusses the CHARSTAR proposal, its design and evaluation.
- Chapter 6 discusses the SlackTrap proposal, its design and evaluation.
- Chapter 7 discusses the REDSOC proposal, its design and evaluation.
- Chapter 8 discusses the SHASTA proposal, its design and evaluation.
- Chapter 9 discusses the TNT proposal, its design and evaluation.
- Chapter 10 concludes the dissertation and discusses related ongoing/future work.

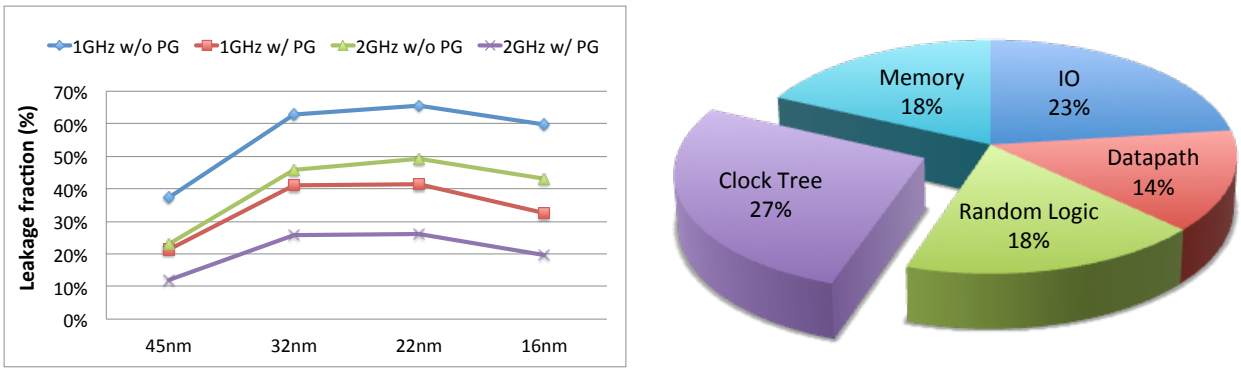
2 OVERVIEW OF THESIS CONTRIBUTIONS

In the previous chapter, we highlighted the potential for bottom-up system integration and the benefits from breaking the clock abstraction and exploiting its functional, spatial and variational implications. Further, we briefly introduced the research proposals in this dissertation, in the context of the above implications. Now we provide a self-contained overview of each of these proposals, highlighting their novelties and benefits.

2.1 CHARSTAR: Clock Hierarchy Aware Resource Scaling in Tiled Architectures

The demand for highest performance from current-day processing architectures has resulted in several challenges to achieving optimum energy efficiency. In order to exploit the maximum available parallelism in applications, these architectures are often significantly over-provisioned with execution resources. But more often than not, a significant portion of such resources remain underutilized. An ideal energy efficient architecture should dynamically avoid incurring any power or energy overhead from such unused resources.

The two main sources of avoidable power consumption in a processing core are excess leakage and clock tree power. Figure 2.1 illustrates the fractions of leakage power and clock power in modern architectures. In Fig.2.1.a leakage power is measured over multiple technology nodes on McPAT [233, 135] for an IBM POWER7 model [233]. The leakage power forms a formidable fraction of chip power, especially in designs operating at lower frequencies. It is evident that in designs without power gating, the leakage fraction can



(a) Leakage fraction over tech. nodes

(b) Mali GPU Power Distribution

Figure 2.1: Power Consumption

go even up to 70%. While leakage power is significant, so is the power dissipated by the clock distribution network - even more so if the leakage power fraction declines. In Figure 2.1.b, clock power is observed to consume more than a quarter of the total power on ARM Mali GPUs [28]. Prior work has shown that the power consumed by the clock distribution system (CDS) can be as high as 50% of the total circuit power, or, at the very least, consumes a major portion of the dynamic power [137, 49, 160, 140]. Such inefficiencies would increase with architectures designed even more aggressively for greater peak performance, and intelligent clock/resource aware mechanisms are necessary to optimize for better energy efficiency.

Prior works on architecture reconfiguration perform intelligent power gating to shut down *inefficient or underutilized* resources during particular periods of an application's execution. When resources are power gated, the leakage power that they would have dissipated if awake, is saved. These implementations can naively be extended to shut down immediate portions of the clock distribution system (clock gating) which solely feed these particular resources, reducing clocking power as well. *But these proposals neither study gating further up the clock hierarchy nor do they involve the clock hierarchy in influencing the*

reconfiguration decisions, both of which could have a large impact on energy efficiency.

The primary contribution of this work is optimizing reconfiguration mechanisms to be CDS-aware (Section 5.1). Shutting down of a particular resource in a CDS-unaware mechanism might only expect a linear increase in power savings, but if shutting down that resource allows the clock gating of a large portion of the clock hierarchy, then power savings are more significant. We show that the reconfiguration decisions made by a CDS-aware vs. a CDS-unaware mechanism are reasonably different, with a large impact on energy efficiency. Our proposed reconfiguration mechanism is aware of the clock hierarchy and the power consumed by each node in the tree. It selects resources to power down in a greedy manner and further, appropriately shuts down entire branches of the clock tree whenever possible.

Apart from proposing clock-aware reconfiguration and analyzing the same for multiple state of the art clock distribution systems, this proposal makes some secondary contributions as well. First, we argue that for best benefits from reconfiguration, the mechanism should operate at the appropriate *spatio-temporal* granularity to efficiently capture application characteristics (Section 5.2.2). This balances the ability to adapt to an application's needs quickly (temporal granularity) and accurately, in terms of resources, (spatial granularity) and at the same time, keep overheads to a bare minimum.

Second, we jointly optimize both DVFS and clock-aware power gating (PG-DVFS) to achieve the ideal configuration for each phase of an application, in terms of both ILP as well as frequency (Section 5.2.1). Jointly optimizing reconfiguration and clock rate has been studied for multi-processors—optimizing Thread Level Parallelism (TLP) vs. clock rate—but there have been no concrete proposals for *intra-core* dynamic control to balance

Instruction Level Parallelism (ILP) against core frequency.

Finally, our analysis shows that the combined savings from clock hierarchy aware integrated PG-DVFS is significant, but requires sophisticated control to predict the most efficient resource configuration for each application phase. We show that standard linear control mechanisms perform poorly in comparison to an oracular approach. We advocate the use of a lightweight machine learning-based control mechanism by showing near ideal accuracies of a multi-layer perceptron in this domain of reconfiguration (Section 5.2.4).

While the benefits of clock-aware reconfiguration extend to all processing frameworks, this work limits complexity of reconfiguration control by focusing on spatial/tiled architectures (Section 5.2.3).

We evaluate *CHARSTAR* on a high performance tiled microarchitecture, in which multiple resource tiles exploit the maximum available ILP from applications. Our implementation power-gates one or more of these tiles and their respective clock hierarchies when such ILP opportunities is limited, improving energy efficiency and transforming it into a *dynamically adaptable heterogeneous core* [88]. Evaluation is discussed in Section 5.4 with methodology in Section 4.1.

Corresponding Publications: *Gokul Subramanian Ravi and Mikko H. Lipasti. "CHARSTAR: Clock Hierarchy Aware Resource Scaling in Tiled Architectures". ISCA 2017.*

2.2 SlackTrap: Aggressive Slack Recycling via Transparent Pipelines

Modern processing architectures are designed to be reliable. They are designed to operate correctly and efficiently on diverse workloads across varying environmental conditions. To achieve this, the work performed by any functional unit (FU) in a synchronous design should be completed within its clock period, every clock cycle. Thus, conservative timing guard bands are employed to handle wide environmental (PVT) variations as well as all legitimate workload characteristics that might activate the critical path in any FU. In the common non-critical cases, this creates clock cycle *slack* - the fraction of the clock cycle performing no useful work. Under typical conditions and workload characteristics, each clock cycle produces slack averaging more than 25% of the clock period and sometimes even as much as 40% [77]. Performance and/or energy efficiency are thus sacrificed for reliability. Moreover, scaling to lower technology nodes creates an increasing gap between worst-case and nominal circuit delays, requiring even larger guard bands [116].

Timing Speculation (TS) is a state-of-the-art mechanism, which cuts into traditional timing guard bands, providing better execution efficiency at the risk of timing violations. When coupled with error detection and recovery, it presents a functionally correct, efficient, processor design. Its prior implementations in the synchronous domain have focused on adaptive variation of the operating points (F,V) by tracking the frequency of timing errors occurrences [55] or by estimating impact of PVT variations on slack [77, 134] and so on.

Prior synchronous TS solutions suffer two fundamental constraints. First, they are

bounded by the possibility of timing errors from *every* computation, in *every* synchronous FU or operation stage, and on *every* clock cycle. Second, the dynamic mechanisms among these are implemented by varying frequency/voltage over time and thus, can only be *reconfigured* at a reasonably coarse granularity of time (at best, over epochs of 1000s of cycles). Thus, ensuring no (or minimal) timing errors over the entire epoch forces these operating points to be set rather conservatively, constrained by timing requirements of each operation in the entire epoch. Otherwise, they run the risk of increased timing violations. While recovery mechanisms [55] maintain reliable operation in the face of timing errors, they impose significant penalties on performance and energy efficiency.

On the other hand, purely asynchronous solutions are inherently suited to slack conservation [138]. Varying execution times among operations which could cause timing errors in an aggressive synchronous TS design, can be avoided by allowing such varied delays to be balanced within the entire asynchronous execution window. But pure asynchronous solutions suffer from other functional complexities resulting in low throughput and/or high overhead implementation costs, making them a less popular solution.

To leverage the benefits of asynchronous solutions within the synchronous (pipelined) computing realm, we propose SlackTrap: ① Simple "asynchronous" execution engines are integrated seamlessly into synchronous pipelines. ② These engines are implemented as transparent pipelines with synchronous control - resulting in relatively low design complexity (Section 6.1.2). ③ Multiple asynchronously executable operations, bounded by synchronous boundaries, are grouped together into a transparent multi-cycle execution flow. This allows the timing speculation mechanism to cater to the average slack across these grouped executions rather than the most critical operation itself - allowing more aggressive

timing speculation (Section 6.1.1). ④ The above is enabled by a reliable slack estimation (Section 6.2.1) and slack tracking (Section 6.2.2) mechanisms. ⑤ Finally, benefits from timing speculation are obtained by clocking synchronous boundaries (to the sequences) early, rather than increasing frequency or decreasing voltage (Section 6.2.3). ⑥ Atop the CRIB tiled architecture, SlackTrap achieves absolute speedups up to 20% and relative improvements vs. competing mechanisms of up to 1.75x (Evaluation in Section 6.4 and methodology in Section 4.2).

Corresponding Publications: (*1*) *Gokul Subramanian Ravi and Mikko H. Lipasti. "Aggressive Slack Recycling via Transparent Pipelines". ISLPED 2018.* (*2*) *Gokul Subramanian Ravi and Mikko Lipasti. "Timing Speculation in Multi-Cycle Data Paths". IEEE CAL 2016.*

2.3 REDSOC: Recycling Data Slack in Out-of-Order Cores

As discussed in Section 2.2, conservative timing guard bands are employed to handle all legitimate workload characteristics that might activate critical paths in any EU/op-stage and wide environmental (PVT) variations that can worsen these paths. Improvements in performance and/or energy efficiency are thus sacrificed for reliability. In the common non-critical cases, this creates clock cycle *Slack* - the fraction of the clock cycle performing no useful work.

Slack can broadly be thought to have two components: ① *PVT Slack*, caused under non-critical PVT conditions, and ② *Data Slack*, caused due to non-triggering of the executional critical path. *PVT Slack*, with its relatively low temporal and spatial variability, can more easily be tackled with traditional solutions [77, 55, 214, 183]. On the other hand, *Data*

Slack is strongly data dependent and varies widely and manifests intermittently across different instructions (opcodes), different inputs (operands) and different requirements (precision/data-type).

The focus of this work is on *Data Slack*, and as analysis in Sec.3.3.1 shows, its multiple components can cumulatively produce even greater than half a cycle's worth of slack. The available *data slack* has been increasing, since instruction set architects are under pressure to increase execution bandwidth per fetched instruction, leading to data paths with increasingly rich semantics and large variance from best-case to worst-case logic delay. Furthermore, in spite of rich ISA semantics, or perhaps because of them, even optimum compilers are able to use these complex features only some of the time, but these *richer* data paths contribute to the critical timing all the time [42]. This trend is exacerbated by workload pressures, specifically the emergence of machine learning kernels that require only limited-precision fixed-point arithmetic [220].

The end-product of our proposal REDSOC is to recycle this data slack, to be utilized across multiple operations, and improve system performance. There are three domains of prior work that have explored this goal in different forms, which are discussed below.

The first is *timing speculation* (TS) which was introduced in Section 2.2. TS solutions suffer from the fundamental constraints that they are bounded by the possibility of timing errors from *every* computation, in *every* synchronous EU/op-stage, and on *every* clock cycle. Since data slack has wide variations across operations and since (F,V) operating points can only be altered at reasonably coarse granularity of time, these proposals are forced to be configured conservatively. Moreover, the design overheads in implementing timing error detection and timing overheads from recovery are significant [134].

The second domain is *specialized data-paths*. When specialized data-paths are built to accelerate certain hot code, specific function elements are combined together in sequence and the timing for that data-path can be optimized for the particular chain of operations [193, 236]. But such data-paths do not provide flexibility for general-purpose programming and also suffer from low throughput or very large replication overheads. Thus, they cannot be easily integrated into standard out-of-order (OOO) cores.

The third domain is static and dynamic forms of *Operation Fusion*. These proposals involve identification of sequential operations that can be fit into a single cycle of execution [165] and further, rearranging instruction flow to improve the availability of suitable operation sequences to fuse [23]. Optimizing the instruction flow is a significant design/programming burden, while unoptimized code provides only limited opportunity for single-cycle fused execution in the context of our work.

REDSOC, on the other hand, avoids all of these issues. REDSOC aggressively recycles data slack to the maximum extent possible. It identifies the data slack for each operation based on opcode and operand characteristics (Section 7.1). It then attempts to cut out (or recycle) the data slack from a producer operation by starting the execution of dependent consumer operations at the exact instant of completion of the producer operation (Section 7.2). Further, REDSOC optimizes the scheduling logic of OOO cores to support this aggressive operation execution mechanism (Section 7.3). Recycling data slack in this manner over multiple operations allows acceleration of these data sequences. This results in application speedup when such sequences lie on the critical path of execution (Evaluation in Section 7.4 and methodology in Section 4.3).

REDSOC is timing non-speculative, and thus does not need costly error-detection

mechanisms. Moreover, it accelerates data operations without altering frequency/voltage, making it suitable for fine-grained data slack. It is implemented in general OOO cores, atop the data bypass network between ALUs via transparent flip-flops (FFs with bypass paths) and is suitable for all general-purpose execution. Finally, it cumulatively conserves data slack across any naive sequence of execution operations and neither requires adjacent operations to fit into single cycles nor any rearrangement of operations.

Corresponding Publications: *Gokul Subramanian Ravi and Mikko Lipasti. "Recycling Data Slack in Out-of-Order Cores". HPCA 2019.*

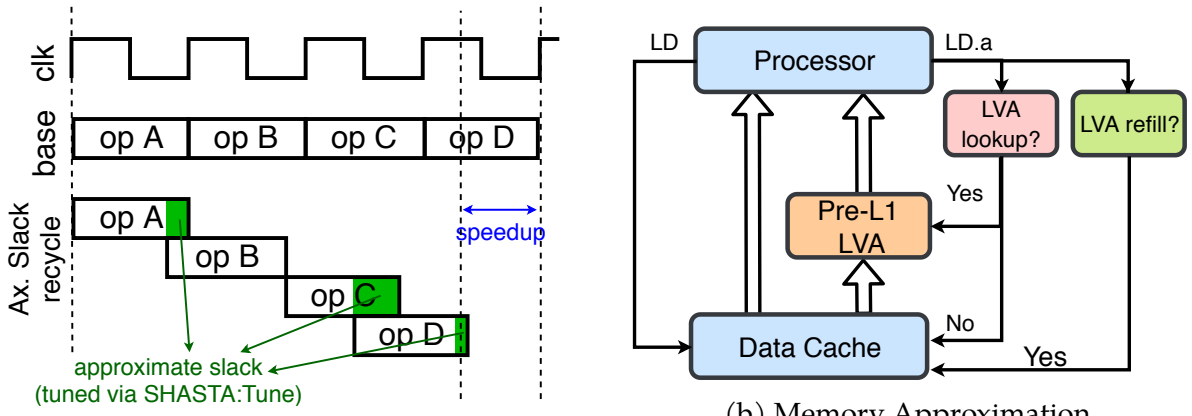
2.4 SHASTA: Synergic HW-SW Architecture for Spatio-Temporal Approximation

The mainstream adoption of approximate computing and its use in a broader range of applications is predicated upon the creation of programmable platforms for approximate computing [223]. The key requirement for efficient general purpose approximate computing is an amalgamation of (i) general purpose hardware designs flexible enough to leverage any amount/type of approximation that an application engenders, and (ii) tuning mechanisms intelligent enough to reap the optimum efficiency gains (in terms of performance and enegy) given a hardware design, an application and a resiliency specification. In this regard there are many existing limitations and opportunities in prior proposals, both in terms of general purpose hardware designs as well as in terms of approximation tuning mechanisms.

We propose SHASTA, a cross-layer solution for building optimal General Purpose Approximation Systems (GPAS) i.e. approximation systems suited to a wide range of general purpose error-tolerant applications, each with unique and potentially fine-grained approximation characteristics. SHASTA tackles the opportunities discussed above to achieve significant benefits in execution efficiency. We classify the goals for building optimal GPAS and how SHASTA addresses them, into three broad categories below.

Enabling fine grained spatio-temporal diversity in hardware approximation: An ideal GPAS should provide the flexibility to control each executing operation uniquely, as accurate or approximate. Moreover, each approximate computation should be ideally allowed to have its own individual/unique amount of approximation. This approximation diversity can be thought of as two components - spatial and temporal. Spatial diversity in approximation implies that each static approximate operation (i.e. every approximate variable in an approximate application's code) should be allowed unique approximation control. Temporal diversity in approximation implies every dynamic instance of static approximation operations (for example, every iteration of an approximate variable) should be allowed different approximations i.e. the approximation applied to the static operation should be allowed to evolve over time (say, across the loop iterations). Further, the system should be able to dynamically reconfigure these approximation settings with low overhead. While such flexibility is easier to explore in software, fine granularities and spatio-temporal diversity of approximation is more challenging in hardware.

For compute approximation, SHASTA proposes a new variant of Timing approximation called *Slack-Control Approximation* (Section 8.1.1). *Slack-Control Approximation* is inspired by our prior proposal REDSOC (introduced in Section 2.3). SHASTA extends REDSOC ideas



(a) Compute Approximation

(b) Memory Approximation

Figure 2.2: Hardware Approximation

to approximate computing, performing more aggressive slack recycling and achieving a flexible approximation scheme which can be controlled on a per-clock-cycle and per-computation-unit basis, thus allowing fine granularity and spatio-temporal diversity. An overview of SHASTA's compute approximation is shown in Fig.2.2.a. The following points can be noted. First, approximation is achieved by reducing computation time, allowing a sequence of operations to be completed faster than the accurate baseline, resulting in speedup. Second, not all computations in the sequence have to be approximate. Third, different approximate computations are allowed varying degrees of approximation (hence varying amounts of slack).

For memory approximation, SHASTA implements *Dynamic Pre-L1 Load Approximation* (Section 8.1.2). Inspired by LVA [148] but going beyond, this technique approximates loads prior to L1 cache access, by reading values out of a small approximator which is integrated close to the datapath - resulting in latency and energy benefits. The key advancement to prior work is the ability to perform fine-grained spatio-temporally diverse approximation. Our implementation enables the control of each unique load's approximation degree (how

often the data in the approximator is refilled) and approximation confidence (how often the data is approximated i.e. looked up in the approximator) independent of other loads. This allows more efficient use of load approximation - creating better fine-grained trade-offs between error and efficiency. An overview is shown in Fig.2.2.b.

Automated Hardware-cognizant approximation tuning: We call the amount of approximation assigned to each approximation operation as the *approximation configuration* of the application. An ideal GPAS requires an intelligent tuning mechanism to identify any given application's optimum approximation configuration. This is especially important in systems allowing fine granularities of spatio-temporal approximation diversity wherein there is potentially considerable efficiency difference between optimal and sup-optimal configurations.

Fig.2.3 shows an example micro-application with two approximate elements. The data points on the scatter plot show various approximation configurations. The points falling on the red line are those configurations satisfying the application's specified error tolerance (for example, 90% accuracy). Among these, the blue dot (HW-A) is the configuration selected by a hardware-agnostic mechanism (akin to [106, 151, 149, 223]), which optimizes towards application's error tolerance in a pre-specified order: in this example, first on element A and then on element B. This configuration can be sub-optimal because the tuning mechanism is not taking into account the impact of this configuration on the hardware's execution efficiency.

On the other hand, SHASTA is novel in its hardware-cognizant approximation-tuning (Section 8.2). Tuning is performed by dynamically evaluating the approximation's actual effect on hardware. The tuning mechanism, implemented over a gradient descent algorithm,

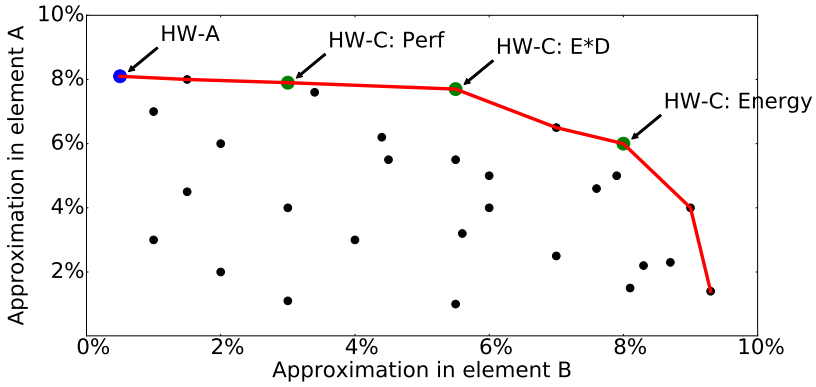


Figure 2.3: Hardware-cognizant Approximation Tuning

iterates through a sequence of approximation configurations and evaluates the application error and hardware execution metric (eg. performance/ energy) for those configurations. Gradually it moves the approximation configuration of the application in the direction of the steepest efficiency-error gradient i.e., one which achieves the best efficiency metric improvement for the lowest change in error, until the optimum configuration is reached. Fig.2.3 shows the approximation configurations chosen by the hardware-cognizant mechanism (HW-C) for different metrics: performance, energy and energy-delay. Not only are the chosen configurations unique for each metric (and optimum for that metric), they are also different from the one chosen by the hardware-agnostic one.

Synergic optimization of varied forms of approximation: An ideal GPAS should be able to employ multiple approximation (eg. compute, memory etc.) techniques in conjunction. This is especially important in general purpose systems wherein the benefits obtained from any one form of approximation might be minimal for some applications. For optimum approximation to be achieved, the hardware-cognizant tuning mechanism described earlier should be able to manage every fine grained and diverse approximate computation of all the enabled approximation techniques, and the prescribed approximation configuration

should be cognizant of the interactions between these various approximation techniques.

In SHASTA, the hardware and software proposals are built into a robust GPAS, aided by a) an ISA with approximation extensions, b) a compiler which takes approximation annotated applications to generate an executable utilizing these ISA extensions, and c) a runtime which is run periodically over the lifetime of an application, which leverages the tuning mechanism to tune the application's approximation configuration in accordance with the application's execution characteristics. We show that this cross-layer approximation system is able to achieve better error tolerant execution efficiency in comparison to prior work and moreover it is able to achieve synergy among the multiple approximation techniques. System details and synergic effects are discussed in Section 8.3.

SHASTA is implemented on top of an OOO core and achieves mean speedups/energy savings of 20-40% over a non-approximate baseline for greater than 90% accuracy - these benefits are substantial for applications executing on a traditional general purpose processing system. SHASTA can be tuned to specific accuracy constraints and execution metrics and is quantitatively shown to achieve 2-15x higher benefits in terms of performance and energy, compared to prior work. Evaluation is discussed in Section 8.4 and methodology in Section 4.4.

Corresponding Publications: (*1*) *Gokul Subramanian Ravi, Joshua San Miguel, and Mikko Lipasti. "Synergic HW-SW Architecture for Fine-Grained Spatio-Temporal Approximation". TACO 2020.* (*2*) *Gokul Subramanian Ravi and Mikko Lipasti. "Axl: Accelerating Approximations via Slack Recycling". WAX 2018.*

2.5 TNT: Attacking latency, modularity and heterogeneity challenges in the NOC

Communication in the exascale era: The capabilities of large-scale computer systems with 10s to 100s of cores are often limited by the inefficiency of on-chip communication. Prior research has shown that server workloads executing on a CMP can lose as much as half of their potential performance due to long latency LLC accesses [81, 61, 82]. A key contributor to the long LLC access latency is the need for data/coherence traffic to often travel across many cores when slices of the LLC are distributed among them. Thus, the design of highly efficient low-latency on-chip communication is fundamental to further the exascale era. In this work, we identify that challenges and opportunities stemming from the need for modularity and the prevalence of heterogeneity in the Network-On-Chip (NOC), are critical to reducing the latency of on-chip communication.

Modularity: As the number of on-chip cores increases, the ability to build NOCs in a modular tile-scalable manner becomes critically important to streamline design and verification [199, 44, 15]. Such a modular design is composed of self-contained NOC tiles, replicated across the entire network. Each of these tiles are made up of the controlling router and the communicating links. Importantly, communication occurs in the form of hops between adjacent routers. The canonical topology for modular tile-scalable networks is the 2-dimensional mesh [95, 96, 227]. While the design-time benefits of modularity are abundantly clear, modularity can severely impair a rather important feature of the network - transmission latency. This is explained below.

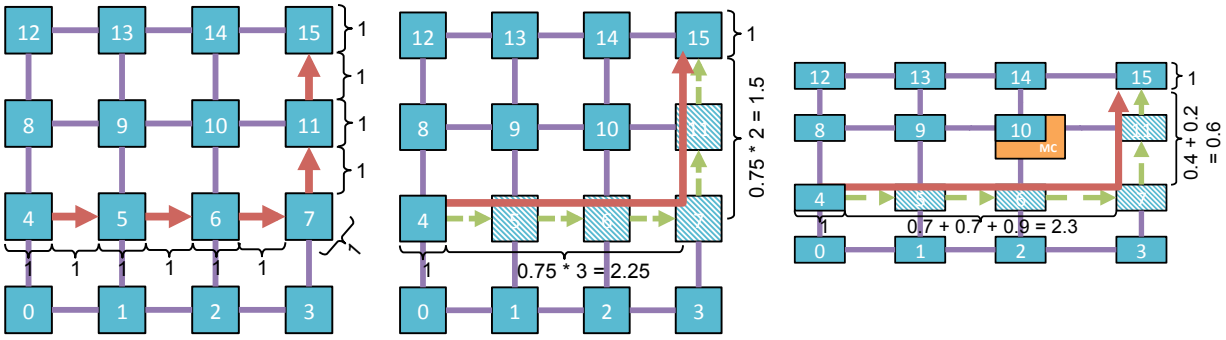
$$T_{base} = H.t_r + H.t_w + T_c + \frac{L}{b} \tag{2.1}$$

$$T_{wire} = (\eta_{chip}.H).t_w + \frac{L}{b} \tag{2.2}$$

$$T_{wire}(Physical) = (\Sigma_1^H \eta_{link}).t_w + \frac{L}{b} \tag{2.3}$$

$$T_{TNT} = 1.t_r + (\Sigma_1^H \eta_{link}).t_w + T_c + \frac{L}{b} \tag{2.4}$$

First, note that the ideal latency (T_{base}) of a packet in the network is described [45] by Eq.2.1, where H is the number of hops, t_r is the router pipeline delay, t_w is the wire (between two routers) delay, T_c is the contention delay at routers, and L/b is the serialization delay for the body and tail flits, i.e. time for a packet of length L to cross a channel with bandwidth b. In the 2-dimensional mesh, the average hop counts increase linearly with expansion in each dimension of the mesh, linearly increasing average packet latency.



(a) Traditional mesh (b) Transparent flow (c) Accounting for heterogeneity

Figure 2.4: 5 hop request in a 4*4 mesh. (a) Traditional mesh takes 11 cycles - 1 per router / link. (b) In TNT, the request flows through in a single pass bypassing intermediate routers. The lookahead requests enabling this are shown in green. In this homogeneous design, the link+switch wire traversal capability is $\eta_{chip} = 0.75$, thus the request completes in 6 cycles. (c) In this non-homogeneous design, nodes have a greater width than length, and node 10 is attached to a large memory controller. TNT is still able to account for per-link delay, allowing request completion in 5 cycles.

Latency: Next, we examine the network's true traversal capability. The ideal latency for a packet to traverse through the network would be the delay incurred from wire traversal alone (i.e. without restrictions imposed by routers and congestion). The wire delay along a packet route of H hops, is proportional to H times the *delay fraction per hop* (termed as η, with $0 < \eta <= 1$), and is a function of frequency, technology node, etc. Thus, the ideal *wire-only* time delay for an uncontested packet to traverse H hops is given by Eq.2.2. Note: this uses η as a conservative chip wide (η_{chip}) estimate.

Thus, it is clear that in a canonical modular network topology like the mesh, the latency of a packet to traverse the network (T_{base} in Eq.2.1), is significantly higher than the *wire-only* delay through the network (T_{wire} in Eq.2.2). The main limiter to achieving lower latency is the impact of modular design: the quantization of network traversal into hops, introduced by the packet's interaction with routers at each node in the network. Hop quantization primarily increases packet latency in two ways: a) it introduces router delays at each hop and b) it under-utilizes the wiring's per-cycle traversal capability, i.e. traversing only a single hop in a cycle even if the wire is capable of multiple hops worth of traversal per cycle ($1/\eta$). This is a cause for concern because for larger systems in the exascale era, the high hop counts would lead to horrendous on-chip network traversal latency and energy, creating a stumbling block to core count scaling [123].

Heterogeneity: Previously, when we discussed wire traversal capability (η_{chip}), we remained agnostic to the impact of physical chip heterogeneity. We assumed that all links are of same length and the wire delay is constant throughout the chip. In reality, even given a logically homogeneous system, the physical design is often heterogeneous. Heterogeneity stems from physical die characteristics: ① Unequal sizes and aspect ratios

of cores, memory controllers (MCs), ② Their sparse distribution across the system, ③ Within-die process variations. Thus, in a real physical layout, the η across a chip is not constant and is dependent on the route of interest, with each link having its individual η_{link}. Thus Eq.2.2 is modified to Eq.2.3.

TNT - coping with modularity and heterogeneity for ideal latency: To reach near-ideal packet latencies in traditional NOCs, the network design should attempt to achieve the above T_{wire} packet delay, but without straying away from the modular design philosophy. This is clearly a challenge considering the conflicting elements in both goals. Further, a solution that is exploiting inherent design characteristics (wire delay), should be able to cope with the physical heterogeneity that is part and parcel of the design. We propose *TNT or Transparent Network Traversal* to overcome these challenges - ideally achieving end-to-end network traversal in a single pass at the best physically capable wire delays, while performing only neighbor-to-neighbor control interactions.

TNT pushes closer to the ideal wire-only latency by attempting source to destination traversal as a single multi-cycle *long-hop*, bypassing the quantization effects of intermediate routers via *transparent* data/information flow, reducing the packet latency to Eq.2.4 (Section 9.1). In TNT, the *long-hop* from source to destination, if without conflicts, results in ⓐ only a one-time router delay and ⓑ allows the physical wiring's traversal capability to be utilized via multiple, possibly non-integral, hops worth of traversal per cycle. Further, ⓒ by tuning η individually for each potentially heterogeneous link/tile, TNT exploits any route's traversal capability to the best extent.

TNT's transparent traversal is achieved by designing *A Modular and Flexible Lookahead Network* - a novel control path (Section 9.2). The control path performs simple delay-aware

neighbor-to-neighbor interactions to enable end-to-end transparent flit traversal. Note that neighbor-to-neighbor interactions create potential for conflicting requests under high traffic. Thus, to allow for TNT's capability of traversing multiple (and variable) hops per cycle, multiple conflict resolutions would need to be performed in sequence within a cycle, without being hindered by the synchronous quantization introduced by conventional sequential logic. To achieve this, the control path is designed as follows. Control is built atop a light-weight mesh. It carries lookahead requests to set up transparent paths, tracks wire delay to monitor accumulation of path delay from one link to the next (Section 9.2.1), performs simple combinational lookahead-request conflict resolutions at the routers (Section 9.2.2) and finally, respects timing (violation) constraints - a must-have for multi-cycle designs (Section 9.2.3). Fig.2.4 illustrates the above discussion.

Analysis on Ligra graph workloads shows that TNT is able to to reduce LLC latency by up to 43%, improves performance by up to 38%, reduces dynamic energy by as much as 35%, compared to the baseline 1-cycle router NOC. Further, it achieves more than 3x the benefits of best alternative research proposals across different metrics. Evaluation is discussed in Section 9.4 and methodology in Section 4.5.

Corresponding Publications: (*1*) *Gokul Subramanian Ravi, Tushar Krishna, and Mikko Lipasti. "TNT: Attacking latency, modularity and heterogeneity challenges in the NOC". Under Submission 2020.* (*2*) *Gokul Subramanian Ravi, Tushar Krishna, and Mikko Lipasti. "McMahon: Minimum-cycle Maximum-hop network". AISTECS 2019.*

2.6 Chapter Summary

This chapter provided an overview of each research proposal discussed in this dissertation. CHARSTAR focuses on intelligent reconfiguration which is cognizant of the power consumed by the clock hierarchy. SlackTrap introduces the idea of slack recycling via transparent datapaths, implemented on a spatial architecture and focused on statistical slack modeling. REDSOC extends the slack recycling proposal to the out-of-order core - requiring optimizations to the out-of-order scheduler as well. Further, it is accurate in its recycling, with specific slack identified in each instruction / computation based on its operation and operand properties. SHASTA builds atop the REDSOC proposal for slack recycling - extending this idea to approximate computing. With the above accompanied by memory approximation, SHASTA is able to perform multiple forms of approximation in conjunction. SHASTA is especially novel in its ability to perform hardware cognizant per-operation approximation tuning. Finally, TNT investigates slack recycling in communication over the on-chip network and proposes a solution with near-ideal wire-only latency. Further is is able to account for physical NOC heterogeneity and takes a modular design approach.

Each of these proposals are discussed in depth over Chapters 5 - 9. Further, their background / motivation and methodology are discussed next, in Chapters 3 and 4 respectively.

3 BACKGROUND AND MOTIVATION

This chapter covers the entire background for the proposals introduced in Chapter 2. Relevant background to each proposal is specified below:

- **CHARSTAR:** Tiled Architectures (Section 3.1.1), CRIB: Consolidated Rename, Issue and Bypass (Section 3.1.2), Reconfiguration in Hardware (Section 3.1.4), Clock Power Distribution (Section 3.2)
- **SlackTrap:** Tiled Architectures (Section 3.1.1), CRIB: Consolidated Rename, Issue and Bypass (Section 3.1.2), Cycle Utilization in Compute (Section 3.3), Timing Verification (Section 3.6)
- **REDSOC:** Out-of-Order Execution (Section 3.1.3), Data Slack (Section 3.3.1), Timing Verification (Section 3.6)
- **SHASTA:** Out-of-Order Execution (Section 3.1.3), Data Slack (Section 3.3.1), Approximate Computing (Section 3.4)
- **TNT:** On-Chip Network Traversal and Cycle Utilization (Section 3.5), Timing Verification (Section 3.6)

3.1 Processor Architecture

3.1.1 Tiled Architectures

In tiled or spatially distributed architectures, resources (functional units, buffer entries, registers and, in some cases, even caches) are structured in the form of multiple small tiles or partitions. Tiled processor architectures which have been proposed with a variety of objectives in mind include TRIPS [194], RAW [213], Wavescalar [209], WiDGET [226], Sharing Architecture [239] and CoreFusion [103]

Tiled designs are not limited to microprocessor architectures and, in fact, are more common among other compute engines. GPUs are a common example. Accelerators are often designed in a spatial framework as well. For example, neural network accelerators such as NPUs [57], Eyeriss [33], the DiaNao family [51, 34] all consists of a sea of compute nodes called processing engines(PEs). General purpose accelerators such as DySER [69] are also organized as a spatial framework.

In this dissertation, our proposals related to tiled architectures (CHARSTAR - Chapter 5 and SlackTrap - Chapter 6) have focused on the CRIB architecture [76], though the benefits and findings are applicable to a broad class of tiled machines [194, 209, 213]. The benefits of tiled architectures towards CHARSTAR's reconfiguration and SlackTrap's slack recycling are discussed in Section 5.2.3 and Section 6.3 respectively.

3.1.2 CRIB: Consolidated Rename, Issue and Bypass

The primary goal of the CRIB processor [76] is to perform an alternate form of out-of-order execution where explicit register renaming is no longer necessary. By removing explicit register renaming, the corresponding supporting structures such as the register alias table, the reservation station, and the reorder buffer can be eliminated. This results in in significant power reduction up to as much as 25% of the total chip power.

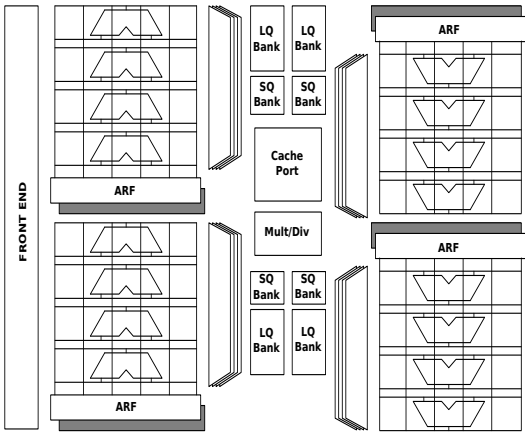


Figure 3.1: Tiles in the CRIB design [76]

The functionality of those structures is consolidated into a single structure. In the CRIB core, the RAT (register alias table), the RS (reservation station), and the ROB are consolidated into one structure namely, the consolidated rename/issue/bypass block, or CRIB. The CRIB consists of multiple partitions, as shown in Fig.3.1. Explicit register renaming is replaced with the concept of spatial renaming. A large consolidated multi-ported physical register file is replaced with multiple small spatially-organized architected RFs (ARF) which consist of a simple rank of latches.

The monolithic CRIB is divided into partitions - with multiple CRIB entries (we use 4)

per partition and one architected RF per partition. Each CRIB entry consists of a computation unit (eg. ALU) and its corresponding routing logic. The routing logic connects logical register columns to the ALU as well as the ALU result back to the appropriate register column. Each instruction in the CRIB taps its source operands from the register columns, evaluates the result and then overwrites its destination register column accordingly. The source tags are used to grab source operands from the ARF. The destination tag is used to overwrite the destination register of the instruction. When all entries in a partition complete execution and the corresponding data is written to the ARF, the partition is committed after which new instructions are inserted into it. The partitions are connected in a circular fashion with instances of ARF between the partitions. Only the ARF at the head of the partition has the committed state of the program and as every partition completes execution, the head or commit pointer is moved to the next ARF instance. Ready instructions across multiple CRIB partitions can evaluate concurrently, exposing OOO-like parallelism. The CRIB partitions are the units of reconfiguration in this dissertation's CHARSTAR proposal in Chapter 5. The benefits towards CHARSTAR's reconfiguration is discussed in Section 5.2.3.

Further, CRIB is novel in its data movement approach. Data forwarding between instructions in the CRIB is accomplished without latches. To enforce fully synchronous execution, only the control bits which are indicative of dependence and completion are latched. This departure from traditional fully latched designs also results in power savings and serves as a starting point for this dissertation's slack recycling proposals (specifically SlackTrap - Chapter 6).

3.1.3 Out-of-Order Execution

We provide a summary of out-of-order processor execution based on [68].

Fetch: The first part of the pipeline is instruction fetch. It includes (a) an instruction cache, which stores instructions, and (b) a branch predictor and target buffer which determine the address of future instructions.

Decode: The next part involves instruction decoding. This includes decoders and ROMs, which identify attributes of the instruction such as type and resources that it will require. In REDSOC (Chapter 7), per-instruction data slack is identified at the decode stage whenever possible. This is discussed in Section 7.1.

Renaming and Dispatch: Afterward, the instructions flow to the allocation, which performs register renaming and dispatch. Register renaming removes false dependencies by mapping to a larger set of physical registers maximizing the potential for instruction-level parallelism. This is performed via tables with information on the mapping of logical register to current physical ones as well those physical registers which are unused. The instruction dispatch consists of reserving different resources (if available) that the instruction will use in the future, including reorder buffer, issue queue and load/store buffer entries.

Issue: The next phase in the pipeline focuses on instruction issue. In a reservation station based model for scheduling, after instructions are renamed, they wait in reservation stations for their sources to become ready. In a conventional design, each reservation station entry (RSE) has 2 parent (or source) tags which are identifiers for the source operands.

The wakeup logic is responsible for waking up the instructions that are waiting for their source operands and execution resources to become available. This is accomplished

by monitoring each instruction's parent (producer) instructions as well as the available resources. Once the tag matches occur, the instruction is woken up and a request is placed to the Select logic.

The selection logic chooses instructions for execution from the pool of ready instructions waiting in its reservation station entries (RSEs). Priority-based scheduling (e.g. oldest-first) is required when the number of ready instructions are greater than the number of available functional units. This happens when tags from parents of multiple instructions become available; these instructions are all awakened and send requests to the select logic. If selected, the instruction's destination tag is then broadcast on the tag bus, so as to wake up following consumer instructions in subsequent cycles(s).

The issue stage of processor execution is optimized in this dissertation's REDSOC proposal (Chapter 7) and is also utilized by SHASTA (Chapter 8). These scheduling optimizations enable the slack recycling mechanisms to be effective in an out-of-order core. They include enhancements to the wakeup logic (Section 7.3.2), the select logic (Section 7.3.4) and the reservation station entries (Section 7.3.3).

In the past, multiple works have optimized scheduling to break-down its critical loop [147, 158, 26, 205, 167] - our scheduling enhancements are influenced by some of these.

Execute and Bypass: When issued, instructions are sent for execution. There are a variety of execution units for different operations, including integer, floating-point, SIMD and logical operations.

An important component of the execution pipeline is the bypass logic. Bypass consists of wires which move results from one unit to the input of other units. This is performed as required - associated logic determines whether the inputs to the units should use the

bypass or should be obtained from the register file.

REDSOC (Chapter 7) replaces traditional opaque latches / flip-flops in these bypass paths with transparent capabilities. These enhancements are discussed in Section 7.2 and are also utilized by SHASTA (Chapter 8). Transparent capabilities are similar in SlackTrap (Chapter 6), except that the transparent flow is between connected processing nodes of the spatially distributed architecture - this is discussed in Section 6.1.2.

Commit: At the end of execution, instructions move to the commit phase which guarantees that the out-of-order non-sequential execution of events within the processor is abstracted away from the outside world by providing the appearance of sequential execution. The reorder buffer plays an important role in the commit phase and is checked to see if older instructions are committed before younger instructions. Once completed and committed, instructions are removed from the pipeline, making appropriate updates and releasing held resources.

Mispeculation and Recovery Sometimes, events performed by the processor have to be undone due to some incorrect speculative sequence of events(eg. branch misprediction and the subsequent execution of incorrect instructions). When this happens, instructions have to be flushed, and some storage (e.g., register file) has to be reset to a previous state.

3.1.4 Reconfiguration in Hardware

In CHARSTAR (Chapter 5), we propose an intelligent clock-aware reconfigurable processor architecture. We look at some related reconfiguration background and proposals below.

Granularities of Adaptivity: In order to enable effective reconfiguration of resources,

granularities of adaptivity across the different adaptive dimensions need to be chosen appropriately. As clearly elucidated by Lee et al. [131], the adaptive computing paradigm provides flexibility of microarchitecture in two dimensions - temporal and spatial. The *temporal dimension* corresponds to the rate at which resources can be efficiently reconfigured. This could range from domain-level to application-level to phase-level adaptivity. The *spatial dimension* represents the microarchitectural scope of reconfigurations - the number of unique resources which can be resized and the number of configurations they are each capable of attaining. This might range from the level of a cluster or a single core to finer levels like pipeline stages or resources such as register files and execution units.

Temporal Granularity: To capture fine grained phases of applications, in the order of hundreds or thousands of instructions, recent proposals such as Composite Cores [141, 157] and the MorphCore [117] propose a temporally fine-grained design by building two contrasting engines within a core. Similar works on resizing of resources, even at the granularity of tens of cycles [122], enable doubling or halving of buffer sizes (eg. ROB) to adapt quickly to the application's needs. These cores are able to adapt very quickly largely due to the simplicity of the control mechanism and very limited resource configurations to choose from (such as the big and little engines in the Composite Core). Thus, such architectures that target very fine grained temporal adaptivity are incapable of finer variations in the spatial dimension.

Moreover, while these architectures are *theoretically capable* of adapting to the smallest of application changes over time, the ability is somewhat underutilized. At very small switching intervals on the order of hundreds of cycles, the minuscule variations within most applications are not significant enough to warrant shifts between coarsely granular

architectural configurations (such as Big and Little engines). Such configurations are far apart on the spectrum of performance and power, thereby frequently rendering such schemes ineffective.

Spatial Granularity: At the other end of the adaptability spectrum, prior works have aimed at mutable architectures, capable of simultaneously resizing multiple processor resources. These resources could range from the front end, to functional units and/or to the back end [169, 52], to enable the processor to adapt precisely (resource-wise) to the needs of the application. By varying multiple microarchitectural parameters, each across a range of values, these processors often encounter an adaptable design space with billions of nodes (configurations) [52]. The large design space means that searching through these nodes to find the ideal configuration can happen only at a coarse grained temporal granularity of millions of cycles. Such schemes lose out on finer temporal variations, again skewing the spatio-temporal adaptivity balance.

CHARSTAR chooses balanced reconfiguration granularities across both dimensions of adaptivity for optimal reconfiguration. This is motivated and discussed in Sections 5.2.2 and 5.3.1.

Coupled reconfiguration: Core folding and DVFS are implemented in CMPs, but usually in a decoupled manner. Works such as [142], in fact, suggest that per-core power gating and DVFS should be implemented in a decoupled fashion due to increased complexity and difference in characteristics between the mechanisms. On the other hand, others [222] have shown the smart combination of both mechanisms can only improve efficiency at the CMP level. The few works exploring these as coupled mechanisms [222, 171] are restricted to the CMP level and only perform heuristic based control. In contrast, CHARSTAR targets

joint optimization of power gating and DVFS within a single core (Section 5.2.1) using a lightweight machine learning predictor.

Intra-core reconfiguration: Within a single core, Albonesi et al. [5] have studied complexity-adaptive hardware i.e. the dynamic control of clock-rate/latency vs. resource size for particular resources such as issue queue and caches. Sen et al. [197] explore opportunities in shutting down portions of cache resources and increasing the core clock frequency. At the circuit level, significant prior work analyzes various features of clock trees, their gating and related design optimizations [156, 47, 49, 198, 229].

Prediction mechanisms: Prior reconfiguration works include prediction mechanisms that track unique resources, such as the occupation of the instruction queue [170], IPC variation [27], multiple L1 cache misses [94], L2 caches misses [122] or the contribution of the most recently enabled tile [63], but these are less effective in modeling performance. Prior work towards CMPs include improving uncore energy efficiency via DVFS [230], anticipating the system-level performance impact of resource allocation across multiple cores at runtime [18], adapting multiple cores with *lanes* to suit stringent power budgets by sampling on multiple configurations [169] and to uniformly scale multiple cores and their resources [66]. As far as a single core is concerned, Dubach et al. [52] propose a high-end multi-dimensional machine learning based scheme to perform a limits analysis in spatial granularity - scaling a large number of resources within a single core, each with multiple resource sizes.

The underlying constraint in all of these works is that they make use of very complex (and often online-based) ML models as they generally target fine grained architectural (spatial) variations. This requires searching through a complex design space making them

unsuited to fine temporal granularities. On the other hand, CHARSTAR shows the utility of a low-overhead lightweight offline MLP neural network even in small design spaces (Section 5.2.4).

3.2 Clock Power Distribution

In CHARSTAR (Chapter 5), reconfiguration is novel in its cognizance of the clock hierarchy's power distribution. An illustrative example of the impact of clock hierarchy consideration on the efficiency of gating / reconfiguration is provided in Section 5.1.

The clock distribution network consumes a significant portion of the total power in processors and SOCs, prior work claims 30-50% of total chip power and/or up to 70% of the dynamic power [137, 49, 160, 140]. A clock distribution system (CDS) is typically the largest net in the circuit netlist and operates at the highest speed of any signal within the entire synchronous system, hence consuming significant power [132, 48, 72, 215].

Clock power has increased with technology scaling [132, 100]. First, long global interconnect wires have become significantly more resistive as wires become thinner [132, 91]. Clock signals are affected by this increased wire resistance and require precise control of clock-signal arrival times, as they otherwise severely limit the maximum performance of the entire system. This has boosted the demand for repeaters in clock networks, increasing their power profile and complicating their synthesis. Second, with shrinking cycle times, the impact of process, voltage and temperature (PVT) variation also impacts clock skew and reliable clock networks have become more costly in terms of area and power [132].

3.2.1 Different Clock Distribution Systems

Clock distribution systems can broadly be divided into 3 styles - trees, grids (meshes) and hybrids. Different tree distributions and mesh/hybrid are shown in Fig.3.2 and Fig.3.3 respectively.

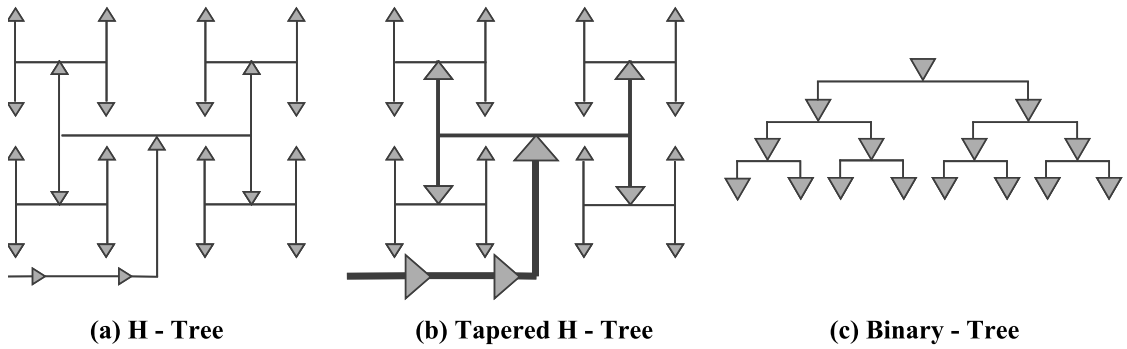


Figure 3.2: Clock Trees

Trees: H-trees and binary trees are commonly used for clock distribution. Trees are attractive because of their low power and low metal usage [232]. If a H-Tree is completely balanced, it exhibits identical nominal delay and identical buffer and interconnect segments from the root of the distribution to all branches and thus would exhibit low skew. But if unbalanced, skew could be very high. In general, idealized buffer placements associated with a balanced H-tree may be difficult to achieve.

In the case of a tapered H-tree, the trunk widths increase geometrically toward the root of the distribution to maintain impedance matching at the T-junctions [64]. This strategy minimizes reflections of the high-speed clock signals at the branching points. Circuit analysis shows that the impedance of the conductor leaving each branch point must be twice the impedance of the conductor providing the signal to the branch point in a tapered H-tree structure [64].

A binary tree is uni-directional and delivers the clock in a balanced manner in either the vertical or horizontal dimension [232]. Similar to H-Trees, all branches exhibit identical buffer-interconnect segments, zero structural skew, and similar PVT tracking. In contrast to the H-tree, the buffers in a binary tree can be placed in close proximity along a centralized stripe. The closer physical proximity of the buffers in a binary tree can result in sensitivity to on-die variation but minimize floor-plan disruptions and are easier to design in a balanced manner.

