

Coordinating Software and Hardware for Performance Under Power Constraints

by

Ziqiang Patrick Huang

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

Benjamin C. Lee, Supervisor

Andrew D. Hilton

Alvin R. Lebeck

Brian M. Rogers

Daniel J. Sorin

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University

2019

ABSTRACT

Coordinating Software and Hardware for Performance Under
Power Constraints

by

Ziqiang Patrick Huang

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

Benjamin C. Lee, Supervisor

Andrew D. Hilton

Alvin R. Lebeck

Brian M. Rogers

Daniel J. Sorin

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Electrical and Computer
Engineering
in the Graduate School of Duke University
2019

Copyright © 2019 by Ziqiang Patrick Huang
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

For more than 50 years since its birth in 1965, Moore’s Law has been a self-fulfilling prophecy that drives computing forward [59]. However, as Dennard scaling ends [18], chip power density presents a challenge that becomes increasingly severe with every process generation. Consequently, a growing subset of transistors on a chip will need to be powered off in order to operate under a sustainable thermal envelope, a design strategy commonly referred as “dark silicon” [27, 90].

Although dark silicon poses a major challenge for consistently delivering higher performance, it also inspires researchers to rethink how a chip should be designed and managed under a power and thermal constrained environment. The historical way of extracting performance, whether single-thread or multi-thread, by throwing complicated and power-hungry hardwares is no longer applicable. Instead, we need to rely more on software, but the hardware needs to provide new mechanisms. In this thesis, we present three pieces of work on software and hardware codesign to demonstrate how coordinating softwares like compilers and runtimes with underlying hardware support can help boosting performance in a power-efficient way.

First, out-of-order (OoO) processors achieves higher performance than the in-order (IO) ones by aggressively scheduling instructions out of program order during execution. However, dynamic scheduling requires sophisticated control and numerous bookkeeping structures—*e.g.*, reorder buffer, load-store queue, register alias table—that increase complexity, area, and most importantly power. Observing that

a compiler produces better static schedules when the instruction set defines simple operation, we propose an ISA extension that decouples the data access and register write operations in a load instruction. We show that with modest system and hardware support, we can improve compilers’ instruction scheduling by hoisting a decoupled load’s data access above may-alias stores and branches. We find that decoupled loads improve performance with geometric mean speedup of 8.4% for a wide range of applications, bringing a step closer to OoO performance on IO design.

Second, sprinting is a class of computational mechanisms that provides a short but significant performance boost while temporarily exceeding the thermal design point. Using phase change material to buffer heat, sprinting is a promising way to deliver high performance in future chip designs that are likely to be power and thermal constrained. However, because sprints cannot be sustained, the system needs a mechanism to decide when to start and stop a sprint. We propose UTAR, a software runtime framework that manages sprints by dynamically predicting utility and modeling thermal headroom. Moreover, we propose a new sprint mechanism for caches, increasing capacity briefly for enhanced performance. For a system that extends last-level cache capacity from 2MB to 4MB per core and can absorb 10J of heat with phase change material, UTAR-guided cache sprinting improves performance by 17% on average and by up to 40% over a non-sprinting system. These performance outcomes, within 95% of an oracular policy, are possible because UTAR accurately predicts phase behavior and sprint utility.

Finally, applications often exhibit phase behaviors that demands for different types of system resources. As a result, management frameworks need to coordinate between different sprinting mechanisms to realize the full performance potential. we propose UTAR+, an extended version of UTAR that not only determines when to sprint, but also the type of resource as well as the sprinting intensity. Building upon UTAR’s phase predictor and utility and thermal-aware policy, UTAR+ quickly iden-

tifies the most profitable sprinting option to maximize performance/watt. For a system that offers multiple sprinting options, UTAR+-guided multi-resource sprinting improves performance by 22% on average and by up to 83% over a non-sprinting system, outperforming UTAR+-guided single-resource sprinting for a variety of applications.

To my dear parents and wife.