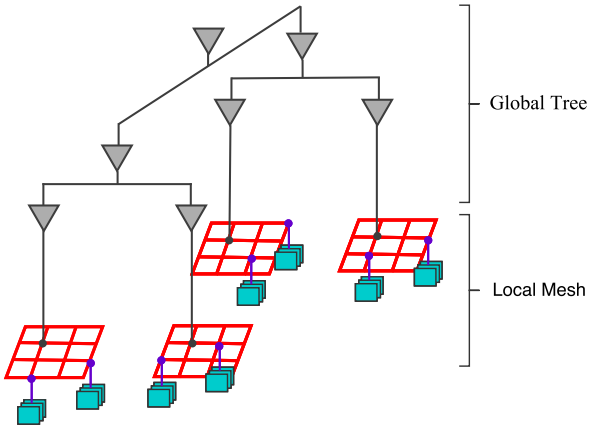


Figure 3.3: Hybrid Clock Distribution

Grids/Meshes: The lowest clock skew is achieved with Grid style networks but they require lot of metal resources and dissipate a lot of power [232]. A clock grid resembles a mesh with fully connected clock tracks in both dimensions and grid drivers located on all four sides (shown in red in Fig.3.3). Local clocks are supplied by directly connecting to the grid. The grid effectively shorts the output of all drivers and helps minimize delay mismatches. The shorted grid node helps balance the load non-uniformities and results in a more gradual delay profile across the region.

Hybrids: Hybrid clock distribution systems combine one or more of the above tech-

System	Pentium4	Itanium2	SPARCIV	AMDK7	AlphaEV7	Xeon	Opteron	POWER6
CDS	Tree/Grid	Tree	Grid	B-Tree	Tree/Grid	Tree/Grid	H-Tree/Grid	H-Tree/Grid

Table 3.1: Commercial Clock Distribution Systems

niques as shown in Fig.3.3 Some hybrid systems have different parts of the chip clocked using different distribution techniques. Most current processors use hybrid techniques as they simplify skew reduction and have relatively low power dissipation.

Table 3.1 lists clock distribution systems employed by popular commercial processors from the previous decade [132, 177]. The analysis in this thesis assumes a standard tree-grid hybrid, with the use of grids to clock local intra-tile resources and a global tree to drive the local grids. CHARSTAR's evaluation (Section 5.4.1) analyzes clock aware reconfiguration across H-Tree, Tapered H-Tree and Binary Tree hierarchies. Portions of these global tree topologies can be gated when not in use, while the local grid can only be gated as a whole.

3.2.2 Estimating Clock Node Power

$$P_{Clk_M} = C * V^2 * f \tag{3.1}$$

$$P_M = N * C * V^2 * f \tag{3.2}$$

$$P_{M-1} = (N/k) * C * V^2 * f \tag{3.3}$$

$$\sum P_i = [\frac{1-(1/k)^M}{1-(1/k)}] * N * C * V^2 * f \tag{3.4}$$

In this section, we present a model for clock distribution power [232]. Let us consider the switching power of a single unconditional clock at the final distribution stage M with capacitance C (load + portion of interconnect) and voltage V at frequency f. This is shown

in Eq.3.1. Considering there are N clock nodes in the stage, the total dynamic power in stage M leads to Eq.3.2. Assuming a fan-out of k at each stage, the power in stage $M-1$ would be given by Eq.3.3. Summation over all M stages gives the total clock distribution power, as shown in Eq.3.4. If sub-trees within this distribution network are power gated, that portion of power can be reduced from the above accordingly. CHARSTAR's evaluations (discussed in Section 5.4) incorporate the above power model in estimating clock tree gating based power savings.

3.3 Cycle Utilization in Compute

Traditional processors fix operating points conservatively so that even the most critical computations and variations do not violate timing. In the common case this results in clock cycle slack. Traditionally, slack can broadly be thought to have two components: ① *PVT Slack*, caused under non-critical PVT conditions, and ② *Data Slack*, caused due to non-triggering of the executional critical path. *PVT Slack*, with its relatively low temporal and spatial variability, can more easily be tackled with traditional solutions [77, 55, 214, 183]. On the other hand, *Data Slack* is strongly data dependent and varies widely and manifests intermittently across different instructions (opcodes), different inputs (operands) and different requirements (precision/data-type). Our proposals SlackTrap (Chapter 6) ad REDSOC (Chapter 7) perform accurate slack recycling i.e. eliminating the excessive slack within computations and thus improving computing performance.

3.3.1 Data Slack

More often than not, a circuit finishes a computation before the worst-case delay elapses, because the critical paths of the circuit are inactive. Xin et al. [234] analyze timing for ALPHA and OpenRISC ALUs, post synthesis and place-and-route. Their analysis shows that roughly 99% of timing critical paths are triggered by less than 10% of all computations. Similarly, Cherupalli et al. [38] perform data slack analysis for a fully synthesized, placed, and routed openMSP430 processor and show that more than 75% of the timing paths have greater than 30% clock cycle slack.

In order to recycle the considerable data slack it is important to categorize it based on its sources. These sources/categories are discussed below:

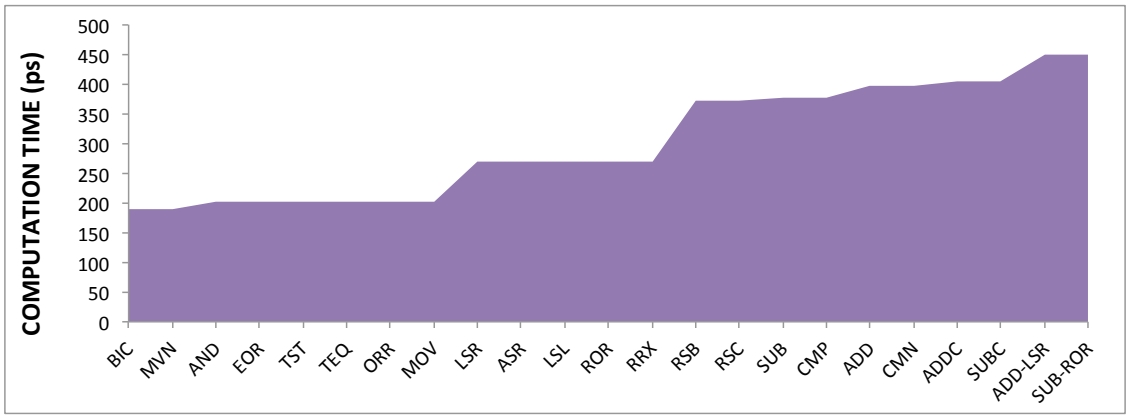


Figure 3.4: Computation Time for ALU Operations (designed for 2GHz)

Operation Type (Opcode-Slack): In general purpose processors, it is common to have functional units that perform multiple operations, with different opcodes. In a conventional timing conservative design, the functional unit would be timed by the most-critical computation in order to be free of timing violations. Thus, many of the executing opcodes/-operations do not trigger the critical path of the functional unit and end up producing data

slack.

Further, the semantic richness of current-day ISAs means that multiple modes of operations are supported via the same data paths. For example, the ARM ISA-based designs support a *flexible second operand* input to the ALU to perform complex operations such as a shift-and-add instruction. Supporting such complex operations via the enhancements to the standard datapath further elongates the critical path of execution. These rich/complex semantics are frequently unused, resulting in even higher data slack.

Fig.3.4 shows the critical computation times for different operations on a single-cycle ARM-style ALU, coded in RTL and synthesized (2 GHz target) for a TSMC 45nm standard-cell library using Synopsys Design Compiler. It is evident that a large number of ALU operations (eg. logical) have significantly lower computation times than more critical arithmetic operations. And even these arithmetic operations produce some data slack in the absence of modifications to the second operand. It is, therefore, intuitive that ALUs would produce considerable data slack across common applications and that this data slack can be intermittently distributed depending on the application characteristics. This form of slack is easily identifiable for the operations, simply by means of the instruction opcode.

Data Width of Operands (Width-Slack):

High-end processor word widths are usually 32-64 bits while a large fraction of the operations are narrow-width (large number of leading zeros). The execution of such operation on a wide(r) compute unit means that there is low spatial and temporal utilization of the compute unit. Low spatial utilization refers to the higher-bit wires and logic-gates which are not performing useful work, while low temporal utilization refers to data slack

from non-triggering of the entire critical propagation paths.

Computations with a significant number of higher-order zero bits are especially common in machine learning applications; many synapses have very small weights, a characteristic exploited in multiple prior works to improve spatial utilization [111]. Low spatial utilization (resulting in unnecessary leakage power) has also been attacked in traditional architectures by aggressive power gating of functional units and operand packing [25], among others.

But the problem of low temporal utilization for narrow-width computations has not been explored. Fig.3.5 shows the varying length of the critical path on a 16-bit Kogge Stone adder for different bit-widths. The colored paths show increasing critical delays/paths for computations of increasing widths. When only a smaller portion of the total data-width is in use, the length of the critical carry-bit propagation path (and thus, the critical delay) reduces, roughly proportional to $\log(datawidth_{eff})$. This form of slack can be estimated via data-width identification. Data-width identification at the time of execution can be performed via simple logical operations at the input ports to the functional units [25]. Prediction methods for identification of data-width early in the execution pipeline, have also been very successful [139, 53].

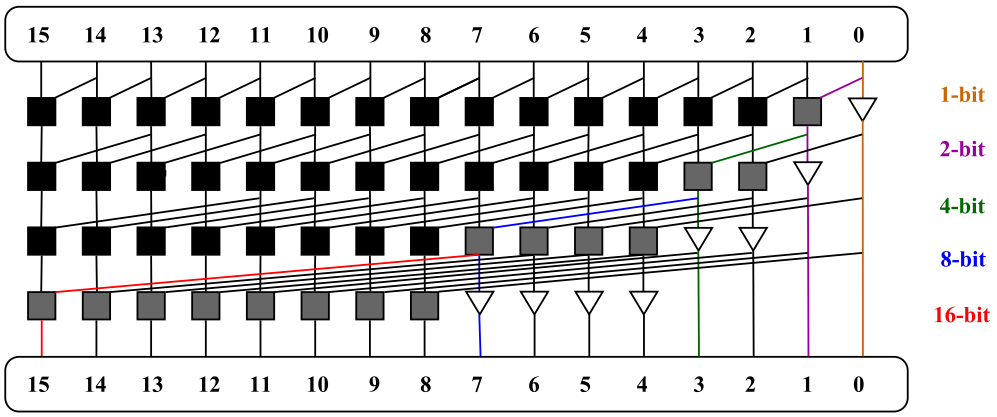


Figure 3.5: Critical paths for KS-Adder

Data Type of Operands (Type-Slack): Sub-word parallelism, in which multiple 8/16/32/64-bit operations (i.e. sub-word operations of *smaller* precision/data types) are performed in parallel by a 128-bit SIMD unit, is supported in current processors via instruction set and organizational extensions (eg. ARM NEON, Intel MMX). This is yet another form of improving spatial utilization and another case of opportunity to improve temporal utilization. The varying compute latency for different data-widths is similar to Fig.3.5, but the method of identification is from the ISA itself, rather than from observing the bits of the inputs. Low-precision computation has especially gained popularity in machine learning applications over the past few years [110], often enabling the use of narrow data types, specified directly by software.

To summarize, current-day applications often exhibit a diverse distribution of operations with plentiful data slack. An effective mechanism to recycle this data slack, in order to speed up sequences of operations, can therefore have substantial opportunity to accelerate these applications. Further, conventional epoch-based voltage and frequency scaling is not effective for capturing this type of slack, since it isn't pervasive, but only manifests intermittently in ALU operations. Hence, we need a scheme to identify and track slack on an instruction-by-instruction basis, and a very fine-grained recycling mechanism, to benefit from it. REDSOC's decoder enhancements to identify data slack are discussed in Section 7.1 whereas SlackTrap uses a statistical model for slack estimation (Section 6.1.1). Further, compute slack recycling mechanisms are discussed for REDSOC in Section 7.2 and for SlackTrap in Section 6.1.2

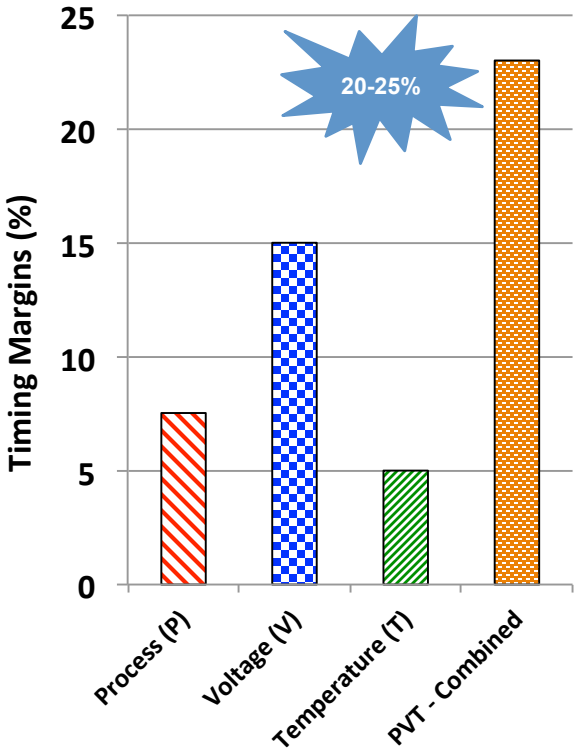


Figure 3.6: Timing guardband impact of PVT

3.3.2 PVT Slack

As chip designers attempt to reduce supply voltage to meet power targets, parameter variations are a serious problem. Environment induced variations which affect the functioning of a processor fall into three categories: process, voltage and temperature. Process variations are caused due to wafer characteristics, doping fluctuations etc., leading to potentially large variations in device attributes [77]. They are broken down into Die-to-die (D2D) and Within-Die (WID) variation. WID consists of a systematic component characterized by spatial correlation and a random component with no correlation characteristics [195]. Random variability increases sharply as supply voltage scales down: scaling from 1.0V to 0.3V increases variability by 6x, making this a significant component at NTV [138].

Supply voltage and on-chip temperature also vary with workload and environment. Voltage variations result in current fluctuations on the order of 10s to 100s of cycles [77] and can also exacerbate thermal hot spots. Thermal variations cause changes to leakage current and restrict permissible voltage and TDP in the chip's environment. The breakdown of the contribution of each of these parameters to the timing guardband is analyzed in Tribeca, with voltage being the largest contributor [77] and is shown in Fig.3.6 In all our compute slack recycling proposals, this traditionally wasted timing guardband stemming from PVT is recycled for better performance. Methodology for modeling the effect of PVT is discussed in Sections 4.2 and 4.3. The slack identification and recycling mechanisms for SlackTrap is discussed in Chapter 6 and for REDSOC in Chapter 7.

3.3.3 Prior works

To reduce the timing guardband needed (especially to accommodate PVT variations), prior works attempt to predict or detect the variations themselves or their effects on timing/voltage. The best known technique to adaptively control voltage guard bands in the presence of data dependent variation is Razor [55]. Razor [55] performs core DVS, tuning voltage based solely on frequency of timing errors. Multiple prior works have focused on static variations alone, recognizing that variation affects some pipeline stages more than others and accordingly tune delays per stage at fabrication [214, 136]. POWER7 [134] introduced Critical Path Monitors to estimate the delays caused by PVT variations and implemented feedback controllers to adjust V/F. Tribeca [77] uses a last value predictor for V/F settings over discrete time intervals based on dynamically gathered PVT information.

It also optimizes recovery mechanisms for low cost detection and recovery from timing errors. TIMBER [40] detect timing errors after the clock edge and borrow slack from the next pipeline stage. Xin et al. [234] predicted likely erroring instructions and allow them an extra cycle of execution, but giving them an entire extra cycle might be inefficient. Multiple "Better than worst case" approaches have been proposed in prior work [55, 70, 9, 10], especially targeting PVT variation [77, 214, 134].

Prior works optimize narrow data-width based execution to improve EU utilization [25], effective register capacity [53], issue width [139] and energy reduction in multiple parts of the core [75]. Fast ALU computations are implemented in some Intel processors [90].

3.4 Approximate Computing

Workloads from several prevalent and emerging application domains, such as vision, machine learning, and data analytics, possess the ability to produce outputs of acceptable quality in the presence of inexactness or approximations in a large fraction of their underlying computations [223]. Allowing computations to be approximate can lead to significant improvements in processor efficiency because it alleviates the "correctness tax" [57] imposed by always accurate systems. This "correctness tax" can take multiple forms. In software, this could refer to overzealous functional accuracy or excessive loop iterations whose execution has low impact on the application's accuracy [191]. In hardware, it could refer to the higher power/latency of accurate compute operations or the effectively lower throughput stemming from full bit-width memory operations.

Our proposal SHASTA (Chapter 8) advances general purpose approximate systems in

many ways. First, it enables fine grained approximation in compute, building atop RED-SOC, as well as in memory (Sections 8.1.1 and 8.1.2). Second, it tunes the approximation configuration of the application in an intelligent hardware cognizant manner, so as to maximize the execution efficiency for a given error tolerance (Section 8.2). Third, it is able to coordinate among multiple forms of approximation in a synergic manner (Section 8.3). Related background on all these three fronts are discussed next.

3.4.1 Hardware approximation with fine spatio-temporal diversity

Prior advancements at the application, compiler and ISA levels [192, 223, 29, 21, 161, 150] have enabled fine-granularities and diverse approximation at the software level. Unfortunately, reaping the benefits of software enabled fine granularity and spatio-temporally diverse approximation in hardware is complicated, especially when targeting general purpose approximate systems (GPAS).

```
@Ax(0.9) float kmeans(dataSet, k) {
   while(it < MAX || condition) {
      ----;
      ----;
      #Euclidean distance
      for (int i = 0; i < v1.length; ++i) {
         float d = @Ax(v1[i] - v2[i], k1[it]);
         sum = @Ax(sum + @Ax(d*d, k2[it]), k3[it]);
      }
      ----;
      ----;
      it++;
   }
   return centroids;
}
```

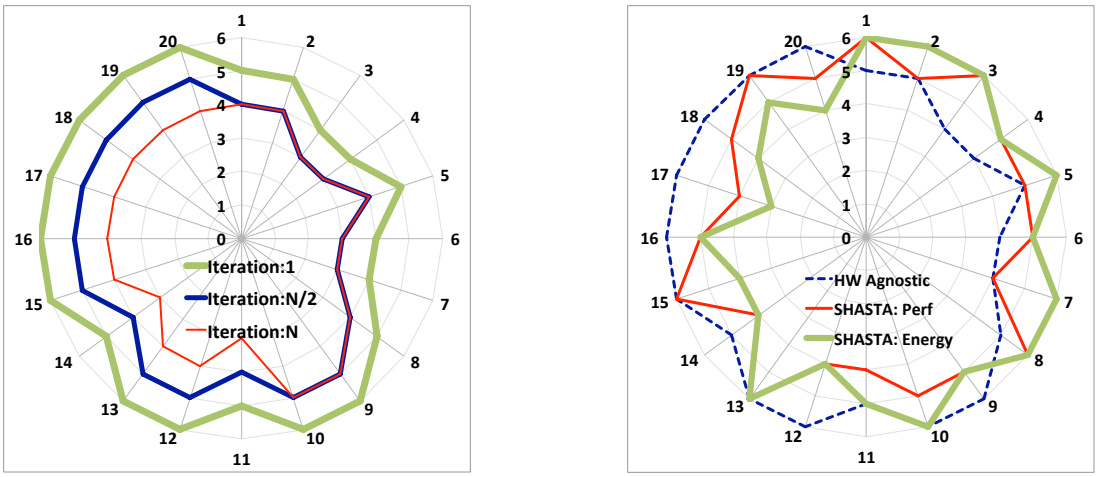
Listing 3.1: Approximation enabled programs

Consider the code snippet in Listing 3.1 that performs K-Means clustering under some

specified error tolerance. The snippet shows one portion of the application which involves calculation of Euclidean distance. It involves 3 specific computations marked as approximate: subtraction, multiplication and addition. All other operations are accurate - for instance, the for-loop induction variable *i* is not approximated to avoid compromising control flow. It is evident that approximate applications like Listing 3.1 tend to contain a fine-grained mixture of accurate and approximate operations. Further, in Listing 3.1 the spatial and temporal diversity in approximation is evident. Spatial diversity is enabled by a unique k_i for each approximate operation, which denotes the 'level' of approximation for op_i. Temporal diversity is indicated by the k[it] i.e. a static approximate operation can have different approximation 'levels' over different iterations of the while-loop. In terms of diversity, while recent efforts have explored either one form, mostly in software [21, 161, 150] but some in hardware [223], no prior work has tackled an optimal amalgamation of both.

Spatio-temporal diversity is further illustrated in Fig.3.7.a for a toy approximate application (figure details described in caption). The different radial values across the circumferential variables indicates spatial diversity. Further, the changing configuration over iterations of the application (moving from green to blue to red) indicates temporal diversity. The approximation across the variables increases from iteration 1 to iteration N/2, typical of clustering-style applications (eg. K-Means) which settle close to appropriate clusters in early iterations. Moreover, the load operations become even more approximate from iteration N/2 to iteration N, indicative of saturating of stored memory values over time (thus allowing more approximation).

In light of applications with fine-grained intermingling of approximate and accurate



(a) Spatio-Temporal Diversity (b) HW Cognizant Tuning

Figure 3.7: The approximation configuration for a toy application with 20 approximate elements, each with 7 different approximation levels. The circumferential axis denotes the variables while the radial axis denotes the approximation level for that variable. The lower the radial value, the greater the approximation. Also, the first 10 approximate elements are compute operations while the last 10 are load operations.

operations such as the code in Listing 3.1, most prior approximate compute hardware for general-purpose applications employ dedicated functional units for performing both accurate and approximate operations concurrently. Resulting hardware challenges include resource duplication (a side effect being resource under-utilization), operation scheduling complications, design complexities [39, 56], circuit overheads [56], and thus might offer control only at coarse granularities such as vector operations [223] or code blocks. Note that this still does not support diversity - in order to support diverse approximation in conjunction, these proposals would require even more duplicate resources (one for each approximate level), severely exacerbating hardware and management complexities. Note: In SHASTA, in the context of temporal diversity we specifically tackle iterative applications i.e. we allow a static operation's approximation to evolve from one iteration to the next.

SHASTA is able to achive fine-grained spatio-temporal approximation diversity in

hardware. This is discussed in Sections 8.1.1 and 8.1.2.

3.4.2 HW-cognizant Approximation Tuning

An ideal approximation tuning mechanism needs to be able to perform the following:

① Identify the tolerable approximation for an application (In the Listing 3.1 Code snippet, this is 10%). ② Identify application elements i.e. variables/computations that can be approximated (Code snippet: 3 compute operations and related loads). ③ Identify potential approximation configurations such that their assignments to application elements meets the application's error tolerance requirements. ④ Choose the right approximation configuration so as to maximize any specified execution metrics such as performance, energy efficiency or power.

From a hardware-software co-design perspective, the fourth challenge is most significant. Most of the prior proposals which target general purpose computing perform tuning which is relatively "hardware agnostic". This means that even though the overall goal of the approximation is to improve execution efficiency, the approximation tuning mechanism is mostly oblivious to the quantitative effects of the chosen approximation configuration on hardware execution (eg. IPC / Energy). They ignore target hardware characteristics and only algorithmically maximize the element-wise approximation until the application is closest to its overall error tolerance [106, 151], or use only minimal statically obtained hardware metrics [149, 223].

While some solutions which perform system cognizant approximation tuning have been proposed (with varying knowledge of the system-level effects of their particular

approximations) [206, 92, 93, 172, 175, 174, 83] these solutions are mostly not designed for general purpose systems and further, are less suited to fine granularities of diverse approximation. On the other hand, SHASTA targets general purpose approximate systems (GPAS) and specifically focuses on fine-grained and diverse approximation, with potentially 100s of approximate elements, which significantly complicates the implementation of hardware cognizance.

Fig.3.7.b shows different approximation configurations chosen by tuning mechanisms. All the configurations are for an error tolerance of 10%. The figure shows an approximation configuration chosen by a hardware-agnostic) tuning mechanism in blue. It also shows the configurations selected by SHASTA in green and red. While the red configuration uses highest performance as the optimization metric, the green uses lowest energy.

The hardware agnostic configuration is the result of a greedy optimization algorithm over the variables (as used in prior work) - it approximates variables 1-5 aggressively (which it tunes first) almost maxing out on the error tolerance, and thus is forced to be very conservative on later variables 5-20. Note that the above tuning is being performed simply one variable after the other, until the error tolerance is reached, irrespective of the execution efficiency impact. SHASTA's configurations, on the other hand, are achieved by hardware execution cognizant tuning over each variable - more aggressive approximations for variables that provide better improvements to execution efficiency. Note that the configurations generated by SHASTA are different when the execution metric changes. The energy-optimum scenario (green) tunes the memory variables (10-20) more aggressively because memory approximations not only reduce latency (for better performance) but also reduce memory access which directly impacts energy. Details on SHASTA's tuning

mechanism are found in Section 8.2.

3.4.3 Synergic Approximation System

Finally, we stress the need for synergic capabilities of the approximation HW-SW framework. In Listing 3.1 and Fig.3.7, two forms of approximation are being employed: compute approximation and memory/load approximation. In an ideal approximate platform: ① the different approximation techniques in the hardware stack should coexist independent of each other, ② the hardware-software interface should abstract away the particular technique of approximation used and ③ the software stack should be able to tune all forms of approximation in synergy.

For example, while compute and load approximation techniques are independent, the tuning mechanism should be cognizant of their intersecting dataflow i.e. the approximate loads might be consumed by the approximate compute, thus the errors from each of those operations can constructively or destructively intersect with each other. In this example, we refer to constructive intersection as (a portion of) the error in the load operations being masked away by the approximation in the compute operation. Conversely, destructive interference would amplify the individual errors.

SHASTA's ability to coordinate among multiple forms of approximation in a synergic manner is discussed in Section 8.3.

3.4.4 Survey of Approximation Techniques

General Purpose Timing Approximation: We identify Truffle [56] to be the closest related prior work to SHASTA in terms of compute approximation - it focuses on GPAS and performs voltage scaling based timing approximation. It also allows timing violations in the SRAM (as a form of memory approximation). We compare it with SHASTA in terms of SHASTA's characteristic features. Truffle is unable to achieve fine-grained latency reduction nor is it able to alternatively use the same resources for both approximate and accurate execution. Further, it does not support multiple levels of approximation and does not attempt to intelligently balance or tune different forms of approximation. Thus it is unable to achieve fine granularity spatio-temporally diverse approximation. Quantitatively, Truffle provides energy reduction of 2%-10% at non-definite application error. In comparison SHASTA provides around 30+% energy reduction at well defined 90% application accuracy. SHASTA's evaluation is discussed in Section 8.4 while the applications and evaluation methodolgy is found in Section 4.4.

Application-specific Approximation Systems: Prior works have proposed building approximation systems from the ground up which target a single application or domain. These systems often achieve synergy in approximation as well as perform system aware approximation tuning. For example, Hashemi et al. [83] target biometric security systems, building an end-to-end flow from an input camera to the final iris encoding with intermediate approximate computational steps. Approximations at various stages are chosen based on their effects on the system, with the help of a Recurrent Neural Network (RNN). While suitable to application specific system, they do not discuss extensions to general purpose

approximation. Further, the approximation methods do not target fine granularities like in SHASTA. The use of an RNN at such fine granularities can cause explosive overheads.

Similarly, prior works from Raha et al. [174, 175] have also targeted Smart Camera Systems [175, 172] and Reduce-and-Rank kernels [174], again achieving synergy among different forms of approximation while focused on an application-specific subsystem. While also using a gradient descent style tuning mechanism (like SHASTA - Section 8.2), these proposals again do not target general purpose approximation with fine-granularities or spatio-temporal diversity.

Approximation at the Application-level: Approximation proposals from Hoffmann et al. [93, 92] have also analyzed trade-offs between accuracy and system energy, but at the application level. While suitable to many applications, these approximations and knobs are hand chosen from one application to the next [93] so are not easily malleable towards diverse general-purpose workloads. Apart from focusing on software-level solutions, these proposals do not target fine-granularities or spatio-temporal diversity in their approximation methods. Moreover, in [93], the approximation tuning searches through the entire configuration space, making it less suitable for a vast tuning space which is imposed by finer granularities and diversity of approximation. Further, the reinforcement learning approach proposed by [92] is again less suited to the finer granularities which is SHASTA's focus.

Similarly Panyala et al. [161] perform application-specific approximation at the software level targeting graph algorithms - they use techniques such as loop perforation, incomplete graph coloring and synchronization. While loop perforation and synchronization are suited to general purpose iterative applications, in this work they are tuned specifically to

PageRank and Community Detection. These methods are orthogonal to the approximation techniques in SHASTA and can be used in conjunction. Loop perforation is closest to the temporal diversity targeted by SHASTA. In their work, the benefits obtained by loop perforation alone for Community Detection is very limited - this is because of a) the absence of spatial diversity i.e. loop perforation eliminates entire iterations and b) absence of an intelligent tuning mechanism to select the right iterations for perforation/approximation. On the other hand, SHASTA shows that performing diverse fine grained approximation with intelligent tuning has substantial efficiency benefits (evaluation in Section 8.4).

Prior work such as EnerJ [192] enable programmers to annotate programs specifying operations or variables to be accurate or approximate via extensions to the programming language, compiler and ISA. Recent advancements at application [29, 21] and ISA [223] levels have furthered software approximation capability beyond only a binary distinction between accurate and approximate regions [192] to allow for spatially diverse approximation control. Software approximation solutions have studied temporal approximation in the context of loop perforation which skips a specific iteration subset of an iterative application [161, 150]. They have shown that iterative clustering algorithms like K-Means stabilize their solutions in an early fraction of the iterations (less than 1% change beyond 20% of the iterations).

Compute Approximation: Dedicated approximate units can run at low voltage [56], can be of specific low precision [39] or can perform temporally coarse-grained precision scaling via power/clock gating [223]. These solutions can incur significant design overheads, i.e. more functional units, dual (accurate/approximate) voltage rails, and scheduling logic overheads. Such proposals do not support multiple approximation levels concurrently, as

this might require dedicated voltage rails (voltage scaling) or uniquely partially power-gated data paths (precision scaling) for each level of approximation supported. Thus, the employed approximation in hardware has to be kept conservative or runs the risk of hitting intolerable operation/application error. Timing approximation is a form of approximation that controls the computation timing [187] - specifically it reduces computation timing sacrificing accuracy for gains in execution efficiency. Timing approximation is usually achieved by under-volting [56] and is constrained in terms of its flexibility and ability to be reconfigured at fine granularities. Other forms of hardware-level approximation include, but are not restricted to, synthesis techniques [143, 154], self-healing designs [67], libraries for approximate-circuits based design space exploration [152] and design methodologies for approximate CGRAs [3]. Compute Approximation in SHASTA is discussed in Section 8.1.1

Tuning for Approximation: It is generally assumed that the tolerable amount of approximation for an application is known as part of the application's or domain's specification. Moreover, it is usually assumed that the programmer can annotate the application to identify which elements can be approximate. Some software proposals have optimized this by requiring the programmers to only specify that a program can be approximate and automatically infer the operations and data that can be safely approximated [162, 189]. After elements for approximation and application error tolerance are identified by programmer/domain specification, prior software proposals have analyzed program-flow and input data sets to identify how much approximation each of the approximate elements can tolerate while maintaining the overall application error tolerance [21, 106, 151, 162, 149]. These satisfy the first 3 challenges identified in Section 3.4.2. Some approximation

tuning proposals use greedy algorithms in random order over the approximate elements, maximizing approximation on one element before moving to the next [106, 151]. Alternative proposals order the approximate elements by energy per operation [149, 223] and then perform greedy tuning. While this is a reasonable heuristic for optimizing for power savings, it is not very useful for performance or energy consumption, since application performance is too dependent on dynamic application dataflow characteristics to be interpreted as a static per-operation metric. Finally, previous work has utilized gradient descent for application-specific approximation, to combine approximation of memory and compute, targeting Reduce and Rank applications [173]. Our work, on the other hand, utilizes gradient descent based tuning towards fine-grained approximation for general purpose approximation systems (details in Section 8.2).

System-wide Approximation: The importance of maximizing the execution metric of interest while managing application error has also been stressed in prior work [206], though their focus is on applications whose error and hardware execution metrics can be completely modeled mathematically, which can be challenging for complex applications with increasing approximate variables, especially when executing on diverse hardware platforms. Similarly, the need for hardware cognizance in approximation has been discussed in some system-level proposals [92, 93], but these works are not tuning approximation at an operation-level granularity, nor are they leveraging spatio-temporally diverse hardware approximation solutions. Further, application-specific approximation systems proposals [174, 175, 83] have build systems wherein each application-specific subsystem's approximation is managed intelligently depending on the effects of the application on that subsystem. In most scenarios, these approximations are coarse grained, implementing a very limited

number of approximation knobs (often just one) per sub-system. Given an application's overall error tolerance, prior works targeting general purpose approximation [21, 106, 151, 162, 149] have proposed software-centric solutions (analyzing program-flow and input data sets) to obtain some feasible approximation configuration. These proposals perform tuning which is "hardware agnostic" i.e. the tuning mechanism is oblivious to the effect of the chosen approximation configuration on hardware execution. Such solutions are sub-optimal since they do not target a specific execution metric (eg. performance or energy) and ignore features of the execution hardware. While some application-centric systems have analyzed the system as a whole to perform approximation tuning [174, 175, 83, 79, 11], these proposals are either not designed with a general purpose system in mind or they are unsuited to fine-granularity and spatio-temporal diversity in approximation.

3.5 On-Chip Network Traversal and Cycle Utilization

Multi-core processors with 10s to 100s of cores are commonplace in today's servers, supercomputers and high-end workstations. Intel's line of Xeon Phi processors have 50-100 cores per chip targeting high performance computing and Machine Learning. Moreover, we are ushering in the exascale era in which there is an exponential increase in data processing requirements from cloud computing, ML and scientific applications - forcing futuristic CMPs to target 100s to 1000s of cores and other compute nodes on each chip [20, 6, 130, 74, 109]. With long communication being a critical hurdle to maximizing large-scale computing performance, optimizing NOC traversal latency is of paramount importance. In TNT (Chapter 9), we reduce traversal latency as close to the only-wire delay as possible, along

with maintaining design modularity as well as accounting for physical NOC heterogeneity.

3.5.1 State-of-the-art NOCs

Microarchitecture: NOC routers primarily perform the following actions [45]: Buffer Write (BW), Route Compute (RC), Switch Allocation (SA) and VC Selection (VS). Innovations within routers have allowed it to move from serial execution to parallel execution, via lookahead routing [45], simplified VC selection [129], speculative switch arbitration [145, 153], non-speculative switch arbitration via lookaheads [124, 128] to bypass buffering and so on. Router delay has dropped to 1-cycle in academic prototypes [125, 163] - we use this 1-cycle router as our baseline. Winners of SA proceed to Switch Traversal (ST), and Link Traversal (LT). ST and LT can be performed within a cycle [96, 163] allowing $t_w = 1$. We use this as TNT's evaluation baseline (Section 9.4), incurring 2-cycles-per-hop.

Modular design: It is evident from Eq.2.1 that packet latency decreases linearly with reductions in hop count (H). Reducing hop count via high-radix routers [45] and point-to-point designs is challenging because adding more direct links to distant routers results in an increase in i) serialization delay due to use of thinner channels (i.e. smaller b) and ii) router delay t_r and power, due to higher number of router ports. Such designs can increase complexity and complicate layout, etc., since multiple point-to-point global wires need to span across the chip, resulting in loss of modularity. Also, loss of design modularity often goes hand in hand with increasing overheads of the design, again impacting scalability. Therefore, industry-standard on-chip networks are often designed with regular topologies (eg. mesh), short interconnects of fixed length and self-contained tiles which can be

optimized and built modularly using regular repetitive structures, easing the burden of verification, especially as number of on-chip cores increase [8, 199, 44, 15]. Despite TNT's goal of end-to-end traversal in a single pass, it is still built in a modular fashion. This is discussed in Section 9.2.

3.5.2 Traversing the NOC

Heterogeneity in physical dimensions: The logical topology of a multi-node system and its NOC can often be different from the physical design. The floor plan of the entire chip needs to carefully consider the contents of tiles (cores, caches, memory controllers, etc.), the distribution of the tiles, and the placement of links relative to these tiles. Wire congestion, delay and energy overheads, as well as the use of global / semi-global metal layers, all factor into floor planning considerations. Such factors can impose significant physical heterogeneity on an otherwise logically homogeneous system. In TNT, we are specifically concerned with the impact of this heterogeneity on link lengths in the NOC, which in turn impacts the traversal latency. The specific factors we discuss here are: the types of nodes in the system, their aspect ratios and their distribution/connectivity.

The nodes in the CMP are usually made up of cores plus cache slices, or memory controllers (MC). As the number of processor cores grow, it is not practical to assume each tile will have a MC directly attached - they are limited by packaging constraints, primarily the number of available pins. For instance, Intel Skylake servers [203] can have 10-28 cores with 2-4 MCs, connected by a mesh topology, with MCs at the top and bottom of the fabric. Prior work [1] has shown that a diamond placement of MCs performs best using

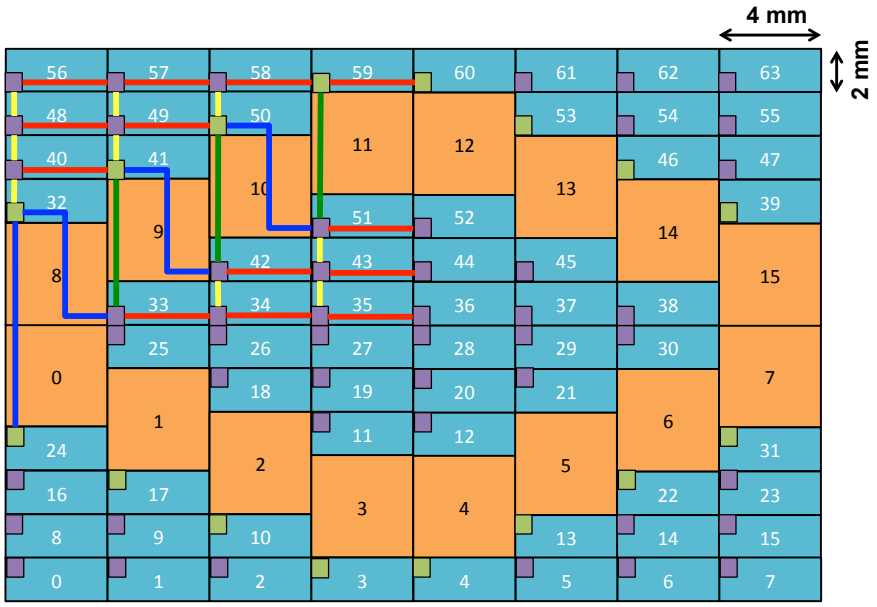


Figure 3.8: A "Typical" Floorplan - 64 cores, 16 MCs.

dimension-ordered routing, because it is able to spread traffic across all rows and columns. Further, the NOC nodes potentially have non 1:1 aspect ratios. Die images of the Intel Icelake server shows that its Sunny Cove cores are twice as wide as they are long, with dimensions of roughly 4mm by 2mm [208]. Also, the MC nodes can be multiple times the size of the core nodes - die images of the AMD Zeppelin system shows MCs twice the area of its cores, with 1 MC for every 4 cores [237].

Based on such publicly available die information [237, 203, 208], a typical floorplan might look as illustrated in Fig.3.8. The figure shows a 64 core, 16 MC system connected with a mesh. The blue tiles depict 4mm * 2mm cores, while orange tiles depict 4mm * 4mm MCs. The top left quadrant also depicts the physical mesh links of the resulting system. Link lengths are as follows - yellow: 2mm, red: 4mm, green: 6mm, blue: 8mm. Thus, it is evident that even logically homogeneous system are impacted by heterogeneity imposed by physical floorplan constraints, resulting in link lengths (and traversal times)

being different across different routes on the chip.

TNTs' time stamp based delay tracking mechanism which enables support for these forms of heterogeneity is discussed in Section 9.2.1. Evaluation over a wide range of heterogeneity scenarios is presented in Fig.9.15 and its corresponding discussion. The floorplan / methodology details for TNT are discussed in Section 4.5.

Wire transmission distance: The time delay for bits to traverse across a wire is managed via repeaters. Repeaters are sized appropriately to allow bits to traverse wires of a particular length in a required amount of time. The limiting constraint to longer wires and higher frequencies is timing closure failure and unacceptable energy consumption. In Fig.3.9, we use DSENT [207] to plot the achievable transmission distance over wires per clock cycle, against the consumed energy. We analyze the data for different technology nodes and for different frequencies. At a frequency of 1 GHz, link drivers are able to theoretically drive up to as much as 20mm across 45/32/22nm technology nodes. At higher frequencies, the traversal capability drops. The reason for the slowdown is the slew rate of the signal [126]. At higher frequencies (i.e., shorter clock periods), the repeater is unable to produce the extra current required to reduce the signal's rise/fall time proportionally [126]. At lower technology nodes, while transmission energy shrinks, there isn't a significant impact on the wire delay due to compensatory effects from scaling resistances and capacitances [126].

TNT is able to benefit from a wide range of per-cycle wire transmission distances. TNT's evaluation considers both high and low transmission distances in Section 9.4. The floorplan / methodology details for TNT are discussed in Section 4.5.

Impact of variation: Process variations are caused due to wafer characteristics, doping fluctuations [22], imperfections in manufacturing process, etc., leading to potentially large

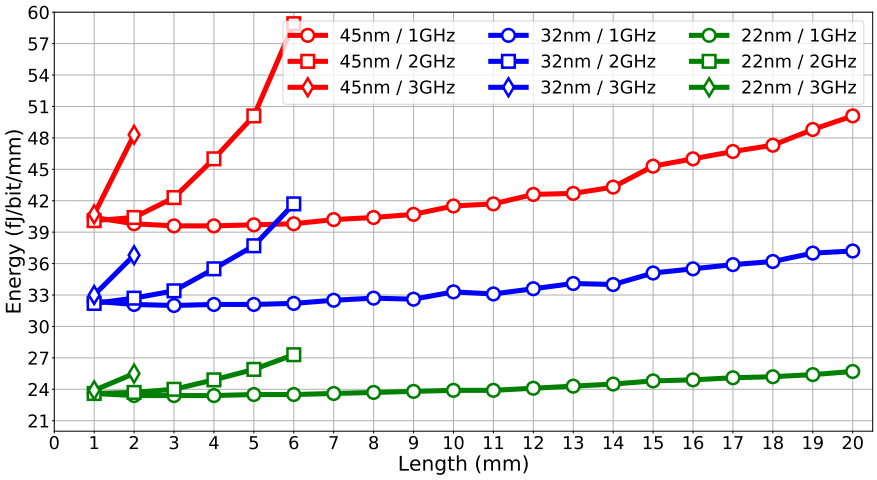


Figure 3.9: Transmission Distance per Clock Cycle

variations in device attributes [77]. Since on-chip networks connect distant parts of the chip, they are especially vulnerable to within-die parameter variations. The systematic portion of process variation can be characterized based on 3 parameters - μ (the mean wire delay), σ (the standard deviation) and ϕ (the spatial correlation) [195]. NOCs are usually designed to work under the most unfavorable parameter values in the chip [7] - thus the latency of traversing a single hop is estimated with worst case delay characteristics. This potentially results in a considerable fraction of the switch/link traversal clock cycle being wasted as slack. Prior works have noticed more than 30% difference in the minimum voltage required for error-free functioning of routers across a 64-node NOC [7]. It is intuitive that capturing the delay characteristics across each of the different tiles in a chip at design time, and tuning the local η accordingly can improve the traversal capability along variation-robust routes.

TNT's benefits over different variation characteristics are shown in Fig. 9.9 and its corresponding discussion.

Putting it all together: Thus, it is evident that with a typical wire traversal capability of 8mm per cycle (suited to the 1-1.5 GHz frequency range), the links in the typical NOC

floorplan depicted in Fig.3.8 would consume 0.25-1 cycles per hop ($\eta = 0.25/0.5/0.75/1$). And further, exploiting process variation characteristics can improve upon the worst-case traversal capability in different parts of the chip. Given the considerable NOC slack to exploit, is it possible to simply run the NOC at a higher frequency? The answer is no and the reasons are multi-fold. First, and most importantly, the router logic is often the real limiter to network frequency [123], not the wire delay. Second, increasing frequency results in a super-linear drop in the wire capability. Third, there would be a quadratic increase in power and energy penalties. Finally, on-chip heterogeneity would still require a worst-case clock period or pipelined links with longer credit loops, requiring larger buffers (VCs), which can already dominate router area and power. Clearly, more innovative solutions are required to fully utilize link traversal capability.

3.5.3 Related NOC design

Asynchronous NOCs: In the past, purely asynchronous NOCs have been proposed [12, 19], targeting deterministic traffic. Such a network is programmed statically to preset contention-free routes for QoS. On the other hand, TNT 'reconfigures' paths on every cycle handling general-purpose CMPs with non-deterministic traffic and variable contention. Asynchronous Bypass Channels [107] target chips with multiple clock domains across a die. They need to buffer/latch flits at every hop speculatively, discarding them thereafter if flits are successfully bypassed. Also, the switching between bypass and buffer modes cannot be done cycle-by-cycle, which increases latency. In contrast, TNT currently targets a single clock domain across the entire die, allowing routers to switch between bypass/buffer

each cycle.

Lower hop count topologies: TNT is not restricted to a mesh topology. Other topologies might reduce hop count in comparison to the mesh, but the hop count still grows as the number of nodes on the chip scale up. For example, the average number of hops on a torus is 0.75x that of the mesh [108] - lower, but still growing with chip size. Moreover, the torus can have unequal link lengths (due to the wrap around links) or have links double the size of the mesh if all links are equally sized [108] - both of which can significantly increase TNT's benefits. Similar to FB [119], other high-radix router designs [13, 45, 113, 73] are topology solutions to reduce average hop counts, and advocate adding physical express links between distant routers. Each router now has an increased number of ports and the channel bandwidth is often reduced proportionally to maintain the same area/power as a mesh router. Further, many of these solutions need multiple global links spanning across the chip, reducing the modularity of the network. As these solutions strive to achieve substantial hop count decrease (especially in large systems) area and power grow dramatically. We compare TNT against two such topologies in Section 9.4.2.

Reducing router complexity: TNT utilizes a baseline 2-cycle per hop with a 1-cycle router, which is state-of-the-art with VC-based router designs. VC-based designs have been commonly adopted in industry and academic prototypes, including relatively larger networks [164, 46, 97]. Other proposals have adopted VC-free designs when scaling to larger sizes, which reduces router complexity and can allow for 1-cycle per hop [188, 212, 228, 146]. We compared TNT against an optimistic VC-based 1-cycle per hop design in Fig.9.8d, highlighting TNT's benefits. Note that TNT's transparent flow based approach can be adopted atop 1-cycle per hop designs as well, but is beyond current scope.

3.6 Timing Verification / Closure

State-of-the-art: In the CAD / EDA domain, multiple advancements have allowed for efficient timing closure, including the enabling of aggressive timing guardbands [219, 127, 112, 37, 225], and support for multi-cycle paths [190, 224, 238, 43].

REDSOC: The introduction of transparent FFs (i.e. selective FF bypassing) adds some complexity to timing analysis/closure of the execution unit and data bypass network. For the simplified 2-EU system shown in Fig.7.2, traditional timing paths (in a standard FF design) to analyze for timing closure would be $(F_{1i} - F_{1o})$, $(F_{2i} - F_{2o})$, $(F_{1o} - F_{2o})$ and $(F_{2o} - F_{1o})$. These would require to be single cycle timing paths. The introduction of FF bypassing introduces 2-cycle timing paths which also need to be verified. In the same example, these would be $(F_{1i} - F_{2o})$ and $(F_{2o} - F_{2o})$ when M_{12} is enabled for transparent dataflow. Similarly, there would be $(F_{2i} - F_{1o})$ and $(F_{1o} - F_{1o})$ when M_{21} is enabled for transparent dataflow. These paths are marked as 2-cycle paths during timing analysis, the rest of the design's timing remains traditional. Note, M_{12} and M_{21} are never transparent at the same time - this precludes combinational loops.

TNT: TNT uses wire delay information and timing based tracking. While this causes some increase in design and verification complexity, we have discussed in Section 9.3 how these delay measurements can be performed realistically: accounting for dynamic fluctuations if required and tolerating measurement inaccuracies with safeguard mechanisms. It is important to point out that our design verification requirements do not violate our goal of modularity. Specifically, the design *does not require timing closure over any multi-cycle*

multi-hop routes. Timing measurements are only required on a per-link basis - TNT accumulates these over routes and makes dynamic routing decisions. In a way, our timing accumulation and safeguard mechanisms perform top-level STA (static timing analysis) on the fly and avoid actions that are potentially timing concerns.

In the specific context of TNT, IR-ATA [219] significantly reduces timing guardband pessimism by generating path-based timing margins across an ASIC. Each timing path is margined specific to its unique topology and noise characteristics, removing the requirement for global worst-case margins. This method of timing analysis is well suited to TNT's use of heterogeneous per-link timing delays. Further, CAD tools already require separate place-and-route passes for chip wide interconnects like clock and reset - it is plausible that similar passes can be implemented for the TNT NOC paths if required.

3.7 Chapter Summary

This chapter provided background and motivation for the research proposals discussed in this dissertation. CHARSTAR was motivated by the considerable power consumed by the clock distribution system and its unique distribution across the clock hierarchy. SlackTrap was motivated by the presence of considerable timing guardbands due to conservative design of traditional architectures. REDSOC's per-instruction slack recycling was motivated by the fact that the amount of slack can vary drastically between instructions, but can be identified form instruction characteristics, often early in the processing pipeline. SHASTA was motivated by the need for fine-grained approximation in hardware, as well as the need for tuning mechanisms which can take into account both the requirements of the

application and the capabilities of the hardware. Finally, TNT was motivated by analysis of wire traversal capability in the NOC, especially in the face of physical heterogeneity. Apart from the above, this chapter also provided background on different types of architectures and optimizations atop which our proposals are built or against which our proposals are compared, as well as other related prior work.

4 IMPLEMENTATION AND EVALUATION METHODOLOGY

In this chapter we discuss the methodology towards the implementation and evaluation of the various proposals in this dissertation. Across all proposals, this includes details on workloads and evaluation / simulation infrastructure. For slack recycling proposals, we also provide details on slack modeling, specific features of the design and discussion on timing closure. For SHASTA, we provide details on error modeling as well.

4.1 CHARSTAR

Resource	**Configuration**
CRIB	16 x 4-entry Int. CRIB; 16 x 2-entry FP CRIB
Compute	Int. ALU (1 cycle); FP add (4); FP mult. (4)
Core Mem.	32 LQ/SQ; 2-way 64KB L1I (2); 4-way 32KB L1D (2)
Uncore Mem.	L2: 2MB, 8-way (12); Off-chip mem (168)

Table 4.1: CRIB Specification

We extend the Gem5 Simulator [17] to support *CHARSTAR* atop the CRIB architecture. Performance and per-quantum statistics are obtained from Gem5 by running multiple Simpoint [168] slices (of size 100 million instructions) of the SPEC CPU2006 benchmark suite compiled for ARM ISA. The CRIB section of the processor is implemented in RTL and analyzed with the Synopsys Design Compiler. Power numbers for other processor resources are obtained from McPAT [135, 233]. All power and area results are obtained at a technology node of 22nm. Architecture specifications of CRIB with 16 partitions are presented in Table 4.1. Our results provide numerical benefits for the processor portion including tightly coupled L1 and L2 caches but excludes the rest of the memory subsystem.

4.2 SlackTrap

Slack modeling: Mean slack estimates for PVT variation and standard deviation for systematic components are obtained from McPAT-PVT [210]. Standard deviation for random variations are obtained for nominal and high estimates from prior work [138, 116]. Numerical values are shown in Table.4.2. Data slack estimates are obtained via enhancing SoftInj,

Variation	**Parameters**	**Values**
PVT sys. (slack %)	Mean, 99.99th	30.5, 19
PVT random	σ/μ (nominal, high)	1.5, 8.5

Table 4.2: Variation Parameters

a software fault injection library that implements the b-HiVE error models [217]. Fig.4.1 showing statistical slack estimates over an ALU, for datasets corresponding to random inputs, gcc benchmark and the overall SPEC and MiBench benchmark suites. Slack estimates are shown for single operations and asynchronous operation sequences of length 2 and 4. Operations lie closer to mean value for longer sequences, as motivated in Fig.6.1

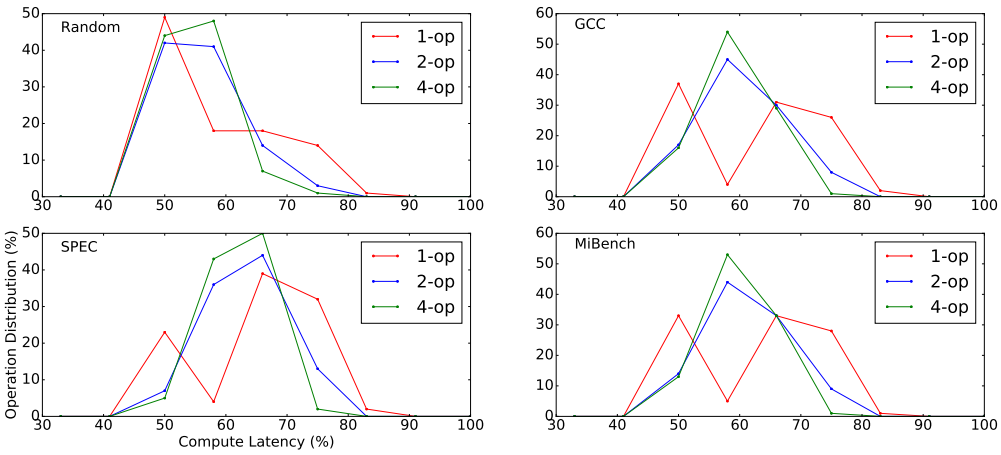


Figure 4.1: Data Slack Analysis

Simulation: We extend the Gem5 Simulator to support timing speculation atop CRIB. CRIB configuration is same as shown in Table 4.1.

Workloads: We choose SPEC CPU2006 for performance evaluation due to its dynamically varying DFG characteristics, irregular memory access patterns, and hard to predict control flow. Results are obtained from Gem5 by running multiple Simpoint slices (each 100M instructions) of the workloads, compiled for ARM ISA.

4.3 REDSOC

Simulation: We extended the Gem5 [17] simulator to support Slack Recycling atop standard out-of-order cores. We model 3 cores labeled *Big*, *Medium* and *Small*. The description of the cores can be found in Table.4.3.

Parameter	**Small**	**Medium**	**Big**
Frequency	2 GHz		
Front-End Width	3	4	8
ROB/LSQ/RSE	40/16/32	80/32/64	160/64/128
ALU/SIMD/FP	3/2/2	4/3/3	6/4/4
L1/L2 Cache	64kB/2MB w/ prefetch		

Table 4.3: Processor Baselines

Benchmarks: The benchmarks for analyzing results are 2-fold. The first set encompass relatively compute intensive applications from the SPEC CPU 2006 [85] and the MiBench benchmark [78] suites. They are run via multiple Simpoints [168], each of length 100 million instructions. The second set consists of kernels from ARM Compute Library [101] for computer vision and machine learning with support for ARM NEON SIMD (brief

details in Table.4.4). Benchmarks are all compiled for the ARM ISA; NEON vectorization flags are turned on for the ML kernels.

Kernel	Description
CONV	Convolution: Gaussian 3x3
ACT	Activation: ReLU
POOL 0/1	Pooling: 2x2 Max/Average
SOFTMAX	Softmax function

Table 4.4: Kernels for Machine Learning

Influence of PVT variation: Pure data slack estimates correspond to worst-case design corner to isolate it from PVT (process, voltage, temperature) variations. Thus, the data slack estimates expect to stand true across all PVT conditions. Executing under nominal PVT conditions provides some exploitable guard band [134, 77] and adds a small additional component to the total slack.

In real design, guard band variations from PVT can be measured with Critical Path Monitors (CPMs) [134]. To exploit PVT guard band, our design only requires localized CPMs in the proximity of the ALUs and bypass network. Conventional CPM based guard band estimates (eg. Power7 [134]) are more conservative since they are located in the most timing/power critical regions of the entire chip.

To account for PVT variation, slack LUTs are re-calibrated on-the-fly, thus supporting changes to voltage, temperature, aging etc. We adhere to a tuning granularity of 10,000 cycles as is prescribed in Tribeca [77]. There are no design-time/testing overheads for PVT based slack calibration, which is simply tracked dynamically with CPMs.

Data slack measurements: In this work, we only target single cycle data slack for the

integer ALU as well as in specific integer SIMD operations. We do not target data slack from other multi-cycle operations such as FP ops. Slack is modeled via RTL design in Verilog and synthesis using the Synopsys Design compiler. We synthesize the execution pipeline stage with a 0.5 ns cycle time (i.e. 2 GHz) constraint. As explained in Sec.7.1, REDSOC only requires to categorize operations into 14 different slack buckets - we do not need accurate slack estimations of each operation. This completely simplifies CAD timing analysis.

Our slack analysis agrees with estimations from prior work [217] as well characterization via gate-level C-models. Considering the low effort, we expect state-of-the-art CAD tools to be capable of (or extendable to) such analysis. During processor execution, appropriate slack bucket is selected simply based on opcodes and operands: no dynamic timing analysis is involved.

Slack Tracking Precision in the RSE: We quantized slack / timing corresponding to different precisions (up to 8-bits) in our architecture simulator and analyzed performance. Performance saturated at 3-bits (or 1/8th of a clock cycle). Hence, 3-bit values are sufficient for slack recycling.

4.4 SHASTA

Applications: We evaluate SHASTA across 8 applications suitable to approximation from different domains - highlighting that SHASTA is a general purpose solution to approximation. The applications include those from the AxBench [235], Polybench and Parsec [16] benchmark suites and others. Operations to approximate are identified by hand, based

on guidelines established in prior work [148]. Applications are tuned to target 3 different accuracy levels of 99%, 95% and 90%, as is common requirement for these applications. Note that they can be tuned for other accuracy levels as well. Accuracy measurement metrics for the AxBench applications are as defined in the benchmark suite [235]. Accuracy measurement of other applications are as performed in prior work [106, 148]. Applications are compiled for the ARM ISA to run on the Gem5 [17] architecture simulator.

Table 4.5 shows the different applications along with the number of approximate operations identified in the program code, the iterations of the application as well as the resulting percentage of dynamic instructions which are approximate (Dynamic Approximation %).

Application	Domain	Annotations	Iterations	Dyn. Approx %
Blackscholes	Finance	111	1	30.63%
Mat Mul	ML	6	1	17.91%
Inversek2j	Robotics	60	1	14.25%
K-means	Mining	39	32	34.1%
FFT	Sig Proc	60	1	11.25%
Canneal	CAD	75	32	7.2%
PageRank	Graph	25	32	30%
MLP	ML	30	20	27%

Table 4.5: Approximate Applications and their Characteristics

Slack modeling: Slack is modeled via RTL design in Verilog and synthesis using the Synopsys Design compiler. We synthesize the execution pipeline stage with a 0.5 ns cycle time (i.e. 2 GHz) constraint. Our timing analysis agrees with estimations from prior work [217] as well characterization via gate-level C-models. Considering the low effort, we expect state-of-the-art CAD tools to be capable of (or extendable to) such analysis. During processor execution, appropriate slack bucket is selected simply based on opcodes and operands: no dynamic timing analysis is involved.

Error modeling: Compute operations identified to be suitable for approximation are approximated via our approximation model build on top of the SoftInj error injection framework [217]. Memory operations suited to approximation are approximated by implementing the approximator described in Sec. 8.1.2 in software. Overall application error for a given approximation configuration is evaluated by natively running the application to completion and comparing with a known precise output.

Parameter	**Values**
Frequency	2 GHZ
Front-end width	4
ROB/LSQ/RSE	80/32/64
ALU/SIMD/FP	4/2/2
L1/L2 Cache	64kB/2MB w/ prefetch
Approximator / LHB	64 / 4 entries

Table 4.6: Processor Configuration

Simulation: Application IPC for given approximation configuration is obtained via Gem5 [17] architectural simulation. We implement load approximation and slack recycling for out of order cores in Gem5. Power numbers are estimated in accordance to McPAT [135]. The processor configuration is described in Table 4.6.

Tuning: Application tuning is performed via wrapper functions written in C which run a numerical gradient descent tuning mechanism. The algorithm iterates until convergence. We also implement a hardware agnostic greedy algorithm (for comparison) as described in multiple prior works such as [106]. The algorithm finds a suitable approximation for the first variable while keeping others exact, then it fixes the approximation of first variable and moves on to the second variable while keeping the others exact, and so on - independent of hardware efficiency.

4.5 TNT

Design: Evaluations are carried out at 1GHz (22nm node) and assume an 8mm per cycle wire traversal capability - a conservative assumption at 1GHz, and generally suitable at less than 2GHz. This NOC frequency range is in line with designs built by industry and academia [204, 46, 164, 14, 97]. Benefits at higher frequencies would vary with wire traversal capability, as discussed in Section 3.5.2.

Simulation Performance results for both Synthetic Traffic and Full System are obtained using GEM5 [17] + Garnet2.0 [2] infrastructure, which provides a cycle-accurate timing model. The evaluated system configuration is shown in Table 4.7. Energy and area numbers for router and link components are estimated with the help of DSENT [207].

Processors	64 3-wide OOO	**Freq/Tech**	1 GHz / 22 nm
L1 Cache	Pv. 32kB, 2 cyc	**Link/Flit**	1-8 mm / 128bit
L2 Cache	Sh. 1MB/core, 12 cyc	**Topology**	8-ary 2-Mesh
Coherence	MESI	**Routing**	XY
Vir. Net	3 (req, fwd, resp)	**Vir. Chan**	4+4+4

Table 4.7: Target System

Synthetic Traffic: We show results for Uniform Random, Bit Complement and Transpose. We primarily focus on 1-flit packets to understand benefits without secondary effects due to flit serialization, and VC allocation across multiple routers etc.

Full System Benchmarks: We run graph applications from Ligra [200], a lightweight graph processing framework. We evaluate on accompanying input graphs, specified in PBBS [201] format. Applications are compiled for RISC-V ISA and run with 64/256 threads.

Link Delay: Link delay varies with NOC heterogeneity, as discussed in Section 3.5.2.

We model the floorplan considerations based on the typical floorplan in Fig.3.8 and also sweep through a wide design space in Fig.9.15. Process variation is modeled similar to prior work [195] and the effect of different variation characteristics are discussed in Fig.9.9. In a real design, we expect these characteristics to be measured post-fabrication, interpreted as clock cycle slack, and written into the per-router lookups (discussed in Section 9.2.1). As studied/shown in prior work [36, 181], we expect state-of-the-art CAD tools to be capable of (or extendable to) such analysis. Dynamic variations to these measurements can be caused by DVFS, voltage/temperature fluctuations, ageing etc. Industry-standard methods of critical path monitoring (CPMs) and adaptive timing guardband control, which, for example, have been utilized by multiple generations of high frequency IBM Power processors [134, 133, 50] can be utilized to capture such dynamic variations and update the router look-up values accordingly. Note that while more accurate timing measurements are beneficial, the design is robust to less accurate measurements, thanks to the timing safeguard mechanism discussed in Section 9.2.3 and evaluated in Tables 9.2a, 9.2b. A less accurate design can employ a larger timing safeguard window, at the cost of losing a fraction of TNT's benefits.

Floorplan: TNT results are shown for 3 scenarios (against a mesh baseline):

① *TNT:Minimum* shows the "minimum" benefits of TNT over the baseline. This is achieved on a floorplan wherein all tiles are homogeneous and 8mm, with no exploitable variation guardband. Thus $\eta = 1$. TNT gets minimal benefits only from avoiding buffering/restarting at intermediate routers.

② *TNT:Typical* shows the "typical" floorplan illustrated in Fig.3.8 wherein link lengths range from 2mm to 8mm. Typical variation characteristics are also assumed, which leads

to a roughly a 20% variation in wire delays across the chip [7] (detailed discussion on variation in Fig.9.9). This leads to $\eta = 0.2 - 1$ range across the chip.

③ *TNT:Maximum* shows "maximum" benefits of TNT, achieved on a homogeneous floorplan wherein all links are 1mm (smaller lengths are unlikely). Further, a maximum exploitable variation guardband (up to 35% variation in delay) is assumed. $\eta = 0.085$ roughly across the chip. Note: A wider design space is briefly explored in Fig.9.15.

4.6 Chapter Summary

In summary, this chapter provided details on evaluation and implementation methodology. The CRIB architecture and its corresponding simulation infrastructure is build upon by both CHARSTAR and SlackTrap. The other proposals develop upon the standard out-of-order processor simulation infrastructure (in Gem5). Compute slack modeling is performed by writing RTL, synthesis via the Synopsys Design Compiler and measuring characteristics with the synthesis tool. In the case of TNT, wire delay modeling is performed via DSENT. Evaluation and implementation based on the methodology from this chapter, are discussed in Chapters 5 - 9.

5 CLOCK HIERARCHY AWARE RESOURCE SCALING IN TILED ARCHITECTURES

In this chapter, we explore the CHARSTAR proposal. In terms of breaking the clock abstraction, CHARSTAR stands separate from the other contributions in this dissertation in that it focuses on the clock distribution system and its power consumption (while the others target different forms of clock cycle slack).

We discuss its primary contribution - clock hierarchy aware reconfiguration (Section 5.1), as well as the secondary contributions - integration with DVFS (Section 5.2.1), an ML assisted prediction mechanism (Section 5.2.4) and optimally balanced temporal / spatial granularities of reconfiguration (Section 5.2.2). In Section 5.3, we look at its design atop tiled architectures (focusing specifically on the CRIB architecture). We present an evaluation of its performance and efficiency benefits (Section 5.4) as well as an analysis of its overheads (Section 5.3.2). Finally, a discussion on generality, constraints and extensions of CHARSTAR is provided in Section 5.5. Section 5.6 summarizes the proposal.

An introduction to CHARSTAR was provided in Section 2.1, background / motivation in Chapter 3 and methodology in Section 4.1.

CHARSTAR's contributions are summarized below:

① We propose *CHARSTAR*, a highly energy efficient, dynamically resizable and frequency scalable tiled architecture, implemented atop the CRIB processor.

② Through a comprehensive limit study, we explore the benefits of *intra-core* integrated frequency scaling and architectural reconfiguration to achieve the ideal configuration which balances ILP and clock rate. We are not aware of prior work specifically in dynamic joint

optimization of DVFS and power gating *within a core*.

③ We improve the reconfiguration mechanism by making it aware of the non-linear hierarchical clock-tree power dissipation. We are not aware of prior work specifically on clock-tree-aware power gating of microarchitectural resources.

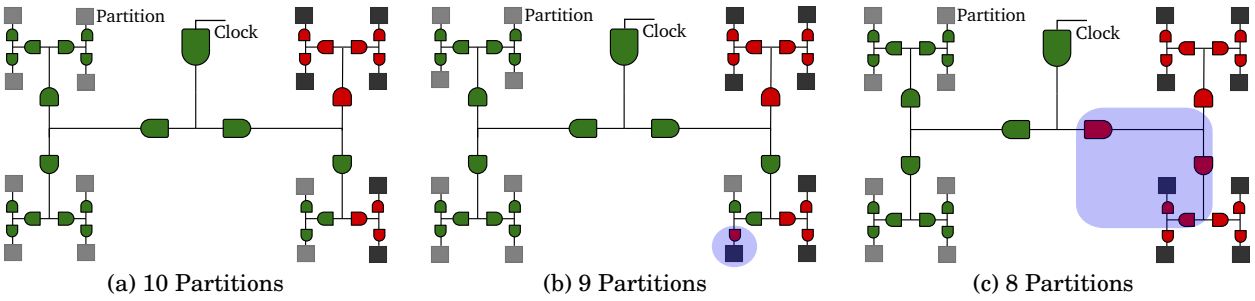
④ We advocate tiled architectures as the ideal reconfiguration fabric and study the benefits of striking a balance across the temporal and spatial dimensions of adaptivity.

⑤ We make use of a lightweight two-dimensional multi-layer perceptron predictor to accurately predict the resource and frequency levels for each phase (quantum) of instructions. Most existing machine learning based control mechanisms focus on highly complex design spaces, to predict at coarse temporal granularities. We believe ours to be the first in targeting fine temporal granularities instead - with a simple prediction space, low design complexity and high accuracy.

⑥ Experimental results from the SPEC CPU2006 [85] benchmark suite shows that our mechanism, when tuned to saving energy, improves energy efficiency by 20-25% with performance loss limited to 1-5% on average. Alternatively targeting speedup achieves performance improvements of more than 10% under a constrained power budget and close to 20% when unconstrained but optimized for energy-delay.

5.1 Clock Tree Aware Power Gating

A background on reconfiguration and gating within the core is provided in Section 3.1.4. When portions of resources (ALUs, compute nodes, or shader cores) are not in use, power gating saves leakage power, but gating the portion of the clock hierarchy supplying these



(a) 10 Partitions (b) 9 Partitions (c) 8 Partitions

Figure 5.1: Example motivating Clock-Tree aware reconfiguration

resources (clock gating) also has huge energy saving potential. Many reconfiguration mechanisms either focus solely on power gating execution resources alone or in addition, simply turn off the clock tree segment (local mesh and global tree's leaf node) which supplied the clock to those resources.

Clock tree aware gating techniques, will be capable of further shutting down upper levels of the global clock tree hierarchy if the corresponding sub-tree is not in use. In addition, *Clock tree aware reconfiguration mechanisms* can make intelligent resource reconfiguration decisions to maximize gating of resources - providing best energy efficiency while guaranteeing required performance. CHARSTAR's reconfiguration mechanism is aware of the clock tree hierarchy and the power consumed by each node in the tree, and selects resources to power down in a greedy manner that shuts off entire branches of the clock tree whenever possible. Our proposal is aware of the portion of the clock tree that can be turned off with every resource configuration. It accounts for the power consumption of both resources *and* the clock tree in the reconfiguration algorithm. We present a simple example highlighting how significantly clock-tree awareness can impact reconfiguration.

Fig. 5.1 illustrates 3 scenarios with different number of tiles or partitions (10/9/8) enabled. Light and dark gray squares indicate awake and gated partitions respectively.

Green and red gates indicate enabled and disabled channels in the clock tree respectively. If the performance gap between the 3 configurations is within reasonable limits, the reconfiguration mechanism would have to choose between the 3 configurations to maximize energy efficiency.

A naive reconfiguration mechanism would assume that transitioning from 10 -> 9 -> 8 partitions provides linear savings in power (proportional to the partitions being power gated). Thus even a marginal performance improvement with 9 partitions as compared to 8, might result in the 9-partition configuration being chosen as the ideal setting providing best energy efficiency. In contrast, the figure shows that moving from 10 -> 9 partitions, saves just one extra node on the clock tree (apart from the extra partition) but moving from 9 -> 8 partitions enables the shut down of the entire half of the clock tree leading to a larger (15x) clock tree savings. The additional partitions and portions of the clock hierarchy shut down are shown by the blue shaded regions.

This can also means that even if the drop in performance is *not insignificant* from shutting down each partition, choosing the 8-partition configuration can be largely more energy efficient in comparison to the 9 or 10 partition configurations, due to significant power reduction from a large portion of the clock tree being turned off. This illustrates that full knowledge of the clock hierarchy can produce very different results and larger energy savings than prior, naive mechanisms. Higher power savings from the clock hierarchy could instead even be used to run at higher frequencies, possibly providing higher performance than the 9/10 partition scenarios. This is discussed further in Section 5.2.1.

It should be noted that clock-tree aware reconfiguration is not restricted to tiled architectures alone and can be performed on other compute fabrics as well. The potential benefits

as well as complexity involved would depend on the reconfiguration style and how the clock tree feeds the different reconfigurable nodes. Architectures with flat clock trees or non-uniform clock distribution to various reconfigurable resources may be structurally limited in how large a portion of the clock hierarchy can actually be power gated. For instance, an out of order processor power gating some portion of its Register File, ALUs and/or LSQ, will potentially need a complex clock-tree gating mechanism and might still provide only limited gains. On the other hand, a regularly arranged spatial architecture like a GPU might shut down some number of shader cores based on the amount of parallelism available, with comparatively lesser complexity. Finally, clock hierarchies could be designed from the ground up to better suit reconfiguration. We leave exploration of this option to future work.

5.2 Dynamic Reconfiguration Control

Having motivated the importance of a reconfiguration mechanism aware of the clock distribution system, we next discuss the implementation of the control mechanism itself. We raise key questions that need to be answered to design an efficient control mechanism for a generic reconfiguration fabric, and then discuss them in the following sub-sections.

① What global set of resources are made reconfigurable? (Sec 5.2.1) ② How often do we want to reconfigure resources? (Sec 5.2.2) ③ How many different levels of reconfiguration for each resource? (Sec 5.2.2) ④ How to group resources to control reconfiguration efficiently? (Sec 5.2.3) ⑤ How to assess requirements dynamically and perform reconfiguration accurately? (Sec 5.2.4)

Note that while the following discussion is largely focused on clock-aware reconfiguration of tiled microprocessors, the overall approach is applicable to power-gating on a variety of substrates.

5.2.1 Integrating resource/frequency scaling

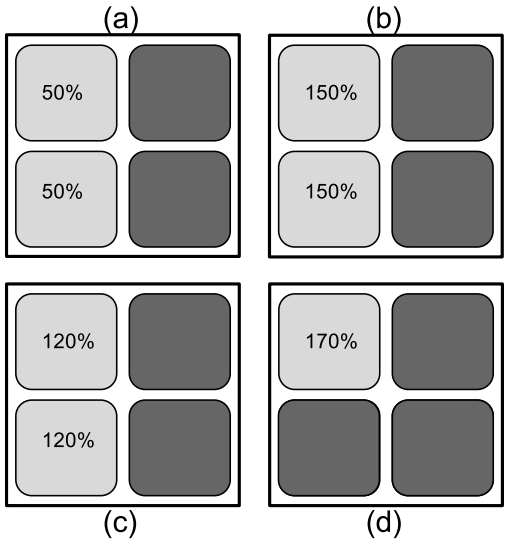


Figure 5.2: PG-DVFS Example

A background on integrated/coupled reconfiguration is provided in Section 3.1.4.

CHARSTAR identifies program phases with limited ILP, and power gates resources that are not necessary for reaching that ILP level. This leads to more energy-efficient execution under a fixed performance constraint. Alternatively, given a fixed power budget, the savings from power gating can be applied to increase voltage and frequency to deliver better performance for the baseline power. Our mechanism integrates the scaling of both resource and frequency *within the core*. As we show, joint optimization for both ILP and clock rate often leads the controller to choose configurations that provide greater benefit than choosing either in isolation. An integrated PG-DVFS mechanism has significant

potential within a single core because energy/performance gains from frequency variations are closely dependent on which resources are actually in use.

The following example (Fig. 5.2) motivates DVFS-PG integration. The example builds on a baseline processor with four tiles running at nominal (100%) frequency. If this processor encounters an application phase wherein ILP is limited, different reconfiguration mechanisms will affect the processor in different ways. These scenarios are depicted in Fig.5.2, (a) to (d). Darker and lighter gray shades of tiles refer to power gated and awake tiles respectively. Also, the percentage values within awake cores refers to their running frequency in proportion to the baseline.

• If DVFS and resource sizing are completely decoupled, both mechanisms could kick in independently and shut down multiple tiles and as well as reduce frequency, unaware of the other. This results in scenario A wherein two tiles are shut down *and* the frequency drops to only 0.5x resulting in performance dropping below what can be ideally achieved.

• If the controller performs resource sizing first, followed by naive throttling of frequency (in accordance to the baseline power budget), as shown in scenario B, the increased frequency might be largely beyond the requirement for maximum performance. For instance, shutting down 2 tiles may theoretically allow a 50% increase in frequency at constant power budget, but the 1.5x frequency might be so high that the memory becomes the bottleneck. This results in higher than ideal power dissipation.

• An integrated mechanism, on the other hand, acknowledges both the available ILP and the frequency requirements in conjunction, resulting in scenario C which has enough tiles (two in this example) for maximum ILP and the minimum sufficient frequency (1.2x) to achieve maximum throughput providing better efficiency.

• An integrated mechanism can further increase efficiency if situations exist (scenario D) wherein shutting down more tiles than the ILP bound (only one tile awake though max ILP is achieved with two tiles) and increasing the frequency further (1.7x, in comparison to scenario D with 1.2x), might actually be more advantageous. This is akin to the example illustrated in Fig.5.1 wherein it is possible that enabling 8 partitions allows large potential power savings in the clock tree which in turn allows increasing the frequency sufficiently to provide speedup above the 9/10 enabled partition scenarios (as described earlier in Section 5.1).

Prior work in trading off power gating and clock rate (TLP vs. DVFS) has focused on boosting the frequency of one or more cores while power-gating others (e.g. Intel Turbo Boost [31]). In contrast, CHARSTAR targets joint optimization of power gating and DVFS *within a single core*, which has not been explored in prior work. While the parallelism vs. frequency argument is the same in both cases, the architectural trade-offs, when analyzed, are very different. Intra-core PG-DVFS involves higher orders of dependencies between the usage of intra-CPU resources and the core frequency (in comparison to TLP vs. clock rate) and requires a sophisticated control mechanism, as proposed here.

5.2.2 Design for Spatio-Temporal balance

A background on the dimensions and granularities of adaptivity is provided in Section 3.1.4.

It is evident that in designing any reconfiguration control mechanism, it is necessary to be able to efficiently capture and adapt to application characteristics. The ability to

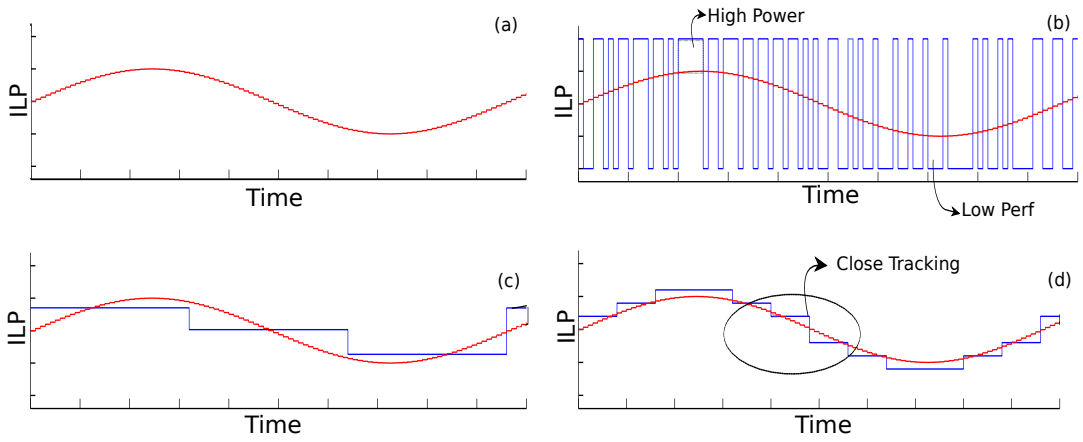


Figure 5.3: Fine grained adaptability across two dimensions.

achieve extremely fine granularities of adaptivity in both temporal and spatial dimensions is limited by complexity of control and power gating mechanisms. We propose a balanced design over the two dimensions of adaptivity with the following qualitative analysis. While it is intuitive that balanced fine granularities in both dimensions provides benefits from both dimensions of adaptivity, we also show that coarse granularity on one dimension in fact hinders the effectiveness of fine grained adaptability along the other - *the effects of the two dimensions are not independent*.

From Figure 5.3, we make the following observations:

• Figure 5.3.a uses a simple sinusoidal curve to illustrate the variation in ILP of an application over time. Ideally, achieving extremely fine grained adaptability in both dimensions would allow resizing of resources, receptive to the smallest application variations, but this is infeasible due to unreasonable overheads.

• Figure 5.3.b portrays one solution allowing very fine temporal granularities but with coarse grained resource levels (only 2 levels, akin to Big/Little). The core can change its configuration very frequently but is allowed a very limited number of configurations.

Thus, there are considerable periods when the core configurations overshoot the ideal requirement, resulting in excess power consumption with no performance gain and other periods when the core configuration undershoots the ideal, causing performance loss.

• Figure 5.3.c portrays an architecture allowing multiple possible resource configurations, but large switching times. Many of the *possible* resource levels (12 in this example) are never reached due to averaging across coarse temporal granularities and again, the large area between the curves show inefficiencies in performance and energy.

• The optimum solution as present in 5.3.d is a balance between adaptability along both dimensions - the adaptability is *reasonably* fine grained in both dimensions but not as fine grained as the previous solutions. It can be seen that the region between curves is minimal and the core is able to efficiently adapt to the needs of the application.

5.2.3 Tiled Architectures

The combined savings from clock hierarchy aware integrated PG-DVFS is significant, but at the same time is complex to implement. The inherent advantages of tiled architectures, via clustering of resources into small groups can limit these complexities and thus make them an ideal fabric for reconfiguration. Tiled architectures provide easy boundaries for powering down clusters of resources with minimal complexity and routing overheads [118]. Resizing resources by dynamically power-gating and waking up these reasonably sized clusters allows fine spatial granularities of adaptability. At the same time, the consolidated structure of these tiles reduces the reconfigurable design space significantly and their overall well defined topology minimizes circuit overhead, allowing for reasonably fine temporal

granularities. Moreover, tiled architectures with their regular shape and arrangement, present a well structured hierarchical clock tree. This allows straightforward control of disabling branches of the clock tree as well. In all, a tiled architecture provides inherent advantage for spatio-temporally balanced reconfiguration with minimal overheads and with enough simplicity and regularity to be managed by a lightweight prediction mechanism. Details on different types of tiled architectures are found in Section 3.1.1.

Through a comprehensive limit study (Section 5.3.1), we analyze the overhead and benefits of fine- vs. coarse-grained spatial and temporal reconfiguration for tiled architectures, and choose a moderately fine-grained operating point for both dimensions. Our results indicate that, *though CHARSTAR does not exploit the finest possible adaptive granularity in either dimension, it reaches a balance across both and minimizes overheads and design complexity, thereby improving ease of actual implementation and achieving benefits nearly matching the ideal opportunity.* While this result is applicable to all reconfiguration mechanisms, this is especially important in relatively complex mechanisms targeting multiple optimization opportunities such as our own clock-aware PG-DVFS.

5.2.4 ML-based Prediction Scheme

A background on prediction schemes for reconfiguration is provided in Section 3.1.4.

To adapt to application phases, CHARSTAR must predict the best tile/frequency configuration for each quantum, using as inputs the behavior of the previous quantum. Studies [114, 60] have shown that while a first order model of performance prediction based on microarchitectural events can provide rough estimates (of resource requirements), they are

often inaccurate due to overlap between a large portion of these events. Moreover, standard controllers with simple modeling may be sufficient when focused solely on reconfiguration or DVFS, whereas in the case of a more complex clock-tree aware PG-DVFS mechanism, the inaccuracies are further exacerbated.

For instance, the Composite Cores architecture [141] feeds multiple statistics from each quantum into a linear regression model which predicts the execution mode (big engine vs. little engine) for the next quantum. But the prediction accuracy is limited (Section 5.4.3), as linear models are unable to understand non-linear relations among architectural events. More prior works are discussed in Section 3.1.4. Our results show that linear regression models or those that track a unique microarchitectural resource are inadequate for clock aware PG-DVFS prediction.

Machine learning techniques are capable of adapting to non-linear relations in inputs. While we believe in the potential for different ML techniques towards resource reconfiguration (eg. decision tree learning etc.), our chosen technique is a Multi-Layer Perceptron (MLP). This is due to a large resource of recent prior work in modular design implementation of neural networks allowing ease of analyzing the trade-off of area/power overheads vs. MLP accuracy. Our results show high predictor accuracy with low effective computation and communication latency along with reasonable area/power overheads (Sections 5.3.2 and 5.4.3).

Our predictor makes use of a MLP with one hidden layer, capable of fitting any finite input-output mapping problem, enabling prediction of configurations with high accuracy. At runtime, the MLP is dynamically fed with the following statistics for every quantum of instruction: ① Branch mispredictions, ② I-Cache misses, ③ D-Cache misses, ④ L2-Cache

misses, ⑤ Average % of the tiles occupied, ⑥ Number of times the tiles were full, ⑦ IPC, ⑧ Clock hierarchy, and ⑨ Topology overheads, to predict the configuration for the next quantum. These statistics are selected based on prior work in performance regression analysis [114, 60, 141].

The MLP is trained using a typical cross-validation approach, by partitioning the data collected from a testing portion (1%) of each benchmark into two sets - a training set with 70% of the data, and a validation set with the remaining 30%. The training is performed on MATLAB with random seeds using the Levenberg-Marquardt optimization based back-propagation algorithm. While our training and prediction is focused on the SPEC CPU2006 benchmark suite, we expect similar prediction accuracy across a wider range of applications. In general, a one time offline training with small portions of the standard applications that an architecture expects to execute, is sufficient for high prediction accuracy; this conclusion is supported by our results. This is due to the iterative nature and presence of hot functions in most applications.

The different metrics used for MLP training are performance, energy efficiency and energy-delay. The *configuration* is the combined selection of the enabled (awake) tiles plus the core frequency. The selection is usually constrained by the power budget - fewer tiles kept awake would mean that frequency could be raised further if it proves beneficial. This power constraint is relaxed in some cases, discussed in Section 5.4.2. The clock hierarchy power gating is completely tied to the number of tiles awake and is not a separate reconfigurable resource. The clock tree awareness influences the energy efficiency while training - enabling different number of tiles requires considerably different portions of the clock hierarchy branches to be enabled (as discussed in Section 5.1). Moreover, the

different clock hierarchies (H-Trees, TH-Trees, B-Trees) affect the energy metric differently. This would train the MLP differently and the reconfiguration decisions vary accordingly.

5.3 CHARSTAR in a Tiled Architecture

5.3.1 Granularities of adaptivity

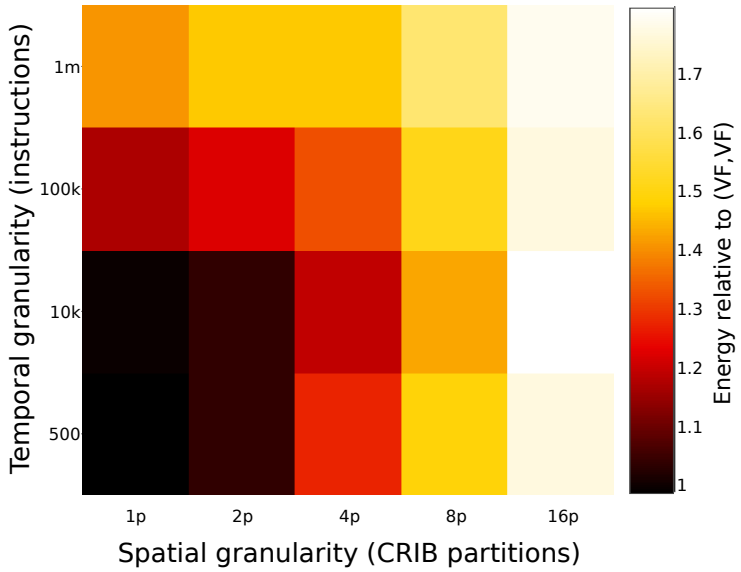


Figure 5.4: Bi-dimensional granularity impact on Energy

In this section we perform quantitative analysis to choose optimum temporal and spatial granularities on the CRIB processor, our tiled architecture of choice. Background on CRIB can be found in Section 3.1.2 and in prior work [76]. We analyze the gains from spatio-temporally balanced tile reconfiguration (without DVFS) across both adaptive dimensions in conjunction. Figure 5.4 illustrates the energy saving potential across the two dimensions of adaptivity in comparison to the ideal case of highly fine granularities in both dimensions - 500 instructions (temporal) and a single partition (spatial). This analysis is averaged

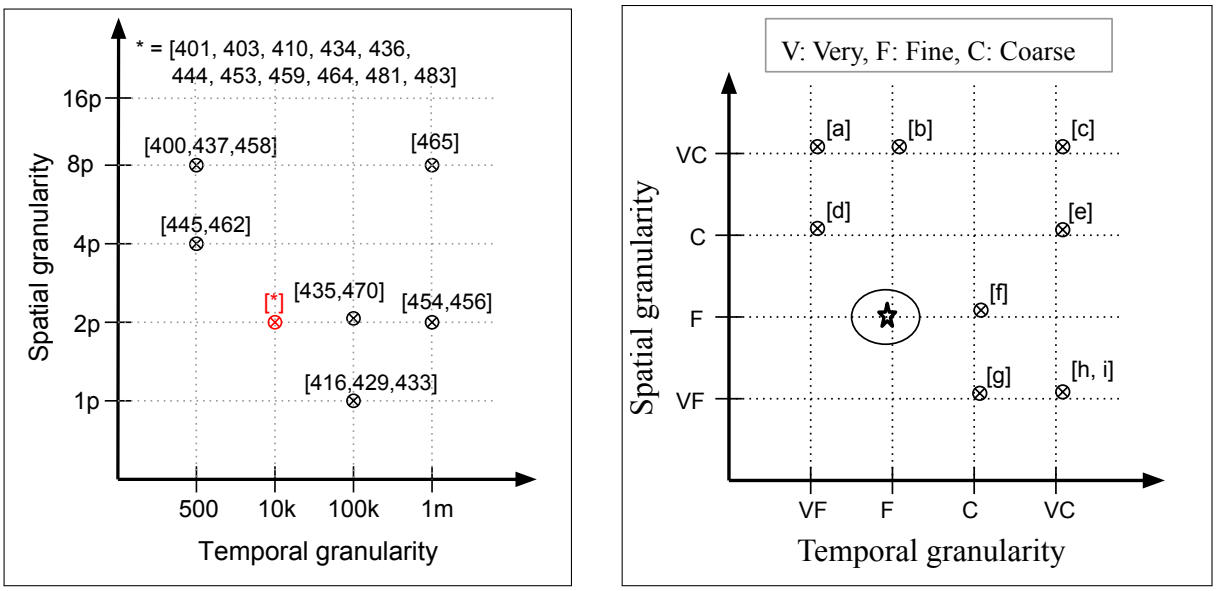
across the SPEC CPU2006 benchmark suite. Note that darker portions of the heat map indicate higher energy efficiency. The range of spatial and temporal granularities are in accordance with prior work.

Temporal: Moving from coarse (1m instructions) to fine temporal granularities (500) of resizing improves energy efficiency but the improvements slowly begin to saturate. Despite low break even points for gating resources, circuit overheads as well as architectural steps such as draining/squashing the pipeline start becoming more significant at very fine granularities.

Spatial: The ability to allocate or cut back on resources at finer levels without losing out on performance improves energy efficiency in the case of finer spatial granularities. The benefits slowly saturate due to penalties caused by dependent instructions in different partitions and increased overhead from prediction mechanisms.

As noted earlier, the ideal configurations at the bottom left - (500,1p), (500, 2p), (10k,1p) cannot be achieved due to increased overheads of implementation. Among other options, it is evident that the balanced configuration of (10k, 2p) is more energy efficient than the extreme configurations across either dimension - this is in agreement with our qualitative analysis. Considering the above analyses, we choose a temporal granularity of *ten thousand* instructions and a spatial granularity of *2 partitions*. This provides a spatio-temporal balance across both dimensions and sacrifices benefits only marginally on either dimension.

Fig. 5.5a illustrates the sweet spot across the two dimensions of adaptivity for each benchmark in the SPEC CPU2006 benchmark suite. It is evident that while a majority of the benchmarks achieve best efficiency at the balance point of (2p,10k), there are a few differing in their sweet spot based on the specific benchmark characteristics. We believe such forms



(a) Granularity sweet-spots for the SPEC CPU2006 benchmark suite

(b) Prior works: a [157], b [141], c [239], d [122], e [66], f [170], g [169], h [226], i [52]

Figure 5.5: Spatio-Temporal Balance

of classification will go a long way in designing ideal reconfigurable architectures suited to their native applications, and are a precursor to architectures which can *dynamically adapt their granularities of reconfiguration* itself.

In contrast to our observation for the need for balanced adaptivity, most of the existing proposals cited in Fig.5.5b propose reconfiguration techniques that aggressively tackle one dimension of adaptivity, but usually at the cost of the other, thereby achieving less than ideal efficiency. In figure, the axes labels denote V - Very, F - Fine, C - Coarse.

5.3.2 Quantifying Overheads

Clock tree awareness: Our implementation models the clock-tree as a binary/H-tree and tracks the per-node power consumption in a manner akin to well established clock-tree power modeling [47, 49, 198] and as discussed in earlier sections. The per-node power (as

a fraction of the entire clock hierarchy) varies in proportion to the node's *level* in the clock hierarchy as well as its *fan-out*. The influence of the clock power adds a layer of heterogeneity to a homogeneous set of tiled resources and accordingly increases the training complexity of the control mechanism. The inference/deployment complexity remains unaffected since the design space remains the same (in comparison to clock unaware reconfiguration) - tile configuration and frequency level. For instance, if 8 tiles are required to be enabled, the chosen tiles are as shown in Fig5.1.c rather than any other distribution of 8 tiles (as that would enable more clock tree branches).

Power Gating and Awakening logic: The prediction mechanism passes resizing directions to the PG/awakening logic and the specific number of tiles are put to sleep or awoken. Based on the disabled tiles, the corresponding portions of the clock tree are also shut down or powered up. The latency overhead involved in power gating and waking up functional resources is found to be to the order of 10 or, at most, a 100 cycles [99, 218, 216]. Since the granularity of resizing in our implementation is every 10,000 instructions, the impact of draining the CRIB partitions followed by power gating or waking up of partitions are minimal ($< 2\%$). Moreover, the delays from power gating a subset of partitions have no impact on the other awake partitions and they are ready to use without any overheads. Similarly, when partitions need to be awoken, already ready partitions are first filled up with instructions while others are being awoken in parallel. Therefore, the overheads from utilizing the newly awoken ones are reduced. The area overheads are negligible in comparison to the size of the tiles and the widespread clock-tree.

Fine-grained voltage-frequency control: The hardware resources required to control the voltage and frequency of the core for each quantum is minimal. Relative to standard

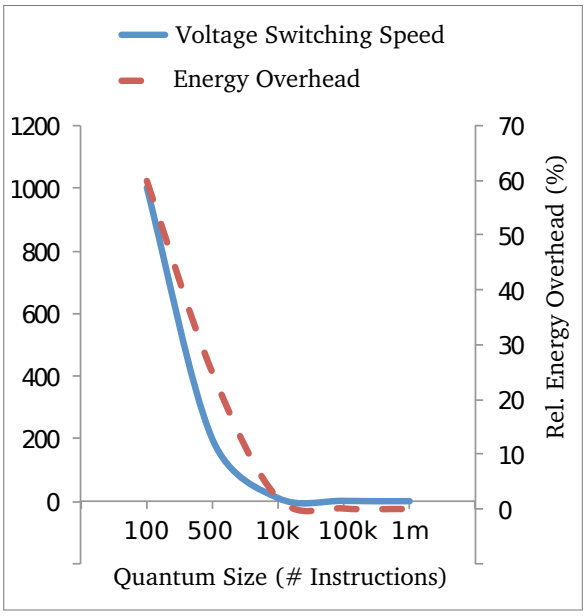


Figure 5.6: Voltage Switching Overheads

DVFS techniques, this falls in the realm of fine grained DVFS, achievable using on-chip voltage regulators [59, 121]. Based on the voltage-frequency relation established in [80], we require a voltage range of 1V - 1.2V corresponding to a frequency range of 800 MHz - 1.3 GHz. To enable switching across this entire domain every quantum (an aggressive assumption) we would need fine grained voltage regulation with a switching capability of 0.2V between quanta. A quantum size of 10000 instructions and allowing an overhead of 1% for DVFS would require the ability to switch a maximum of 0.2V over 35 ns (@ 1.3 GHz), allowing a safe assumption of a design towards a switching speed of 6 mV/ns. Eyerman et al. [59] consider voltage switching speeds up to 200 mV/ns while Kim et al. [121] design regulators capable of switching at 50 mV/ns with a 15% energy and 2-3% area overhead.

The voltage switching and energy overheads increase super-linearly at smaller temporal granularities (depicted in Fig.5.6) due to increased complexity in regulator design. For example, quantum sizes of 500 instructions results in energy overheads of 20-25%, preventing

voltage/frequency control at very fine granularities from being a viable option. Overheads can be marginally controlled if the processor operates through voltage transitions by either quickly ramping down frequency before voltage ramps down or quickly switching to higher frequency just as the voltage is settling at the higher levels [41, 121]. Considering this analysis and the prior work [59, 121], we believe that our requirement is within reasonable limits, with insignificant overheads (2%) in energy and even lesser in area.

NPU design and training: Our mechanism searches within our limited design space representable in 10 bits (8 (1-hot) bits for tiles + 2 bits for 4 frequency levels) and predicts ideal configurations with high accuracy. Considering the design space, a rudimentary predictor managed by a condensed neural network with few neurons is sufficient. Initial design space exploration showed that 10 neurons in the hidden layer provided high levels of accuracy. The lightweight design with relatively few neurons and only a single hidden layer keeps complexity and overheads minimal, *aptly suited for architectures aiming to strike a spatio-temporal balance in adaptivity*. The use of a tiled architecture baseline significantly reduces the design space complexity over prior work since only two parameters (number of tiles and frequency level) need to be predicted.

Different implementations of neural processing units (NPU) have been explored in prior works [57, 71, 58] with varying trade-offs in performance, power, area and complexity. We adopt a simple digital NPU implementation proposed by Esmaeilzadeh et al. [57]. Similar to this, we consider communication latency (tightly-coupled NPU) and computation latency (one hidden layer and 10 neurons) to both be 10s of cycles. Further, latency could be possibly hidden by starting the MLP computation marginally before the completion of the previous quantum. The power overhead is expected to be marginal ($< 1\%$) as it has few

neurons and is used only for 10s of cycles every 10k cycles and power gated otherwise. Based on estimates of neural functional unit area [51], processor area [76] and design synthesis, the area overhead is roughly 0.5%.

5.4 Evaluation

Experimental methodology is discussed in Section 4.1.

5.4.1 Impact of Clock Hierarchy awareness

In this section, we compare results from clock hierarchy aware power gating, normalized to a baseline of savings obtained from power gating only (i.e. no clock gating). We show results for 3 techniques: a) *Leaf*: which naively clock gates only the leaf node of the clock tree along with power gating the under-utilized resource, b) *Unaware*: which gates further up the clock hierarchy but the reconfiguration mechanism is unaware of (and uninfluenced by) the power saved from the clock nodes, and c) *Aware*: which uses our proposed clock-hierarchy aware gating mechanism. The benefits from these techniques are stacked upon one another in each of the figures 5.7, 5.8 and 5.9. These results are analyzed across H-Trees, tapered H-Trees (TH-Tree) and B-Trees for 3 different reconfiguration optimization goals.

Fig.5.7 shows increase in energy efficiency (over a PG-only baseline) obtained from reconfiguration optimized for lowest energy consumption when constrained by a requirement (or service level agreement i.e. SLA) that the performance falls no lower than 95% of the baseline. Key characteristics are highlighted below -

① First, it is evident that there are higher energy savings possible in H-Trees and B-Trees

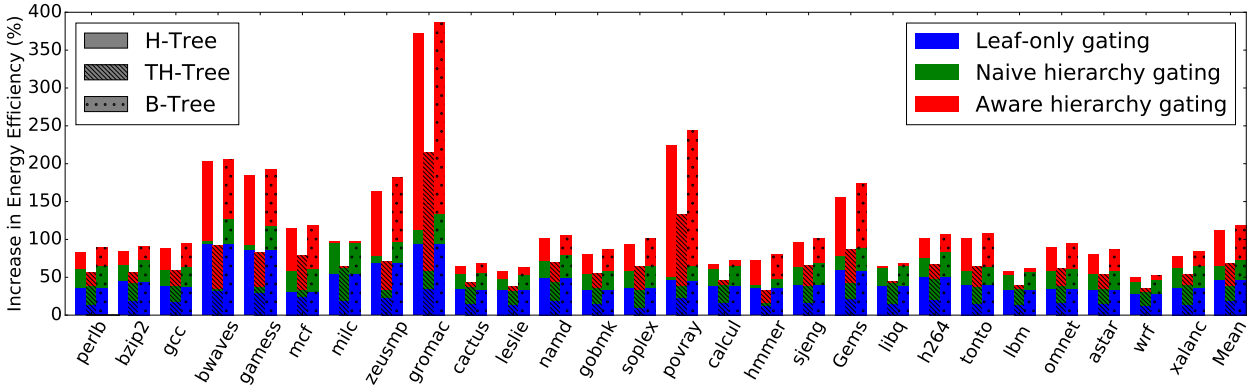


Figure 5.7: Clock-gating benefits w/ Perf. Constraint

in comparison to the tapered H-Tree. This is because nodes higher up the clock hierarchy consume more power in tapered trees, meaning that shutting down an extra *lower* node (i.e. a node that is, or is close to, a leaf) has a lower impact on the overall energy benefits. ② Results are marginally better for B-Trees in comparison to H-Trees because there are deeper hierarchies in B-Trees in comparison to H-Trees (the number of nodes double at each level in B-Trees, while they quadruple in H-Trees), thereby providing more opportunities to gate unused common nodes in the former. ③ Clock gating leaf nodes alone provides 20-45% higher energy savings in comparison to the PG-only baseline, highlighting the importance of clock gating. ④ The benefits of gating higher up the clock tree, even without clock hierarchy aware reconfiguration provides 20-27% higher energy savings compared to only leaf node clock gating. This quantitatively reiterates the need for designing gating mechanisms to shut down clock nodes further up the clock hierarchies and not just clock gating leaf nodes alone. ⑤ Finally, clock hierarchy aware reconfiguration provides a 27-39% increased energy savings over clock hierarchy unaware reconfiguration in tune with our qualitative motivation earlier. ⑥ Overall, optimized CDS-aware reconfiguration provides 65%-111% greater relative energy savings in comparison to PG-only reconfiguration.

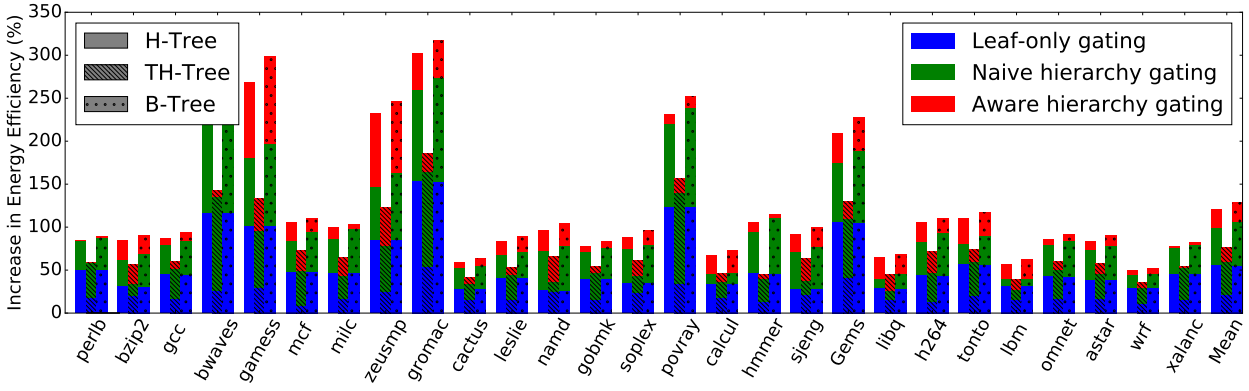


Figure 5.8: Clock-gating benefits w/o Perf. Constraint

We also show benefits for reconfiguration tuned to optimum energy (Fig. 5.8) and optimum energy-delay (Fig. 5.9) when unconstrained by any strict performance requirements. The observed benefits are similar to what was described above.

In the unconstrained optimum energy case (Fig. 5.8), the energy benefits of gating up the clock hierarchy (**Unaware** vs. **Leaf**) is higher than the constrained performance scenario (improvements are 36-45% here) while benefits of clock aware reconfiguration (**Aware** vs. **Unaware**) is lower (15-20% improvements here). The reason is that since there is no performance constraint in this scenario, the chosen configurations are mostly low resource configurations. This is because the most energy efficient configurations are usually ones with low resources with low performance - huge power savings at a loss in performance still provides higher energy savings. Therefore, there is significant scope to even naively gate higher up the clock hierarchy which increases its benefits in comparison to gating only leaf nodes. Since naive gating of upper hierarchy already provides considerable benefits, the extra benefits from smarter clock awareness correspondingly reduces. Overall benefits are higher: optimized CDS-aware reconfiguration provides 72%-118% increased unconstrained energy savings in comparison to PG-only reconfiguration.