Contents

Abstract	iv
List of Tables	xi
List of Figures	xii
Acknowledgements	xv
1 Introduction	1
1.1 Organization and Overview of the Dissertation	2
1.1.1 Decoupled Load for Nano-Instruction Set Computers	3
1.1.2 Utility and Thermal Aware Runtime for Online Sprinting Management	4
1.1.3 Coordinated Multi-resource Sprinting	4
1.1.4 Key Contributions of the Thesis	5
2 Decoupling Loads for Nano-Instruction Set Computers	7
2.1 Decoupled Loads	9
2.1.1 Design Objectives	10
2.1.2 Instruction Semantics	12
2.1.3 Microarchitectural Support	14
2.1.4 System Support	17
2.2 Compiler Support	18
2.2.1 Hoisting Over (May-)Aliasing Stores	19
2.2.2 Hoisting Over Branches	22

2.2.3	Scheduling Policy	25
2.2.4	Load Tag Allocation	28
2.3	Experimental Evaluation	29
2.3.1	Performance Analysis	31
2.3.2	Sensitivity Analysis	33
2.3.3	Side Effects	36
2.4	Related Work	37
2.5	Conclusion	39
3	UTAR: Utility and Thermal Aware Runtime for Online Sprinting Management	40
3.1	UTAR Design	42
3.1.1	Management Architecture	43
3.1.2	Phase Classifier	44
3.1.3	Phase Predictor	48
3.1.4	Utility and Energy Predictor	49
3.1.5	Thermal Tracker	50
3.1.6	Sprinting Coordinator	51
3.2	Cache Sprinting Architecture	52
3.2.1	Case for Cache Sprinting	52
3.2.2	System Architecture	53
3.3	Experimental Methodology	55
3.4	Evaluation	58
3.4.1	UTAR Performance	59
3.4.2	Sprinting and System Dynamics	62
3.4.3	UTAR Prediction Accuracy	63
3.4.4	Sensitivity Analysis	65

3.5	Related Work	69
3.6	Conclusion	71
4	Coordinated Multi-resource Sprinting	72
4.1	Case for Multi-resource Sprinting	73
4.1.1	Heterogeneous Resource Demands	73
4.1.2	Tradingoff Resources for More Intense and Longer Sprints	75
4.1.3	Performance Improvement Opportunity	76
4.2	Managing Multi-resource Sprinting	77
4.2.1	UTAR Overview	78
4.2.2	UTAR+: What's Unchanged/Changed	78
4.2.3	UTAR+: Optimizations	79
4.3	Experimental Evaluation	82
4.4	Conclusion and Future Directions	85
5	Conclusions	86
	Bibliography	89
	Biography	98

List of Tables

2.1	Machine Model.	30
3.1	System Specification	56

List of Figures

2.1	In-order pipeline extended with load tag table (LTT).	15
2.2	SAXPY code example	20
2.3	SAXPY schedule with conventional loads	21
2.4	SAXPY schedule with decoupled loads	21
2.5	SPEC Hmmer Code Example.	22
2.6	Hoisting decoupled loads over branch (shared predecessor).	23
2.7	Hoisting decoupled load over branch (multiple predecessors).	24
2.8	Performance speedup from decoupled loads over baseline with conventional loads.	31
2.9	Load breakdown, relative to number of dynamic instructions.	32
2.10	Contributors to decoupled load performance.	33
2.11	Performance sensitivity to machine width.	34
2.12	Performance sensitivity to prefetching.	34
2.13	Performance sensitivity to scheduling policy.	35
2.14	Performance sensitivity to load tag table (LTT) size.	36
2.15	Code size and overhead.	37
3.1	Example program execution in phases	43
3.2	UTAR overview	44
3.3	Phase Classifier Example	46
3.4	Phase Classifier Example	48