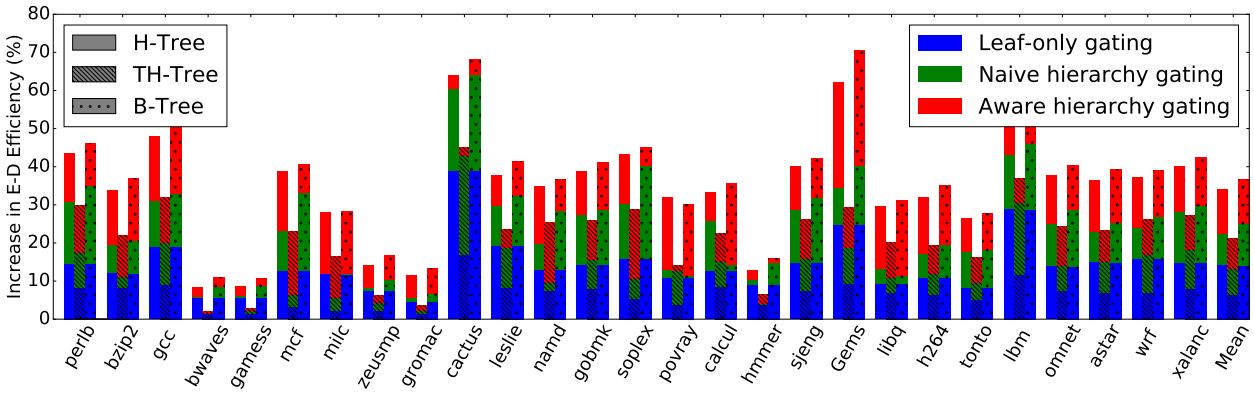


Figure 5.9: Benefits for optimum Energy-Delay

Energy-Delay optimization (Fig. 5.9) improves both energy efficiency and performance. Performance gains are obtained via boosting the frequency when nodes are power gated (as a part of PG-DVFS). More details of PG-DVFS benefits are discussed in the following section. ED improvements range from 21% to 36% in comparison to PG-only baseline.

5.4.2 Impact of integrated PG-DVFS

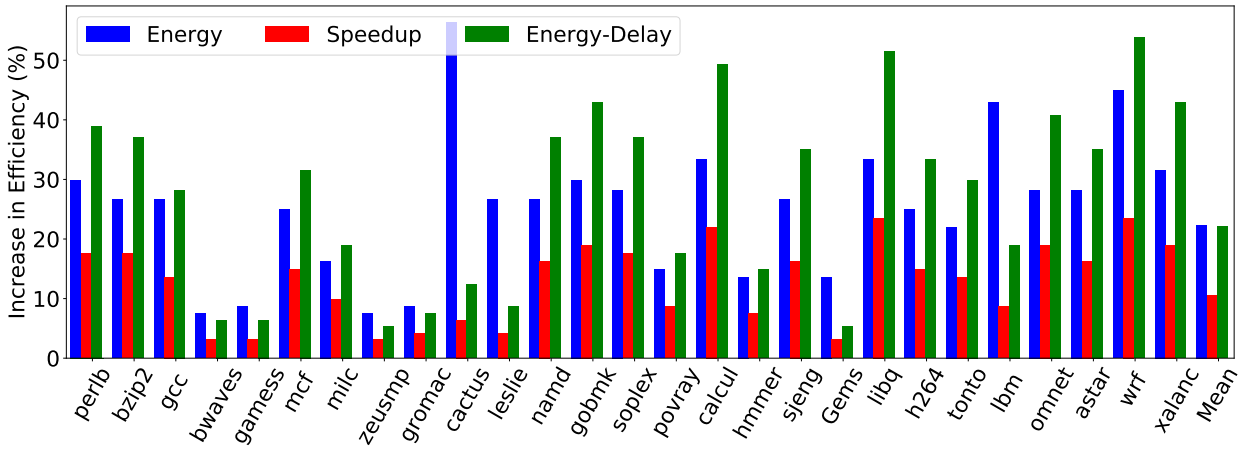


Figure 5.10: Benefits from integrated PG-DVFS (Constrained).

Next, we examine the performance, energy and E-D efficiency improvements from integrated resource-frequency scaling in comparison to a baseline without reconfiguration

(Fig.5.10 and Fig.5.11). The results utilize clock-tree aware reconfiguration, averaging the clock power to be 33% of the total power consumption.

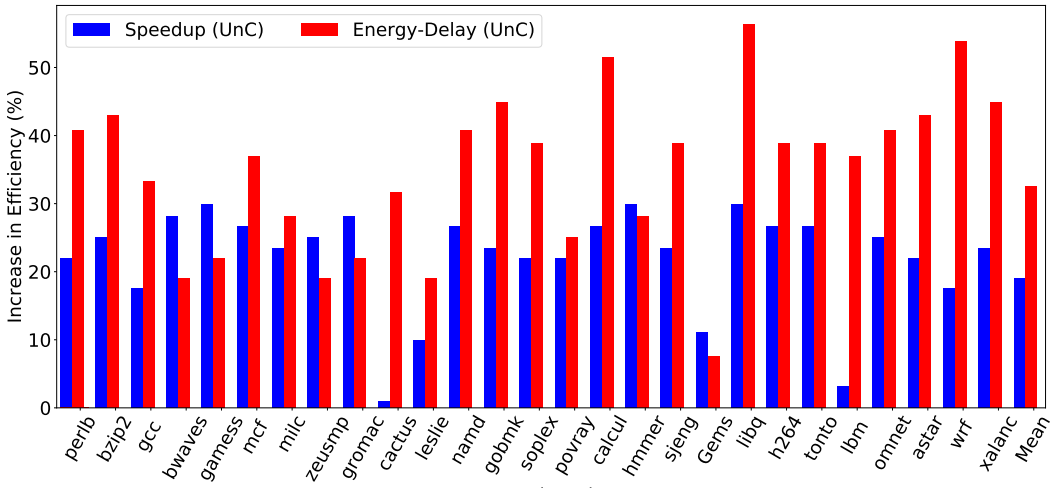


Figure 5.11: Benefits from integrated PG-DVFS (Unconstrained).

① Column 1 in Fig.5.10 shows the energy savings from integrated PG+DVFS, when optimizing for best energy efficiency under a minimum performance target of 95% (SLA) in comparison to the baseline. ② Columns 2 and 3 in Fig.5.10 show performance gains and energy-delay reduction with a control mechanism targeting maximum performance under the condition that each quantum operates within the baseline power budget. ③ Columns 1 and 2 in Fig.5.11 show performance gains and energy-delay reduction when the control mechanism optimizes for lowest energy-delay over every quantum with no strict constraint on the power budget (UnC = unconstrained).

In summary, the figures shows energy savings of 19% under a 95% SLA, performance improvements of 11/18% and ED reduction of 22/26% under power constrained and unconstrained environments. The utility of the integrated mechanism is evident from significant improvements in all above metrics. While not shown in the graph, the energy savings increases by 1.9x when moving from a naive PG+DVFS mechanism to the integrated

mechanism proposed here (qualitatively discussed via scenarios A-D in Section 5.2.1).

5.4.3 Impact of control mechanism

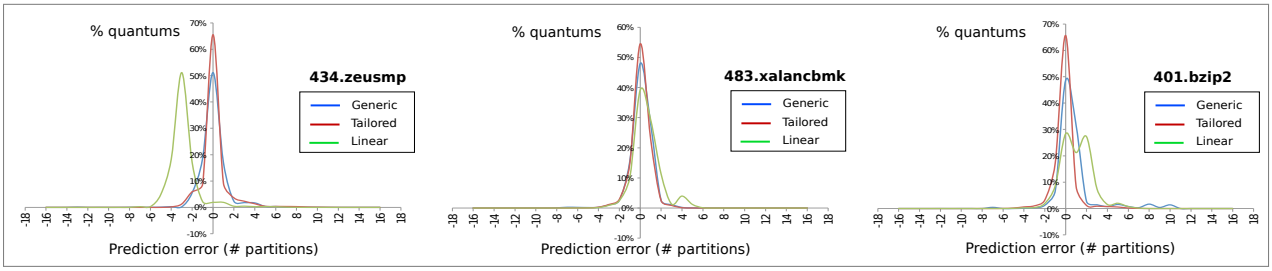


Figure 5.12: Accuracy of prediction: Tailored vs Generic vs Linear

To motivate the accuracy of a generic one-time offline trained effort we make the following comparison. We compared the accuracy in prediction of a *generic* MLP (trained one-time offline) with a *tailored* MLP trained on a per benchmark basis and a linear regression model, across the SPEC CPU2006 benchmark suite. The *tailored* MLP provides a limiting model as to the best accuracy achievable with an MLP based predictor with reasonable overheads. Figure 5.12 shows the comparison for 3 benchmarks in terms of their prediction errors. It is evident that the accuracy of *generic* closely follows that of the *tailored* while *linear* is inaccurate. On average (in 16-partition CRIB), *generic* shows error of 0.45 partitions-per-prediction, *tailored* sees 0.28, and *linear* sees 1.8. When the training set for *generic* is reduced to samples from a random half of the SPEC benchmarks, error increases to 0.49. Further reduction of the training set to samples from a random quarter of the SPEC benchmarks increases the error to 0.67 partitions-per-prediction. These are still significantly more accurate than the linear predictor. The high accuracy of the generic MLP predictor suggests that *a generalizable non-linear function, trained offline, captures the behavior of many workloads*.

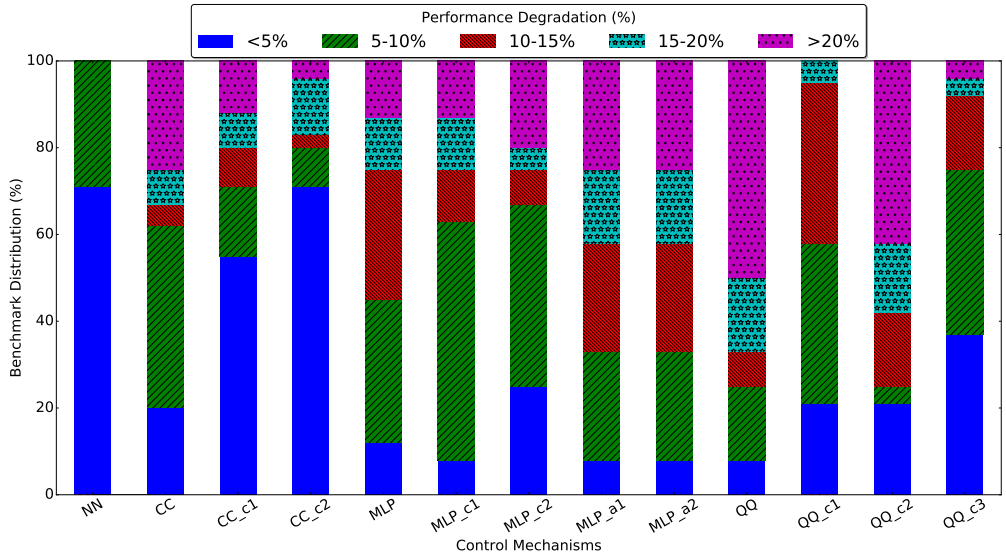


Figure 5.13: Perf. degradation across mechanisms

Figure 5.13 compares the deployment accuracy of multiple prediction mechanisms. It illustrates the performance degradation suffered due to prediction inaccuracies across the SPEC benchmark suite, for each prediction mechanism. The degradation is in comparison to the performance obtained by an ideal oracular predictor. Our proposed mechanism using a neural network is shown as *NN*. Existing prediction mechanisms shown are - CC [141], MLP [122](memory level parallelism) and QQ [170]. To broaden our comparison, we also implement more aggressive (eg. *MLP_a1*) as well as more conservative (eg. *QQ_c1*) versions of these.

Performance degradation from prediction is important to consider so as to avoid violations of SLAs and issues with tail latency [98]. For instance, if we consider an SLA criterion that requires at least 75% of benchmarks in the suite to fall within 10% performance degradation, the only prediction mechanisms meeting these criteria are our own mechanism (*NN*), along with *CC_c2* and *QQ_c3*. It is important to note that even within this limit, only *NN* meets an almost ideal prediction accuracy of 70% of the benchmarks

within 5% degradation and 100% of the benchmarks within 10% degradation. Analysis of ED^2 product [24, 115], also shows that the *NN* predictor closely matches an ideal predictor and provides roughly 20% better ED^2 compared to the competing mechanisms.

5.5 Discussion

Some key discussion points are put forth below:

- Firstly, the ideas of PG-DVFS and using MLP based control for reconfiguration are applicable to any general architecture, and not limited to tiled architectures alone. The ideas of clock tree awareness and spatio-temporal balance are suited to any architecture with well defined clock hierarchy and resource clusters - so suited to all tiled architectures.

- Second, the focus of this work is on single-threaded applications. CHARSTAR could be equally useful in a multi-threaded or multi-programmed context, but MLP complexity would increase based on number of threads/programs allowed simultaneously. Each thread's activity (cache misses, branches etc) would ideally need to be uniquely monitored and resources should be uniquely controlled (in terms of DVFS and PG). This could increase the complexity of the MLP super-linearly. But the current size of the MLP is very small (10 neurons) so the increase in its size to control a reasonable number of threads may not cause a huge overhead. Our future work hopes to enable running multi-programmed and multi-threaded workloads on tiled architectures with the proposed optimizations.

- Third, it is also important to note that there are also other parameters that can affect tile configuration, such as inter-tile communication latency, thermal hot spots, etc. In designs with distributed shared memory, data reuse across tiles is another influential factor. Clock hierarchy aware configuration selection bodes well with keeping inter-tile communication latency low and spatial data reuse, but may not be the best configuration thermally. Further, if the tiles themselves are heterogeneous, then some tiles could be prioritized over others even if it might detrimental to clock power, communication latency, hot spots and so on. Designing a control mechanism that is cognizant of all these aspects could achieve the best overall reconfiguration efficiency and is potential future work.

- Finally, we discuss some topological constraints. In the CRIB architecture, when dependent instructions lie in different partitions, there is a one cycle overhead per partition to transfer data from older to younger dependent instructions. When a number of the partitions are power gated, the overhead involved in transferring data across multiple power gated partitions becomes significant. Such overheads are applicable to all tiled architectures. In order to minimize this overhead, we make use of *shortcut links* between partitions to bypass the penalty of transferring data across power gated partitions. Exploration of different topologies shows increasing overheads with more shortcut links due to increasing size of routers, input buffers, crossbars and allocation logic etc. Optimizing for the right amount of bypassing for a control mechanism cognizant of the different configuration-influencing parameters (discussed above) is also worthy of future exploration.

5.6 Chapter Summary

This proposal explored novel techniques to improve energy efficiency and/or performance through dynamic reconfiguration, by effectively reducing both leakage and clock power. We introduced *CHARSTAR*, integrated power gating and DVFS within the core while also considering the power consumed by each node in the clock tree hierarchy. Our control mechanism is aware of the power consumption of each node in the clock tree, thereby choosing an ideal resource configuration which minimizes the combination of clock tree power *and* leakage power. Varying benefits are analyzed for different types of clock trees. Integrated PG-DVFS scaling is shown to be more effective at saving energy and/or achieving higher performance in compared to naive decoupled mechanisms, achieving optimum configuration per application phase in terms of both ILP and frequency. We explored the inherent advantages of tiled architectures towards dynamic reconfigurability and optimized it for balanced spatio-temporal adaptivity. Finally, we make use of a lightweight MLP predictor, suited to fine temporal granularities, to accurately predict the configurations for each application phase.

CHARSTAR, when deployed on the spatial CRIB architecture, shows improved processor energy efficiency by 20-25% in comparison to an unoptimized baseline, with efficiency improvements of roughly 2x in comparison to naive power gating mechanism. It can alternatively improve performance by 10-20% under varying power/energy constraints.

6 AGGRESSIVE SLACK RECYCLING VIA TRANSPARENT PIPELINES

In this chapter, we explore the SlackTrap proposal. SlackTrap is the first step among the various slack recycling mechanisms discussed in this dissertation. Similar to CHARSTAR, it is implemented atop a tiled architecture (CRIB). The presence of considerable computing resource on spatial fabrics, as well as the eager allocation of these resources, is favorable to SlackTrap's implementation of "asynchronous" timing speculation or transparent dataflow based slack recycling. SlackTrap takes a speculative approach to slack recycling which is aided by statistical estimations of slack. Thus, while it has no design overheads in the micro-architecture front-end, it requires error detection and recovery mechanisms. In contrast REDSOC takes an accurate per-instruction slack estimation approach (Chapter 7).

First, in Section 6.1 we present the potential for asynchronous timing speculation: its motivation from statistical theory and implementing it via transparent pipelines. Next, in Section 6.2 we discuss the control mechanism involving slack estimation, slack tracking / accumulation and the early clocking approach which provides the performance benefits. In Section 6.3, we discuss implementation over the tiled architecture baseline. Finally, in Section 6.4 we present an evaluation of its performance as well as an analysis of its overheads. Section 6.5 summarizes the proposal.

An introduction to SlackTrap was provided in Section 2.2, background / motivation in Chapter 3 and methodology in Section 4.2.

SlackTrap's contributions are summarized below:

① We explore the inherent advantages of transparent pipelines towards efficient timing

speculation.

② We show that a design that allows asynchronous multi-cycle execution of a sequence of computations, is able to employ aggressive timing speculation that caters to the average slack across the sequence rather than the slack of the most critical operation.

③ We redesign the CRIB processor [76] to enable transparent pipeline based slack recycling, thus building the foundation for aggressive timing speculation.

④ Based on statistical theory, we show that the longer the asynchronous execution sequence is, the more aggressively timing speculation can be performed.

⑤ With this design, clock cycle slack stemming from systematic process variation, systematic temperature/voltage variation, random process variation and random data variation is recycled, resulting in performance gains.

⑥ Atop CRIB, SlackTrap achieves absolute speedups up to 20% and relative improvements (vs. competing mechanisms) of up to 1.75x.

6.1 Asynchronous Timing Speculation

6.1.1 Motivation from statistical theory

A popular technique to adaptively control timing guard-bands is Razor [55] which tunes the supply voltage by monitoring the error rate during operation. The commonality among Razor and most TS approaches is that, they all *focus on reducing slack on a per operation basis* and are *constrained by the possibility that timing errors might be caused by every operation, in every unit and on every cycle.* These techniques are tuned to be relatively conservative - having to

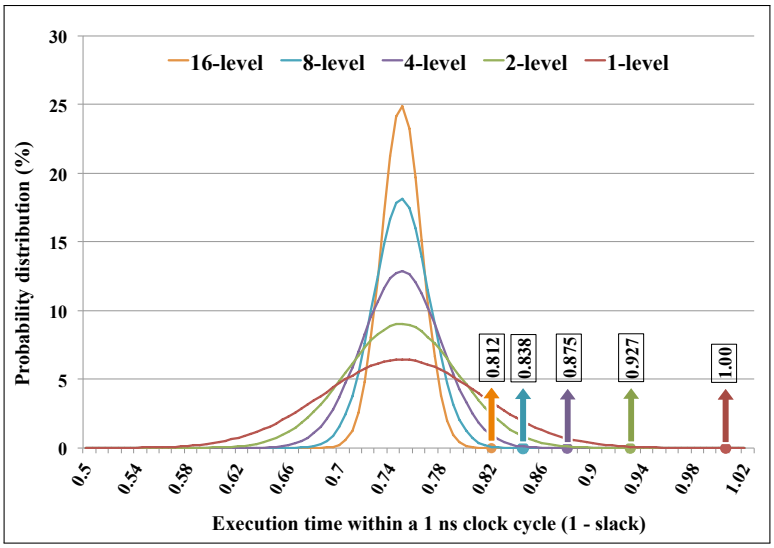


Figure 6.1: Slack distribution

cater to the most critical operations or execution stages, to prevent mispeculation, or suffer the risk of increasing possibilities of timing errors. The following analysis describes the potential for more aggressive timing speculation and motivates our primary proposal.

Fig.6.1 shows different slack distributions. Consider the flattest distribution, the *1-level* curve in red. This represents independent and identically distributed logic delay within an FU - averaging at about 75% of the clock cycle but with a reasonably large variance. This curve corresponds to the logic delay experienced on every clock cycle, by a standard design wherein every operation executes synchronously. The corresponding red arrow refers to the 99.99% confidence interval mark, which is the clock period that, if set, allows not more than 1 in 10000 operations to hit a timing error. The arrow is roughly at the 1.00 mark and thus, in this example, using the 99.99% estimate as guard band is unable to cut out any slack.

On the other hand, assume a design wherein multiple independent operations are executed asynchronously in a sequence i.e. the operations are not separated by clocked

elements. The higher *N-level* curves in Fig. 6.1 correspond to the resultant slack distribution when *N* operations are executed in a multi-cycle internally asynchronous sequence, bounded by synchronous elements.

Via transparent data-flow, the slack accumulates across this sequence and the mean slack estimates are influenced by the number of operations that can be combined together and executed as an asynchronous chain. Note that the greater the lengths of these chains, the higher the average slack per operation - this is explained below. This is because, for independent variation, the estimated mean slack is averaged out over the entire sequence, which would predominantly consist of non-critical operations. The longer the sequence, the more the number of operations that tend to lie closer to the mean value (curve peak increases and width narrows). This would mean that that outliers with a longer critical path can be cushioned by more non-critical operations which consumed less than the high confidence execution time estimate. This allows a lower guard band (i.e. more aggressive clock) for the same confidence interval. In this example, combining 16 operations together (16-level) allows a 20% reduction in the 99.99% guard band estimate.

This statistical representation is a direct interpretation of the central limit theorem (CLT). CLT loosely states that the larger the sample size obtained from a population with a finite level of variance, the more probable it is that the mean across all the samples will be approximately equal to the mean of the population. CLT further states that all of the samples will follow an approximate normal distribution pattern, with all variances being approximately equal to the variance of the population divided by each sample's size.

From this example, it is evident that multi-cycle execution of asynchronous operation sequences provides abundant potential for increased aggressiveness in timing speculation.

6.1.2 Utilizing transparent pipelines

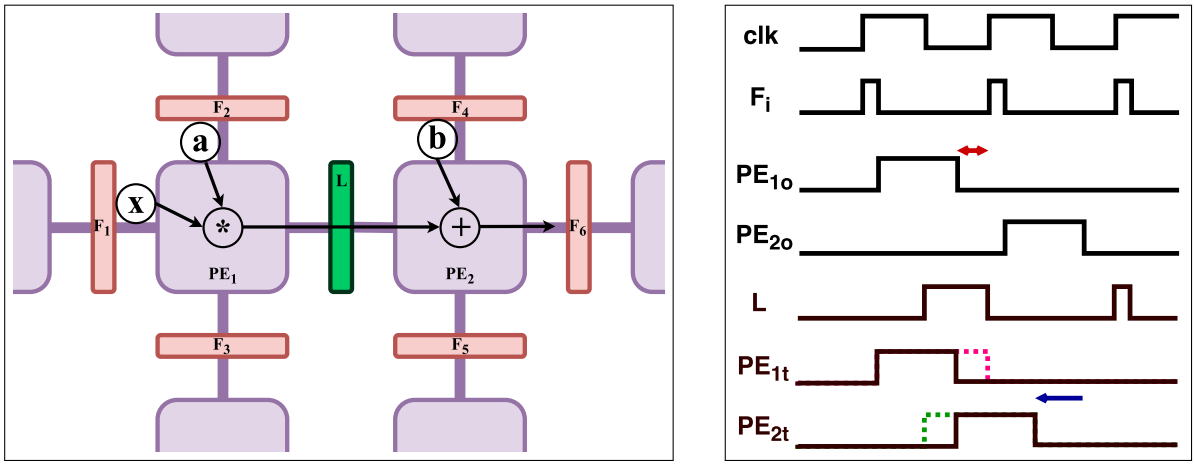
To exploit the opportunity motivated above, we seek an asynchronous engine design that can integrate seamlessly with standard synchronous pipelined systems/interfaces. Typical circuits with some asynchronous characteristics are: Purely combinational multi-cycle data paths (MDP) [193], Asynchronous Elastic Pipelined (AEP) logic [155] and Transparent pipelines via intelligent latching [104, 87]. All of these are capable of conserving timing slack by executing a sequence of operations in an asynchronous group but the below discussion motivates the better capabilities of transparent pipelines in comparison to the others.

MDPs for DFG-style execution have been explored in specialized data paths for executing specific (hot) basic blocks [193]. They execute a basic block for each pulse of a *slow clock*, foregoing pipeline registers and saving area/power. Such data paths are not pipelined, resulting in low throughput (or need for extensive resource replication). Further, being specialized, they lack flexibility for general-purpose programming.

AEPs can theoretically match the throughput of a synchronous pipeline [30, 155]. But they suffer from other inefficiencies. Completion detection within pipe-stages, the handshake mechanisms between pipe-stages (eg. 4-phase signalling) and feedback loops are complex to implement [155]. This restricts high-frequency implementations, causes high area overheads and complicates DFT support. Moreover, interaction with synchronous interface is harder to implement, requiring complex Async-Sync FIFOs [32].

Thus, we explore transparent latch based pipelines. A transparent latch is a storage element with an input, an output, and an enable. When the enable is active, the output

transparently follows the input. When the enable becomes inactive, the latch becomes opaque and the output freezes. Latches between FUs are made transparent at appropriate times to allow data to flow through at non-clock boundaries. At other times, the latch is kept opaque preventing dataflow. As described earlier, data paths with transparent latching inherently suit tolerating variations, since varying execution times among instructions which could cause timing errors in an aggressive purely synchronous guard band design can be avoided by allowing such varied delays to be balanced anywhere within the transparent execution window. This allows varied logic delays among operations to be balanced anywhere within the transparent (i.e. asynchronous) execution window. Our work speculates across sequences of multiple instructions by utilizing these multi-cycle data paths and is constrained only by the average criticality across these sequences, and is cushioned against limitations imposed by the rare critically-timed instruction. The primary benefits explored earlier from transparent pipelines include reducing clocking power [104, 76] as well as interlocked synchronous pipelines, which reduce stall related overheads [105].



(a) Transparent Latch example (b) Timing Diagram

Figure 6.2: Transparent Dataflow

Fig.6.2a illustrates the use of a transparent latch (L, in green) between 2 processing elements (PEs). The figure shows the mapping of a function $a * x + b$ onto the 2 PEs, where PE_1 performs a single-cycle multiply and PE_2 performs single-cycle addition. Assume that a, x and b are available at the PE inputs. Note that PE_2 is idling until the multiply operation completes atop PE_1.

The timing diagram (Fig.6.2b) shows 2 different scenarios - the baseline scenario (refer PE_{1o}, PE_{2o} in figure) which assumes a standard positive edge triggered flip-flop separating the PEs and the transparent scenario (refer PE_{1t}, PE_{2t}) which uses the transparent latch between them. In the former, the flip-flop (F) opens only for a short period of time at the positive clock edge, allowing data to pass through. In the latter, the latch (L) is made transparent for appropriate slack-controlled periods of time. Note: In the figure, a high level (1) for PE_i indicates some computation being performed on that PE.

In the baseline scenario, the multiply operation (on PE_{1o}) is allowed an entire cycle to execute despite the presence of timing slack (shown in red). The addition on PE_{2o} begins only begins after the second positive clock edge. On the other hand, in the transparent design, slack-aware latch control allows L to be open for a period of time which covers the instant at which the multiply completes on PE_{1t}. This allows transparent data flow of $a * x$ into PE_{2t}. Which, in turn, allows PE_{2t} to start real addition computation at the instant of completion of PE_{1t} - thus conserving slack and completing function execution faster than the baseline. Note: the pink dashed line at the end of PE_{1t}'s execution is the error checking period - required due to slack estimation being a speculative mechanism (Sec.6.2.4).

In summary, we propose a *synchronous slack tracking and opportunistic early clocking* mechanism implemented atop a *transparent pipeline* execution engine. Our proposal *utilizes*

otherwise idle functional units to capture slack and reduce execution latency.

6.2 Control Mechanism

6.2.1 Slack Estimation

Background on different forms of clock cycle slack is discussed in Section 3.3.

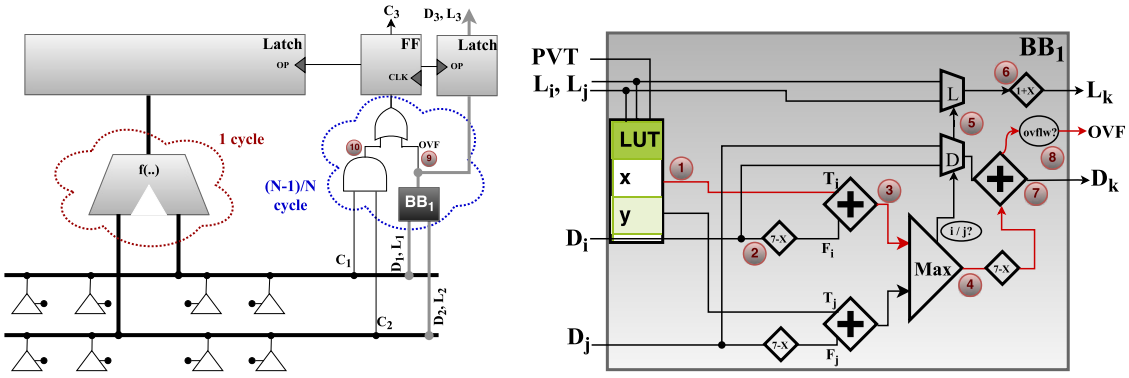
The complexities in accurately estimating the available slack for every operation are tremendously high and therefore exact measurements on a cycle-by-cycle basis are impractical. On the other hand, it is reasonable to make synchronous-domain style slack estimates using the following: ① static design-time information, ② feedback based slack predictor, ③ building a normal distribution model, and ④ extending this to multi-operation sequences using statistical theory. In our analysis, the slack model includes components from the following variations: systematic process, systematic temperature/voltage, random process and data-based. The latter two are modeled as independent/identically distributed (IID) across each operation while the former are correlated across operations.

Tribeca [77] proposed the use of a simple last-value predictor to predict circuit delay behavior under PVT variations, which is then used to tune the processor V/F settings. The predictor chooses the *setting* for the current epoch based on the previous setting and the number of timing violations in the previous epoch. The proposed predictor is within 2% accuracy of oracle prediction. To capture **systematic variations**, we use such a predictor in our design, since our baseline requirement is the same. Such predictors can be appropriately distributed across the chip, so as to pass on localized timing guard

band predictions to the proximate compute node(s). Critical Path Monitors [134] could be added to improve slack estimation accuracy further, but we don't explore this possibility. Similar to Tribeca, we make use of a tuning resolution of 10K cycles - but our resolution could be more fine-grained since we do not require costly frequency or voltage tuning.

Fine-grained spatial and temporal **random variation**, along with data-based variations, are not captured from the above. Therefore, to model slack more aggressively (and more accurately) we use statistical modeling aided by static design-time information. We follow prior works [116, 195] that model slack from the probability density function (PDF) of the logic delay as a Gaussian normal distribution with calculated σ and μ values. These distribution parameters are calculated based on estimates of the effects of data inputs as well as PVT variations on logic delay. Sec.4.2 discusses the estimation of these components in our work.

Statistical slack modeling is especially useful for multi-cycle coalescing based slack estimates, which was discussed earlier in Sec.6.1.1. The statistical execution time estimate at a particular confidence interval mark (we use 99.99%) is estimated for different length asynchronous operation sequences. These values are written into a look-up table (LUT). The LUT is addressed by the length of the sequence (L) and each entry contains the execution time estimate for the L^{th} operation in an asynchronous sequence. Based on the position of an operation within its asynchronous sequence, its slack estimate is obtained from the LUT and used appropriately.



(a) Modified FU (b) Special functions

Figure 6.3: (1) L_i addresses into LUT to obtain estimation computation times of current operation: T_i, based on i's DFG. Similar for j. (2) D_i, the slack accumulated via i's DFG, provides F_i (= 3'b111 - D_i), the completion instant of i within its completion cycle. Similar for j. (3) $F_i + T_i$ is completion time estimate for k based on i. Similarly with j. (4) Conservative estimate for k is assumed from the above, via the Max() operation. (5) Depending on i/j being the Max(), muxes select constraining producer's D and L. (6) L_k is obtained as 1 + constraining L. (7) Max() output is converted into slack, and is added to constraining D to create: D_k, the cumulative slack. (8) If the cumulative slack overflows, OVF is set. (9) OVF set means slack crosses integral boundary and hence early clocking is performed: clocking the operation in the same cycle as the last parent. (10) If not, standard clocking is performed, one cycle after completion of the last parent (assuming 1-cycle baseline).

6.2.2 Slack Accumulation

A computational PE, along with routing logic and slack tracking mechanism is shown in Fig.6.3.a. Each unit is provided with additional control bits - the executing operation's level in its DFG (L) and the cumulative slack in the operation (op) sequence (D). These bits are propagated along with data flow. Sensitivity analysis for sizing L and D showed 4-bit L (i.e. 16 levels) and 3-bit D (i.e tracking accuracy of 1/8th of cycle) were sufficient for maximizing speedup at minimal design costs. In this scenario the first op after a synchronous boundary with no slack would have L = 4'b0000 and D = 3'b000. A following dependent op which takes 75% of a cycle to compute, would have L = 4'b0001 and D = 3'b010.

The hardware performs timing estimation akin to design-time static timing analysis. BB_1 in Fig.6.3.b shows the dynamic slack estimation mechanism assuming 2 producers (i, j) and one consumer (k). The goal is to estimate D_k and L_k for the consumer (current) op. Fig.6.3's caption details the design. The slack tracking circuitry is completely in parallel with the data-path and, due to simple logic, has a shorter critical path: thus, no impact on the design's cycle time.

6.2.3 Early Clocking

Each PE is provided with a completion bit (C) which, when set, indicates that the op it executed has completed. The control mechanism is kept synchronous, so when an op is deemed to be complete at some instant, C is to be set on the subsequent clock cycle boundary.

In this design, let N be the shortest sequence of dependent ops which can accumulate enough slack to shave off one cycle from its execution time (i.e. the synchronous boundary will need to be clocked one cycle early). This would mean that the chain of N ops would complete in $N-1$ clock cycles. This requires N dependent ops' completion bits to be synchronously set in $N-1$ clock cycles. This can be achieved if slack information propagation for this N op sequence can complete in $N-1$ cycles. To achieve this, the slack computation delay per op should be no greater than $(N-1)/N$ cycles (highlighted in Fig.6.3.a). This is our design's only timing constraint. Further, the propagation of computed slack information should form a (transparent latch based) multi-cycle path. This can be observed in Fig.6.3.a and Fig.6.4 wherein slack information propagation bypasses the flip-flop.

The above timing constraint is converted into a bound on the maximum timing slack per op that can be recycled in our design. Let this maximum recyclable slack be s fraction of the clock period - we necessarily forgo any benefit where slack is greater than s. It is intuitive that an N op sequence would complete in $N-1$ clock cycles for the smallest N *iff* each op has maximum slack $(=s)$. Thus $N-1 = N - N*s$ or $N = 1/s$. Substituting this in the earlier result, the slack computation delay per op should thus be less than or equal to $(1-s)$ fraction of a cycle. In other words, the lower the slack computation time, the higher the maximum slack that can be recycled. Note: $s = 0.5$ is the maximum slack we allow in our design and synthesis of slack computation logic easily meets the 0.5 cycle requirement. It should be possible to design for higher or lower slack requirements.

For each computation node, the data latch is made transparent when allocated and is turned opaque in two ways. It could be turned opaque on the next cycle after the last parent op by noting the synchronous completion bits of the parents (i.e. standard synchronous dataflow). It could also be turned opaque on the same cycle as the last completing parent, if the OVF-bit is set. This is because, OVF is set when accumulated slack crosses an integral value, (which causes the slack accumulator to overflow: Fig.6.3.b). Cumulative slack crossing an integral value means that the current M^{th} op can be clocked in the $M-1^{th}$ cycle (rather than the M^{th} cycle).

Data flow over 3 single-cycle operations A-C atop this design, is shown in Fig.6.4: ① Starting at $T = 0$, operation A computes for $0.6ns$. It is synchronously clocked in at the clock boundary i.e. at $T = 1.0ns$ and the slack accumulated is $(1 - 0.6) = 0.4ns$. ② Via transparent data-flow, operation B can start computing at $T = 0.6ns$ and computes for another $0.6ns$, ending at $T = 1.2ns$. It is clocked in at $T = 2.0ns$ (one cycle after A) and

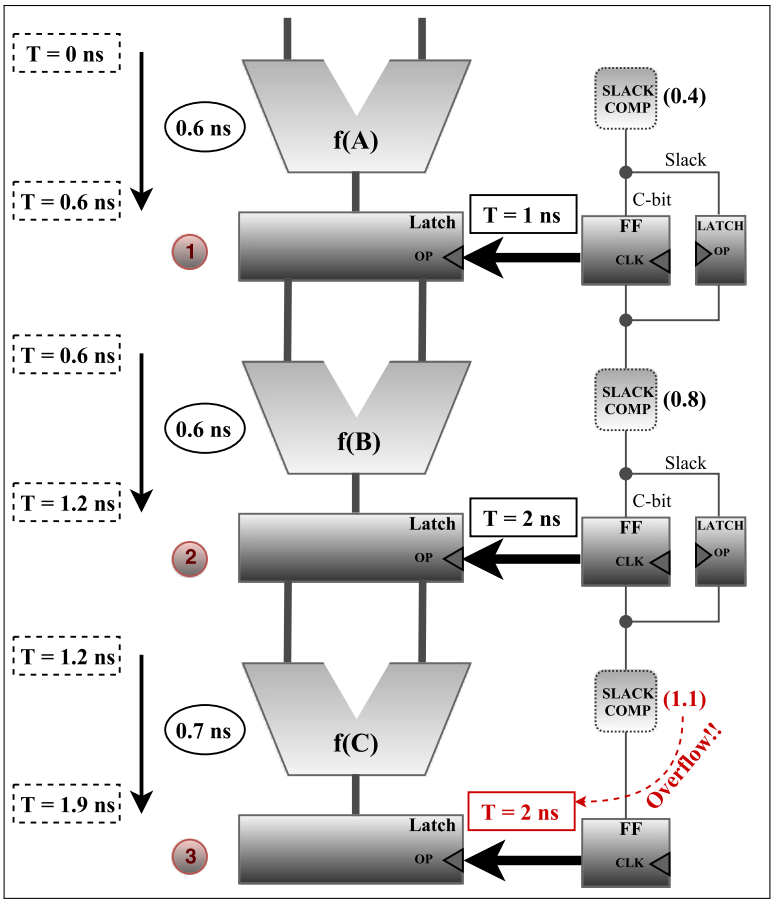


Figure 6.4: Slack-aware transparent data flow

slack accumulated in total is $(0.4 + 0.4) = 0.8ns$. ③ Similarly, operation C starts compute at $T = 1.2ns$ and completes at $T = 1.9ns$. The total slack accumulated thus far becomes $(0.8+0.3) = 1.1ns$ which causes an overflow (by crossing integral boundary). The overflow results in C being clocked in $T = 2.0ns$ i.e. the same clock cycle as B.

6.2.4 Error Detection and Recovery

We optimize a Razor-like error detection mechanism to suit our requirements. For any single computation, the stable correct output will surely be set within the next s clock cycle fraction. Thus, the inputs to the functional unit are retained for this extra s clock fraction

(to avoid short path problems [55]) and error-detection is performed. This detection can complete within the same cycle as the estimated completion or in the next cycle. In the latter case, the inputs are retained at the compute node for the extra cycle.

Error detection is performed via an XOR comparison between the latched *Early* output and the later *Shadow* output. If the PE's output data changes a timing violation has occurred. The latch is then made transparent to capture the correct value and recovery is triggered.

Local data recovery within asynchronous operation sequences involves negligible overheads since transparent data flow is self-correcting across all consumers. Since the latched value in the erroneous FU is corrected after detection, the correct system state can be established by delaying the setting of completion bits (and data latch capture) of younger operations by a single cycle. Further, the slack accumulation is forced to 0 for recovery sequences so that latching correct data is solely controlled by synchronous completion bits. In case recovery spills across synchronous boundaries, overheads are akin to synchronous designs and involves reissue of the synchronous boundary operations and following ones.

6.3 Spatial Architecture Baseline

The design described in Sec.6.1.2 (Fig.6.2a) expects a dependent operation (ADD) to be present early (i.e. waiting) at an idle compute unit (PE_2). Such a design can be easily achieved atop a pipelined spatial computing fabric. Spatial fabrics are generally overprovisioned with enough compute resources for high throughput when sufficient application parallelism exists. Under low utilization scenarios these idling/waiting resources can perform slack conservation. Availability of idling/waiting PEs also eases transparent

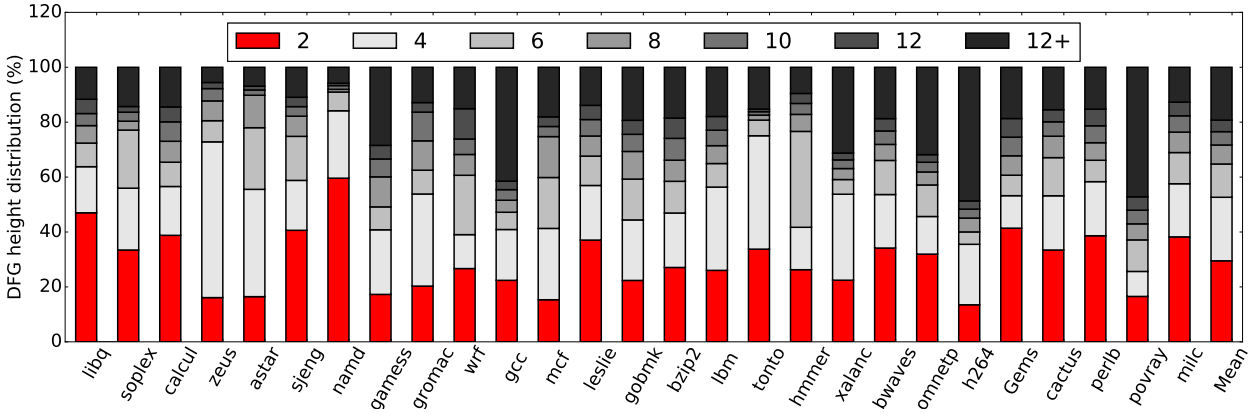


Figure 6.5: DFG Height Analysis

latch control. In traditional transparent pipelines [104, 87], latch management is more complex due to concurrently executing tasks in both the producer stage and the consumer stage, resulting in the need for scheduling bubbles [87], etc. But this is avoided in spatial frameworks which allocate tasks only to free PEs, often far ahead of actual execution.

We implement and evaluate our proposal atop the CRIB spatial architecture [76]. Details on CRIB can be found in 3.1.2 and in the original work [76]. Design specifications are presented in Table 4.1. We implement our proposal by provisioning transparent latches between CRIB PEs and adding other components as described in Sec.6.2. The primary impacting synchronous boundaries are front-end dispatch and memory operations.

6.4 Evaluation

Experimental methodology for SlackTrap is discussed in Section 4.2.

DFG Height Analysis: Fig.6.5 shows the distribution of DFG heights for SPEC benchmarks. These are the DFGs formed between synchronous boundaries and slack can accumulate across them. For instance, with 25% slack averages, DFGs of height 4 or more

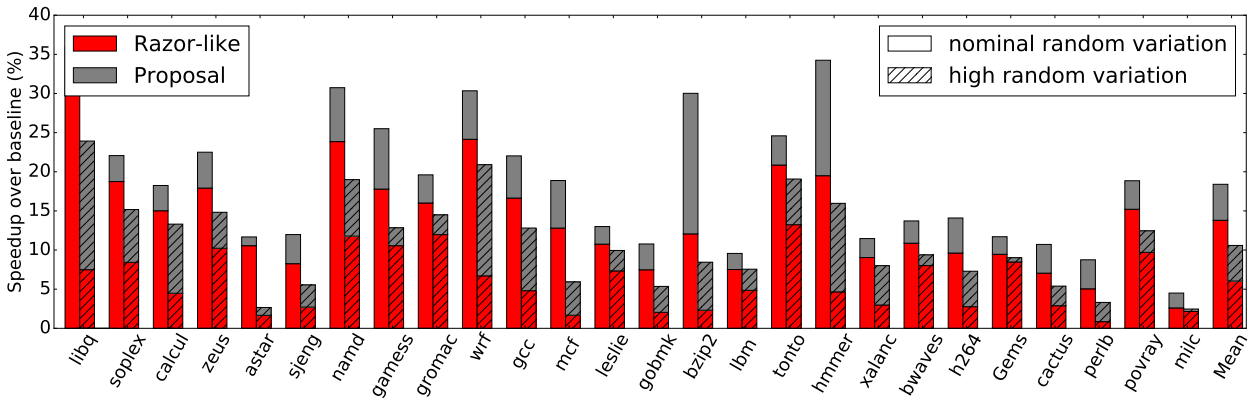


Figure 6.6: Speedup over baseline

accumulate enough slack to clock the end boundaries early. More than 50% of DFGs across most benchmarks allow early clocking.

Performance Speedup: Fig.6.6 shows speedup from our proposal. Speedup is shown for two random slack distributions: *nominal* and *high* (Table 4.2). The obtained speedup is broken into two components: benefits obtained from a synchronous *Razor-like* mechanism and atop that, additional benefit obtained from our *Proposal*. The *Razor-like* mechanism is reflective of state-of-art timing speculation where there is no slack accumulation across asynchronous sequences. The *Proposal* components adds additional speedup due to slack accumulation. On average, the total speedups obtained are 18.4% and 10.6% under *nominal* and *high* variations respectively. Within this, *Proposal* provides 32% (*nominal*) and 76% (*high*) higher speedup respectively, over the *Razor-like* implementation (grey vs red portions).

Average speedup is higher under *nominal* random variation (in comparison to *high*) since there is more estimated slack available at the 99.99% confidence requirement. On the other hand, the portion of speedup obtained from *Proposal* is greater under *high* random variation in comparison to *nominal*. This follows from the high confidence requirement,

which prevents the *Razor-like* implementation from capturing slack due to slow paths in the slack distribution tail. But the averaging effect of *Proposal*-based speculation moves the slack estimate higher by cushioning latencies of critical instructions with the latencies of more probably non-critical ones.

Design Overheads: Area and energy overhead is calculated by implementing the entire design in RTL and synthesizing with Synopsys Design Compiler at 45nm node. Each PE is provided with slack tracking logic with LUTs and error detection/recovery. The LUTs are designed with 2 read ports and 1 write port, and contain 16 entries, each 3-bit wide. The area overhead of slack tracking logic (including LUTs and error logic) is 0.95% atop CRIB.

Energy overhead is 1.12% relative to FU computation energy. This includes 2 LUT reads on each computation, slack accumulation, and error checking logic. LUT writes and error recovery occur roughly once every 10K ops and only marginally add to energy overheads.

6.5 Chapter Summary

In this work, we proposed a design for aggressive slack recycling by using transparent pipelines. Grouping operations together into an "asynchronous" multi-cycle execution sequence allows timing speculation to cater to the average slack across the group rather than the worst-case individual. This allows more aggressive timing speculation in comparison to completely synchronous mechanisms.

We designed a slack accumulation mechanism and appropriate latch control for early clocking, to achieve slack recycling. Estimated slack is modeled mathematically, with dependence on PVT/data variations along with the height of the multi-cycle DFG, built

atop a self-correcting feedback mechanism. The proposal is evaluated on CRIB and shows significant performance speedup under different variation and design constraints.

7 RECYCLING DATA SLACK IN OUT-OF-ORDER CORES

In this chapter, we explore the REDSOC proposal. REDSOC extends the fundamental slack recycling proposal from SlackTrap (Chapter 6) to the out-of-order core. This involves novel contributions to the out-of-order cores scheduling scheme - providing it with the ability to wake up instructions early, track and accumulate slack and prioritize the appropriate instructions. Further, it targets accurate data slack recycling, with unique data slack identified for each instruction / computation based on its operation and operand properties. REDSOC's per-instruction slack recycling was motivated by the fact that the amount of slack can vary drastically between instructions, but can be identified from instruction characteristics, often early in the processing pipeline.

First, in Section 7.1 we present the design for estimating data slack at the decode stage (whenever possible) based on instruction opcodes, types and operands. Next, in Section 7.2 we discuss how transparent dataflow is implemented in the data bypass network. In Section 7.3, we discuss the optimizations to the scheduler which allows for slack recycling in out-of-order cores. Overheads of design are discussed in the above sections as well. Finally, in Section 7.4 we evaluate REDSOC's performance benefits and compare against prior work. Section 7.5 summarizes the proposal.

An introduction to REDSOC was provided in Section 2.3, background / motivation in Chapter 3 and methodology in Section 4.3.

REDSOC's contributions are summarized below:

① The REDSOC proposal redesigns a traditional out-of-order core to support slack

recycling.

② REDSOC classifies execution operations into different slack buckets based on the opcode and input precision.

③ REDSOC employs "transparent flip flops" and slack-aware control between execution units, to allow slack recycling across multiple operations.

④ Further, REDSOC proposes Slack-Aware instruction scheduling via Eager Grandparent Wakeup and Skewed Selection, which allows for efficient slack recycling.

⑤ The REDSOC implementation achieves average speedups in the range of 5% to 25% across the different cores and application categories. Further, it is shown to be more efficient at improving performance in comparison to prior proposals.

7.1 Design for Slack Estimation

Background on data slack and its different components is provided in Section 3.3.1.

Slack Look-Up Table: Static circuit-level timing analysis at design time can measure computation times (i.e. Clock Period - Slack) for different classes of operations. These values are then stored in a slack look up table (LUT). We only break down the computation times into coarse blocks: a) based on operations being arithmetic vs logic, b) based on having a shift component and c) based on 4 different data-widths/types. The 5-bit address to perform a LUT lookup is shown in Fig.7.1. The Arith/Logic and Shift bits are *don't cares* for sub-word parallel SIMD instructions. The Width/Type bits use predicted data-width for normal instructions and data-type for SIMD. There are a total of 14 possible slack categories/buckets arising from the above. Operations are simplified classified into one of

the slack buckets. Details on complexity of above analysis is discussed in Sec.4.3.

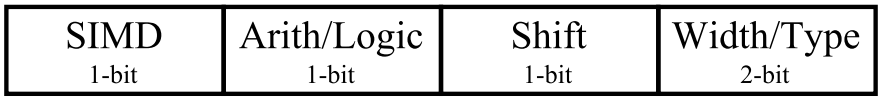


Figure 7.1: 5-bit slack lookup

Data-Width Predictor: Both *opcode slack* and *type slack* can be found out as early as the decode stage in the processor pipeline since the opcode and data type (for SIMD) are encoded with the instruction. *Width slack* (via data-width), on the other hand, is often not available until the execution stage itself. This is because register values or data bypass values are often not available until just prior to execution. For prior work on partial power gating of functional units or combining multiple operations into a single execution on the functional unit, it is sufficient to identify data-width at the time of execution. But in our work, the data-width/operand-slack information is required in the scheduling stage (more on this in Sec.7.3.3). We therefore use a data-width predictor as proposed by Loh [139] and also used by others for optimizations such as Register packing [53].

We utilize a resetting counter based predictor as proposed by Loh [139]. The predictor is addressed by the instruction PC and two pieces of information for each instruction - the most recent data-width of the instruction and a k-bit confidence counter that indicates how many consecutive times the stored data-width has been repeated. On a lookup, if the confidence counter is less than the maximum value of $2^k - 1$, then the predictor makes a conservative prediction that the instruction is of maximum size. Otherwise, the prediction is made according to the stored value of the most recent data-width. If there was a data-width misprediction, the data-width field is updated and the counter is reset to zero. On a match, the counter is incremented, saturating at the maximum value of $2^k - 1$. We use 4

possible prediction outputs indicating high to low data-width. Note: in REDSOC, operand widths are based on higher-order bits being 0s. Negative operands of low precision (with higher-orders bits being 1) thus do not benefit from the above.

Inaccuracy in prediction is detected at execute stage when the operands are available for execution by simply checking the higher order bits. Incorrect predictions are of two kinds - aggressive and conservative. Conservative incorrect predictions result in lost opportunity to recycle data-width slack but do not result in functional errors. Aggressive incorrect predictions would result in correctness violations if allowed and therefore such instructions need to be conservatively re-executed. Recovery is performed similar to cache miss replays via selective reissue of instructions.

Overheads/Accuracy: Prior analyses [139] have shown that a resetting predictor allows aggressive errors in the range of only 0.1-0.6%. We use a 4K-entry prediction table which results in an aggressive misprediction of around 0.3-0.4%. Such a predictor requires a total state of 1.5KB. In comparison, current day branch predictors use prediction tables with as much as 64KB of state [196].

Considering the very small sizes of the LUT and predictor (in comparison to, say, register file and branch predictor) their overheads in terms of area and access energy are only 0.52% and 0.5% of the OOO core.

7.2 Recycling Slack via Transparent Flip-flops

The previous section highlighted the prevalence of considerable data slack in executing operations. In order to execute consumer operations immediately after the producer com-

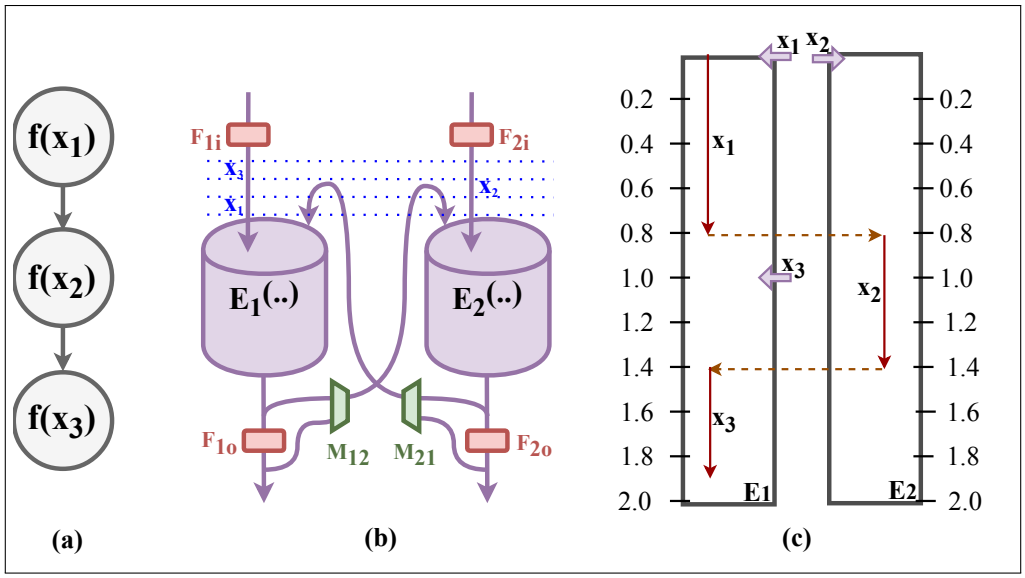


Figure 7.2: Data Slack Recycling

pletes (i.e. to recycle this data slack), we make use of transparent dataflow via intelligent FF control. Note that we incorporate transparent datatflow only within data bypass network between execution units. Via this design, ALU operation sequences can execute "transparently". Other operations such as multi-cycle, FP and memory operations are still "true synchronous" operations and do not themselves benefit from transparent execution.

A transparent mode FF design is a simple implementation consisting of a standard FF but with a bypass path [86]. A mux at the end of the 2 paths can select the "opaque" stored FF value or the bypassed "transparent" value, based on an enable input. In our work, transparent mode is enabled in the bypass path between ALUs whenever data is required to flow through at non-clock boundaries. This allows varied delays across operations to be balanced anywhere within the transparent execution window. Note that such a design can also be implemented via latches [62], which is prevalent in Intel designs [231]. The primary benefits from transparent pipelines explored earlier in the research domain include reducing clocking power [104, 76] as well as interlocked synchronous pipelines which

reduce stall related overheads [105].

We propose a synchronous slack tracking and opportunistic early clocking mechanism implemented atop a transparent execution pipeline. Our proposed mechanism *reduces the execution latency at the cost of increased EU utilization*. We introduce the concept with a simple discussion on applying transparent dataflow to a generic pair of execution units (E_i) as shown in Fig.7.2.b. We assume that the units have forwarding paths to each other (shown in figure) and back to themselves (not shown). Also, forwarding logic is simplified to only show forwarding to a particular input of the execution units (i.e. right input of E_1 and left of E_2) but actual design would support both inputs of each, and would extend to more execution units as well. Moreover, we focus on single-cycle combinational execution. These units could be thought of as the ALUs in standard OOO processors.

Consider the data flow graph Fig.7.2.a. It shows a sequence of 3 dependent operations and need to be executed in sequence. The functional flow atop a pair of execution units (EUs) is depicted in Fig.7.2.b, wherein the stream of inputs x_i are distributed in sequence over the two E_i. In conventional design, this system is entirely executing at a throughput of 1 operation per cycle i.e. not executing at peak throughput, and consumes 3 clock cycles to complete. Note that the operations could have any other distribution across the 2 EUs, but the throughput is always limited to 1 operation per cycle. In other words, in each cycle one execution unit is always idle in this system.

Assume the presence of data slack for each $f(x_i)$, i.e. for each operation's computation on an execution unit. In standard synchronous design, EUs are lodged between opaque FFs and inputs and outputs pass through only at clock edges, causing this slack to be wasted. This is indicated in the figure by F_{1i} and F_{1o} bounding E_1 and similarly for E_2. Our

proposed mechanism cuts out this slack by introducing "transparent FF" based data bypass between the execution units. The ability to bypass the data around the output FF is also shown in Fig.7.2.b. Mux M_{12} can enable bypassing of F_{1o} when forwarding E_1's output to E_2. Similar bypassed dataflow is possible from E_2 to E_1.

Our proposal performs 3 distinct intelligent tasks (ITs):

- IT_1: For a producer operation with slack, a consumer operation is brought early to an idling EU (if available).
- IT_2: The FFs are made transparent (via mux-control) in the bypass paths between the EUs holding the producer and consumer operations respectively, for the period of time that the producer is available at its functional unit.
- IT_3: An operation is held to a EU for two cycles or one cycle depending on whether its execution (via the above mechanism) might cross a clock boundary or not, respectively.

Fig.7.2.c describes this functional flow over the 2 EUs, in more detail, via an example. Consider that the three operations (x_i) described earlier, can execute on the EUs with latencies of 0.8ns, 0.6ns, 0.5ns respectively. The red solid arrow indicates estimated execution time and the yellow arrows show dependencies.

① At t=0ns, x_1 is brought to the input of E_1. This begins computation and would complete at t=0.8ns. ② In parallel with x_1, x_2 is brought to input of E_2 (an IT_1). $f(x_2)$ isn't ready for computation yet, since x_1 is yet to complete on E_1 but is brought in early so that $f(x_1)$'s slack can be completely utilized. ③ Also at t=0ns, the transparent bypass path from

E_1 to E_2 is selected via mux M_{12} as it is estimated that $f(x_1)$ completes in this cycle (IT_2). The value passes through and stabilizes to the correct $f(x_1)$ value at t=0.8ns. Further, $f(x_2)$ starts correct computation at t=0.8ns and finishes at t=1.4ns. ④ Note that x_1 is held at E_1 for one cycle while x_2 is held at E_2 for 2 cycles. This is because computation time estimates indicated that x_1's execution does not cross a clock boundary while x_2's does (IT_3). ⑤ At t=1.0ns, x_3 is brought early to E_1 (IT_1) and bypass path $E_2 - E_1$ is made transparent while bypass path $E_1 - E_2$ is made opaque (IT_2). Further x_3 will hold the unit only for 1 cycle since it computes correct data from t=1.4ns to t=1.9ns (IT_3). ⑥ A true-synchronous operation after x_3 (eg. Store instruction) can clock at t=2.0ns. Some slack is lost but the computation is still 1 cycle faster than the pure synchronous baseline which took 3 cycles.

Summary: It is important to understand that this mechanism *does not require per-operation slack to be so significant that multiple operations can execute within a single cycle*. It only requires *one or more cycles worth of slack to accumulate over an entire sequence of operations*. This translates to higher performance and better energy efficiency via those accelerated sequences that lie on the critical path of program execution.

7.3 Slack-Aware OOO Scheduling

The transparent dataflow of dependent operations between functional units can be used to recycle data slack *IF* a slack aware scheduling mechanism is in place. The scheduler is responsible for issuing instructions to execution units, based on some priority scheme, when all required resources (source operands and execution units) are available. Our slack-aware optimization focuses on three components of the scheduler: the wakeup logic,

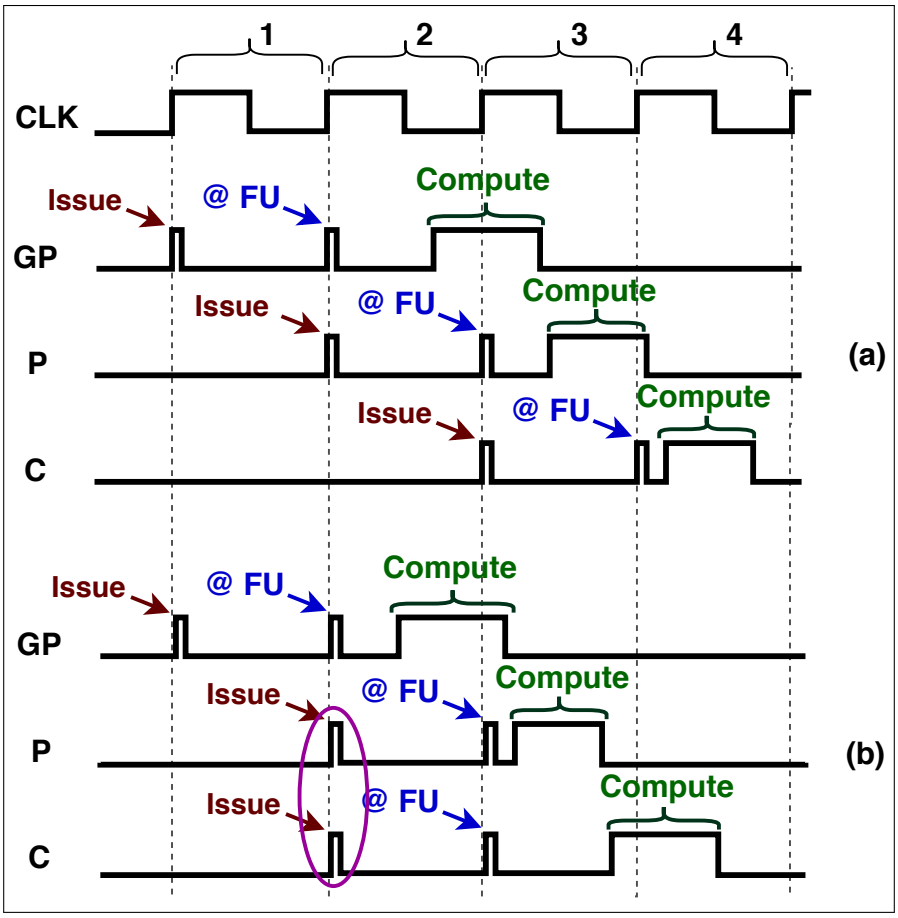


Figure 7.3: Timing Diagram of Execution Pipeline

the issue buffer / reservation stations and the select logic. An overview of the out-of-order core and its scheduling logic is provided in Section. 3.1.3.

Note that our slack aware scheduling mechanism is *focused only on single-cycle operations*. We do not attempt to recycle slack in multi-cycle operations, which reduces some potential overheads which are beyond the scope of this proposal.

7.3.1 Motivation

Implementing slack-aware scheduling in OOO processors requires some challenges to be addressed. In state-of-art deeply pipelined processors, the instruction scheduler is

decoupled from actual execution. Using a fixed latency assumption for each instruction, appropriate dependents are scheduled to wake up and pickup their operands off the bypass at the correct time. Accounting for data slack means that the scheduling logic has to be made aware of the potential early completion of operations within their clock cycle. This requires augmenting the scheduler with data-slack information. Moreover, when a producer operation is expected to produce slack, the scheduler needs to schedule a consumer operation early enough (onto an idle functional unit), so that the consumer can begin evaluating immediately after the producer's completion.

An illustration of how the timing of instruction issue is integral to recycle slack via transparent dataflow is shown in Fig.7.3. The figures show 3 instructions (named GP: grandparent, P: parent, C: child) being executed in a processor pipeline. This simple illustration shows the pipeline issuing instructions one cycle before they arrive at the functional unit and become available for compute to begin. (Note that this is not a design assumption and is only for illustrative purpose.) In Fig.7.3.a, GP is issued at the beginning of cycle 1, and becomes available for execution on an FU at the beginning of cycle 2. Assuming it is made to wait for some previous producer operation (aka great grandparent) to complete in cycle 2, (as described earlier), it then begins evaluating immediately within cycle 2, and completes at some instant in cycle 3. Even via conventional single-cycle tag broadcast, GP's tags can be broadcast in cycle 1 and can wake up instruction P to issue (if selected) at the beginning of cycle 2. P then becomes available at the FU at the start of cycle 3, and begins evaluating after GP is complete and then evaluates into cycle 4. Similarly, C is woken up and selected at the beginning of cycle 3 and is prepared for execution. In this example, operations GP, P and C only need to be issued on consecutive cycles as enabled

by conventional scheduling logic.

On the other hand, a different scenario is shown in Fig.7.3.b. While GP and P are issued as was discussed in the first scenario, a difference arises here because P finishes evaluating within cycle 3 (due to high data slack). To recycle P's slack, C needs to begin evaluating in cycle 3 as well, so it needs to arrive at its FU (note: a different FU from the one P is computing on) at the beginning of cycle 3. To achieve this it needs to issue at the beginning of cycle 2, i.e. at the same time as its parent, P. This scenario is not possible with existing scheduling logic as the scheduling loop requires one cycle; this motivates our modifications to the scheduler, which are discussed below.

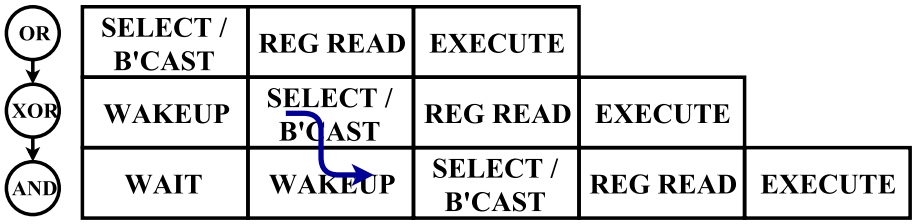
1. **Eager Grandparent Wakeup**: speculative wakeup based on grandparent operations (a modified design based on [205]) so that child operations can be issued in parallel with parent operations.

2. **Slack Tracking**: Calculating and tracking an operation's completion time based on execution times and producers' completion times.

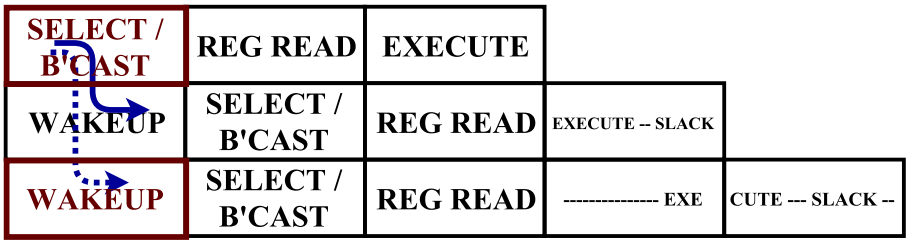
3. **Skewed Selection**: Select logic which prioritizes non-speculative operations over speculative (grandparent-awoken) operations.

7.3.2 Eager Grandparent Wakeup

Grandparent wakeup (GPW) is a speculative wakeup technique used to wake up a child operation based on the broadcasted tags of its grandparent operations [205]. In the original proposal by Stark et al. [205], GPW is used to prevent pipeline bubbles when the scheduling



(a) Conventional Wakeup



(b) Eager Grandparent Wakeup

Figure 7.4: 1-cycle scheduling loop

loop (wakeup-select-broadcast) is pipelined. The goal behind their work was that as pipeline stages and clock frequency grow, it is imperative to break down the timing critical scheduling loop into multiple stages. Pipelining this scheduling loop naively would result in inefficiency: not being able to execute dependent operations in consecutive cycles. But if tags of the grandparent(s) are used to wake up the child instruction, this inefficiency can be avoided. The idea was motivated by the notion that if the grandparents of a child instruction have been selected for execution, then it is likely that the parent will be selected in the following cycle (considering single-cycle operations). The child can then be executed in the cycle following its parent. More details can be found in the original proposal.

Clock frequency and pipeline depth have stabilized in the last decade, so current day schedulers can support single cycle scheduling without the use of grandparent wakeup. The conventional pipeline schedule for a 3-operation dependency graph is shown in Fig.7.4.a. The single cycle scheduling loop is performed in cycle one for waking up the XOR operation based on the OR operation. Similarly, in cycle two, XOR broadcasts and wakes up the AND.

However, the need for eager scheduling to recycle data slack (as motivated in Fig.7.3) creates a need for a grandparent scheduling-like mechanism. We modify GPW to create Eager GPW (EGPW), to wake up the child operation in the same cycle as the parent operation. While this is unnecessary in standard pipelines, it is useful for slack recycling: consumer operations can be sent to idle functional units early (in the same cycle that the producer operation completes) so that the slack from the parent operation is recycled. For the same DDG, assume that the XOR operation has data slack which can be exploited by the AND if it can start execution on the same cycle as the XOR. The corresponding pipeline schedule via EGPW is shown in Fig.7.4.b. The OR instruction wakes up its child (XOR) and its grandchild (AND) in the same cycle. XOR wakeup is conventional, while AND wakeup is achieved by the speculative EGPW mechanism. This allows the AND instruction to arrive in parallel with the XOR at a functional unit and wait for the XOR output to transparently flow through. It then begins useful computation in the same cycle and effectively recycles the XOR operation's slack. As also seen in figure, if the AND reads a second operand from the register file, this also happens early (in parallel with the XOR) based on conventional RF port availability.

In the original implementation [205], GP-mispeculation can potentially occur when the grandchild instruction is woken up with the grandparents tags, but the parent does not get selected. This is verified by checking for the eventual broadcast of parent tags. They show that these mispeculation rates are very low when sufficient functional units are available. Our skewed selection mechanism deprioritizes GP-wakeups and can largely (or even completely) eliminate GP-mispeculation. This is discussed further in Sec.7.3.4.

Note that EGPW only wakes up the grandchild instruction. The conditions for this

grandchild to be selected for issue are explained in the following two sections on Slack Tracking and Skewed Selection.

7.3.3 Slack Tracking

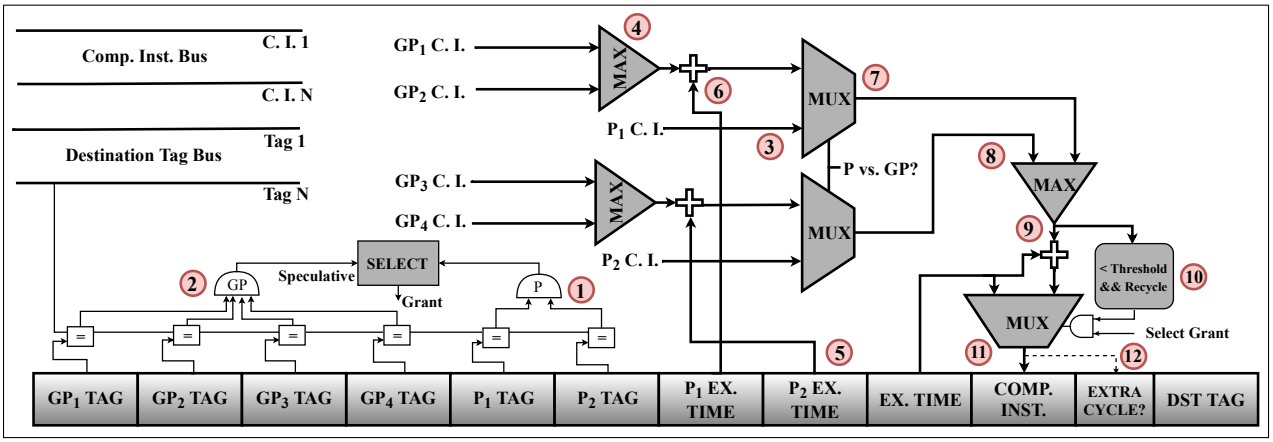


Figure 7.5: Illustrative design for Slack aware RSE (Steps 3-10 occur in parallel with selection)

We assume a reservation station based model for scheduling, as described in Section 3.1.3. After instructions are renamed, they wait in reservation stations for their sources to become ready. In a conventional design, each reservation station entry (RSE) has 2 parent (or source) tags which are identifiers for the source operands. Once the tag matches occur, the instruction is woken up. A request is placed to the Select logic and if selected, it receives a grant. If selected, its destination tag is then broadcast on the tag bus. A more detailed description can be found in prior works [205]. Our goal is to augment this baseline design with slack-awareness.

The following discussion will put forth two designs for slack-aware scheduling. The first is *Illustrative*: its discussion aids in explaining our technique in a step-by-step man-

ner. The second is *Operational*: it is the actual design we employ, suitable for practical implementation.

Illustrative Design for Slack-Aware Scheduling: Our proposed augmented RSE entry is shown in Fig.7.5. In the RSE, slack is tracked with a 3-bit fractional representation i.e. slack precision of 1/8th of the clock period (details in Section 4.3). The timestamp within a clock cycle at which an instruction completes is its 3-bit Completion Instant (CI). The CI is calculated for a given instruction based on its parent/grand-parent CIs and slack information, and is written into the COMP-INST field of the RSE. This is explained as part of the mechanism below.

① Conventional parent tags (P_1, P_2) which are identifiers for source operands, are shown. If both tags comparisons hit (i.e. a match with tags broadcast on the destination bus), the instruction is awoken and a request is sent to the select logic for selection. ② Similar to the grandparent scheduler design by Stark et al. [205], we add grandparent tags (GP_1 - GP_4) to enable grandparent based instruction wakeup. If all tags hit, a *speculative* request is sent to the select logic. Differentiating between a normal select and a speculative select by the *skewed* select logic will be explained in Sec.7.3.4. ③ In case of a parent based wakeup, the estimated CI of the parents are used to determine the starting instant of the child (or current) instruction. P_iC.I. are the CIs, which are broadcast along with the tags (obtained from the CI bus). ④ Similarly, in case of a grandparent based wakeup, estimated CI of the grandparents are obtained off the CI bus. The Max logic estimates the later CI (i.e. last completing) from each set of grandparents. ⑤ The 3 EX-TIME fields in RSE indicate the estimated execution time for this particular instruction and its parents respectively, each of which is a 3-bit value. These values are calculated at decode (read out of the

slack LUT: described in Sec.7.1) and are written into the RSE and the Register Alias Table (RAT). The child instruction obtains the EX-TIMEs for the parents from the RAT during register renaming. (6) In case of Parent-based wakeup, the current instruction can start executing immediately after the last completing parent. In case of GP-based wakeup, the execution time of the Parent instructions should be accounted for. For GP-based wakeup, each parent's EX-Time is added to the latest CI of the corresponding grandparent set, thus producing the parent CI. (7) Based on the instruction wakeup being parent-based or GP-based, the appropriate CI is selected (via a mux) for the two source operands to the current instruction. (8) Among the 2 parent CIs, the later one is selected via the MAX operation (as the child would start executing after this). (9) The completion instant of the child is then calculated by adding its EX-TIME to the last completing parent's CI. (10) A child operation would issue in the current cycle only if a) slack recycling is enabled, b) the completion instant of the last parent is expected within the current cycle (like operation P in Fig.7.3.b) and further, is within some *slack threshold* (discussed later) and c) a grant is obtained from the select logic. (11) The appropriate CI is written into the current instruction's CI field, and then broadcast along with the tag. This could either be the CI, as calculated in (9), or, in a scenario where slack recycling does not happen and the current instruction is executed from a clock cycle boundary (of a later cycle), the value written in would be the operation's EX-TIME itself. (12) Further, if the execution of the operation is expected to cross a clock boundary (such as GP and C, but not P, in Fig.7.3.b), the execution unit is allocated for an extra cycle (i.e. 2 cycles for traditional single cycle operations).

The slack threshold discussed in (10) is used to achieve a balance between potential benefit from slack recycling vs. potentially excessive FU utilization caused by the 2-cycle

allocation requirement. A higher threshold would recycle slack more aggressively, starting consumer operations in the producer's completion cycle even when if there is very low slack in that cycle. The potential benefit then depends on enough small slack increments accumulate to cross a clock cycle boundary, reducing exposed latency in the dataflow graph. The potential detriment is that the FU underutilization caused by 2-cycle allocation might cause slowdowns under high FU demand. Ideally, a simple but intelligent dynamic mechanism can be used to increase or decrease this threshold based on overall observed benefits. In this initial work, we tuned this value via a design sweep for each set of applications (refer Sec.7.4.3).

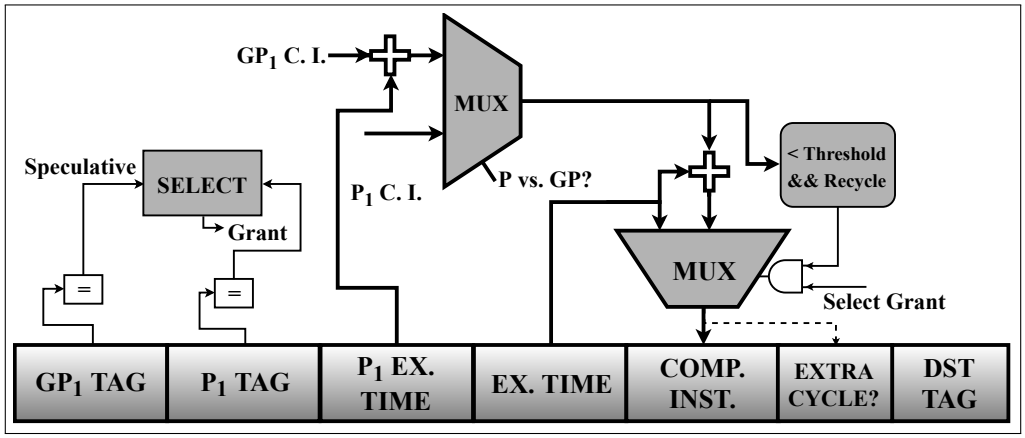


Figure 7.6: Operational design for Slack Aware RSE

Operational Design Slack-Aware Scheduling: From the physical design perspective, increasing the number of tags in the RSE is quite expensive because all wakeup buses should be connected to all source tag comparators in all entries. This can significantly increase the load capacitance on the bus and the wakeup logic drivers [118].

In order to reduce potentially detrimental energy/area overheads from the Illustrative design and for the implementation to be practical, we propose an Operational design which

closely matches (within 1%) the former's performance. It is based on two key observations: 1) a significant fraction of arithmetic computations have only a single source operand [118], and 2) even within the fraction of operations with multiple source operands, the last arriving source operand (tag) is predictable with high accuracy [54].

Based on the above observations, we predict the last arriving parent for each single-cycle arithmetic operation. Further, this information is passed from a parent to its child operation during rename (via the RAT), meaning that a child operation uses a prediction for both its last arriving parent and its last arriving grandparent.

This last-arrival prediction mechanism tremendously reduces design complexity and is shown in Fig.7.6. Only 2 tags are now required in each RSE, one each for the last arriving parent and grandparent respectively. The RSE will require only 2 EX-TIME fields – one being its own execution time, and the other being that of the last arriving parent. Their usage was described in the earlier Illustrative design. Moreover, the slack calculation logic gets significantly simplified, as there is no requirement to compare and estimate the last arriving source operands.

The prediction of the last-arriving tags must be validated to ensure that the instruction did not execute before all of its operands are available. The prediction is correct if the operand predicted to be not arriving last is already available when the instruction enters the register read stage of the pipeline. We utilize a small register scoreboard mechanism from prior work [54] to achieve the same. If the prediction is incorrect, error recovery is required, in a fashion identical to latency mispredictions (but with lower penalty). Considering the almost perfect prediction accuracy (Sec.7.4.2), the performance impact is nearly zero.

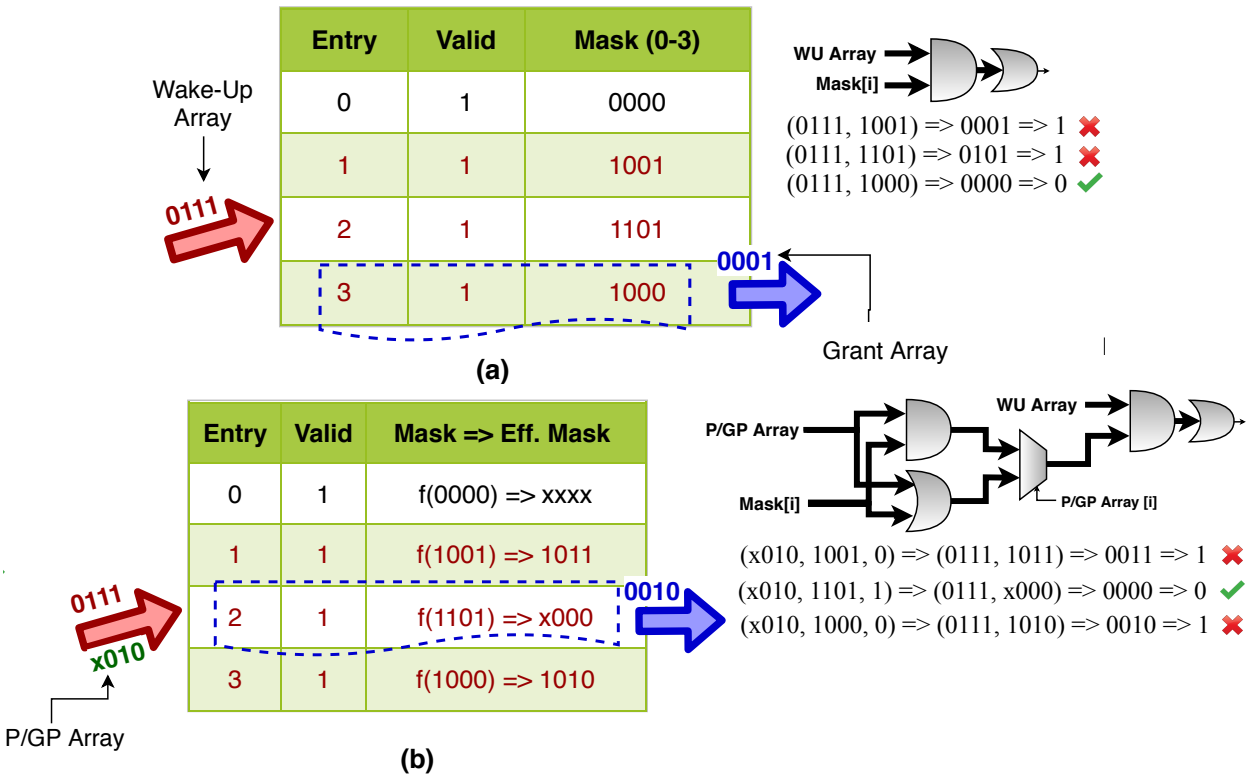


Figure 7.7: Skewed Select Logic (Note: gate-level design is illustrative)

7.3.4 Skewing the Select Arbiter

We *skew* the selection logic to prioritize conventional requests over speculative GP-requests. Only if there are remaining FUs after allotment to conventional requests will they be allotted to GP-requests. Thus, no conventional request suffers from not being serviced due to *other* selections. The mechanism to skew the selection logic is shown in Fig.7.7.

Conventional Selection: Fig.7.7.a shows conventional N:1 selection logic (representative of standard processors) implementing an oldest first priority mechanism. Valid entries are filled up into the selection table in parallel with the reservation station. For an N-entry table, an N-bit mask is used to indicate the priority order. In any entry's mask, a 1 at the i_{th} bit from the left indicates that the i_{th} entry is older (i.e. has higher priority). For example,

the 0_{th} entry has highest priority (mask is all 0s) while 1_{st} entry has a lower priority than entries 0 and 3. Ready instructions send requests to the select arbiter - indicated via the wake-up array. Here, instructions corresponding to entries 1, 2 and 3 have woken up and are requesting grants. The circuit shown adjacent to the table, decides which entry gets the grant, i.e. which among the woken up entries has the highest priority. The 3rd (producing a 0 through the circuit) is found to be the highest priority awake entry and is given the grant. In the figure, the grant array represents the output from the selection logic indicating which instruction gets the grant.

Skewed Selection: Fig.7.7.b shows our proposed skewed selection logic. Skewed selection prioritizes non-speculative requests (from parent based wake-ups) over speculative requests (from GP based wake-ups) while respecting the original priority scheme among each group of requests. The P/GP array is an additional input to the selection logic, which indicates which requests are speculative (GP) and which are non-speculative (P). P requests are shown as a 1 and GP requests show up as a 0 in our design. Again considering the same example, entries 1-3 are woken up. Further, the example assumes entry 2 wakes up non-speculatively while entries 1, 3 wake up speculatively. Since entry 2 is the non-speculative request, it has priority over the other 2 speculative requests. This is implemented by calculating the "effective mask". The circuit implementation is shown adjacent to the table. In the example, entry 1 has its mask altered from 1001 to 1011, since the 3rd entry is a non-speculative wakeup. Similar alteration occurs for entry 4. Conversely, entry 2 has its mask altered from 1101 to x000 i.e. bits corresponding to speculative entries are made 0. The 'x' indicates a don't care since entry 0 is not woken up. After this, the selection circuit from earlier calculates the appropriate entry for selection, which is the

2nd entry in this scenario. It should be noted that the skewed logic is laid out as a simple (but inefficient) sequence of gates, simply for illustrative purposes. The actual (negligible) increase in delay is discussed in Sec.7.3.5.

Discussion: There are two key reasons to skew the selection policy of the select arbiter. They are motivated below.

The first motivation is to improve FU utilization. Previously, we had discussed how speculative GP-wakeups and non-speculative conventional parent-based wakeups both send requests to select logic to obtain grants. We also discussed that the grandparent based early wakeup is useful only when the child instruction needs to be issued in the same cycle as the parent (i.e. when there is slack beyond the completion instant of the parent, refer Sec.7.3.1). This means that any grants provided to the grandparent-based wakeup are unutilized if there is no slack to recycle in that particular cycle. This is indicated by ANDing the select grant with the recycling decision in figure 7.5 and 7.6. In such a scenario, execution units go under-utilized in those cycles when the select logic selects a GP-wakeup request instead of a conventional parent-wakeup and there is no slack to recycle.

The second motivation is to prevent (or reduce) mispeculation from grandparent scheduling. GP-mispeculation occurs when a child operation woken up by a grandparent is selected for issue without the parent also being selected (Sec.7.3.2). Skewed selection prioritizes conventional (parent-wakeup based) requests over speculative GP-wakeup based requests. This means that within an arbitration window, a GP-wakeup can never race ahead of a conventional wakeup. Therefore, a child would never be selected for execution ahead of its parent as long as they are a part of the same select arbitration window.

The arbitration window depends on the design of the select logic. Assume that the

processor selects M instructions for execution (on M units) from N requests. This can be implemented as a) a global arbitration window performing N:M selection [158] or b) M/K local arbitration windows, each performing N:K selection. In the first scenario, there would be no GP-mispeculation thanks to the skewed selection logic. In the second scenario there would be no GP-mispeculation within each window but there could be GP-mispeculation across windows. In this work we assume global arbitration.

7.3.5 Summary of Overheads

It is key to note that the entire slack aware mechanism described above happens in parallel with select logic. Select requests are issued at wakeup oblivious to slack, and select grants are returned at the end of the cycle. The instruction's execution is then finally determined by the grant as well as the slack/CI calculation described above. Moreover, the slack-aware computations are only 3 bits wide, resulting in the critical path of slack-computation being significantly shorter than select logic arbitration. Thus this primary design component of slack based scheduling does not increase the critical paths in scheduling logic.

Area/power overhead of the proposed Operational design is negligible, the main additions only being 10 extra bits per RSE, two 3-bit adders (with overflow) and muxes, and a comparator, contributing to an area overhead of 0.3% and an energy overhead of 0.8%. Note: the adder overflow determines if the computation's execution crosses a clock boundary, the use of which was explained in Sec.7.3.3.

Synthesis of the skewed selection logic shows that the additional delay in select logic amounts to only 3 ps additional delay over the baseline 100 ps select logic. Further, consid-

ering the significant wire delay that exists in the select arbitration tree [158], this increase in delay would be reflected negligibly in real design.

The marginal increase in critical path delay via skewed selection and the absence of any additional critical timing component in the slack tracking mechanism described earlier means that there is hardly any change to the timing of the scheduling loop (which can be a near timing critical, sometimes dominated by load-store unit and the fetch loop [214]).

7.4 Evaluation

A discussion on the experimental methodology is provided in Section 4.3.

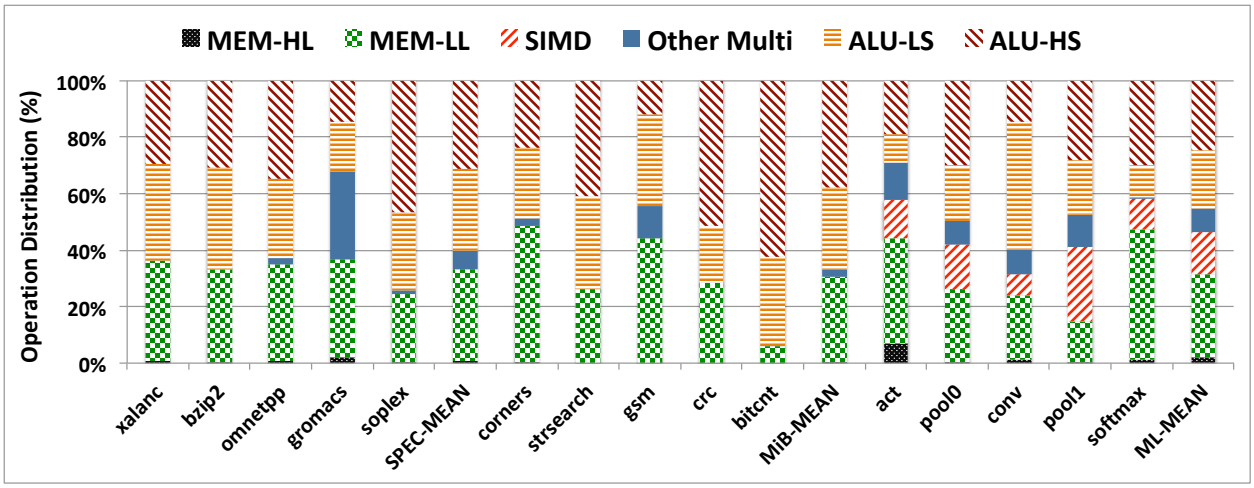


Figure 7.8: Benchmark Operation Characteristics

The benchmarks and their operation characteristics are shown in Fig.7.8. The characteristics shown are: memory operations with high/low latency (MEM-HL/MEM-LL; HL refers to L1 cache misses), NEON SIMD operations, other multi-cycle operations (eg. fp) and high/low slack single-cycle ALU operations (ALU-HS/ALU-LS, HS refers to data slack greater than 20% of the clock cycle). While many SIMD operations are pipelined and

multi-cycle, accumulate, multiply-accumulate etc. support late-forwarding of accumulate operands from similar ops, allowing sequential single-cycle execution [102].

MiBench benchmarks contain a high percentage of ALU-HS (high slack) and a low percentage of memory operations, allowing them to get significant benefits from REDSOC. On the other hand, SPEC benchmarks are more memory-intensive and their compute operations are predominantly ALU-LS, resulting in fewer opportunities for slack recycling. ML kernels have portions of low precision SIMD operations, though some of the kernels (eg. ACT) have high MEM-HL fractions, resulting in less significance of slack recycling, even if large opportunities exist. In general, lower the % of memory operations (i.e. compute intensive), more the opportunities for speedup from REDSOC.

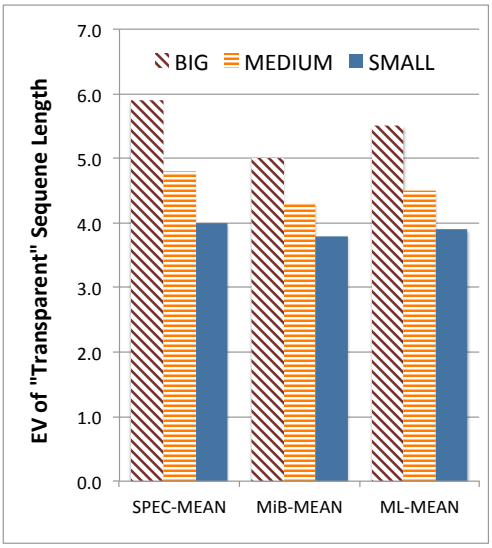


Figure 7.9: Seq. Length

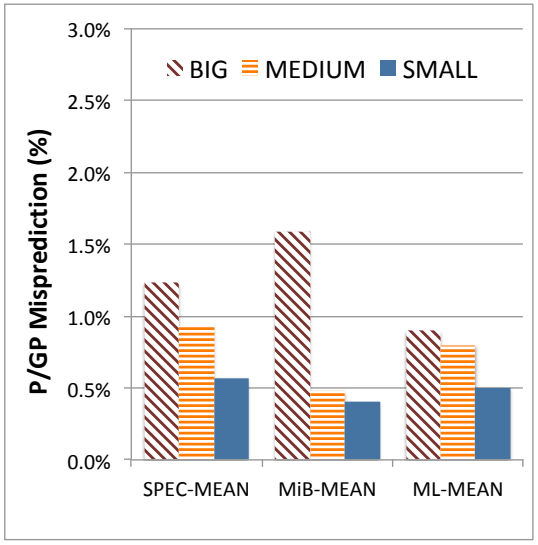


Figure 7.10: Tag Prediction

7.4.1 Potential for Sequence Acceleration

As discussed in Sec.7.2, slack accumulates over a sequence of operations which can be executed in a transparent manner. Fig.7.9 shows the expected value (i.e. weighted mean)

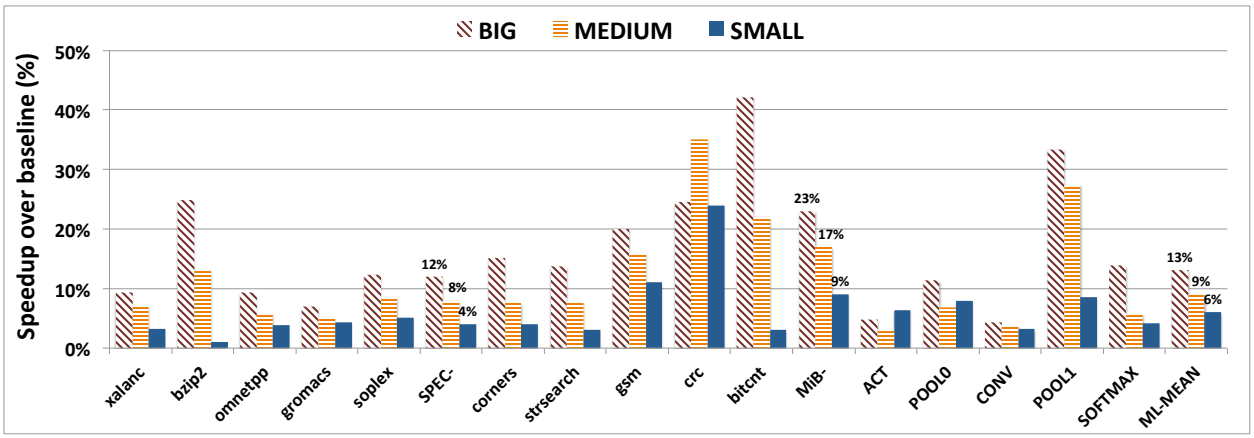


Figure 7.11: Speedup for different cores

of the length of all such sequences. The values shown are averaged across each benchmark class and evaluated for each core type. It can be seen that the observed average length of these transparent sequences is between 4-6 operations. Slack per operation can usually vary between 10%-60% to of the clock-cycle. Thus, the average length of these transparent sequences is sufficient to accumulate one or more cycles of slack, resulting in early clocking of sequence-ending "true synchronous" operations, providing speedup.

7.4.2 Last parent / grand parent prediction

Fig.7.10 analyzes accuracy of tag prediction in the *Operational* design, using a prediction table with 1K entries. The table is addressed by PC-bits and uses 1 bit for prediction per entry to indicate if the particular operand is last to arrive. High accuracy keeps mispredictions to around 1%. Accuracy is lower for larger cores due to higher scheduling traffic.

7.4.3 Performance Speedup

Fig.7.11 shows the speedup obtained over a standard baseline without slack conservation for different core sizes.

The first observation is that speedups are lower for SPEC benchmarks compared to MiBench. This is partially due to SPEC having a significant percentage of high dependence memory operations. Moreover, the average percentage of high slack ALU operations in SPEC is only around 30% while it is close to 60% in MiBench. MiBench applications, on the other hand, show significant speedups (23% average on the BIG cores). The *bitcount* application sees over 40% speedup over the baseline. This is not surprising, considering that benchmark characterization (Fig.7.8) shows that this application has less than 5% of memory operations and close to 60% of high slack single-cycle ALU operations.

Second, note that benefits generally increase with size of the core. A larger core provides more idle functional units for data to transparently flow into, which is a requirement for slack recycling. Further, the larger number of reservation stations in the big cores allow for more dependent waiting operations in the RS to be scheduled aggressively, allowing multiple dependency chains within the application to perform slack conservation. Fig.7.12 illustrates how the pipeline stalls from busy FUs increases from the baseline to REDSOC. For smaller cores, this has some limiting effect on the maximum speedup from slack recycling.

Finally, speedup on the ML kernels is from both low-precision NEON SIMD operations and reasonable fractions of high slack single-cycle operations. Due to their working sets, some of these kernels (e.g. ACT) spend a significant portion of time waiting for long-latency memory operations to complete, and this cuts down gains to some extent. Efficient

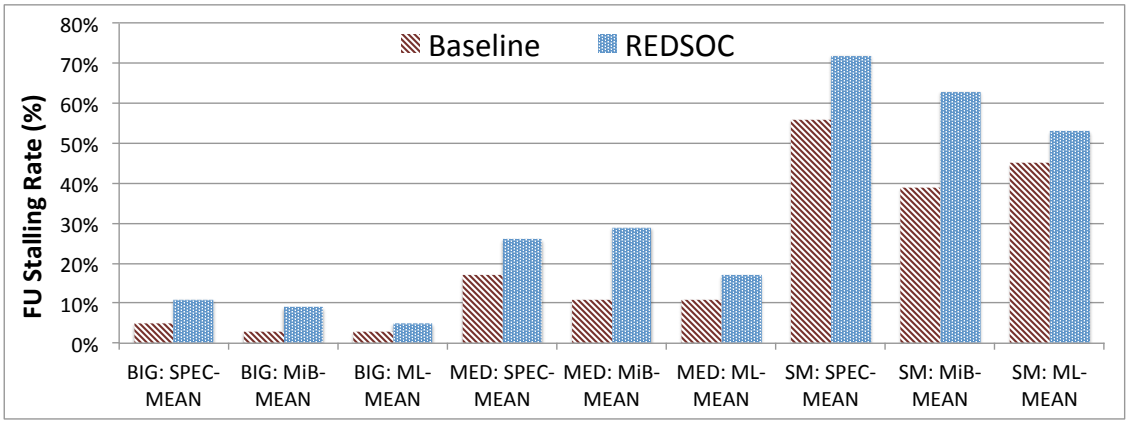


Figure 7.12: Pipeline stall rates from busy FUs

prefetcher tuning and blocking the matrices could increase slack opportunities, so these results might be pessimistic.

To estimate power efficiency at baseline performance, we convert speedup into power savings via application-level V/F scaling. Scaling is modeled on ARM A57 [65]. Mean power savings on chosen SPEC, MiBench and ML benchmarks range from 8-15%, 12-36% and 8-18% respectively.

7.4.4 Comparison with other proposals

We quantitatively compare REDSOC against our own implementations of timing speculation and operation fusion. **TS** is our timing speculation mechanism (similar to Razor) wherein frequency is controlled depending on the error rate in the application. Frequency is statically fixed so as to maintain an error rate between 1% and 0.01% across application execution. Note, we do not model recovery for timing errors; thus, the performance numbers shown for TS can be considered as optimistic. **MOS** is Multiple Operations in Single-cycle - i.e. the implemented operation fusion mechanism. The mechanism dynamically combines multiple operations within a single cycle, if they are capable of fitting within a single cycle.

For example, 2 consecutive logical operations (roughly 50-55% data slack) can executed in a single cycle.

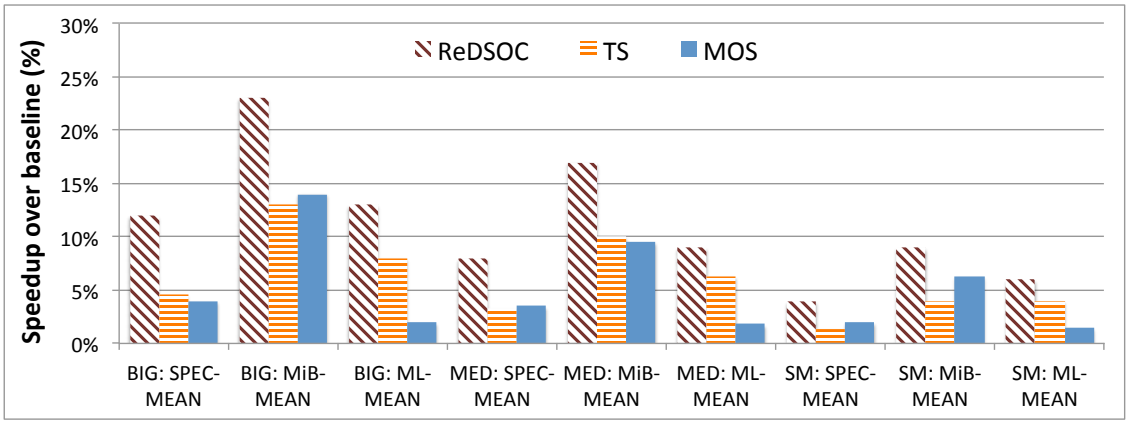


Figure 7.13: Comparison with other proposals

Comparison of these two mechanisms against REDSOC atop the three different core types are shown in Fig.7.13. It is clear that REDSOC significantly outperforms both mechanisms by 2x or more. *MOS* opportunity is limited in most applications, due to an inability to find many sequential operations to combine into a single cycle. It achieves reasonable speedup in MiBench due to higher data slack averages. *TS* is limited by the fact that frequency control can happen only at a coarse temporal granularity, while data-dependent slack varies from operation to operation. Hence, the *TS* setting has to be set rather conservatively to maintain low error rates.

7.5 Chapter Summary

This proposal showed that data slack can often be a significant portion of the clock period, and cutting out this slack provides tremendous opportunity to improve performance. With the increasing popularity of applications that utilize low-precision arithmetic, data slack is

becoming even more prevalent.

REDSOC recycles the data slack from a producer operation by starting the execution of dependent consumer operations at the exact instant of the producer's completion. Recycling over multiple operations executing on ALUs, allows acceleration of these data sequences and improves performance.

REDSOC is particularly beneficial for compute-intensive benchmarks with long data-dependency chains. In the absence of very high ILP due to strict data-dependency, but at the same time when memory is not a bottleneck, REDSOC provides an ideal mechanism to improve performance in an energy-efficient manner, without having to increase processor voltage/frequency. Moreover, its suitability to general purpose processors and its non-speculative nature for circuit timing makes it a reasonable solution for better clock-period utilization in standard OOO cores.

8 SYNERGIC HW-SW ARCHITECTURE FOR SPATIO-TEMPORAL APPROXIMATION

In this chapter, we explore the SHASTA proposal. SHASTA builds atop the REDSOC proposal for slack recycling (Chapter 7) - extending this idea to approximate computing. While the above enables fine-grained compute approximation, this is accompanied by fine-grained memory approximation built upon prior work on Load Value Approximation [148]. SHASTA is able to perform multiple forms of approximation in conjunction and the approximation tuning mechanism synergistically evaluates the efficiency of all forms of approximation, over each corresponding approximate variable. SHASTA's approximation tuning is implemented via a gradient descent algorithm and is novel in that it is cognizant of the execution efficiency of the system - thus being able to achieve optimal error vs efficiency trade-offs.

First, we discuss the design of approximate hardware: Section 8.1.1 presents the timing approximation method for compute while Section 8.1.2 presents the load approximation method for memory. Next, in Section 8.2 we discuss the approximation tuning mechanism, providing a detailed look at the system cognizant gradient descent based tuning algorithm. In Section 8.3, we discuss the end-to-end approximation system and its synergy in approximation. Corresponding overheads are discussed in the above sections as well. Finally, in Section 8.4 we evaluate SHASTA's performance and energy efficiency benefits and compare against prior work. Section 8.5 summarizes the proposal.

An introduction to SHASTA was provided in Section 2.4, background / motivation in Chapter 3 and methodology in Section 4.4.

SHASTA's contributions are summarized below:

① Proposes a flexible and dynamic framework for fine granularity approximation.

② Allows each candidate variable (across the program) to be approximated differently.

③ Allows each dynamic instance of a candidate variable to be approximated differently at different loop iterations.

④ Allows fine-grained compute and memory approximation can support additional forms) - novel compute approximation is implemented via slack recycling.

⑤ Automates application tuning for approximation based on error tolerance target, implemented atop a gradient descent algorithm.

⑥ Application tuning takes hardware execution benefits into consideration.

⑦ Tuning also looks at combined benefits of all forms of approximation on accuracy and cost savings.

⑧ SHASTA is able to achieve considerably better benefits compared to prior work (2-15x) and can provide performance speedups or energy savings in the range of 20% to 40% depending on the goals of the design.

8.1 Design of Approximation Hardware

8.1.1 Compute Timing Approximation

Section 3.4.1 discussed the benefits and challenges of achieving fine-grained spatio-temporally diverse approximation. To achieve these goals SHASTA introduces a new form of Timing approximation called *Slack-Control Approximation* (SCA). SCA builds on top of REDSOC [181],

for accurate computing, which identifies the unused portion of the clock cycle in ALU computations based on opcode and data-type and eliminates it by starting future computations early.

SCA extends this idea further by reducing an operation's compute time while risking higher computation error, but in a controlled manner. Other forms of timing approximation are discussed in Section 3.4.4. SCA is enabled by interpreting a computation's tolerable approximation as a clock cycle slack component: what we call *approximation slack*. Once approximation slack is identified, it is cut out (or 'recycled') by the idea of per-operation slack recycling. Slack recycling is performed by enabling *transparent bypass paths* in a traditional synchronous execution pipeline. Slack recycling recycles the approximation slack in a "producer" operation by starting the execution of dependent "consumer" operations at the instant of completion of the producer operation, undeterred by clock boundaries. Recycling approximation slack in this manner over multiple operations, executing on traditional functional units, allows acceleration of these compute sequences. This results in application speedup when such sequences lie on the critical path of execution.

In SHASTA, SCA is capable of fine-grained spatio-temporally diverse approximation by allowing the following: ① It can uniquely control each dynamic approximation execution's computation time individually, ② It allows same compute units to be used for both accurate and entire range of (timing) approximate compute, thus allowing fine-grained per-operation control without significant overheads and ③ The computation time can be chosen from multiple discrete levels, every clock cycle, for every operation, depending on the amount of approximation the operation can endure.

Viewing Approximation as Cycle Slack: Consider the addition operation on a stan-

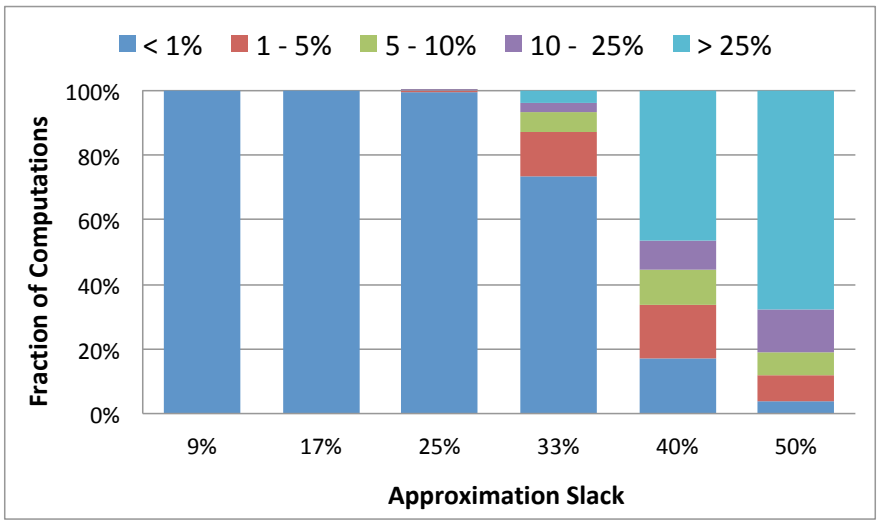


Figure 8.1: ADD: Timing Error Distribution

dard adder design (eg. 16-bit Kogge Stone) as was shown earlier in Fig.3.5. We define the computation time of any pair of operands on this adder as the time required for all 16 bits of the output to settle at the correct values. This is dependent on a) the two operands (i.e. the critical path that they trigger) and b) the previous state of the output bits. For a given fixed state of operands, previous output and operation voltage, it is intuitive that as the allowed computation time for this adder decreases, the potential for some of the output bits to be in an incorrect state increases. In other words, as the "approximation slack" increases, the approximation-error of the adder increases.

Prior work on circuit level error analysis (B-HivE [217]) has shown that timing error as a function of voltage (for a set frequency) can be effectively modeled for different computations (eg. add/multiply). We intuitively extend this to model computation approximation as a function of approximation slack. In actual chips, we expect this modeling to be performed statically at chip design time. Fig.8.1 shows the distribution of error magnitudes experienced by 1 million random add computations for different approximation slack

fractions. For addition operations, there are almost no error magnitudes greater than 1% across all random operations for 25% approximation slack. For 33% slack, more than 90% of operations have less than 5% error magnitude. This suggests (and is supported by results) that not only are average error rates low at reasonable approximation slack, but a majority of operations follow the same trends - meaning that tuning with reasonably characteristic sample inputs and running with inputs in the wild tend to show similar error-rates (even though we do not design to provide guarantees). Once the slack corresponding to different operations are estimated at design time, these values are stored in a slack look up table (LUT).

Every approximation level corresponds to a different slack value. At the decode stage the instructions indicate the amount of approximation on their execution (set by the tuning mechanism). This number is used to look up the amount of approximate slack applicable for this computation, for the particular level of approximation. In this work we only approximate multiply, add and subtract operations. Details on approximation translation from software to hardware are discussed in Section 8.3.

Recycling slack across operations: Once the approximation slack is identified the slack recycling mechanism from REDSOC is used. Slack recycling involves optimizations to the execution and scheduling stages of the processor core pipelne and were discussed in Sections 7.2 and 7.3 respectively.

8.1.2 Memory Load Approximation

Prior proposals for memory approximation already offer a suitable template for fine-grained spatio-temporally diverse approximation [148, 106]. But without a dynamic control mechanism to identify optimal approximation quantities throughout the application, these proposals perform poor exploration of fine-granularities and diversity. SHASTA proposes a modified form of Load Value Approximation (LVA) [148] which enables more aggressive and dynamic use of the previously static approximation technique, enabling fine-grained spatio-temporal diversity and further improving its benefits in other ways.

LVA Benefits and Limitations: LVA is motivated by the idea that applications suited to approximation often exhibit localized value similarity; they tend to reuse similar values. LVA proposes that for applications that can tolerate inexactness, the values associated with cache misses can be approximated. By approximating the load value on a cache miss, the processor can immediately proceed without waiting for the cache response.

Deployment of LVA is essentially controlled by two LVA-Control knobs - *approximation confidence* and *approximation degree*. Note: these knobs are as used in the original work and are paraphrased and explained below. *Approximation confidence* decides how often the data is approximated (based on past comparisons to some fixed error threshold) i.e. how often a value is returned from the approximator instead of a longer latency wait for the accurate value on a cache miss. This enables a trade-off between accuracy and latency of access. *Approximation degree* decides how often the data in the approximator is refilled with values from actual cache access (and thus retrained/refreshed). This effectively trades off accuracy for better energy efficiency in the memory hierarchy.

While latency and energy reduction is significant, there are limitations from LVA that our proposal exploits:

① First (and of most significance to SHASTA's fine granularity + diversity motivation), in LVA the approximation confidence's threshold value and approximation degree are static design time constants and uniform over all approximate loads - but this is not optimal. Among all loads that can be approximate, some approximate loads are less *influential* than others. While the errors from loads might sometimes be large, their effect on the application might be minimal (eg. noise in vision applications). Thus, deciding to approximate based on a particular load's error compared to a static threshold, can have a detrimental impact to overall benefits. Also, it is a common characteristic among inputs that some approximate loads are more *stable* than others. Consider financial applications - it is often the case that some inputs are redundant or change very rarely. For example, in blackscholes, a subset of the inputs takes on only four possible values, two of which occur over 98% of the time. Other inputs may change more frequently. Setting approximation degree to be constant results in higher error under aggressive settings and low benefit in conservative settings.

② Second, LVA invocations are rather infrequent because LVA is only invoked on L1 cache misses and these misses are often low in many applications.

③ Third, in LVA every approximate load still performs all the minimum 'tasks' that a traditional load performs - accessing the load-store queue, accessing the TLB for address translation, address generation computation and accessing the L1 data cache.

Dynamic Pre-L1 Approximation: Our contribution to memory approximation is two-fold. First, we modify LVA to be dynamic - enabling it to tackle fine granularities and spatio-temporal diversity. This is achieved as follows - each approximate load instruction

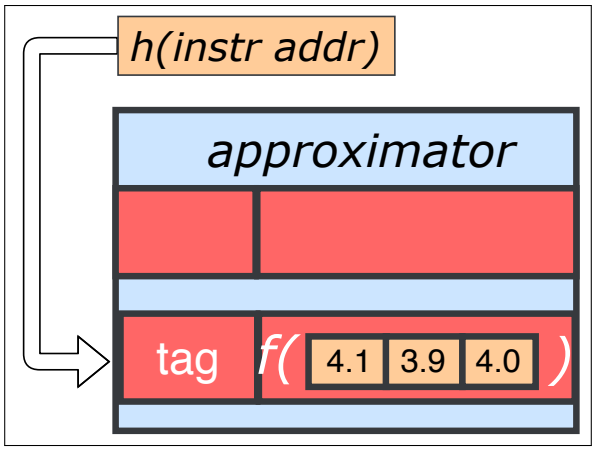


Figure 8.2: Load Approximator

is allowed a unique approximation confidence and a unique approximation degree, an optimum amount as determined by the tuning mechanism. These values are estimated based on application effects and not based on static error thresholds etc. The assigned values are unique per static approximate load as well as based on its iteration instance (if it belongs to a loop, thus exploiting temporal diversity). Therefore, the percentage of time that a static load instruction of some Nth iteration looks up its approximate value in the approximator is tuned uniquely to each such load. Similarly, the percentage of time that the approximator is refilled with an accurate value from the cache is tuned uniquely to each approximate load. Thus "error vs latency" and "error vs energy" trade-offs are controlled in a disciplined manner, unique to each approximate load. The approximation information flow from the tuned application to the hardware is similar to the case of compute approximation, and is detailed in Section 8.3.

Second, our memory approximation implements pre-L1 Approximation, which brings the approximator prior to the L1 cache. This allows the capability for invoking approximation on all loads and not just on L1 caches misses. Our design approximates loads early

in the execution pipeline. Loads which are marked as approximate, and which end up being approximated (based on approximation confidence) perform a *Pre-L1 LVA* lookup, as described earlier in Fig.2.2.b, and thus benefit from a) lower latency than L1 access and b) lower energy than traditional loads by skipping address generation, translation, dependence checks and cache access. Implementing LVA prior to L1 cache access is key to some of the synergic gains in SHASTA - this is discussed in Section 8.3.

Fig.8.2 shows the general structure of our *Pre-L1 LVA* approximator table. The approximator consists of a simple instruction address based hash which performs a lookup into a direct mapped approximator table. Each entry in the table only consists of a tag and a data block. The data block is essentially an approximate local history buffer (LHB) - storing some representation of the accurate reads (from cache) of the most recent approximate load values which match this entry's tag. In figure, the LHB values (4.1, 3.9 and 4.0) are the accurate values of the three previous loads that matched this tag entry. An approximate value is then generated by employing some computation function f (we use average) on the values in the LHB.

8.1.3 Overhead Evaluation

Compute Approximation: The slack recycling implementation in an OOO core suffers reasonable overheads - an area overhead of 0.3% and an energy overhead of 0.8%. The approximation slack LUT has area overhead of 0.52% and negligible access energy. All slack related access and control occur in parallel with the core decode and scheduling logic - there is no increase in critical path latency.

Memory Approximation: Design space exploration results in the choice of 64 entries in the approximator - this results in an overhead of roughly 1KB of storage. Due to the small size of the approximator, access time is very low, pushing the loaded operand off the critical execution path.

8.2 Approximation Tuning Mechanism

The highlighting characteristic of SHASTA's approximation tuning mechanism is that it is a combination of i) being suited to general purpose approximation, ii) supporting fine granularities and diverse approximation and iii) most importantly, enabling hw-cognizant optimization (it obtains actual execution measurements from hardware, such as IPC or energy, by running the application with input sample) when tuning the configuration.

Note that the tuning mechanism is capable of providing unique approximation values for *every dynamic instance of every approximate operation in the application*. For example, if N static program operations marked as approximate and these are iterated over M times, there are N*M dynamic operations for which potentially unique approximate values can be assigned by the tuning mechanism. This achieves the goal for fine-grained spatio-temporal approximation diversity. Remember that the forms of approximation might be different for different approximation operations: for example, in our use case the compute approximation focuses on clock-cycle slack while the memory approximation focuses on approximation loads. All types of approximation can be supported the tuning mechanism as long as each type clearly identifies how the approximation mechanism varies for each 'level' of approximation.

```
def eval_epoch(f,sample,AC,tol)
   """
   f(): function for approximation, sample: input for tuning,
   AC: previous epoch's approx config, tol: the error tolerance
   """
   # Calculate golden output, h/w efficiency metric
   Out_gold = f(sample,0)
   Eff_gold = HW(f(sample,0))
   # Outer-Loop: until convergence/tolerance
   while True:
      AC_prev = AC;
      AC = eval_AC(f, sample, AC_prev, Out_gold, Eff_gold)
      ΔAC = AC-AC_prev
      if ΔAC < ϵ || f(sample,AC) > tol:
         break
   return AC
def eval_AC(f, sample, A, Out_gold, Eff_gold):
   """
   Calculate grad to create new approx configuration
   """
   # Application error at current approximation
   Out = f(sample,A)
   Δerror = Out - Out_gold
   # Approx. Application's execution benefit
   Eff = HW(f(sample,A))
   Δexe = Eff - Eff_gold
   # Loss dependent on error and efficiency
   Loss = L(Δexe,Δerror)
   # Inner-Loop: Iterate over all approx. ops
   while not A.finished:
         # Perturb approximation of ith op in A (ai)
         A_i = modify(A, ai, δ)
         # Calculate application error at A_i
         Out_mod = f(sample,A_i)
         Δerror_mod = Out_mod - Out_gold
         # Calculate h/w metric at A_i
         Eff_mod = HW(f(sample,A_i))
         Δexe_mod = Eff_mod - Eff_gold
         # Loss at A_i
         Loss_mod = L(Δexe_mod,Δerror_mod)
         # Compute the partial derivative
         grad[ai] = (Loss_mod - Loss) / δ
         # Step to next approximation and reset current
         A.iternext()
   # Calculate the new approximation configuration
   A = A - step_size*grad
   return A
```

Listing 8.1: Gradient descent approximation tuning

Pseudocode for the tuning mechanism atop an application is shown in Listing 8.1 and features of the mechanism are discussed below.

① Each tuning epoch involves tuning the application's approximation configuration with some characteristic sample input.

② The tuning epoch starts with capturing the golden accurate application output i.e. with no approximation, and its corresponding golden hardware execution metric (eg. IPC/energy, using hardware/software counters).

③ The tuning mechanism will then run multiple 'outer-loop' iterations of the application, each performing a set of 'inner-loop' iterations.

④ Every 'outer-loop' iteration of the tuning mechanism starts with a current approximation configuration. The goal at the end of the iteration is to find the new approximation configuration which is the steepest move (in terms of the efficiency vs error gradient) from the current configuration.

⑤ In order to achieve this, the current approximation configuration is independently perturbed by δ in each dimension (i.e. each variable). Each resulting configuration is a 'test' approximation configuration.

⑥ The application is run with each such 'test' configuration over the course of tuning mechanism's 'inner-loop'. For each of these 'inner-loop' iterations, the application output and its hardware efficiency metric are obtained and compared with the golden values to obtain a 'Loss' value. The loss is directly proportional to the application error and inversely to the efficiency improvement.

⑦ The gradient of the Loss function along each approximation dimension is calculated by evaluating how much the loss function changed along each dimension's δ perturbation.

This information is used to obtain the new approximation configuration (at the start of the next iteration) and this ends the current 'outer-loop' iteration.

⑧ In the next 'outer-loop' iteration, the approximations with the steepest gradients from prior move by 'one' approximation level, while the other approximations proportionally move by fractional approximation levels. Fractional approximation levels are interpreted by the hardware as a dynamic percentage.

⑨ Tuning continues until convergence, i.e., when there is marginal change to the loss function / gradient on consecutive iterations of the 'outer-loop'. This ends the tuning epoch and the application will run with the final approximation configuration until the next tuning epoch.

Application	SHASTA TI	Greedy TI	Err @ 10%	Ovhd (ms)
Blackscholes	9	71	9%	2.7 (24)
Mat Mul	7	30	4.5%	0.15 (1)
Inversek2j	10	101	11%	1.5 (15)
K-means	22	130	9%	0.95 (21)
FFT	5	20	12.1%	1.4 (7)
Canneal	14	120	8.5%	0.22 (26)
PageRank	20	110	8.3%	1 (20)
MLP	18	80	5%	1.05 (19)

Table 8.1: Approximation Tuning

Overhead Evaluation: Table 8.1 shows that the number of global iterations the gradient descent mechanism takes to convergence is low for our approximate applications (discussed in Section 4.4). Moreover, these statistics are from cold starts wherein the mechanism is not in a tuned state prior to the training. In most scenarios, there is low change input characteristics from one epoch to the next, meaning tuning reaches optimum state for a new epoch in just 1 or 2 iterations. Thus average-case tuning overheads are only 1-2 iterations of

tuning per epoch and this greatly reduces the overhead of dynamically running the tuner. For comparison, the number of iterations for a greedy tuning algorithm are alshow shown.

While cold start tuning overheads range from 1-30 ms depending on the number of approximate variables, time to converge, training sample size etc, average case overheads are less than a ms. The tuning overheads in time (ms) are shown in the last column - invoking the tuning mechanism once every second results in overheads of less than 0.1%. Worst-case cold start overheads are in parenthesis.

The table also shows the actual error estimated at test time (i.e. on the test inputs) when tuning for a 10% tolerance target on the training inputs of the applications.

8.3 SHASTA: System and Synergy

System Overview: Figure 8.3 illustrates the high level overview of SHASTA showing the entire system flow from Ⓐ Static: Approximation specifications within the program and compiling to an approximation-enabled ISA, Ⓑ Dynamic Tuning: A tuning mechanism which iterates the application through multiple approximation configurations (along optimum gradient) on the target hardware, and Ⓒ Dynamic Running: program execution with tuned approximations on hardware.

PL / Compiler / Runtime Support: Programs are annotated as shown in Listing 3.1. The programmer annotates variables and computations that are amenable to approximation and specifies the target approximation for the application, as is done in prior work [192, 56, 29]. Among computations, this work currently only targets add, sub, mult operations for integer and floating point. Approximate load operations are inferred based on annotated

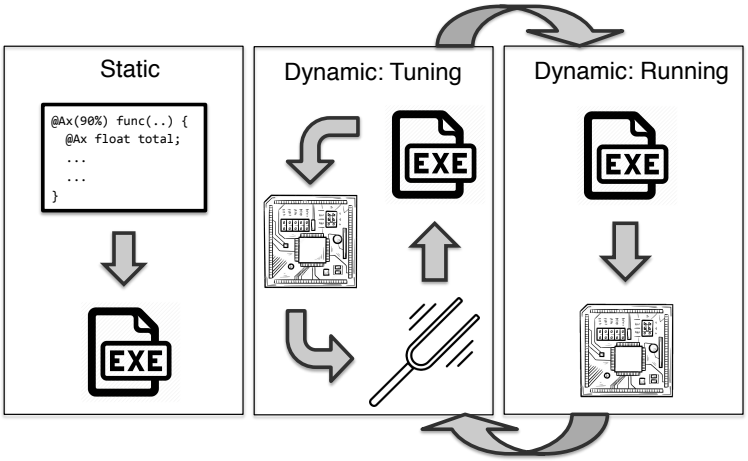


Figure 8.3: SHASTA system overview

variables (eg. v1, v2, d, total in Listing 3.1). Note that the tuning mechanism is agnostic to the fact there are multiple types of approximation (eg. memory vs compute). As long as the levels of approximation for each approximation-type are well defined, this tuning mechanism is suited to any form of approximation.

SHASTA's compiler takes the annotated application and generates an application executable that uses approximate instruction extensions to the ISA (we target ARM, but expect compatibility with other ISAs) for approximation. These instructions are {V}ADD.a, {V}SUB.a, {V}MUL.a and {V}LDR.a and resemble traditional instructions apart from the approximation fields described below. It is important to note that the programmer does not have to specify the amount of approximation for each approximate operation - she only has to specify the approximation requirement at an application/function level and also specify the variables which can be approximated. The values themselves are automatically inferred by the tuning mechanism.

The approximate compute operations use a 3-bit value to indicate the amount of approximation slack which this operation should recycle. A value of 2 (on scale of 0 to 7)

means the approximation slack is roughly 20% of the clock cycle (or computation time is 80% of the clock cycle). Approximate load instructions use two fields - the level of approximate degree (3-bit) and the level of approximate confidence (3-bit). The values of the degree/confidence are indicative of the percentage of time the load approximation related action is performed. For example, an approximation confidence value of 5 (on the scale of 0 to 7) means that the load is looked up in the Pre-L1 LVA roughly 70% (i.e. 5/7) of the time. Note that the compiler does not generate any approximation amounts for the approximate instructions, that is done only by the tuning mechanism by running on the target hardware.

Finally, we enable the interaction between the application and the tuning mechanism via pragmas, as pursued in prior work [176, 221]. In the application, the programmer is expected to add pragmas prior to an approximable function's declaration (such as the K-Means function in Listing 3.1), providing its error tolerance, error metric, hardware efficiency metric, tuning granularity and a pointer to some training input sample. This passes information to the runtime which invokes the tuning mechanism (details in Section 8.2) at the appropriate tuning granularity (possibly every second, for $< 0.1\%$ overheads). The runtime also captures hardware measurements and provides information to the tuning mechanism.

Overall, the programmer burden only involves annotating approximating variables/-computations and specifying approximate function pragmas. In our experience with our chosen approximate applications, this one-time overhead is very reasonable.

Synergy across HW approximation techniques: As discussed in previous sections, SHASTA performs approximations across both compute and memory. With SHASTA's

intelligent tuning mechanism, the benefits from the synergy between compute and memory approximation is greater than combining individual benefits from memory-only/compute-only approximation (under same total error tolerance). This is because of the following:

① Tuning across memory+compute provides a larger tuning space (i.e. more approximate operations), increasing potential for better efficiency-error sweet-spots.

② When an approximate compute operations performs compute on already approximate memory operation ("an intersecting memory-compute approximation"), the resulting error is often less than effects of errors from "non-intersecting" memory and compute approximations. A portion of the error in the load operations is masked away by the approximation in the compute operation. Since approximation tuning involves actual running of the application on hardware, such effects are respected by the tuning control.

③ Slack-Control Approximation produces performance benefits by slack recycling over transparent chains of operations. As discussed earlier, slack recycling only happens within a transparence boundary - one which is established by opaque memory operations. Approximating loads by skipping cache access removes this transparence boundary which breaks chains - meaning that slack recycling can occur over longer chains.

While the importance of synergy has been discussed in prior work targeting application-specific systems (see Section 3.4.4), SHASTA enables synergy among multiple forms of approximation, suitable to General Purpose Approximate Systems (GPAS). Further, SHASTA creates a platform to achieve synergy across diverse fine-granularities of approximations across multiple approximation domains which, to our knowledge, is insufficiently explored in prior work.

In this work, SHASTA employs forms of approximation which requires the hardware to

be capable of a) slack recycling i.e. have transparent paths in the data path, optimizations at scheduler and decode and b) load value prediction/approximation tables etc. While these mechanisms allow SHASTA to be fine grained and diverse in its approximation, the SHASTA system of general purpose approximation could be employed to support other forms of approximation as well.

8.4 Evaluation

The experimental methodology is discussed in Section 4.4.

8.4.1 Performance speedup

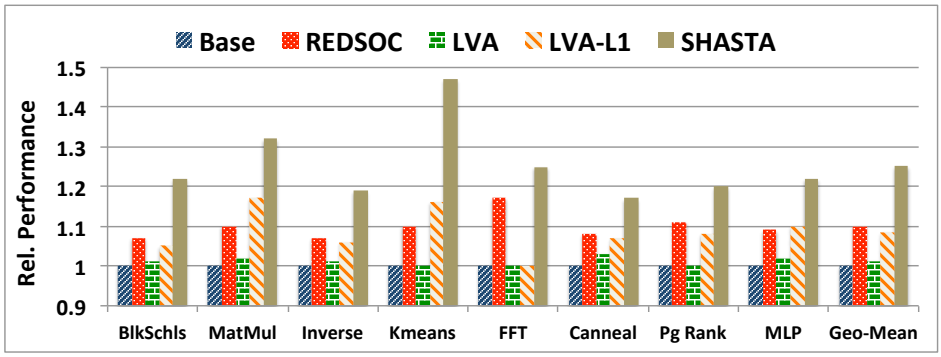


Figure 8.4: Performance Speedup

Fig.8.4 shows the speedup obtained by SHASTA in comparison to a) a traditional baseline without any approximation features, and prior proposals: b) REDSOC [181], which performs slack recycling but only targeting accurate compute and c) LVA [148], which load approximates only on L1-misses and further, an optimization of prior work: d) LVA-L1, which load approximates on L1-hits as well. We include LVA-L1 because with our chosen processor configuration (L1 cache size etc.), the applications we run see low

miss rate and render traditional LVA ineffective. LVA mechanisms and SHASTA results are shown for a 10% error tolerance. Further, SHASTA is set to tune for the performance (IPC) metric. Speedup is a function of the fraction of dynamic approximations, influence of non-approximate instructions (eg. their latencies) and application dataflow.

SHASTA clearly outperforms the competing mechanisms achieving a mean speedup of 25% across the applications with a maximum speedup of 47% with K-Means, in comparison to the traditional baseline. Highest speedups are seen for K-Means which has high error tolerance - this follows observations from prior work which discusses that very few bits of precision are required for reasonable accuracy in kmeans [106]. In comparison, REDSOC, LVA, LVA-L1 achieve speedups of 10% / 2% / 9% respectively - clearly showing that achieving the design goals introduced earlier have a significant benefit to approximation systems.

8.4.2 Reduction in Energy Consumption

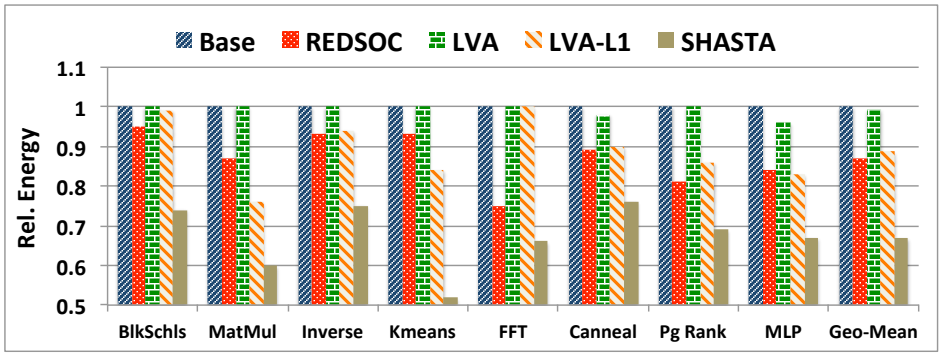


Figure 8.5: Energy Savings

Fig.8.5 shows energy savings from SHASTA in comparison to the baseline, REDSOC, LVA and LVA-L1, again for a error tolerance of 10%. Here, SHASTA is tuned for the

energy metric. Trends from energy savings are similar to those observed in performance speedup. An interesting observation in tuned configurations is that tuning for energy reduction resulted in more aggressive memory approximation in comparison to tuning for performance. This intuitively make sense because while compute approximation directly benefits only performance, memory approximation benefits both performance (by latency reduction) and energy (by cache/memory access reduction).

We find memory approximation to be less beneficial in applications like blackscholes (matching earlier observations [148]). LVA-L1 is of high benefit in a simple matrix multiply kernel (due to redundant as well as similar values) while traditional LVA is less useful because the cache miss rate is negligible. SHASTA sees significant benefits across these kernels - in blackscholes it is able to get significant benefits from compute approximation while it is able to leverage benefits from both pre-L1 LVA and compute approximation in matrix multiply, Mean energy savings for SHASTA are 33% with maximum reduction of 46% in K-Means. In comparison, other techniques see energy reduction in the range of 1% to 12%, clearly highlighting SHASTA's benefits.

8.4.3 Approximation Sweep

Fig.8.6 shows benefits from SHASTA (independent results for both speedup and energy reduction metrics) at different error tolerance levels. We sweep over accuracy requirements of 99%, 95% and 90%. In the figure, 'P' bars show performance speedup and 'E' bars show energy savings. In almost all the applications, there is a clear monotonous relation between tolerated error and benefits from approximation - there is at least a gain of 10% in speedup

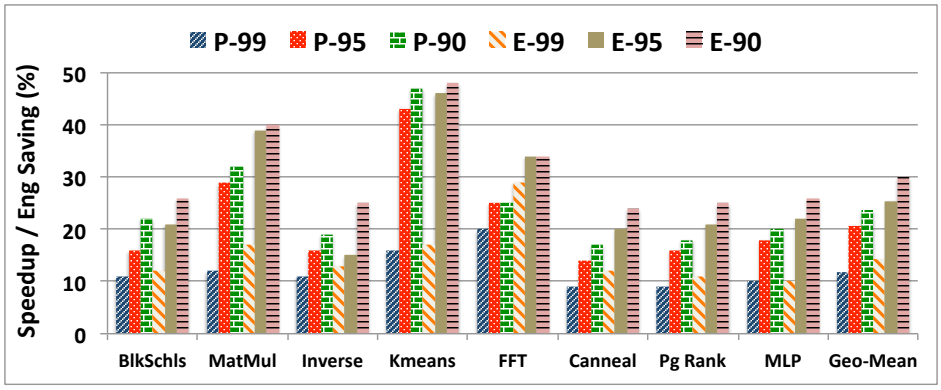


Figure 8.6: Efficiency improvements at varying application accuracy

/ energy reduction when going from 99% accuracy to 90% accuracy.

K-Means sees a 5x difference in benefit between highest accuracy and lowest accuracy. This is because, in K-Means, 95% accuracy can be achieved with very low precision in computation but a there is a significant increase computation precision/correctness required for 99% accuracy.

Even at 99% accuracy, energy savings of 14% and speedup of 12% reflect the importance of approximate systems and their impact even under stringent accuracy requirements.

8.4.4 Breakdown of benefits

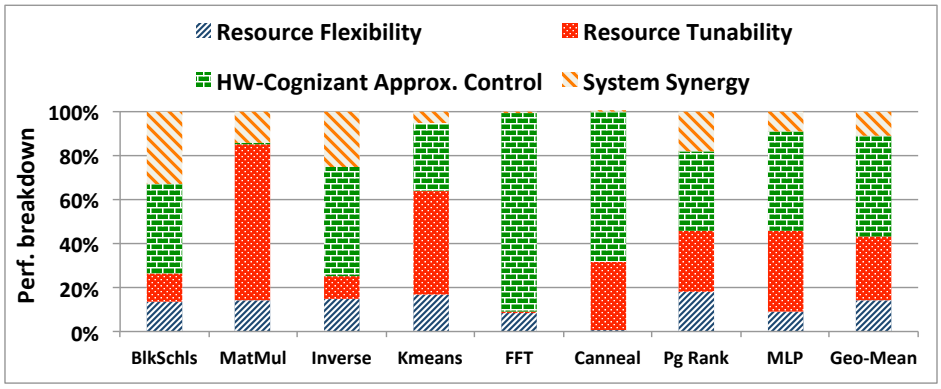


Figure 8.7: Performance benefit breakdown

Next, we breakdown the benefits of SHASTA in terms of the design goals. The analysis is shown in Fig.8.7 wherein we compare SHASTA to a baseline system without SHASTA's highlighting characteristics - a) Spatio-Temporal Diversity at Fine Granularity, b) HW-Cognizant Approximation Tuning and c) Synergic benefits across approximation techniques. To measure the impact of each characteristic, we incrementally add each characteristic to the baseline system - to finally achieve SHASTA.

Spatio-Temporal Diversity at Fine Granularity: We break this SHASTA benefit into two parts - a) resource flexibility i.e. every resource node in SHASTA can handle both accurate and approximate computations, and b) resource tunability i.e. the amount of approximation to be applied by each resource node can be uniquely controlled.

The benefit from resource flexibility is quantitatively observed by comparing SHASTA against the baseline, which provisions exactly 1 compute node for approximate-only compute. Thus not more than 1 approximate compute operation can be processed in parallel. Fig.8.7 shows that SHASTA's resource flexibility contributed to 16% of its total benefits, with higher contributions in applications with more approximate ILP.

Next, Fig.8.7 shows that resource tunability contributes to 30% of SHASTA's overall benefits, obtained by comparing SHASTA against the baseline which allows only fixed/static approximation. The approximation of the resource is tuned in accordance to the spatio-temporal approximation diversity in the application. Temporal diversity is especially important in iterative applications (K-Means, Canneal, Page Rank and MLP) because the accuracy requirements vary across these iterations and need not be conservatively set by the worse-case. In the absence of spatial diversity, all static approximate operations will be tuned to the same approximation - we see this to especially detrimental for simple

applications such as matrix multiply which have only a few static approximations but which can significantly influence the energy/performance of execution. Thus spatial diversity contributes to almost 70% its benefits from SHASTA.

HW-Cognizant Approximation Tuning: To illustrate the benefits of hardware cognizant tuning, we compare SHASTA's tuning mechanism against the baseline's greedy hardware-agnostic tuning. Benefits from intelligent tuning contribute to nearly 40% of SHASTA's benefits as seen in Fig.8.7.

In applications such as K-Means, HW-Agnostic tuning approximates compute variables further than ideal, thereby unable to sufficiently approximate memory variables which could provide significant energy savings. Somewhat similar are Blackscholes and Inversek2j. With a large number of approximate variables, the optimal compute approximations which produce best gradients for approximation (i.e. higher execution benefits but lower error), are deep into the program. A naive tuning mechanism hits the error-tolerance rate just within the first few approximate variables, never reaching the best-gradient variables. Iterative applications such as Canneal benefit greatly from the tuning mechanism because specific iterations of the application (middle/late) provide best energy vs error gradients and are found by the tuning mechanism.

Synergic Benefits: Finally we quantify SHASTA's synergic benefits across the two approximation techniques used. We compare SHASTA against the baseline, in which compute approximation and memory approximation are tuned separately unbeknownst to each other. For this experimental baseline, we tune each of compute and memory for half of the total approximation (i.e. 5% error each), while SHASTA is tuned as a whole. The synergic benefits SHASTA obtains is reflected in Fig.8.7 and contributes to nearly 15% of

the total benefits. Synergic benefits are particularly high in applications with higher error tolerance but more approximate variables such as Blackscholes and Inversek2j, wherein it is especially important to consider the relations between approximate memory operations being used by approximate compute operations.

In summary, it is evident that SHASTA's benefits are obtained from a combination of fine-grained spatio-temporal approximation capability, hardware-cognizant approximation tuning as well as whole system synergy.

8.5 Chapter Summary

SHASTA proposes a novel hardware-software approach to designing efficient *General Purpose Approximation Systems*. It is able to improve hardware approximation capability achieving fine-grained spatio-temporally diverse approximation. At the same time, it adds hardware cognizance to approximation tuning to achieve the optimum execution efficiency under the prescribed error tolerance. Further, it achieves synergic benefits across optimizations, building a closer-to-ideal general purpose approximation system. Via qualitative and quantitative comparisons we show that SHASTA is able to achieve considerably better benefits compared to prior work (2-15x) and can provide performance speedups or energy savings in the range of 20% to 40% depending on the goals of the design, a rather significant improvement on top of traditional general purpose processor architectures.

9 ATTACKING LATENCY, MODULARITY AND HETEROGENEITY CHALLENGES IN THE NOC

In this chapter, we explore the TNT proposal. TNT moves the focus of slack recycling from computation (which we saw in the three chapters prior) to communication in the on-chip network. Here, slack presents itself in the form of sub-clock cycle hop-to-hop wire traversal capability. TNT proposes a solution with near-ideal wire-only latency, built in a modular fashion (which is a must for scalable systems). It is especially novel in tackling physical heterogeneity in the NOC - stemming from sparse placement of memory controllers, asymmetric aspect ratios of nodes and process variation.

In Section 9.1 we presents the transparent network traversal proposal with an illustrative example. Section 9.2 discusses the design of the modular lookahead network which is fundamental to TNT's transparent traversal capabilities: Section 9.2.1 addresses flexible time stamp based delay tracking, Section 9.2.2 addresses the transparent routing scheme, Section 9.2.3 presents safeguard mechanisms to prevent timing violations and Section 9.2.4 provides an overview of other optimizations / details in the proposed design. Section 9.3 discusses TNT implementation as well as its overheads. Finally, in Section 9.4 we evaluate TNT's performance and energy efficiency benefits on both synthetic traffic and graph workloads, and also compare against prior work. Section 9.5 summarizes the proposal.

An introduction to TNT was provided in Section 2.5, background / motivation in Chapter 3 and methodology in Section 4.5.

TNT's contributions are summarized below:

① TNT is the first NOC proposal to incorporate on-the-fly delay tracking to enable

multi-hop many-cycle traversal and approach ideal *wire-only* network latency.

② TNT uses a novel transparent traversal approach, which enables flits to travel the entire route from start node to end node in a single "long-hop". Unlike prior designs, it does not require all-to-all request links in the NOC to achieve this.

③ First design to exploit wire heterogeneity arising from physical constraints like node placements and variation.

④ TNT is a scalable solution - it is built in a modular tile-scalable manner, a key factor for realistic implementation. It performs only neighbor-to-neighbor control interactions but enables end-to-end transparent flit traversal.

⑤ Analysis on Ligra graph workloads shows that TNT is able to to reduce LLC latency by up to 43%, improves performance by up to 38%, reduces dynamic energy by as much as 35%, compared to the baseline 1-cycle router NOC. Further, it achieves more than 3x the benefits of best alternative research proposals across different metrics.

9.1 Transparent Network Traversal

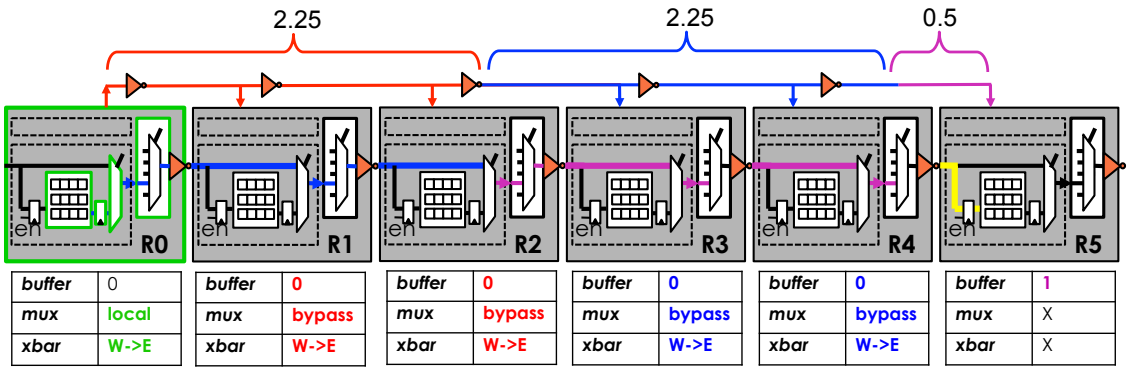


Figure 9.1: Traversal Illustration

A background discussion on the wire capability in NOCs, potentially allowing multiple hops worth of traversal in a single cycle, influenced by physical heterogeneity of the network, is discussed in Section 3.5.2.

TNT uses transparent flow based flit traversal to enable flits to travel the entire route from start node to end node in a single "long-hop," covering multiple hops over the minimum circuit-constrained number of cycles. Only a single pass for the entire flit route, instead of a sequence of hops, avoids the quantization effects of intermediate routers. The flow of a single-flit packet in TNT is described below.

Fig.9.1 show 5 routers with additional per-router control enablers to allow transparent traversal. There are 3 primary control enablers: (a) Buffer Write ($buffer$) at the input flip-flop to determine if an incoming input signal should be latched or not, (b) Bypass Mux (mux) at the input of the crossbar that chooses between a local buffered flit and the incoming bypassing flit on the link, (c) Crossbar select ($xbar$) connecting inports to outports. The packet traverses 5 hops from router R0 to R5 and the chip-wide wire delay analysis (at design time) estimates that $\eta_{chip} = 0.45$ i.e. 2.25 hops per cycle. For the sake of simplicity, this example assumes no heterogeneity. The life time of the 5-hop packet is as follows:

① Cycle 1: A *long-hop* starts from start router R0 where the flit is initially buffered (green outlined in Fig.9.1). Apart from traditional RC/VS, this stage performs **Switch Allocation at Source (SA-S)**, identical to SA in a conventional pipeline: every source router chooses a winner for each outport from among its buffered flits. Assuming the flit wins **SA-S**, TNT will attempt to send the flit all the way to the destination R5 in a single transparent multi-cycle *long-hop*.

② Cycle 2: The next step is to initiate the **Lookahead Request (LR)** (shown in red in Fig.9.1). The LR flows ahead of the actual flit in the same route from source to destination but on a **Lookahead Network (LN)**. The role of LR is to set up router transparence (i.e. router bypass) *just in time for arrival of the data flit*. Note that LR can be a multi-cycle request (since it could take multiple cycles to travel from source to destination). In this example, based on the 2.25 hop per cycle wire delay, the LR has the capability to reach routers R1 and R2 within Cycle 2 and be 0.75 hops from R3. When the LR first reaches intermediate router R1, **Idle Link Takeover (ILT)** is attempted at the router. The role of ILT is to enable transparent use of the router's outport and link, in the required direction, if they are idle (i.e. not set up for use by already buffered flits) in this cycle. This determines if the data flit will be able to flow transparently through the router when it arrives there in the subsequent cycle. If the LR succeeds at ILT at R1, it flows through to R2 where the same process is repeated. If the LR wins ILT at R1, R2, the respective bypass muxes are set to *bypass-mode*. Buffering is disabled and the crossbar is set for West to East traversal.

③ Cycle 3: There is both data flow and control flow, both shown in blue. The data flit traversal is initiated out from R0 to perform the multi-cycle 'long-hop' to destination R5. Base on the control signals set by the LR in the earlier cycle, the data flit is able to bypass buffering and flow through transparently at router R1 and R2. Further, the data flit flows beyond R2 speculatively so as to use up wire traversal capability completely. At the end of cycle 3, the data flit is in flow between routers R2 and R3 (it would have covered 2.25 hops and be 0.75 hops away from R3). Meanwhile, the LR-flow continues ahead of the data and it covers another 2.25 hops reaching routers R3, R4, attempting ILT at the routers, and is halfway between R4 and R5 (assuming successful ILT). At R3, R4 the control signals are

set appropriately, so as to allow transparent data flit traversal in the following cycle.

④ Cycle 4: The data flit covers another 2.25 hops, transparently flows via R3 and R4 and is half a hop away from R5. The LR covers its remaining 0.5 hop and reaches R5, the destination where it enables buffering. All are shown in pink.

⑤ Cycle 5: The data flit travels the remaining 0.5 hop (shown in yellow) and gets latched at R5, completing the 5 hop traversal in a single *long-hop* pass, consuming 5 cycles.

Note that the links/routers over the path from R0-R5 are *not* being all held by this request throughout the duration of the long-hop - the usage is pipelined. For example, in cycle 3, routers R1 and R2 are free to receive new LR requests and in cycle 4 they are free to get new flits.

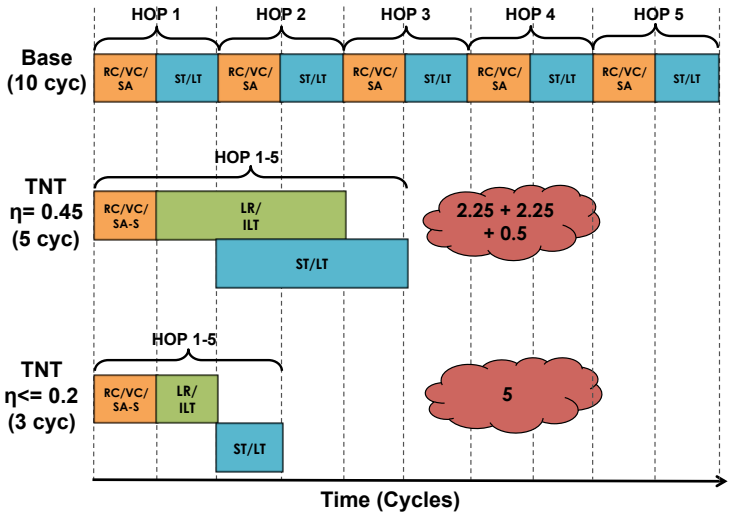


Figure 9.2: TNT Traversal Timing Analysis

The benefits of TNT for the single flit-packet (discussed above) are shown with a timing illustration. Fig.9.2 depicts the timing diagram for the 5-hop traversal with no conflicts. First, it shows the baseline design which takes 2 cycles per hop. Thus the packet's 5 hop traversal consumes a total of 10 cycles. Next, the timing diagram for the previously

described scenario with $\eta = 0.45$ is depicted, consuming a total of 5 cycles. Finally, the timing diagram for the lowest possible latency for this traversal is shown, achievable for $\eta <= 0.2$. The entire 5-hop flit/control traversal can complete in a single cycle each, resulting in a total request time of only 3 cycles. TNT benefits significantly over a wide η range, allowing its beneficial usage over different NOC designs, running at a range of frequencies, with a variety of tile sizes and so on.

9.2 Modular Lookahead Network

An integral part of long-hop traversal model that TNT proposes is the **Lookahead Network** (**LN**) for control requests, which establishes the transparent multi-hop path *just in time* for the flow of the data flits. In accordance with the philosophy of building realistic NOCs in a modular and tile-scalable manner (Section 3.5.1), LN's features are self-contained within each tile and consistent across every tile, even though it has an end-to-end influence on the NOC. LN's features are listed below and discussed in more detail in following sections:

① LN simply adds light-weight links between adjacent routers of the original data flit network (a 2D mesh).

② **Lookahead Requests** (**LR**) flow through the LN. LRs setup sections of the transparent path just prior to arrival of the data flits, up to $\frac{1}{\eta}$ routers in a cycle. LRs are routed through the LN, through the same set of routers as their respective data flits, but arrive at routers one or more cycles earlier.

③ LRs carry routing information along with a **time-stamp** which is used to avoid collisions and prevent metastability, enabling heterogeneity-cognizant transparent traversal.

④ A LR entering an intermediate router's inport performs **Idle Link Takeover (ILT)**, during which it attempts to take over the router's outport and link for a particular cycle. Only if it wins ILT can it flow to next routers. Further, winning ILT guarantees transparent flow at the router for its data flit.

⑤ LR-flow through the router, along with ILT, is completely combinational (since multiple router hops are to be covered in a cycle) and the ILT scheme is implemented to suit combinational conflict resolution.

⑥ The entire design is implemented with a globally synchronous clock and timing violations are avoided via safeguards when requests cross clock edges/boundaries.

⑦ The design has low wiring/area overheads, low energy spent on signaling, no false negatives at routers and numbers of conflicting LRs resembling a traditional mesh.

9.2.1 Time-stamp based delay tracking

The design illustrated in the previous section assumed that link delays are constant across the route, and were the same for data (flits) control (LR) paths. This is challenging to enforce in terms of circuit and layout. Thus, we design our delay-tracking scheme to support heterogeneity. We build a flexible design wherein control flow and data flow can be timed separately, with unique per-link delays. The only requirement is that the control path is faster than the data path - easily ensured with the implementation of LN discussed in Section 9.3. With this constraint, the flexible design is achieved via time-stamp based delay tracking. For each long-hop pass, the LR carries information about the cumulative delay of the LR itself and the data flit respectively, over the route of the long-hop. This is

achieved by accumulating wire delays over each flit / LR link, by means of time-stamps.

The wire delays are a function of multiple characteristics (Section 3.5.2) and can be estimated post-fabrication. Once measured, they are stored in the form of a look-up value at each link/router-outport. At the start of a TNT pass, the time-stamp to be carried through by the LR, for itself and the data flit, are initialized to 0 and these time stamps are updated (accumulated) at every link along the route. Transparent traversal decisions are made based on these time-stamps, which are explained with an example and illustrated design in Fig.9.3.

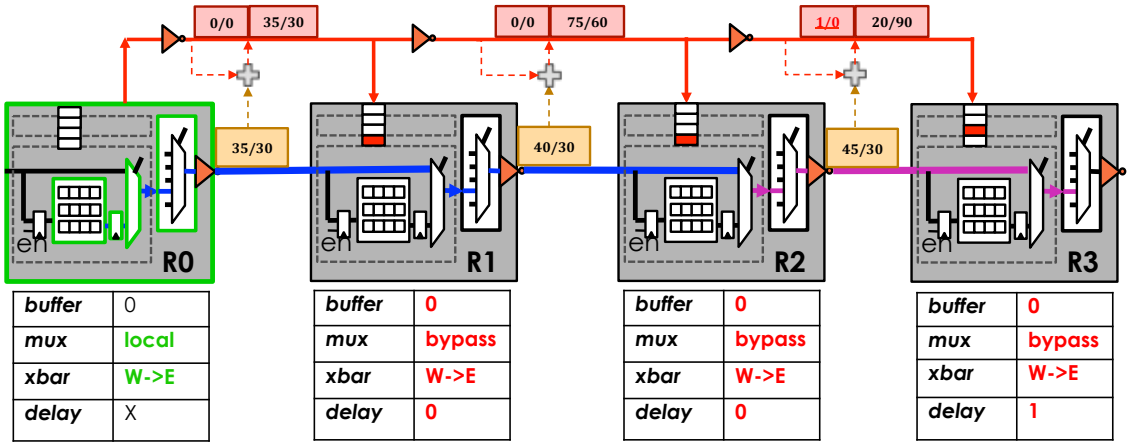


Figure 9.3: Time-stamp based tracking design

The example illustrates 3 hops of traversal wherein $\eta^{LR}_{0/1/2} = 0.30$ i.e. delay per LN control link is 0.30 cycles. Further, $\eta^{Data}_{0/1/2} = 0.35/0.40/0.45$ i.e. the delays for the sequence of 3 data hops in the route are 0.35/0.40/0.45 cycles respectively (including within-router delays). These delays are stored at the routers (yellow). At the top of the figure, we see the accumulation of data/LR delays (red), as well as overflow bits, the use of which are explained below:

① When the LR is sent out from router R0 in clock cycle 0, it accumulates the data link

delay (D^{Data}) and the LR link delay (D^{LR}) and carries the information to the downstream routers (assuming it wins allocation throughout).

② Flowing from router R0 to R1, it carries $D^{LR} = 30$ and $D^{Data} = 35$. This means that LR reaches R1 at 0.30 cycles and the data flit will reach R1 at 1.35 cycles. Thus, R1 is required to be transparent on the next cycle after LR arrival.

③ Flowing from R1 to R2, the cumulative delays are updated to $D^{LR} = 30 + 30 = 60$ and $D^{Data} = 35 + 40 = 75$, meaning that LR reaches R2 at 0.60 and data will reach R2 at 1.75 cycles. Thus R2 should also be transparent in cycle 1.

④ Flowing from R2 to R3, we have $D^{LR} = 60+30 = 90$ and $D^{Data} = 75+45 = 120 > 100$. This results in changes to the overflow bits: Ovf^{LR} is low, while Ovf^{Data} is set to high. This indicates that LR reaches R3 in clock cycle 0 (at 0.90) while data only reaches in cycle 2 (at 2.2). This means that the lag between the control signal and the flit would exceed 1 cycle. Thus, this LR is buffered for a clock cycle at R3 and then LR attempts to win router transparence for cycle 2.

The overflow bits are reset appropriately at the clock boundary. Note that it is possible for greater than 1 cycle cumulative delay between LR and subsequent data. In such scenarios, winning LRs are buffered at the router. We do not observe lag beyond 1-2 cycles, so very small buffers sufficient.

Key takeaway: TNT tracks the timings of heterogeneous links throughout the NOC, and routers are made transparent only in clock cycles that data flits will actually arrive.

9.2.2 Idle Link Takeover

TNT performs a sequence of up to $\frac{1}{\eta}$ ILTs combinationally every cycle. There is ILT contention possible at every router, from requests arriving at different timing instants within the same clock cycle. Thus, in order to design an effective **ILT Priority** scheme to decide which request flows through, it is important to understand the routing challenges involved.

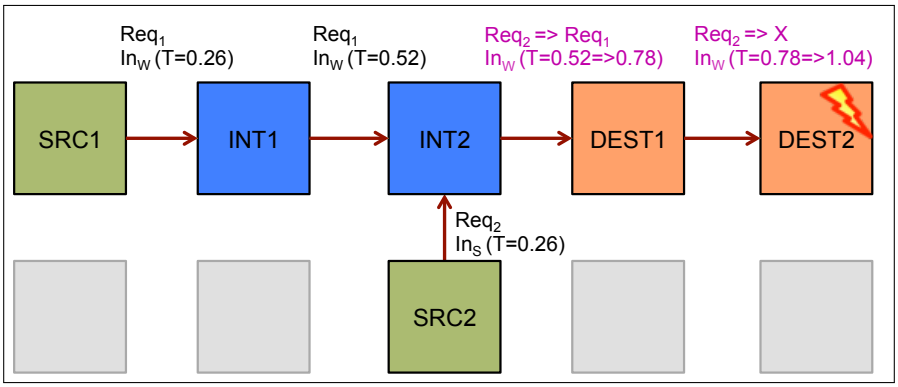


Figure 9.4: Timing for combinational ILT

Consider the example in Fig.9.4 which shows a design with a fixed $\eta = 0.26$. Two routers SRC1 and SRC2 are sending LRs (for $Flit_1$ and $Flit_2$ respectively). DEST1 is 3 hops away from SRC1 with no turns, while DEST2 is 3 hops from SRC2 with one turn. Assume a conventional priority scheme [120] for choosing requests at each router, which prioritizes buffered requests over straight over turns. In this example there are no buffered requests, so only ILT contention. The timeline of events is shown in the figure and is explained below:

① At T=0, LRs are initiated by both source routers.

② At T=0.26, Req_1 attempts ILT at INT1 while Req_2 attempts ILT at INT2. The combinational ILT logic at INT1, INT2 allow Req_1 (straight) and Req_2 (turn) to win respectively

- at this time they are the only requests at those routers.

③ At T=0.52, Req_1 arrives at INT2 - this is still the same clock cycle wherein INT2 allowed Req_2 to pass through earlier. Now, since the naive ILT priority scheme prioritizes straight requests over turns, Req_1 would succeed at INT2 even though Req_2 was already present and INT2 had previously declared Req_2 to be successful at ILT. This makes Req_2 an incorrect LR, since only $Flit_1$ should flow through INT2 in the next clock cycle. At same time Req_2, which prematurely won at INT2, reaches DEST1 and wins ILT and is about to flow on towards DEST2 (its destination).

④ At T=0.78, Req_1, the eventual winner at INT2, reaches DEST1 (its destination) and sets up control signals to buffer its flit there. At the same time, the incorrect winner Req_2 flows to reach DEST2 where it expects to buffer its flit.

⑤ Only by around T=1.04, the information may propagate through to DEST2 that Req_2 is an incorrect LR (i.e. by control bits potentially settling to the correct state thanks to Req_1). But this is likely too late, since the control decisions are clocked in at T=1.0 (assume a 1ns clock cycle). Thus it is possible that the LR Req_2 remains at DEST2. This false positive at DEST2 is detrimental to overall network throughput, though it is does not cause a functional correctness violation. On the other hand, a worse scenario is possible if the late arriving control bits are unsettled at $DEST_2$ at the clock edge leading to a hold time violation and potential meta-stability.

To avoid this, the key requirement for the **ILT Priority** scheme is that it *prioritizes the first arriving request* at a router in any cycle (in this case, Req_2) and block any future requests. In other words, an idle link/outport should be declared busy as soon as it is granted to an LR. The logic to implement this is illustrated in Fig.9.5, which shows LR requests from North,

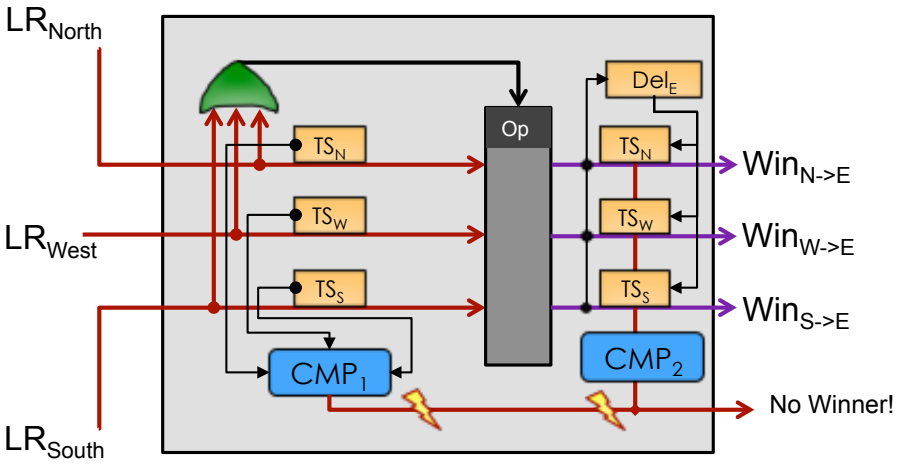


Figure 9.5: Priority + Safeguard detection schemes.

West, South inports of a router competing for the East outport. At the router, the LRs arrive at a latch whose opacity is controlled by the "OR" of all the LRs (i.e. their valid bits). Thus, the latch becomes opaque as soon as the first LR arrives during a clock cycle. This first arriving LR is declared winner and can potentially pass through transparently, followed by its data flit in the next clock cycle. Later arriving LRs all buffer at this router. Via this first-come first-serve priority logic, any potential issues from glitching at future routers are avoided (except as described in Sec.9.2.3). Note: in case of local/buffered request, priority is for the local request and all incoming requests are set to buffer at this router.

Key takeaway: Conventional priority schemes are insufficient for multi-hop singe/multi-cycle traversal. We design a first-come first-serve combinational scheme to address this.

9.2.3 Timing Safeguard

In the above description of the ILT priority logic, it is possible that multiple LR requests arrive at the same time or within very short intervals of each other - which could cause the latch in Fig.9.5 to glitch as it becomes opaque. This is avoided by the **ILT Safeguard**

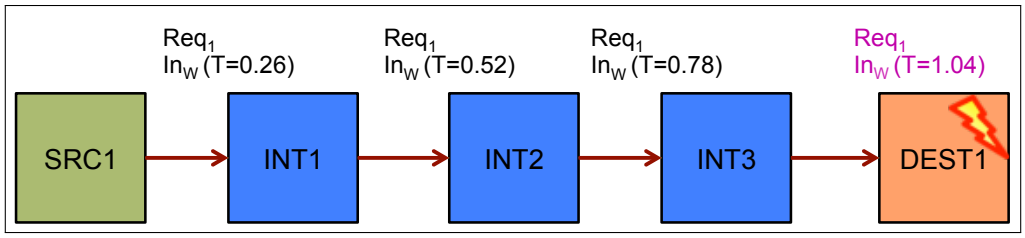


Figure 9.6: Cross-cycle violations (LR Safeguard)

mechanism - a simple time-stamp compare logic CMP1, which compares the time-stamps of the incoming LRs ($TS_{N/W/S}$ in figure, left of the latch). If incoming LRs have time-stamps within a "violation window" of each other, the CMP1 logic triggers a fault, no LRs are allowed to pass through and all are buffered at the router. Thus, the ILT Safeguard prevents the scenario when *multiple* LRs arriving at a similar time and compete for priority.

There is one other potential for timing violation, caused even by a *single* LR request. An example is shown in Fig.9.6. Again, we consider an example design with a fixed $\eta = 0.26$. A single request is shown, a 4-hop request from SRC1 to DEST1. If the LR request is initiated at time T=0, Req_1 can be successful with ILT at INT1, INT2, INT3 at times T=0.26/0.52/0.78, meaning that the flit can traverse through these routers transparently in the next clock cycle. Next, Req1 would reach DEST1 at time T=1.04. Ideally, this would mean that LR reaches DEST1 cleanly within clock cycle 2, during which it sets up buffering for the flit, and the flit itself would reach at T=2.04 and get buffered at T=3.0. But again, it is possible that since the control bits flowing into DEST1 are switching around time T=1.04, very close to the clock boundary at T=1.0, a hold-time violation could occur leading to meta-stability if buffering is required.

This is easily averted via the **LR Safeguard** - simple logic shown as CMP2 in Fig.9.5. Once a LR has been latched at a current router, its arrival time at the next router is estimated.

This is calculated by summing the current time-stamp of the winning LR (one of $TS_{N/W/S}$, seen right of the latch) and the wire delay of the control out-link (shown as Del_E in figure). If this new summed up time stamp of the winning LR falls within a "violation window" of the rising edge of a clock (such as T=1.04 here), then there is potential for timing violation at the next router. If so the LR is buffered at the current router and not allowed to pursue its transparent flow.

In both safeguards, the logic can be made conservative or aggressive by lengthening or shortening the violation window. In our work both CMP1 and CMP2 assume a +/- 5% timing violation window. Section 9.4.1 discusses its trade-offs.

Key takeaway: Even with a suitable prioritization scheme, multi-cycle flows have timing constraints that required to be addressed. We solve these challenges by building safeguard mechanisms which utilize the request time-stamps.

9.2.4 Additional optimizations

Finally, we minimally address other optimizations and details that are important for correctness:

① *Route computation for several hops:* We only allow flits and LR flow to follow deterministic routes (oblivious DOR). Thus, route computation at start router is trivial.

② *Avoiding flit reordering:* This is ensured by the deterministic routing, in conjunction with the priority mechanism which prioritizes buffered/older flits over new incoming flits.

③ *Guaranteeing VCs/buffers at downstream routers:* We use credit-based flow-control. LR can flow from one router to the next, only if buffering downstream is guaranteed. This is

checked in parallel to ILT.

④ *Avoiding premature buffering of Body/Tail flits:* The Head reserves a VC at all its intermediate routers, even though it does not stop there. These are freed by the Tail.

⑤ *Low-load SA-S bypassing [108]:* An incoming flit that is denied transparent flow can speculatively send the next LR without waiting for SA-S (Switch Allocation at Source), if there are no flits ahead of it in the input buffer queue.

⑥ *Preventing short paths / data corruption:* An opaque boundary is always maintained between physically adjacent flits on back-to-back cycles. This can limit gains at high traffic, but we do not see a noticeable impact. If necessary, they can be alleviated by employing path diversity.

9.3 TNT Implementation

Section 4.5 provides details on TNT's implementation methodology.

TNT assumes a chip-wide synchronous clock domain for the NOC routers, as is common in proposed and real designs [97]. In contrast, the nodes attached to each router may (and often do) have private clock domains.

The TNT data-path is modeled as a series of 128-bit 2:1 mux (for bypass) followed by a 4:1 mux (crossbar), followed by 128-bit links. LR signals are 16-bit, primarily carrying destination router id (6-bit), LR delay (4-bit) and flit delay (4-bit). The TNT control-path consists of LR traversal via the mesh-style LN - a series of 16-bit links + logic delay (from ILT) at each router. The LN router was illustrated in Fig.9.5 - there are 2-4 LR links per router, thus up to 4 LRs competing for ILT. Other logic in Fig.9.5: CMP1 and CMP2 are

4-bit modified comparators and delay accumulation is also performed on 4-bit adders. Our designs use link lengths varying from 1mm to 8mm.

	Links	**Requests**	**False Neg**	**Router**
TNT	2-4	1	No	$O(1)$
SMART	$O(\frac{1}{\eta^2})$	$O((\lceil N * \eta \rceil) * \frac{1}{\eta})$	20-40%	$O(\frac{1}{\eta^2})$
FB	$O(4 * N)$	1	No	$O(N^2)$

Table 9.1: Complexity Comparison

Design Complexity: We analyze the signaling network complexity of TNT in Table 9.1, in terms of the number of lookahead/signaling links per router ("Links"), the number of independent signaling requests ("Requests"), the number of false lookahead signals i.e. false negatives ("False Neg"), and the complexity of the routing lookahead logic ("Router"). TNT employs a mesh network for lookahead, resulting in only 2-4 links per router, only 1 request sent out for lookahead which is routed through the mesh and further, a fixed lookahead routing complexity which does not increase with larger networks. The mesh routing also ensures that there are no false lookaheads.

We also provide comparisons with SMART [123] and Flattened Butterfly [119] which are discussed later in Section 9.4.2. It is inuitive that the key factor that makes TNT suitable for realistic and scalable implementation, is that the lookahead signaling uses a mesh. In comparison, prior works use some form of point-to-point signaling / data movement, which do not scale as well [123, 35, 119]. Clearly, none of TNT's signaling components increase in complexity with increasing number of nodes in the system.

Timing: The total conflict-resolution delay for an N-hop request in the LN mesh design is optimally a sequence of N ILT arbitrations on its constant complexity ILT logic. While the number of ILT logic computations is higher in a mesh compared to point-to-point signaling,

energy/delay analysis shows that overall cost is minimal (more details below), making it the all-round favorable option. Analysis on DSENT shows that for 1GHz, TNT's control/data path are able to achieve as much as 12mm per cycle traversal with low overheads. In the control path, arbitration logic accounts for 20% of total delay while the rest is from LR traversal.

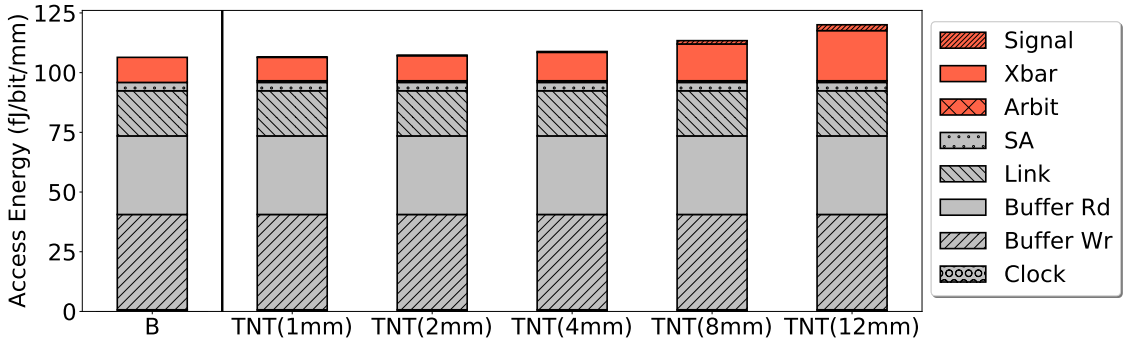


Figure 9.7: Energy per Access

Access Energy: Fig.9.7 plots the energy/bit/hop for each NOC request component for TNT when designed to achieve a traversal capability of 1-12mm in a clock cycle, compared to a traditional baseline NOC. Energy components from clock, buffers and local switch allocation (for the data/base portions of all the NOCs) are similar across all designs. The data Xbar energy increases with traversal capability and comprises of the energy consumed by the data path repeaters for driving the bypass and crossbar muxes. The total energy consumed by the LN is broken into the *Signal* (LR/LN-links) and *Arbit* (ILT/LN-router) components. The *Arbit* also includes costs associated with safeguard mechanisms and delay tracking which are negligible relative to the payload, given the small number of bits required for them. *Signal* energy grows only linearly and *Arbit* energy per hop remains constant (and low). The access energy increase for TNT compared to the baseline is $< 1\%$

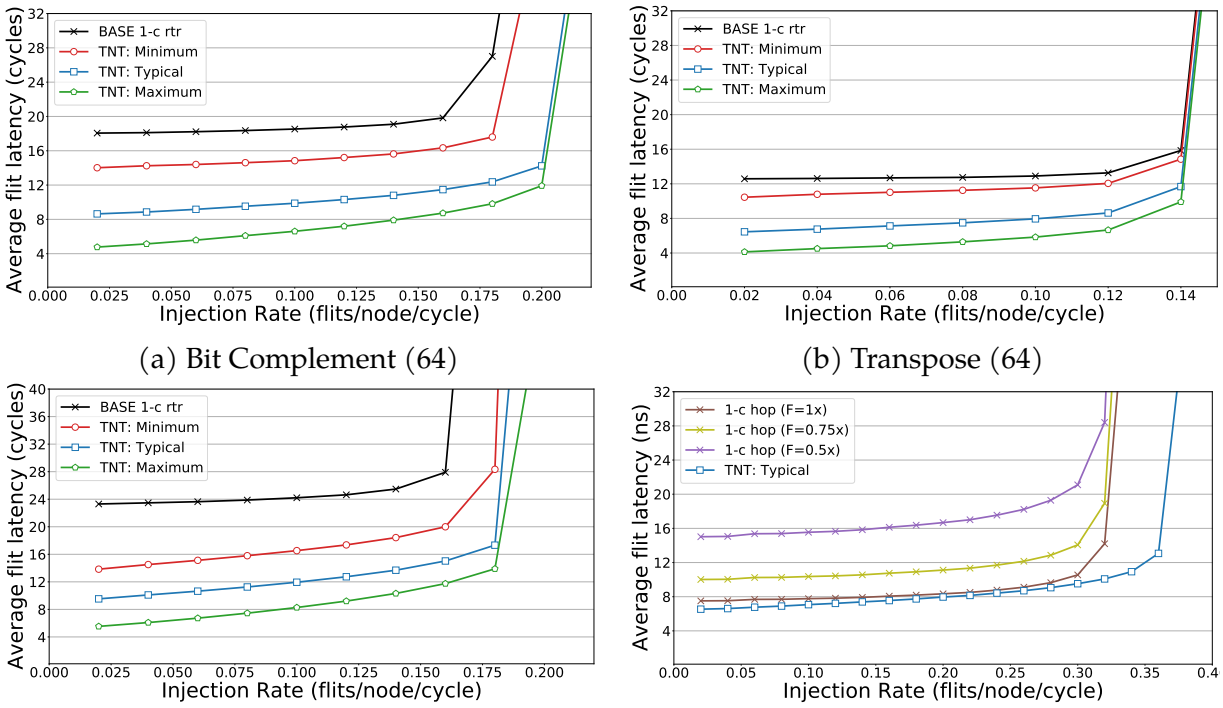
when designed for low traversal capability and $< 10\%$ even at high capability.

Area: Area overheads of TNT range from 0.15% to 5% depending on the wire traversal capability of the design (ranging from 1-12mm per cycle) These are worst-case estimates since it is possible for wiring to be overlapped by NOC nodes/tiles, thus having less impact on the overall area [108, 159, 4].

9.4 Evaluation

Section 4.5 discusses the evaluation methodology for TNT.

9.4.1 Synthetic Traffic



(a) Bit Complement (64)

(b) Transpose (64)

(c) Uniform Random (256)

(d) UR (64): TNT vs 1-cycle per hop

Figure 9.8: Flit latency analysis for TNT

Latency vs Injection Rate: Figures 9.8a and 9.8b show the average flit latency for increasing injection rates (until saturation) for two different synthetic traffic, for a 64 node system. TNT performs consistently better than the baseline, achieving latency reductions of up to 25% for TNT:Minimum, up to 60% for TNT:Typical and up to 74% for TNT:Maximum. TNT is also able to achieve throughput equal to or greater than the baseline mesh, even in scenarios like Transpose which is often adversarial for many optimizations under DOR [73].

Scaling to a larger mesh: Next, we analyze TNT on a larger 256-node mesh for uniform random traffic in Fig.9.8c. TNT is able to achieve latency reductions of up to 80% across the different scenarios, resulting in across-the-chip traversal in as low as 6 cycles. Clearly, TNT is able to scale well with increasing cores - it provides even closer to ideal latencies as the hops for flit traversal increases in larger meshes.

Comparing against an optimistic single cycle design: In Fig.9.8d we compare TNT:Typical against an optimistic design which allows for 1 hop (i.e. router + ST + LT) in a single clock cycle - shown as *"1-c hop"* in the figure. This design is optimistic because we do not model router overheads corresponding to such a design. Further, we model 3 scenarios of running this design at 1x/0.75x/0.5x of the baseline/TNT frequency respectively. For fair comparison, reducing the frequency of *"1-c hop"* is intuitive - combining router + ST + LT in a single cycle can significantly impact the clock frequency by reducing it to as low as 0.5x the original. From figure, it is clear that TNT:Typical outperforms all three *"1-c hop"* scenarios. Despite TNT's longer routing time (shown in Fig.9.2), TNT's gains from multi-link per cycle traversal (thanks to transparent flow and exploiting heterogeneity) outweigh the *"1-c hop"* design's gains from fast routing. Further, as the frequency of *"1-c hop"* is realistically reduced, the benefits of TNT become far more significant.

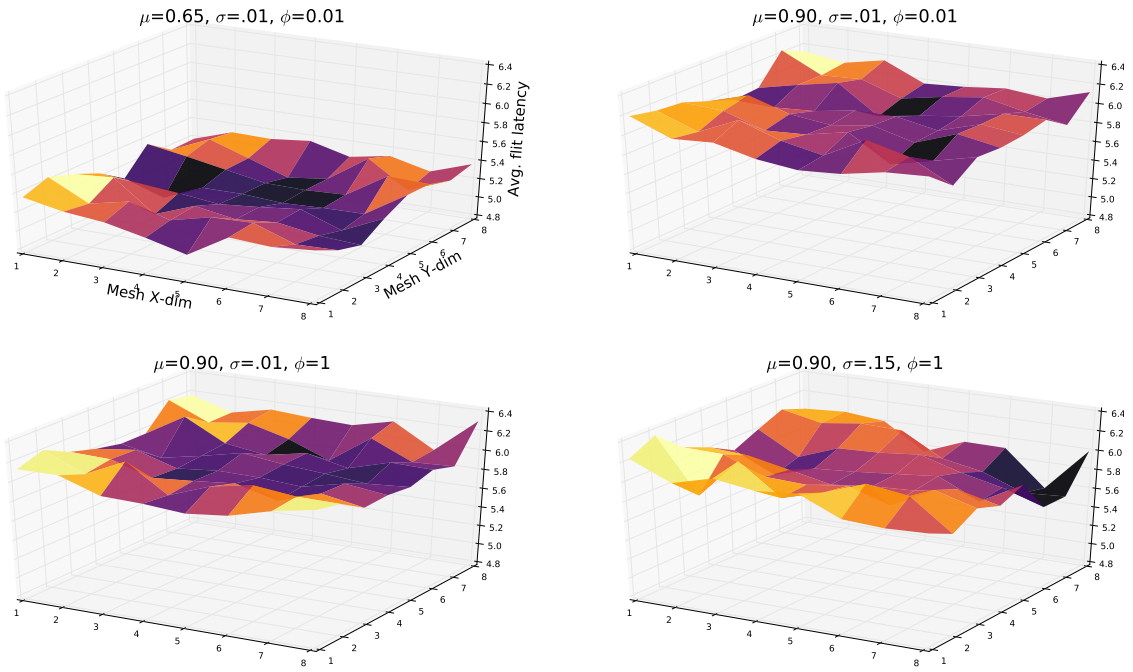


Figure 9.9: UR: 5-hop latency for different variation

Effect of variation: Fig.9.9 shows 4 different variation scenarios (with appropriate μ, σ and ϕ) portraying the average flit latency with uniform random traffic assuming a homogeneous $\eta = 0.5$. In the graphs, the xy-axis show the mesh - each node in the 2D plane is a starting router for all possible 5 hop routes. The value along z-axis $z = f(x, y)$ shows the average latency across all such 5-hop routes starting from that particular source router (X,Y). Within each graph, a darker purple shade indicates a relative lower flit latency while a lighter yellow share indicates a relative higher latency (note that this scale varies across the graphs).

① *Top Left (TL):* shows high mean exploitable guardband ($\mu = 0.65$) resulting in 20% lower average flit latency compared to the other scenarios. ② *Top Right (TR):* shows low mean exploitable guardband ($\mu = 0.9$) resulting in higher average flit latency. ③ *Bottom Left (BL):* shows high spatial correlation ($\phi = 1$) compared to TR ($\phi = 0.01$). The central

area of the mesh in BL constantly experiences lower latencies while corners are high - latencies are more random in TR. ④ *Bottom Right (BR):* shows high deviation from mean ($\sigma = 0.15$) compared to BL ($\sigma = 0.01$). In BR, some areas are very bright and some are very dark, highlighting significant difference in latency across the chip (15%), while they are more evenly colored in BL (within 5%).

η/W	5%	10%	20%
0.4-1	0	0	2.8
0.2-0.4	0	4	10.6
0.1-0.2	1.7	4.9	11.4

(a) LR: Window vs η

IR/W	5%	10%	20%
0.01	0.3	0.59	1.18
0.1	2.7	5.5	10.9
0.3	6.3	12.5	25.2

(b) ILT: Window vs Injection

Table 9.2: Analysis of blocked requests (%)

Timing Safeguards: Recall (from Sec.9.2.3) that timing violation windows are employed by both the ILT Safeguard and the LR Safeguard. If timing measurements are less precise and/or if the chip experiences higher fine-grained heterogeneity, larger windows should be employed. First, in Table 9.2a we analyze different window sizes (W: 5% - 20%) for LR Safeguard and their NOC impact in terms of the % of LRs that are blocked at the routers (over different η). The table shows that a shorter window has no/minimal impact on the number of buffered LRs, across η. With very conservatively large windows (eg. 20%), buffered LRs can increase by around 10% at low η. Second, Table 9.2b shows how different window sizes for ILT Safeguard impact the % of blocked LRs (over different injection rates). At lower IR, there is negligible increase in LR buffering, irrespective of the window size. At higher IR but smaller window sizes, there is minimal increase in LR buffering. These numbers increase to 25% at very conservative timing windows and near-saturation IR. Thus, reasonably large window sizes can usually be employed if required, without

having a significant impact on TNT benefits. Note: this analysis is very pessimistic because it assumes that in the absence of safeguard-based conflict, the LRs will necessarily flow through transparently. This is not always the case, especially at high IR and/or low η.

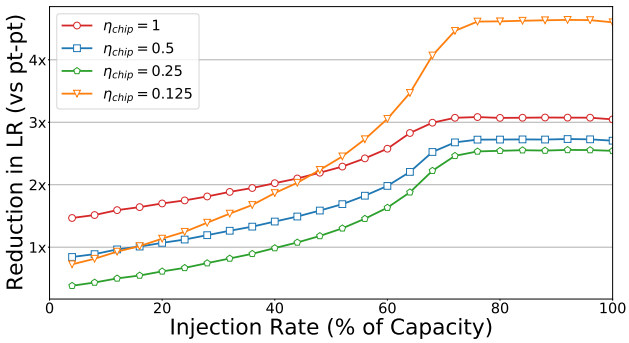


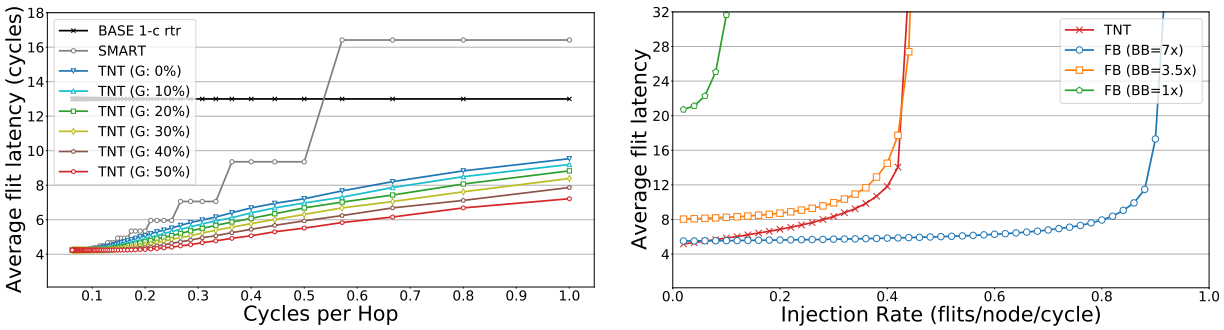
Figure 9.10: Reduced lookahead conflicts (vs pt-pt)

Conflict reduction from LN mesh: Under high injection rate, dedicated pt-pt signaling results in an enormous growth in the number of conflicts among signaling requests, many of these conflicts being duplicated across routers for the same flit's traversal. Such conflicts are significantly reduced via the TNT Lookahead mesh. Fig.9.10 shows conflict reduction for TNT under different η, for UR traffic. TNT shows conflict reduction by as much as 4.5x compared to pt-pt signaling.

9.4.2 Comparisons with prior work

SMART: SMART [123] provides the illusion of dedicated physical express channels. It drive signals up to multiple integral hops (called a *smart-hop*) within a single-cycle. While SMART is a significant step towards achieving ideal T_{wire} delay, it has key differences / limitations compared to TNT:

Heterogeneity: First, note that SMART does not exploit heterogeneity in the NOC and is focused on NOCs which uniformly have a higher hops-per-cycle capability.



(a) TNT vs SMART

(b) TNT vs FB

Figure 9.11: TNT Comparisons

Latency: The key point to note is that SMART transforms the quantization granularity of network traversal from a traditional 1-router hop to a multi-router hop. The underlying limitation is that quantization still exists, resulting in hurdles to lower latency, especially at higher η. In Fig.9.11a we compare the average flit latency of UR traffic for TNT and SMART, on a homogeneous design with an $\eta = 0.1 - 1$ range. The injection rate is fixed at 25% of maximum. In order to mimic heterogeneity (which only TNT can exploit) we plot TNT with different exploitable delay guardbands ranging from 0-50%. Higher guardband implies higher exploitable heterogeneity. TNT and SMART perform similarly at very low η. At high η, SMART performs similar to the baseline or worse - on the other hand, TNT is able to benefit from considerable latency reduction. Overall, TNT can provide 1.6x - 2.3x flit latency reduction over SMART.

Modularity: SMART uses a dedicated point-to-point network within a *smart-hop* neighborhood and sends independent signaling requests across these dedicated links to each path router for every *smart-hop*. SMART's complexity is shown in Table 9.1 and it is clear that its logic complexity, number of links and number of signalling requests grow enormously as the wire traversal capability increases (i.e. lower η) . Related to the point-to-point signaling,

false negatives occur in SMART when the lookahead signal independently reaches a router but the flit is forced to prematurely stop earlier due to conflicts. Thus, TNT is more scalable and has lower overheads. The energy overheads from TNT are lower than those of SMART by almost 50% at lower η.

Flattened Butterfly: We compare TNT against a Flattened Butterfly topology [119] with concentration of 1. Each FB router has 29 ports - (7 I/O in each direction + NIC port). Further, it has dedicated single-cycle links to every node in both dimensions. We assume that the FB router delay is 1-cycle - this is an aggressive assumption, especially since the SA stage needs to perform 15:1 arbitrations (usually assumed to be 4-cycle [108]). We use 8 VCs per port with virtual cut-through so as to allow more buffer resources for FB. We perform analysis on 3 configurations of FB wherein the Bisection Bandwidth (BB) is 1x, 3.5x and 7x that of TNT. Results are shown for Uniform Random traffic in Fig. 9.11b.

At BB=1x, FB loses to TNT, both in terms of latency and throughput, due to heavy serialization delay. At BB=3.5x, FB is able to match the throughput of TNT but TNT is able to achieve up to 40% lower latency. At BB=7x, FB matches or performs at lower latency than TNT and has higher potential throughput. Note that in order to achieve this, the FB incurs heavy overheads in terms of wiring, area and power. The complexity is shown in Table 9.1. As derived with the help of prior work [123], the radix-29 FB router at BB=3.5x incurs an area and power overhead of 9x, 5x respectively over TNT. A better solution would be multiple TNT-based meshes - achieving better latency as well as scalable bandwidth.

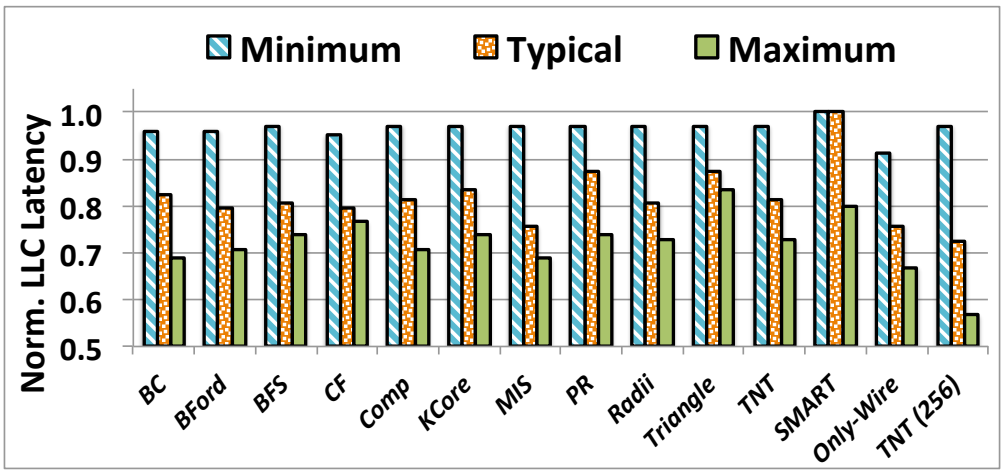


Figure 9.12: TNT Benefits: LLC latency

9.4.3 Full-system Analysis

Latency Impact: First we look at the average LLC access latency in Fig.9.12 and analyze the latency reductions from TNT. Results are normalized to the traditional mesh baseline. We also show mean comparisons to SMART and an ideal only-wire delay network, as well as TNT benefits on a 256-node system. TNT's benefits for "Minimum" scenario are negligible (3%) - this is not unexpected, due to no exploitable wire capability. We see considerable latency reduction with TNT of 19% in "Typical" and 27% in "Maximum". Thus TNT is within 5% of the ideal Only-Wire case for 64 nodes. In comparison, SMART is unable to achieve LLC latency reduction in the "Minimum" and "Typical". SMART is designed for homogeneity in wire traversal capability. Thus it is forced to be conservative in the "Typical" case. SMART is able to achieve 20% reduction in the homogeneous "Maximum" scenario. For 256 nodes, TNT is able to achieve a 27% reduction in "Typical" and 43% in "Maximum".

Performance: Fig.9.13 shows the reduction to application runtime from TNT. Results

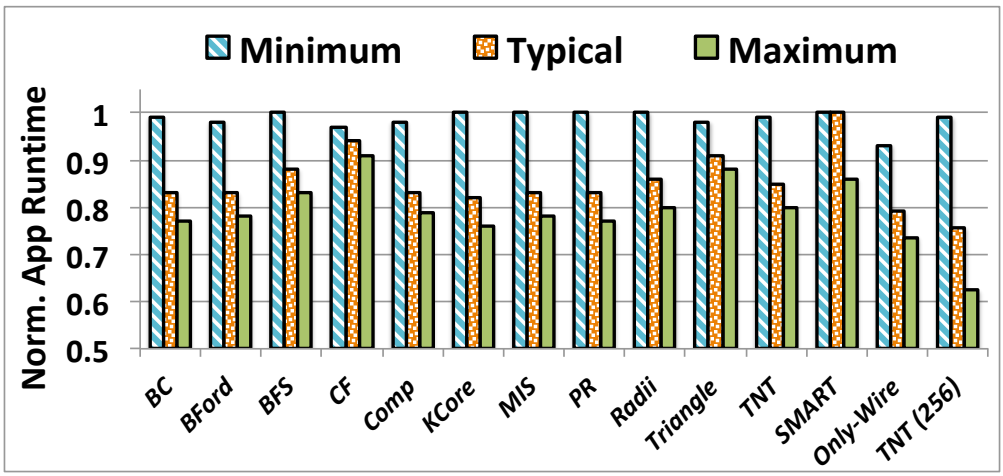


Figure 9.13: TNT Benefits: Runtime

are intuitive based on the LLC access latency reduction discussed above. TNT is able to achieve mean runtime reductions of 1%, 15% and 20% for the "Minimum", "Typical" and "Maximum" scenarios. TNT is again within 5-6% of the ideal case. On the other hand, SMART achieves no reductions in "Minimum" and "Typical" while 14% in "Maximum" scenario. Scaling TNT to 256-nodes increases benefits to 2%, 24% and 38% respectively.

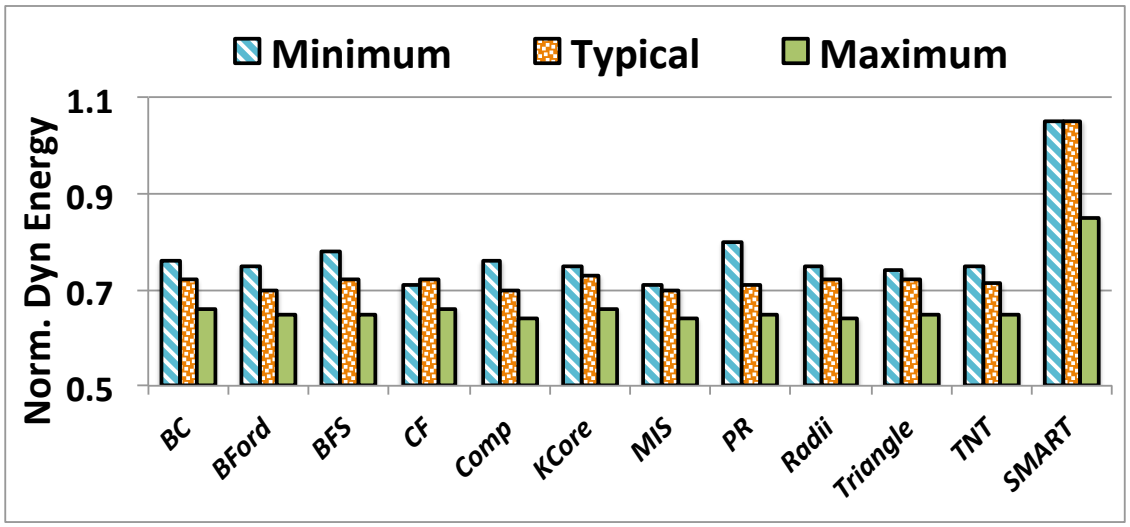


Figure 9.14: TNT Benefits: Dynamic Energy

Network Dynamic Energy: Fig.9.14 measures the total dynamic energy of the network consumed with TNT in comparison to the baseline and SMART. Results are normalized to

the baseline. TNT is able to achieve a mean energy reduction of 24%, 29% and 35% in the three scenarios. Interesting to note is that TNT achieves considerable energy savings even in the "Minimal" case due to reduced buffering at intermediate routers, even though there is no significant latency reduction. In comparison, the energy increases in SMART for the first two scenarios, due to no latency reduction but continued buffering, while it reduces by 15% in "Maximum".

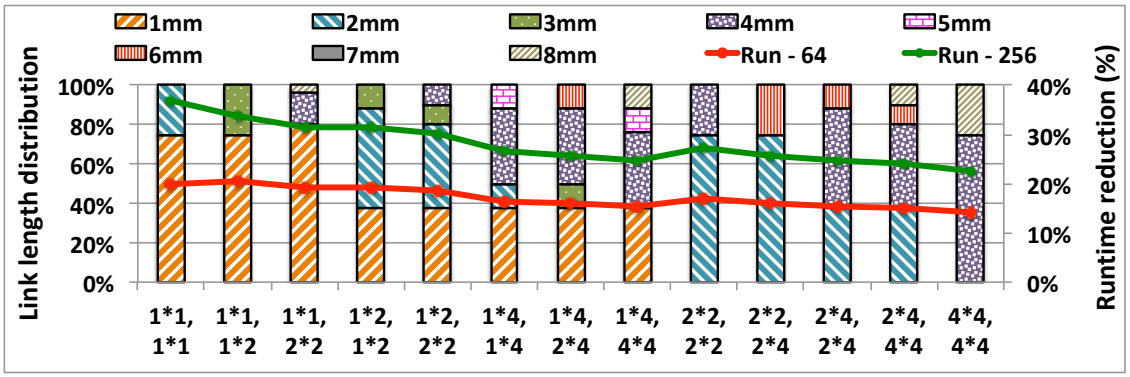


Figure 9.15: Design Space Exploration

Design Space Exploration: Finally, we sweep through a broad design space in terms of core and memory controller dimensions. For each design, we build (by hand) floorplans suited to the diamond shaped MC distribution (similar to Fig.3.8) and estimate the distribution of link lengths. It is intuitive that a distribution with a large % of short links would enjoy greater TNT benefits. Fig.9.15 shows 13 different designs (along the x-axis). Note that any "X*Y, W*V" design point refers to a core size of $X * Y mm^2$ and a MC of $W * V mm^2$.

For instance, the "1*1, 1*1" design has all edge dimensions of 1mm. For this, the diamond-MC based floorplan results in 75% of links being 1mm and 25% of the links being 2mm. The figure also plots the runtime reductions from TNT (vs. baseline) when each of these floorplans are applied to a 64-node system and a 256-node system. This is shown on the

secondary axis. Runtime reductions in the range of 12% - 20% are observed for the 64-node system while a broader range of 22%-38% is observed for the 256-node system.

9.5 Chapter Summary

Scalability of current-day CMPs is largely dependent on the ability to communicate efficiently. But we are far away from near-ideal latencies due to limitations imposed by the inherent traversal quantization of traditional modular NOC designs, as well as the physical heterogeneity of real systems. We proposed TNT to overcome these challenges - achieving end-to-end network traversal in a single pass, at the best physically capable wire delays, while performing only neighbor-to-neighbor control interactions. Overall, TNT aggressively exploits wire capability to make tremendous strides towards near-ideal communication in the exascale era.

10 CONCLUSION AND REFLECTIONS

10.1 Dissertation Summary

This dissertation targets a broad theme of computer architecture research - building vertically integrated computing systems. The philosophy of vertical integration is to achieve better exchange of information between different layers of the system stack, so that each layer can be designed more efficiently. In recent years, as device characteristics have limited the gains within each specific computing abstraction layer, it is key to look to vertical integration for building sophisticated computing systems.

This philosophy is particularly evident in the abundance of hardware accelerators designed today. Designing from 'top-down' with a single application or a single domain in mind dissolves many abstraction layers and results in more efficient architectures to execute that application/domain. While the benefits of 'top-down' integration to build application-specific hardware is clear, it is important to not dismiss the opportunities in 'bottom-up' integration.

With this view, this dissertation targets integration from the gates up in classical computers - enabling upper layers of the computer system abstractions to conveniently exploit the lower hardware-level circuit/device characteristics. Our work specifically focuses on breaking through abstractions of the computer system's clock. Understanding and exploiting clock characteristics is exciting and has tremendous potential, especially for two reasons. First, the clocking system is one of the most power-consuming circuits on a chip - thus understanding its power distribution can have considerable impact on execution efficiency.

Second, quantizing to clock cycle boundaries is fundamental to synchronous computing. Any form of quantization, if conservative, is a detriment to execution performance - thus understanding clock utilization characteristics, both in compute and in communication, can significantly further execution performance.

The research proposals discussed in this dissertation have targeted multiple characteristics of the clock: clock hierarchy power distribution, environment (PVT) impact on the timing guardband, computations' impact on clock cycle utilization (both accurate and approximate), as well as 'hops per clock cycle' wire traversal capability in the NOC.

The proposals have been implemented over multiple execution substrates: out-of-order cores, spatially distributed architectures and on-chip networks. Further, they have targeted a variety of application domains: general purpose, low precision, machine learning, approximate and graph applications.

Apart from exploiting clock characteristics, they have also made other significant architecture and system-level contributions such as: a neural network based control mechanism for resource reconfiguration + DVFS, identifying optimal granularities for reconfiguration, novel contributions to instruction scheduling in out-of-order cores, fine grained memory approximation, gradient-descent based hardware-cognizant approximation tuning which provides synergic support to multiple forms of approximation, as well as novel contributions to asynchronous routing in the NOC.

10.2 Reflections

In this section, I share observations related to the projects pursued in this dissertation and thoughts on extending them.

10.2.1 Non-classical solutions to classical architecture challenges

Classical architecture research is still very important today. My takeaway from interactions with industry is that every few % points of general-purpose improvement is worthy of exploration - there are still too many important applications (or phases of applications) which are not suited to any of the 100s of accelerators in the wild. Despite this, it is my opinion that we (i.e. the architecture community) are stuck in a classical conundrum. On the one hand, classical solutions to classical architecture challenges are often unable to produce worthy gains - this is not surprising considering that architects have been squeezing the maximum efficiency out of our computer systems for decades. On the other hand, non-classical solutions to classical architecture challenges (which is how I view this dissertation) can provide considerable benefits, but there is significant push-back to their acceptance as they are deemed too non-classical to be suited to the legacy of classical architecture. Something has to give - and I believe it has to be the latter. As a community we need to accept that there is a critical need for non-classical solutions to classical general-purpose computing, if we want to continue the aggressive pursuit of efficiency improvements. I'm hopeful that with the current momentum for open-source hardware, there will be more aggressive solutions pursued across the abundant varied

designs and implementations, both in industry and academia, resulting in non-classical solutions to classical challenges becoming the norm and not the outlier.

10.2.2 Gates-up Vertical Integration

In this dissertation, we pursue vertical integration from the gates up in classical computing - exploiting circuit level characteristics across the computing stack. While we focused on the clock abstraction there are multiple other avenues as well. Variation-focused vertical integration has been popular over the years and continues to be critical moving forward. Both scale-down (i.e. lower technology nodes) and scale-out (i.e. more processing nodes on-chip/off-chip) means that there will be considerably diverse variation characteristics across the computing system - exploiting these characteristics is a must for efficient better-than-worse-case computing.

Other circuit-level characteristics are also worthy of exploration - one example is capacitance. A lot of existing classical optimizations are already focused on moderating the effect of capacitance on power consumption - reducing the switching activity to reduce effective capacitance, building smaller structures (eg. buffers / bypass logic) etc. Still, it seems like there is much more scope for capacitance-aware static/dynamic optimizations.

There is also potential for high impact by pursuing gates-up vertical integration in domain specific accelerators. On the one hand, the support for variable application / data characteristics atop these domain specific accelerators means that there might be considerable better-than-worst-case guardbands to exploit. On the other hand, the control of the entire hw-sw stack for these accelerators, might make designing vertically integrated

solutions more feasible with less overheads.

10.2.3 CHARSTAR

Granularity of reconfiguration: One of the important questions we attempt to answer in CHARSTAR is what are the right granularities of reconfiguration across the different dimensions of reconfiguration. While there are constraints imposed by circuit-level overheads like power gating / wake-up latency etc, there is still a large window of granularities to choose from. These granularities themselves can be reconfigurable depending on the characteristics of a particular application and the states of reconfiguration being explored.

Criteria for resource management: In CHARSTAR, we allocate resources in a spatial architecture with the goal of minimizing clock-tree power consumption (while meeting performance targets). There are many other parameters that can affect resource allocation/reconfiguration, such as inter-tile communication latency, thermal hot spots, etc. In designs with distributed shared memory, data reuse across tiles is another influential factor. Further, if the tiles themselves are heterogeneous, then some tiles could be prioritized over others even if it might be detrimental to clock power, communication latency, hot spots and so on. Designing a control mechanism that is cognizant of all these aspects and many more, could achieve the best overall resource management efficiency.

Other spatial architectures: In CHARSTAR we focused on a tiled microprocessor, CRIB. The idea of reconfiguration / resource allocation in a spatially distributed architecture extends much beyond tiled microprocessors. It is especially worthy of exploration in GPUs which have 100s of shader cores which can suffer from under-utilization. For example,

training/inference on current-day deep neural networks (such as SQNNs [166] which sometimes have sequential dependencies) leads to poor shader utilization over phases of training/inference. Using the right set of shader cores in these phases has potential for considerable improvements to energy efficiency. Similarly accelerators for machine learning, graph processing and beyond are usually spatially distributed but often experience phases of sparse utilization. Again, there is potential for multiple hardware characteristics aware resource management.

ML-assisted systems: In CHARSTAR, we used a small neural network based predictor to predict the resource configuration for each application phase. Intel's recent paper [211] compares against CHARSTAR's predictor and builds even more accurate prediction mechanisms for resource reconfiguration. Adding more constraints, metrics to the reconfiguration control mechanism would require more sophisticated ML prediction schemes, also creating more granularity of reconfiguration tradeoffs. ML-assisted systems are important moving forward, even in traditional out-of-order cores, beyond existing uses in branch prediction, prefetching etc. Other prediction / speculative mechanisms within the core can also benefit from ML. Similarly, considering the exponentially large parameterized system and workload configuration space, ML based techniques can help in fast and efficient system design.

10.2.4 SlackTrap

DFG-level quantization: A key aspect to our transparent flow based slack recycling mechanisms is that the transparent data flow is quantized to synchronous boundaries, and it is

these synchronous boundaries which interact with the computing events / components outside the transparent dataflow engines. This keeps design complexity minimal - system components which are outside the transparent dataflow path are oblivious to the existence of transparent dataflow. Within these transparent DFGs (data flow graphs), slack estimation and accumulation is performed dynamically from one operation to another - which provides flexibility for dynamically building DFGs on the fly. It is possible that these transparent DFGs can be scheduled at the coarse granularity of the DFG itself. Building mini-graphs with prior slack estimations could mean that an N-node DFG can be scheduled as a unit, and expected to complete in M cycles, where $1 - \frac{M}{N}$ is the average slack across the DFG. This is especially possible when slack focuses only on PVT variations - which can be expected to remain fixed across the course of a DFG's execution.

Slack recycling in non-traditional processing: The advantage of slack recycling in a spatially distributed architecture with a large number of compute nodes is that a sequence of operations (i.e. a DFG) can be eagerly scheduled onto available nodes ahead of time, and slack can be easily recycled across the DFG. This might be especially attractive in ML accelerators with multiple compute nodes (eg. multiply-and-accumulate units) where activations/weights flow from one node to the next as a stream. Further, in these scenarios there can especially be a lot of slack available when processing at low-precision.

10.2.5 REDSOC

CISC-y quantization for slack recycling: As discussed previously, transparent dataflow is quantized to synchronous boundaries. Thus, N-operation DFGs can be identified whose

execution consumes M cycles. If some flexibility reduction in execution is tolerable, these N operation DFGs can be scheduled as macro-ops or a trace, whose total execution time is estimated prior and stored in some instruction-addressed structure. This may relieve the need for aggressive scheduling optimizations such as grandparent based scheduling, slack accumulation / tracking etc. This can be thought of as a CISC-style synchronous quantization to RISC instructions and is worthy of more exploration.

Critical instruction scheduling: In REDSOC, slack is recycled in a greedy manner whenever possible. When slack is recycled, a sequence of operations can complete in a lesser number of cycles (compared to the traditional synchronous baseline). This results in performance benefits only if the instruction sequence lies in the critical path of the application. There is future potential to identify critical instruction sequences, so that slack is recycled only along those sequences which can actually provide performance benefits. REDSOC's skewed selection mechanism (prioritizing non-speculative instructions) can possibly be optimized to support critical instruction selection.

Slack in bypass: In REDSOC, we focused on slack in the execution units alone. There is also potential for identifying slack in the bypass paths of execution units. With a large number of execution units in big cores, units which are farther away might utilize the entire clock cycle for bypass whereas closer units would have greater bypass slack. Further, critical operations can be scheduled closer to one another, allowing for higher bypass slack, more slack recycling and increased performance benefits from acceleration.

Other slack opportunities in the core: Quantizing to clock cycles is not limited to the execution units - it is prevalent across the entire synchronous processor core. Thus, there is potential for slack recycling in other stages of the pipeline. These include decode,

cache access, register file access etc. In order to be able to recycle slack, there has to be some intelligent mechanism which eagerly allows these operations to be ready for slack recycling and also to track and accumulate slack. Prior works have shown the prevalence of considerable slack across multiple stages of the processor - even from only static analysis. Thus it is also interesting to think about rebuilding the entire processor with a motive of transparent dataflow while maintaining some semblance of synchronous quantization.

IOT: More aggressive slack recycling is possible if we have end-to-end control over the hw-sw stack. Thus, slack recycling might be attractive in the IOT domain. Examining slack at ultra-low voltages and frequency, as well as analyzing error vs performance trade-offs (like SHASTA) has considerable potential. At the compiler level, encoding operations to increase slack opportunity as well as identifying DFGs for slack recycling could be possible. Finally, our REDSOC results showed the slack recycling was most attractive in large out-of-order cores. On the other hands, IOT systems might employ small out-of-order (or in-order) cores. It would be interesting to analyze how to add minimal hw structures to these small cores to make them more amenable to slack recycling.

ILP/TLP vs Slack recycling: Slack recycling requires the availability of idle functional units. Dependent operations are brought eagerly to the idle functional units, so that they can consume the producer/parent operation's outputs as soon as they are ready. The availability of idle functional units depends on the core, the application's instruction level parallelism as well as thread level parallelism (multi-threading). We envision that a core employing both SMT and slack recycling can choose to pause SMT if a particular thread is performance critical and it can benefit from more aggressive slack recycling. This leads to interesting single-thread performance vs thread-level-parallelism trade-offs.

Encoding for better slack: Slack recycling can potentially benefit from data encoding schemes. The amount of slack in compute operations is dependent on the length of the critical path of the computation. For example, in an add operation, the longer the chain of carry overs, the more is the computation time and less is the slack. An effective encoding scheme could reduce, say, a long sequence of 1s, which could reduce the length of the carry over chain, thus increasing computation slack.

Asynchronous solutions in synchronous simulators - a path to realistic design: A common question that comes up in the context of this research is how to simulate these proposals - since all architecture simulators are cycle-synchronous i.e. events are quantized to clock cycles. This often leads to questioning if asynchronous solutions are therefore worthy of exploration at all. In my mind, identifying and solving the challenges in simulating asynchronous slack recycling ideas in a synchronous simulator, was key to coming up with a realistic pseudo-asynchronous solution, i.e. transparent dataflow with synchronous boundaries, which is suited to traditional processor designs. A possible takeaway from this is that, when the path from an aggressive proposal to its realistic microarchitecture (or otherwise) implementation seems challenging, a good idea would be to work through simulating the proposal in some framework which bares resemblance to real implementation. Effectively achieving the proposal in simulation, can clear up many questions about how the proposal would look in realistic implementation.

10.2.6 SHASTA

Other approximation forms: The tuning mechanism proposed in SHASTA enables efficient hw-cognizant tuning for fine-grained approximation. In SHASTA, we used this to optimize timing approximation and load value approximation, but other forms of approximation can be supported as well. For example, precision approximation via gating portions of the datapath as well as voltage scaling are two forms of compute approximation, while memory compression might be used as a form of memory approximation. Software approximation techniques can also be used such as loop perforation etc. Some of these forms of approximation can only be controlled at coarse granularities over space and time. Thus this information needs to be incorporated into the tuning mechanism - this would restrict the state space explored by the tuning in that particular tuning epoch.

Other approximation challenges: Identifying the approximate amount of approximation for different approximation variables is performed efficiently by the gradient-descent based hw-cognizant tuning mechanism in SHASTA. Other major approximation challenges include identifying the right variables for approximation itself - there are both static and dynamic opportunities to explore along this path. Another big hurdle in approximate computing is quality monitoring and error management - providing guarantees on output quality is necessary for approximate computing to be more widely accepted in both research and industry. Building approximate systems similar to systems for machine learning might be promising in this regard.

10.2.7 TNT

Variation-aware NOCs: As we build larger systems with 100s of cores (on a single chip / wafer etc), there is bound to be considerable variation across the system. Considering that the entire NOC might usually be designed to function as a single voltage/frequency domain, it often has to be designed conservatively to be suited to worst case requirements. Thus exploiting variation characteristics is especially important in the NOC. Beyond the bare-wire delay proposal that is explored in TNT, there are multiple other opportunities as well. For example, routing which is aware of timing characteristics / fault tolerance etc. across the NOC can allow for more aggressive as well as reliable designs. Exploiting dynamic variation characteristics across the NOC is particularly interesting and challenging. High microarchitectural activity can lead to voltage droops and thermal hotspots. Routing decisions which avoid such scenarios can lead to better system execution efficiency.

Slack recycling without time stamps: In our proposals, slack recycling was enabled by means of time stamps. Slack estimations in both the NOC as well as in functional units is expected to be performed post-fabrication and can also be re-estimated over time (to account for dynamic variations). It would be interesting to explore direct computation/communication completion detection without the use of time stamps or static slack estimation. Circuits geared towards bit transition detection or current monitoring can potentially be used to deem if a computation is complete, or some data/bits have been communicated. This self-estimated completion detection can then be used to pursue more efficient transparent dataflow but requires deeper understanding of circuit-level challenges.

Data slack opportunities in the NOC: Communication time in the NOCs is also data

dependent. Bit transitions and their impact on cross-talk / capacitive coupling also influence traversal time. Apart from this, compression / encoding schemes / approximation to speed up data traversal in the wire is also worthy of exploration.

Asynchronous routing / time stamps: One of the key enablers to TNT's transparent traversal is the idea of asynchronous/combinational routing i.e. multiple routings are performed in a single clock cycle. The challenge here is that multiple requests from different directions can arrive at a router in a clock cycle and they can arrive at both near or far apart time instants. Moreover, combinational flow also creates timing violations / challenges. Our approach of using time stamps to solve these challenges proved to be a simple and effective solution to solve many of these challenges. Going forward, I believe that the use of time stamps has a lot of potential in the pseudo-asynchronous domain, not only for routing but in many other areas. Time stamps or their equivalent (but quantized to clock cycles) have also been explored in other areas of architecture research like cache coherence. Using time stamps in new areas, as well as extending existing synchronous time stamp proposals to asynchronous domains are worthy of exploration.

10.2.8 Timing concerns

The common criticism of slack recycling (in specific) and proposals exploiting timing guardbands (in general) is that the resulting timing verification / closure challenges are too considerable for realistic adoption. Our response is three-fold. First, over the past two decades we have seen a steady decline in performance improvements of conventional processors. Thus it is imperative that we redesign some, if not all, design methodologies

from the ground up, if we want a new lease of life to processor performance growth. In this spirit, one-time challenges, such as increased complexity in timing verification, would need to be embraced. Many of these challenges and feasible solutions are already being consistently explored in EDA/CAD research. Second, slack measurements, which are the key enabler to our proposals, are already performed in conventional post-fab timing verification. We expect them to be performed in a more detailed manner (for example, capturing per-link wire delay in TNT, or width-based / operation-based data slack in REDSOC) but the setup should be similar to what already exists. Third, our proposals perform some aspects of timing analysis and timing guardbanding on the fly. This is especially visible in TNT wherein, requests are disallowed from reaching routers near the clock edge, as these may lead to timing violations. Further, flexible timing windows can be incorporated depending on the confidence in timing measurements.

10.3 Related Ongoing Work

Learnings from this dissertation's research contributions have influenced our ongoing research proposals. For example, the application of memory and compute approximation in conjunction in SHASTA has partially influenced our recent work on Value Locality based Approximation [202]. Further, the gradient descent based tuning mechanism in SHASTA has benefited our ML-assisted workload design proposal, MicroGrad [186].

10.4 Future Directions

The broad theme of our research, targeting bottom-up vertical integration, has created a foundation for plenty of future work in classical as well as emerging computing domains. Note: many aspects of these future directions are also discussed prior in the reflections in Section 10.2

Classical Computing:

1. *Internet of Things:* There is potential to extend our proposals for clock cycle slack recycling into the IoT / Ultra Low-Power domain. With IoT's reliance on intermittent harvested energy sources, there is abundant value for fine-grained efficiency vs error-tolerance trade-offs.

2. *Clock Abstraction:* Our proposals have shown that considerable circuit potential is discarded by abstractions constrained by worst-case estimations, especially in the clock context. Multiple other core-wide opportunities to exploit clock characteristics (eg. cache / register-file), could unearth handsome benefits.

3. *Approximate Computing:* SHASTA has shown that there is significant scope for robust approximation by designing approximate systems similar to systems for ML. There is potential for much future work in this direction

4. *Intelligent Resource Management:* There are many factors that can affect resource allocation and configuration in a tiled or spatially distributed computing system such as tile heterogeneity, inter-tile communication latency, thermal hot spots, data/memory

sharing etc. Designing a control mechanism similar to CHARSTAR that is cognizant to all these aspects and more has considerable potential. This can especially have a significant impact in the coming age of accelerator level parallelism [89]

Quantum Computing:

There is abundant potential in applying cross-layer learnings from classical computing to the quantum domain, which, from a computer architecture perspective, is at a crucial juncture [144] between needing cross-layer optimization but also requiring feasible abstractions for wider and scalable use. Classical research themes of suitably exposing hardware circuit characteristics to the rest of the computer system and managing interactions within and between different computing layers, fit well with important quantum computing goals. Some opportunities are discussed below:

1. *Handling Errors:* Research is needed to track errors occurring through a quantum program [144]. Moreover, identifying the error tolerance of different logical qubits and different parts of a quantum application/circuit, can improve error-tolerant mapping onto physical qubits. Towards this, there is potential to build upon SHASTA's hardware-cognizant application error tracking.

2. *Variability:* Intelligent mapping is needed [144] to tackle multiple quantum constraints such as: costs of swaps, decoherence times, diversity in executions and so on. SlackTrap and REDSOC's design for tracking dynamic circuit characteristics, as well as CHARSTAR's dynamic resource control are good starting points.

3. *Communication:* As quantum systems scale, the qubits themselves will be divided into a hierarchy of interconnected domains [144], introducing intra-module vs. inter-module communication trade-offs. Decisions regarding the partitioning of applications into modules suited to intra-module and inter-module communication will gain importance. TNT and CHARSTAR provide a foundation to tackle questions on modularity.

10.5 Closing Remarks

In recent times, the advancement of classical computing has considerably slowed down. There is a need to drastically rethink classical strategies - which once upon a time enabled rapid growth, but are now limiting innovation. At the same time, aggressive new strategies are always challenging to implement in the real world.

We believe that the research proposals in this dissertation, while aggressive in their vertically integrated cross-layer approach, can be made compatible with industry-standard design flows. We will strive to advance these proposals in academia and industry, hopefully leading to real-world adoption and making a lasting impact on future computing systems.

BIBLIOGRAPHY

[1] Dennis Abts et al. "Achieving Predictable Performance through Better Memory Controller Placement in Many-Core CMPs". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: Association for Computing Machinery, 2009, pp. 451–461. ISBN: 9781605585260. DOI: 10.1145/1555754.1555810. URL: https://doi.org/10.1145/1555754.1555810.

[2] N. Agarwal et al. "GARNET: A detailed on-chip network model inside a full-system simulator". In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2009, pp. 33–42. DOI: 10.1109/ISPASS.2009.4919636.

[3] O. Akbari et al. "Toward Approximate Computing for Coarse-Grained Reconfigurable Architectures". In: *IEEE Micro* 38.6 (Nov. 2018), pp. 63–72. ISSN: 1937-4143.

[4] F. Alazemi et al. "Routerless Network-on-Chip". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 492–503. DOI: 10.1109/HPCA.2018.00049.

[5] David H. Albonesi. "Dynamic IPC/Clock Rate Optimization". In: *ISCA*. IEEE Computer Society, 1998.

[6] *Angstrom*. http://projects.csail.mit.edu/angstrom.

[7] A. Ansari et al. "Tangle: Route-oriented dynamic voltage minimization for variation-afflicted, energy-efficient on-chip networks". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 440–451. DOI: 10.1109/HPCA.2014.6835953.

[8] *ARM Core link interconnect*. https://www.arm.com/products/system-ip/corelink-interconnect.

[9] T. M. Austin. "DIVA: a reliable substrate for deep submicron microarchitecture design". In: *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. 1999, pp. 196–207. DOI: 10.1109/MICRO.1999.809458.

[10] Todd Austin et al. "Opportunities and Challenges for Better Than Worst-case Design". In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. ASP-DAC '05. Shanghai, China: ACM, 2005, pp. 2–7. ISBN: 0-7803-8737-6. DOI: 10.1145/1120725.1120878. URL: http://doi.acm.org/10.1145/1120725.1120878.

[11] T. Ayhan and M. Altun. "Circuit Aware Approximate System Design With Case Studies in Image Processing and Neural Networks". In: *IEEE Access* 7 (2019), pp. 4726–4734. ISSN: 2169-3536.

[12] J. Bainbridge and S. Furber. "Chain: a delay-insensitive chip area interconnect". In: *IEEE Micro* 22.5 (Sept. 2002), pp. 16–23. ISSN: 0272-1732. DOI: 10.1109/MM.2002.1044296.

[13] James Balfour and William J. Dally. "Design Tradeoffs for Tiled CMP On-chip Networks". In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. Munich, Germany: ACM, 2014, pp. 390–401. ISBN: 978-1-4503-2840-1. DOI: 10.1145/2591635.2667187. URL: http://doi.acm.org/10.1145/2591635.2667187.

[14] Jonathan Balkind et al. "OpenPiton: An Open Source Manycore Research Framework". In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 217–232. ISSN: 0362-1340. DOI: 10.1145/2954679.2872414. URL: https://doi.org/10.1145/2954679.2872414.

[15] L. Benini and G. De Micheli. "Networks on chips: a new SoC paradigm". In: *Computer* 35.1 (Jan. 2002), pp. 70–78. ISSN: 0018-9162. DOI: 10.1109/2.976921.

[16] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2008.

[17] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[18] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach". In: *MICRO*. IEEE Computer Society, 2008.

[19] T. Bjerregaard and J. Sparso. "A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip". In: *Design, Automation and Test in Europe*. Mar. 2005, 1226–1231 Vol. 2. DOI: 10.1109/DATE.2005.36.

[20] B. Bohnenstiehl et al. "KiloCore: A 32-nm 1000-Processor Computational Array". In: *IEEE Journal of Solid-State Circuits* 52.4 (Apr. 2017), pp. 891–902. ISSN: 0018-9200. DOI: 10.1109/JSSC.2016.2638459.

[21] Brett Boston et al. "Probability Type Inference for Flexible Approximate Programming". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 470–487. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814301. URL: http://doi.acm.org/10.1145/2814270.2814301.

[22] K. A. Bowman, S. G. Duvall, and J. D. Meindl. "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration". In: *IEEE Journal of Solid-State Circuits* 37.2 (Feb. 2002), pp. 183–190. ISSN: 0018-9200. DOI: 10.1109/4.982424.

[23] Anne Bracy, Prashant Prahlad, and Amir Roth. "Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth". In: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Portland, Oregon: IEEE Computer Society, 2004, pp. 18–29. ISBN: 0-7695-2126-6. DOI: 10.1109/MICRO.2004.15. URL: http://dx.doi.org/10.1109/MICRO.2004.15.

[24] D.M. Brooks et al. "Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors". In: *Micro, IEEE* (2000).

[25] David Brooks and Margaret Martonosi. "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance". In: *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. HPCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 13–. ISBN: 0-7695-0004-8. URL: http://dl.acm.org/citation.cfm?id=520549.822763.

[26] Mary Douglass Brown. “Reducing Critical Path Execution Time by Breaking Critical Loops”. AAI3187660. PhD thesis. Austin, TX, USA, 2005. ISBN: 0-542-29104-5.

[27] Alper Buyuktosunoglu et al. “A Circuit Level Implementation of an Adaptive Issue Queue for Power-aware Microprocessors”. In: *GLSVLSI*. ACM, 2001.

[28] Inc. Cadence Design Systems. *Best Practices for Implementing ARM Cortex(R)-A12 Processor and MaliTM-T6XX GPUs for Mid-Range Mobile SoCs*. 2013. URL: http://www.armtechforum.com.cn/2013/3_Cadence.pdf.

[29] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. “Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 33–52. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509546. URL: http://doi.acm.org/10.1145/2509136.2509546.

[30] Josep Carmona et al. “Elastic Circuits”. In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 28.10 (Oct. 2009), pp. 1437–1455. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2030436. URL: http://dx.doi.org/10.1109/TCAD.2009.2030436.

[31] James Charles et al. “Evaluation of the Intel® Core(TM) i7 Turbo Boost feature”. In: *IISWC*. IEEE. 2009.

[32] T. Chelcea and S. M. Nowick. “Robust interfaces for mixed-timing systems”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.8 (Aug. 2004), pp. 857–873. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2004.831476.

[33] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. “Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks”. In: *ISCA*. IEEE Press, 2016.

[34] Tianshi Chen et al. “DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning”. In: *ASPLOS*. ACM, 2014.

[35] X. Chen and N. K. Jha. “Reducing Wire and Energy Overheads of the SMART NoC Using a Setup Request Network”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (Oct. 2016), pp. 3013–3026. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2016.2538284.

[36] H. Cherupalli, R. Kumar, and J. Sartori. “Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 671–681.

[37] H. Cherupalli and J. Sartori. “Graph-based dynamic analysis: Efficient characterization of dynamic timing and activity distributions”. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, pp. 729–735.

[38] Hari Cherupalli, Rakesh Kumar, and John Sartori. “Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-low-power Embedded Systems”. In: ISCA. 2016.

[39] V. K. Chippa et al. “Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency”. In: *Design Automation Conference*. June 2010, pp. 555–560. DOI: 10.1145/1837274.1837411.

[40] Mihir Choudhury et al. “TIMBER: Time Borrowing and Error Relaying for Online Timing Error Resilience”. In: *DATE ’10*. 2010, pp. 1554–1559. ISBN: 978-3-9810801-6-2.

[41] L.T. Clark et al. “An embedded 32-b microprocessor core for low-power and high-performance applications”. In: *Solid-State Circuits, IEEE Journal of* (2001).

[42] R. P. Colwell et al. “Instruction Sets and Beyond: Computers, Complexity, and Controversy”. In: *Computer* 18.9 (Sept. 1985), pp. 8–19. ISSN: 0018-9162. DOI: 10.1109/MC.1985.1663000.

[43] Jason Cong et al. “Architecture and Synthesis for Multi-Cycle Communication”. In: *Proceedings of the 2003 International Symposium on Physical Design*. ISPD ’03. Monterey, CA, USA: Association for Computing Machinery, 2003, pp. 190–196. ISBN: 1581136501. DOI: 10.1145/640000.640040. URL: https://doi.org/10.1145/640000.640040.

[44] W. J. Dally and B. Towles. “Route packets, not wires: on-chip interconnection networks”. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. June 2001, pp. 684–689. DOI: 10.1109/DAC.2001.156225.

[45] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 0122007514.

[46] B. K. Daya et al. “SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 25–36.

[47] Monica Donno, Enrico Macii, and Luca Mazzoni. “Power-aware Clock Tree Planning”. In: *ISPD*. ACM, 2004.

[48] Monica Donno, Enrico Macii, and Luca Mazzoni. “Power-aware Clock Tree Planning”. In: ISPD ’04. ACM, 2004.

[49] Monica Donno et al. “Clock-tree Power Optimization Based on RTL Clock-gating”. In: *DAC*. ACM, 2003.

[50] A. J. Drake et al. “Single-cycle, pulse-shaped critical path monitor in the POWER7+ microprocessor”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2013, pp. 193–198.

[51] Zidong Du et al. “ShiDianNao: Shifting Vision Processing Closer to the Sensor”. In: *ISCA*. ACM, 2015.

[52] Christophe Dubach et al. “A Predictive Model for Dynamic Microarchitectural Adaptivity Control”. In: *MICRO*. IEEE Computer Society, 2010.

[53] Oguz Ergin et al. “Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure”. In: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Portland, Oregon: IEEE Computer Society, 2004, pp. 304–315. ISBN: 0-7695-2126-6. DOI: 10.1109/MICRO.2004.29. URL: http://dx.doi.org/10.1109/MICRO.2004.29.

[54] Dan Ernst and Todd Austin. "Efficient Dynamic Scheduling Through Tag Elimination". In: *Proceedings of the 29th Annual International Symposium on Computer Architecture*. ISCA '02. 2002.

[55] Dan Ernst et al. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation". In: *MICRO 36*. 2003. ISBN: 0-7695-2043-X.

[56] Hadi Esmaeilzadeh et al. "Architecture Support for Disciplined Approximate Programming". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 301–312. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2151008. URL: http://doi.acm.org/10.1145/2150976.2151008.

[57] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Vancouver, B.C., CANADA: IEEE Computer Society, 2012, pp. 449–460. ISBN: 978-0-7695-4924-8. DOI: 10.1109/MICRO.2012.48. URL: http://dx.doi.org/10.1109/MICRO.2012.48.

[58] Hadi Esmaeilzadeh et al. "Neural network stream processing core (NnSP) for embedded systems". In: *ISCAS*. IEEE. 2006.

[59] Stijn Eyerman and Lieven Eeckhout. "Fine-grained DVFS Using On-chip Regulators". In: *TACO* (2011).

[60] Stijn Eyerman et al. "A Performance Counter Architecture for Computing Accurate CPI Components". In: *ASPLOS*. ACM, 2006.

[61] Michael Ferdman et al. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 37–48. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150982. URL: http://doi.acm.org/10.1145/2150976.2150982.

[62] M. Fojtik et al. "Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction". In: *IEEE Journal of Solid-State Circuits* (2013).

[63] Daniele Folegnani and Antonio González. "Energy-effective Issue Logic". In: *ISCA*. 2001.

[64] E. G. Friedman. "Clock distribution networks in synchronous digital integrated circuits". In: *Proceedings of the IEEE* (2001).

[65] A. Frumusanu and R. Smith. "ARM A53/A57/T760 investigated: Samsung Galaxy Note 4 Exynos review". In: *https://www.anandtech.com* (2017).

[66] Hamid Reza Ghasemi and Nam Sung Kim. "RCS: Runtime Resource and Core Scaling for Power-constrained Multi-core Processors". In: *PACT*. ACM, 2014.

[67] G. A. Gillani et al. "MACISH: Designing Approximate MAC Accelerators With Internal-Self-Healing". In: *IEEE Access* 7 (2019), pp. 77142–77160. ISSN: 2169-3536.

[68] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. "Processor microarchitecture: An implementation perspective". In: *Synthesis Lectures on Computer Architecture* 5.1 (2010), pp. 1–116.

[69] V. Govindaraju et al. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing". In: *IEEE Micro* 32.5 (Sept. 2012), pp. 38–51. ISSN: 0272-1732. DOI: 10.1109/MM.2012.51.

[70] Brian Greskamp and Josep Torrellas. "Paceline: Improving Single-Thread Performance in Nanoscale CMPs Through Core Overclocking". In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 213–224. ISBN: 0-7695-2944-5. DOI: 10.1109/PACT.2007.52. URL: http://dx.doi.org/10.1109/PACT.2007.52.

[71] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing". In: *HPCA*. IEEE. 2015.

[72] Paul E Gronowski et al. "High-performance microprocessor design". In: *IEEE Journal of Solid-State Circuits* (1998).

[73] B. Grot et al. "Express Cube Topologies for on-Chip Interconnects". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Feb. 2009, pp. 163–174. DOI: 10.1109/HPCA.2009.4798251.

[74] B. Grot et al. "Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. June 2011, pp. 401–412.

[75] Erika Gunadi and Mikko H Lipasti. "Narrow Width Dynamic Scheduling". In: *Journal of Instruction-Level Parallelism* 9 (2007), pp. 1–23.

[76] Erika Gunadi and Mikko H. Lipasti. "CRIB: Consolidated Rename, Issue, and Bypass". In: *ISCA '11*. 2011. ISBN: 978-1-4503-0472-6.

[77] Meeta Gupta et al. "Tribeca: Design for PVT Variations with Local Recovery and Fine-grained Adaptation". In: *MICRO*. 2009. ISBN: 978-1-60558-798-1.

[78] M. R. Guthaus et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15. URL: https://doi.org/10.1109/WWC.2001.15.

[79] Muhammad Abdullah Hanif et al. "X-DNNs: Systematic Cross-Layer Approximations for Energy-Efficient Deep Neural Networks". In: *Journal of Low Power Electronics* 14.4 (2018), pp. 520–534. ISSN: 1546-1998.

[80] H. Hanson et al. "Thermal response to DVFS: analysis with an Intel Pentium M". In: *ISLPED*. 2007.

[81] Nikos Hardavellas et al. "Database Servers on Chip Multiprocessors: Limitations and Opportunities." In: Citeseer.

[82] Nikos Hardavellas et al. “Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches”. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. Austin, TX, USA: ACM, 2009, pp. 184–195. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555779. URL: http://doi.acm.org/10.1145/1555754.1555779.

[83] S. Hashemi et al. “Approximate Computing for Biometric Security Systems: A Case Study on Iris Scanning”. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2018, pp. 319–324.

[84] John L. Hennessy and David A. Patterson. “A New Golden Age for Computer Architecture”. In: *Commun. ACM* 62.2 (Jan. 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: http://doi.acm.org/10.1145/3282307.

[85] John L. Henning. “SPEC CPU2006 Benchmark Descriptions”. In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[86] Eric L Hill and Mikko H Lipasti. “Transparent mode flip-flops for collapsible pipelines”. In: *ICCD 2007*.

[87] Eric Hill and Mikko Lipasti. “Stall Cycle Redistribution in a Transparent Fetch Pipeline”. In: *ISLPED*. 2006. DOI: 10.1145/1165573.1165583.

[88] Mark D. Hill and Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* (2008).

[89] Mark D. Hill and Vijay Janapa Reddi. *Accelerator-level Parallelism*. 2019. arXiv: 1907.02064 [cs.DC].

[90] Glenn Hinton et al. “The Microarchitecture of the Pentium 4 Processor”. In: *Intel Technology Journal* (2001).

[91] R. Ho, K. W. Mai, and M. A. Horowitz. “The future of wires”. In: *Proceedings of the IEEE* (2001).

[92] Henry Hoffmann. “JouleGuard: Energy Guarantees for Approximate Applications”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: ACM, 2015, pp. 198–214. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815403. URL: http://doi.acm.org/10.1145/2815400.2815403.

[93] Henry Hoffmann et al. “Dynamic Knobs for Responsive Power-aware Computing”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 199–212. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950390. URL: http://doi.acm.org/10.1145/1950365.1950390.

[94] Houman Homayoun et al. “Reducing Power in All Major CAM and SRAM-Based Processor Units via Centralized, Dynamic Resource Size Management”. In: *IEEE Trans. Very Large Scale Integr. Syst.* (2011).

[95] Y. Hoskote et al. “A 5-GHz Mesh Interconnect for a Teraflops Processor”. In: *IEEE Micro* 27.5 (Sept. 2007), pp. 51–61. ISSN: 0272-1732. DOI: 10.1109/MM.2007.4378783.

[96] J. Howard et al. "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS". In: *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. Feb. 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077.

[97] J. Howard et al. "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling". In: *IEEE Journal of Solid-State Circuits* 46.1 (Jan. 2011), pp. 173–183. ISSN: 1558-173X. DOI: 10.1109/JSSC.2010.2079450.

[98] Chang-Hong Hsu et al. "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting". In: *HPCA* (2015).

[99] Zhigang Hu et al. "Microarchitectural Techniques for Power Gating of Execution Units". In: *ISLPED*. ACM, 2004.

[100] "Impact of Technology Scaling in the Clock System Power". In: ISVLSI '02. IEEE Computer Society, 2002, pp. 59–. ISBN: 0-7695-1486-3.

[101] ARM Inc. "ARM Compute Library". In: *https://developer.arm.com/compute-library/* (2017).

[102] ARM Inc. "Cortex-A57 Software Optimization Guide". In: *https://infocenter.arm.com/* (2016).

[103] Engin Ipek et al. "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors". In: *ISCA*. ACM, 2007.

[104] Hans M. Jacobson. "Improved Clock-gating Through Transparent Pipelining". In: *ISLPED '04*. 2004, pp. 26–31. ISBN: 1-58113-929-2.

[105] Hans M Jacobson et al. "Synchronous interlocked pipelines". In: *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*. IEEE. 2002, pp. 3–12.

[106] A. Jain et al. "Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783744.

[107] T. N. K. Jain et al. "Asynchronous Bypass Channels: Improving Performance for Multi-synchronous NoCs". In: *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*. 2010, pp. 51–58. DOI: 10.1109/NOCS.2010.15.

[108] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. "On-chip networks". In: *Synthesis Lectures on Computer Architecture* 12.3 (2017), pp. 1–210.

[109] D. Johnson et al. "Rigel: A 1,024-Core Single-Chip Accelerator Architecture". In: *IEEE Micro* 31.4 (July 2011), pp. 30–41. ISSN: 0272-1732. DOI: 10.1109/MM.2011.40.

[110] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: http://doi.acm.org/10.1145/3079856.3080246.

[111] Patrick Judd et al. “Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS ’16. Istanbul, Turkey: ACM, 2016, 23:1–23:12. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926294. URL: http://doi.acm.org/10.1145/2925426.2926294.

[112] J. Jung and T. Kim. “Variation-Aware False Path Analysis Based on Statistical Dynamic Timing Analysis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.11 (2012), pp. 1684–1697.

[113] Y. Kao et al. “CNoC: High-Radix Clos Network-on-Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.12 (Dec. 2011), pp. 1897–1910. ISSN: 1937-4151. DOI: 10.1109/TCAD.2011.2164538.

[114] Tejas S. Karkhanis and James E. Smith. “A First-Order Superscalar Processor Model”. In: *ISCA*. IEEE Computer Society, 2004.

[115] Stefanos Kaxiras and Margaret Martonosi. “Computer Architecture techniques for power-efficiency”. In: *Synthesis Lectures on Computer Architecture* (2008).

[116] S. K. Khatamifard et al. “VARIUS-TC: A modular architecture-level model of parametric variation for thin-channel switches”. In: *ICCD*. 2016. DOI: 10.1109/ICCD.2016.7753353.

[117] Khubaib et al. “MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP”. In: *MICRO*. IEEE Computer Society, 2012.

[118] Ilhyun Kim and Mikko H. Lipasti. “Half-price Architecture”. In: ISCA. 2003.

[119] J. Kim, J. Balfour, and W. Dally. “Flattened Butterfly Topology for On-Chip Networks”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. Dec. 2007, pp. 172–182. DOI: 10.1109/MICRO.2007.29.

[120] John Kim. “Low-Cost Router Microarchitecture for on-Chip Networks”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: Association for Computing Machinery, 2009, pp. 255–266. ISBN: 9781605587981. DOI: 10.1145/1669112.1669145. URL: https://doi.org/10.1145/1669112.1669145.

[121] Wonyoung Kim et al. “System level analysis of fast, per-core DVFS using on-chip switching regulators”. In: *HPCA*. Feb. 2008.

[122] Yuya Kora, Kyohei Yamaguchi, and Hideki Ando. “MLP-aware dynamic instruction window resizing for adaptively exploiting both ILP and MLP”. In: *MICRO*. ACM. 2013.

[123] T. Krishna et al. “Breaking the on-chip latency barrier using SMART”. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2013, pp. 378–389. DOI: 10.1109/HPCA.2013.6522334.

[124] T. Krishna et al. “Express Virtual Channels with Capacitively Driven Global Links”. In: *IEEE Micro* 29.4 (July 2009), pp. 48–61. ISSN: 0272-1732. DOI: 10.1109/MM.2009.64.

[125] T. Krishna et al. “SWIFT: A SWing-reduced interconnect for a Token-based Network-on-Chip in 90nm CMOS”. In: *2010 IEEE International Conference on Computer Design*. Oct. 2010, pp. 439–446. DOI: 10.1109/ICCD.2010.5647666.

[126] Tushar Krishna. “Enabling dedicated single-cycle connections over a shared network-on-chip”. PhD thesis. MIT, 2014.

[127] A. Krstic, Yi-Min Jiang, and Kwang-Ting Cheng. “Pattern generation for delay testing and dynamic timing analysis considering power-supply noise effects”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.3 (2001), pp. 416–425.

[128] A. Kumar, L. Peh, and N. K. Jha. “Token flow control”. In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. Nov. 2008, pp. 342–353. DOI: 10.1109/MICRO.2008.4771803.

[129] A. Kumar et al. “A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS”. In: *2007 25th International Conference on Computer Design*. Oct. 2007, pp. 63–70. DOI: 10.1109/ICCD.2007.4601881.

[130] G. Kurian et al. “ATAC: A 1000-core cache-coherent processor with on-chip optical network”. In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2010, pp. 477–488.

[131] Benjamin C. Lee and David Brooks. “Efficiency Trends and Limits from Comprehensive Microarchitectural Adaptivity”. In: *ASPLOS*. ACM, 2008.

[132] Dong Jin Lee. “High-performance and Low-power Clock Network Synthesis in the Presence of Variation”. PhD thesis. Citeseer, 2011.

[133] C. R. Lefurgy et al. “Active Guardband Management in Power7+ to Save Energy and Maintain Reliability”. In: *IEEE Micro* 33.4 (2013), pp. 35–45.

[134] C. R. Lefurgy et al. “Active management of timing guardband to save energy in POWER7”. In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2011, pp. 1–11.

[135] Sheng Li et al. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures”. In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 469–480. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669172. URL: http://doi.acm.org/10.1145/1669112.1669172.

[136] Xiaoyao Liang, Gu-Yeon Wei, and David Brooks. “ReVIVaL: A Variation-Tolerant Architecture Using Voltage Interpolation and Variable Latency”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 191–202. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.27. URL: http://dx.doi.org/10.1109/ISCA.2008.27.

[137] Hong-Ting Lin, Yi-Lin Chuang, and Tsung-Yi Ho. “Pulsed-latch-based Clock Tree Migration for Dynamic Power Reduction”. In: *ISLPED*. IEEE Press, 2011.

[138] J. Liu. "Soft Mousetrap: A Bundled-Data Asynchronous Pipeline Scheme Tolerant to Random Variations at Ultra-Low Supply Voltages". In: *ASYNC*. 2013. DOI: 10.1109/ASYNC.2013.29.

[139] Gabriel H. Loh. "Exploiting Data-width Locality to Increase Superscalar Execution Bandwidth". In: *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 35. Istanbul, Turkey: IEEE Computer Society Press, 2002, pp. 395–405. ISBN: 0-7695-1859-1. URL: http://dl.acm.org/citation.cfm?id=774861.774903.

[140] Jingwei Lu, Wing-Kai Chow, and Chiu-Wing Sham. "Fast Power- and Slew-aware Gated Clock Tree Synthesis". In: *IEEE Trans. Very Large Scale Integr. Syst.* (2012).

[141] Andrew Lukefahr et al. "Composite Cores: Pushing Heterogeneity Into a Core". In: *MICRO*. IEEE Computer Society, 2012.

[142] Kai Ma and Xiaorui Wang. "PGCapping: Exploiting Power Gating for Power Capping and Core Lifetime Balancing in CMPs". In: *PACT*. ACM, 2012.

[143] Divya Mahajan et al. "Axilog: Abstractions for Approximate Hardware Design and Reuse". In: *IEEE Micro* 35.5 (2015), pp. 16–30.

[144] Margaret Martonosi and Martin Roetteler. *Next Steps in Quantum Computing: Computer Science's Role*. 2019. arXiv: 1903.10541 [cs.ET].

[145] H. Matsutani et al. "Prediction router: Yet another low latency on-chip router architecture". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Feb. 2009, pp. 367–378. DOI: 10.1109/HPCA.2009.4798274.

[146] M. McKeown et al. "Power and Energy Characterization of an Open Source 25-Core Manycore Processor". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 762–775.

[147] Pierre Michaud, Andrea Mondelli, and André Seznec. "Revisiting Clustered Microarchitecture for Future Superscalar Cores: A Case for Wide Issue Clusters". In: *ACM Trans. Archit. Code Optim.* 12.3 (Aug. 2015), 28:1–28:22. ISSN: 1544-3566. DOI: 10.1145/2800787. URL: http://doi.acm.org/10.1145/2800787.

[148] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. "Load Value Approximation". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 127–139. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.22. URL: http://dx.doi.org/10.1109/MICRO.2014.22.

[149] Sasa Misailovic et al. "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 309–328. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660231. URL: http://doi.acm.org/10.1145/2660193.2660231.

[150] Sasa Misailovic et al. “Quality of Service Profiling”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 25–34. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806808. URL: http://doi.acm.org/10.1145/1806799.1806808.

[151] Thierry Moreau et al. *QAPPA: A Framework for Navigating Quality-Energy Tradeoffs with Arbitrary Quantization*. Tech. rep. 2017.

[152] Vojtech Mrazek et al. “AutoAx: An Automatic Design Space Exploration and Circuit Building Methodology Utilizing Libraries of Approximate Components”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC 2019. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257.

[153] Robert Mullins, Andrew West, and Simon Moore. “Low-Latency Virtual-Channel Routers for On-Chip Networks”. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 188–. ISBN: 0-7695-2143-6. URL: http://dl.acm.org/citation.cfm?id=998680.1006717.

[154] K. Nepal et al. “ABACUS: A technique for automated behavioral synthesis of approximate computing circuits”. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, pp. 1–6.

[155] S. M. Nowick. “High-Performance Asynchronous Pipelines: An Overview”. In: *IEEE Design Test of Computers* (2011). ISSN: 0740-7475. DOI: 10.1109/MDT.2011.71.

[156] Jaewon Oh and Massoud Pedram. “Gated clock routing for low-power microprocessor design”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* (2001).

[157] Shruti Padmanabha et al. “Trace Based Phase Prediction for Tightly-coupled Heterogeneous Cores”. In: *MICRO*. ACM, 2013.

[158] Subbarao Palacharla, Norman P Jouppi, and James E Smith. *Complexity-effective superscalar processors*. Vol. 25. 2. ACM, 1997.

[159] D Pamunuwa et al. “Layout, Performance and Power Trade-Offs in Mesh-Based Network-on-Chip Architectures”. In: *in Proc. of the 12th IFIP International Conference on Very Large Scale Integration (VLSI-SoC 2003)*. Citeseer. 2003.

[160] Jatuchai Pangjun and Sachin S. Sapatnekar. “Low-power Clock Distribution Using Multiple Voltages and Reduced Swings”. In: *IEEE Trans. Very Large Scale Integr. Syst.* (2002).

[161] A. Panyala et al. “Approximate Computing Techniques for Iterative Graph Algorithms”. In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Dec. 2017, pp. 23–32. DOI: 10.1109/HiPC.2017.00013.

[162] Jongse Park et al. *Expax: A framework for automating approximate programming*. Tech. rep. Georgia Institute of Technology, 2014.

[163] S. Park et al. “Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI”. In: *DAC Design Automation Conference 2012*. June 2012, pp. 398–405.

[164] Sunghyun Park et al. “Approaching the Theoretical Limits of a Mesh NoC with a 16-Node Chip Prototype in 45nm SOI”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. San Francisco, California: Association for Computing Machinery, 2012, pp. 398–405. ISBN: 9781450311991. DOI: 10.1145/2228360.2228431. URL: https://doi.org/10.1145/2228360.2228431.

[165] Yongjun Park, Hyunchul Park, and Scott Mahlke. “CGRA Express: Accelerating Execution Using Dynamic Operation Fusion”. In: *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES ’09. Grenoble, France: ACM, 2009, pp. 271–280. ISBN: 978-1-60558-626-7. DOI: 10.1145/1629395.1629433. URL: http://doi.acm.org/10.1145/1629395.1629433.

[166] Suchita Pati et al. *SeqPoint: Identifying Representative Iterations of Sequence-based Neural Networks*. 2020. arXiv: 2007.10459 [cs.DC].

[167] Arthur Perais et al. “Cost-effective Speculative Scheduling in High Performance Processors”. In: *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, Oregon: ACM, 2015, pp. 247–259. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2749470. URL: http://doi.acm.org/10.1145/2749469.2749470.

[168] Erez Perelman et al. “Using SimPoint for Accurate and Efficient Simulation”. In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’03. San Diego, CA, USA: ACM, 2003, pp. 318–319. ISBN: 1-58113-664-1. DOI: 10.1145/781027.781076. URL: http://doi.acm.org/10.1145/781027.781076.

[169] Paula Petrica et al. “Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems”. In: *ISCA*. ACM, 2013.

[170] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. “Dynamic Resizing of Superscalar Datapath Components for Energy Efficiency”. In: *IEEE Transactions on Computers* (2006).

[171] Ramya Raghavendra et al. “No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center”. In: *ASPLOS*. ACM, 2008.

[172] A. Raha and V. Raghunathan. “Approximating Beyond the Processor: Exploring Full-System Energy-Accuracy Tradeoffs in a Smart Camera System”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.12 (Dec. 2018), pp. 2884–2897. ISSN: 1557-9999.

[173] A. Raha and V. Raghunathan. “Synergistic Approximation of Computation and Memory Subsystems for Error-Resilient Applications”. In: *IEEE Embedded Systems Letters* 9.1 (2017), pp. 21–24.

[174] A. Raha et al. “Energy-Efficient Reduce-and-Rank Using Input-Adaptive Approximations”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.2 (Feb. 2017), pp. 462–475. ISSN: 1557-9999.

[175] Arnab Raha and Vijay Raghunathan. "Towards Full-System Energy-Accuracy Tradeoffs: A Case Study of An Approximate Smart Camera System". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC 2017. Austin, TX, USA: Association for Computing Machinery, 2017. ISBN: 9781450349277.

[176] A. Rahimi et al. "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters". In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Sept. 2013, pp. 1–10. DOI: 10.1109/CODES-ISSS.2013.6659022.

[177] Nitya Ranganathan and N Jouppi. "Evaluating the potential of future on-chip clock distribution using optical interconnects". In: ().

[178] Gokul Subramanian Ravi, Tushar Krishna, and Mikko Lipasti. "McMahon: Minimum-cycle Maximum-hop network". In: *Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems (AISTECS)* (2019).

[179] Gokul Subramanian Ravi, Tushar Krishna, and Mikko Lipasti. "TNT: Attacking latency, modularity and heterogeneity challenges in the NOC". In: *Under Submission* (2019).

[180] Gokul Subramanian Ravi and Mikko Lipasti. "Axl: Accelerating Approximations via Slack Recycling". In: *Workshop on Approximate Computing (WAX)* (2018).

[181] Gokul Subramanian Ravi and Mikko Lipasti. "Recycling Data Slack in Out-of-Order Cores". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019.

[182] Gokul Subramanian Ravi and Mikko Lipasti. "Timing Speculation in Multi-Cycle Data Paths". In: *IEEE Computer Architecture Letters* 16.1 (2016).

[183] Gokul Subramanian Ravi and Mikko H. Lipasti. "Aggressive Slack Recycling via Transparent Pipelines". In: *Proceedings of the International Symposium on Low Power Electronics and Design*. ISLPED '18. 2018.

[184] Gokul Subramanian Ravi and Mikko H. Lipasti. "CHARSTAR: Clock Hierarchy Aware Resource Scaling in Tiled ARchitectures". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. 2017.

[185] Gokul Subramanian Ravi, Joshua San Miguel, and Mikko Lipasti. "Synergic HW-SW Architecture for Fine-Grained Spatio-Temporal Approximation". In: *ACM Transactions on Code and Architecture Optimization*. TACO '20 (2020).

[186] Gokul Subramanian Ravi et al. "MicroGrad: A Centralized Framework for Workload Cloning and Stress Testing". In: *Under Submission* (2019).

[187] Sherief Reda and Muhammad Shafique. *Approximate Circuits: Methodologies and CAD*. Springer, 2018.

[188] A. Rovinski et al. "A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS". In: *2019 Symposium on VLSI Circuits*. 2019, pp. C30–C31.

[189] Pooja Roy et al. "ASAC: Automatic Sensitivity Analysis for Approximate Computing". In: *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES '14. Edinburgh, United Kingdom: ACM, 2014, pp. 95–104. ISBN: 978-1-4503-2877-7. DOI: 10.1145/2597809.2597812. URL: http://doi.acm.org/10.1145/2597809.2597812.

[190] S. Ryu, J. Koo, and J. Kim. "Low design overhead timing error correction scheme for elastic clock methodology". In: *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2017, pp. 1–6.

[191] Mehrzad Samadi et al. "SAGE: Self-tuning Approximation for Graphics Engines". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540711. URL: http://doi.acm.org/10.1145/2540708.2540711.

[192] Adrian Sampson et al. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 164–174. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993518. URL: http://doi.acm.org/10.1145/1993498.1993518.

[193] J. Sampson et al. "Efficient complex operators for irregular codes". In: *HPCA*. 2011. DOI: 10.1109/HPCA.2011.5749754.

[194] Karthikeyan Sankaralingam et al. "TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP". In: *ACM Trans. Archit. Code Optim.* 1.1 (Mar. 2004), pp. 62–93. ISSN: 1544-3566. DOI: 10.1145/980152.980156. URL: http://doi.acm.org/10.1145/980152.980156.

[195] S. R. Sarangi et al. "VARIUS: A Model of Process Variation and Resulting Timing Errors for Microarchitects". In: *IEEE Transactions on Semiconductor Manufacturing* (2008). ISSN: 0894-6507. DOI: 10.1109/TSM.2007.913186.

[196] D. J. Schlais and M. H. Lipasti. "BADGR: A practical GHR implementation for TAGE branch predictors". In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. Oct. 2016, pp. 536–543. DOI: 10.1109/ICCD.2016.7753338.

[197] Rathijit Sen and David A Wood. *Cache Power Budgeting for Performance*. Tech. rep. UW-CS-TR-1791. University of Wisconsin - Madison Computer Sciences Department, Apr. 2013.

[198] Weixiang Shen et al. "An effective gated clock tree design based on activity and register aware placement". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* (2010).

[199] Li-Shiuan Peh and W. J. Dally. "Flit-reservation flow control". In: *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*. Jan. 2000, pp. 73–84. DOI: 10.1109/HPCA.2000.824340.

[200] Julian Shun and Guy E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 135–146. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442530. URL: http://doi.acm.org/10.1145/2442516.2442530.

[201] Julian Shun et al. "Brief announcement: the problem based benchmark suite". In: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 2012, pp. 68–70.

[202] R. Singh et al. "Value Locality based Approximation with ODIN". In: *IEEE Computer Architecture Letters* 01 (June 5555), pp. 1–1. ISSN: 1556-6064. DOI: 10.1109/LCA.2020.3002542.

[203] *Skylake - Microarchitectures - Intel*. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server).

[204] A. Sodani. "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. 2015, pp. 1–24.

[205] Jared Stark, Mary D. Brown, and Yale N. Patt. "On Pipelining Dynamic Instruction Scheduling Logic". In: *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 33. 2000.

[206] Xin Sui et al. "Proactive Control of Approximate Programs". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 607–621. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872402. URL: http://doi.acm.org/10.1145/2872362.2872402.

[207] C. Sun et al. "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling". In: *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. May 2012, pp. 201–210. DOI: 10.1109/NOCS.2012.31.

[208] *Sunny Cove - Microarchitectures - Intel*. https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove.

[209] Steven Swanson et al. "WaveScalar". In: *MICRO*. IEEE Computer Society, 2003.

[210] A. Tang et al. "Delay and Power Modeling Framework for FinFET Processor Architectures Under PVT Variations". In: *IEEE Trans. on VLSI Systems* (2015). ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2352354.

[211] S. J. Tarsa et al. "Post-Silicon CPU Adaptation Made Practical Using Machine Learning". In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 2019, pp. 14–26.

[212] M. D. Taylor et al. "Scalar operand networks". In: *IEEE Transactions on Parallel and Distributed Systems* 16.2 (2005), pp. 145–162.

[213] Michael Bedford Taylor et al. "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams". In: *ISCA*. IEEE Computer Society, 2004.

[214] Abhishek Tiwari, Smruti R. Sarangi, and Josep Torrellas. "ReCycle:: Pipeline Adaptation to Tolerate Process Variation". In: *ISCA '07*. 2007, pp. 323–334. ISBN: 978-1-59593-706-3.

[215] V. Tiwari et al. "Reducing power in high-performance microprocessors". In: *DAC*. 1998.

[216] James W Tschanz et al. "Dynamic sleep transistor and body bias for active leakage power control of microprocessors". In: *Solid-State Circuits, IEEE Journal of* (2003).

[217] G. Tziantzioulis et al. "A Bit-level History-based Error Model with Value Correlation for Voltage-scaled Integer and Floating Point Units". In: *DAC*. 2015. ISBN: 978-1-4503-3520-1.

[218] Kimiyoshi Usami et al. "Adaptive power gating for function units in a microprocessor". In: *ISQED*. IEEE. 2010.

[219] Ashkan Vakil, Houman Homayoun, and Avesta Sasan. "IR-ATA: IR Annotated Timing Analysis, a Flow for Closing the Loop between PDN Design, IR Analysis Timing Closure". In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ASPDAC '19. Tokyo, Japan: Association for Computing Machinery, 2019, pp. 152–159. ISBN: 9781450360074. DOI: 10.1145/3287624.3287683. URL: https://doi.org/10.1145/3287624.3287683.

[220] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[221] Vassilis Vassiliadis et al. "A Programming Model and Runtime System for Significance-aware Energy-efficient Computing". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA: ACM, 2015, pp. 275–276. ISBN: 978-1-4503-3205-7. DOI: 10.1145/2688500.2688546. URL: http://doi.acm.org/10.1145/2688500.2688546.

[222] Augusto Vega et al. "Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control". In: *MICRO*. ACM, 2013.

[223] Swagath Venkataramani et al. "Quality Programmable Vector Processors for Approximate Computing". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 1–12. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540710. URL: http://doi.acm.org/10.1145/2540708.2540710.

[224] V. Vorisek et al. "Improved handling of false and multicycle paths in ATPG". In: *24th IEEE VLSI Test Symposium*. 2006, 6 pp.–165.

[225] M. Wagner and H. Wunderlich. "Efficient variation-aware statistical dynamic timing analysis for delay test applications". In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 276–281.

[226] Yasuko Watanabe, John D. Davis, and David A. Wood. "WiDGET: Wisconsin Decoupled Grid Execution Tiles". In: *ISCA*. ACM, 2010.

[227] D. Wentzlaff et al. “On-Chip Interconnection Architecture of the Tile Processor”. In: *IEEE Micro* 27.5 (Sept. 2007), pp. 15–31. ISSN: 0272-1732. DOI: 10.1109/MM.2007.4378780.

[228] D. Wentzlaff et al. “On-Chip Interconnection Architecture of the Tile Processor”. In: *IEEE Micro* 27.5 (2007), pp. 15–31.

[229] Shmuel Wimer and Israel Koren. “The Optimal fan-out of clock network for power minimization by adaptive gating”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* (2012).

[230] Jae-Yeon Won et al. “Up by their bootstraps: Online learning in artificial neural networks for CMP uncore power management”. In: *HPCA*. IEEE. 2014.

[231] David M Wu et al. “An optimized DFT and test pattern generation strategy for an Intel high performance microprocessor”. In: *ITC*. 2004.

[232] Thucydides Xanthopoulos. *Clocking in Modern VLSI Systems*. Springer Publishing Company, Incorporated, 2009.

[233] Sam Likun Xi et al. “Quantifying sources of error in McPAT and potential impacts on architectural studies”. In: *HPCA*. IEEE. 2015.

[234] Jing Xin and Russ Joseph. “Identifying and Predicting Timing-critical Instructions to Boost Timing Speculation”. In: *MICRO-44*. 2011, pp. 128–139. ISBN: 978-1-4503-1053-6.

[235] Amir Yazdanbakhsh et al. “AxBench: A multiplatform benchmark suite for approximate computing”. In: *IEEE Design & Test* 34.2 (2017), pp. 60–68.

[236] Sami Yehia and Olivier Temam. “From Sequences of Dependent Instructions to Functions: An Approach for Improving Performance Without ILP or Speculation”. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 238–. ISBN: 0-7695-2143-6. URL: http://dl.acm.org/citation.cfm?id=998680.1006721.

[237] *Zen - Microarchitectures - AMD*. https://en.wikichip.org/wiki/amd/microarchitectures/zen.

[238] H. Zheng et al. “High-level synthesis with behavioral level multi-cycle path analysis”. In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013, pp. 1–8.

[239] Yanqi Zhou and David Wentzlaff. “The Sharing Architecture: Sub-core Configurability for IaaS Clouds”. In: *ASPLOS*. ACM, 2014.