3.5	Performance comparison: scaling frequency vs increasing LLC capacity, baseline 2.1GHz with 2MB LLC	54
3.6	Performance gain: gcc-s04 running with 4MB LLC over baseline 2MB	54
3.7	UTAR Performance: compared against various policies	58
3.8	PDF for utility: (a) bimodal, (b) unimodal	60
3.9	UTAR Sprinting Behavior: (a) ocean_ncp, (b) gcc-s04	61
3.10	Impact of phase signature definition on phase classification accuracy .	62
3.11	Impact of signature boundary on: (a) phase classification accuracy, (b) total phase number	62
3.12	Utility prediction accuracy	65
3.13	UTAR performance sensitivity to (a) TDP (6.5W), (b) total thermal headroom (5J), (c) sprinting intensity (6MB)	66
3.14	Impact of sprinting intensity on sprinting time	68
3.15	UTAR performance against perfect TDP	68
4.1	Performance gain: rayCast running with 2.3GHz + 2MB LLC over 2.1GHz + 4MB LLC (no sprint is: 2.1GHz + 2MB LLC)	74
4.2	Performance gain: gcc running with 2.3GHz + 2MB LLC over 2.1GHz + 4MB LLC (no sprint is: 2.1GHz + 2MB LLC)	74
4.3	Performance gain: bzip running with 1.9GHz + 5MB LLC over 2.1GHz + 4MB LLC (no sprint is: 2.1GHz + 2MB LLC)	75
4.4	Performance comparison: single-resource vs multi-resource sprints over 2.1GHz + 2MB LLC	76
4.5	UTAR overview	78
4.6	Sprinting Mechanism Preference over LLC Miss	80
4.7	Sprinting Mechanism Preference over Cache Sprining Utility	81
4.8	Cache Sprinting Intensity Preference Cahce Sprinting Utility	81
4.9	UTAR+ Performance	83
4.10	Epochs breakdown in multi-resource sprints	84

4.11 UTAR+ Prediction Accuracy	85
--	----

Acknowledgements

The journey towards the Phd dissertation has been long and challenging yet full of reward for me. I have the privilege to work at a prestigious university, work on cutting-edge research problems and most importantly, work with a group of amazing people.

First I want to thank Duke University for providing such a supportive environment. Being far away from home as an international student sometimes can be hard, but Duke has done an excellent job to provide any necessary support. I am and will forever be proud of being a part of this community.

I want to thank my two advisers, Dr. Benjamin Lee and Dr. Andrew Hilton. I fell in love with the field of computer architecture and decided to pursue a PhD because of my great experience in Dr. Hilton's classes and Dr. Lee's lab. Throughout the years working with them, I learned from them both not only the professional skills to conduct high-quality research, but also many necessary characteristics to be better person,

I also want to thank my other committee members, Dr. Daniel Sorin, Dr. Alvin Lebeck and Dr. Brian Rogers. I took many classes from them and had many discussions with them about my research. They have always been honest and patient with me and have provided many constructive advice to me.

I want to thank many colleagues and friends that have made this journey much more enjoyable: Dr. Qiuyun Llull for providing many helpful advice when I first

joined the lab, Dr. Luwa Matthew for bringing much fun and joy to the lab, Dr. Songchun Fan for always being positive and encouraging and Dr. Tamara Lehman for always being there for me when I need help. Finally, I want to thank my dear parents and my beloved wife for always having faith in me. Without their support, this journey would have not been possible.

1

Introduction

For almost four decades since mid 1960s, Moore's Law [58] (the doubling of transistors every 18 months) has been the fundamental driving force for computing. Though several generations of transistor shrinking, coupled with Dennard scaling [19], chip designers were able to aggressively targeting exponential growth in peak-performance without increasing power consumption. Although transistor dimensions are projected to continue shrinking to the end of the decades [4], the accompanying growth in static power due to leakage current precludes further scaling of voltage. As a result, active power grows with every process generation, leading to higher power density for a chip of the same size. Recent trends have predicted power density to increase to over 6x beyond the 8nm process node [4, 11].

The challenge of increased power density was passed along to package designers by specifying higher thermal design power(TDP) for chips. TDP serves as the nominal value for designing chips' cooling systems. However, chips are reaching the thermal limits of both active and passive cooling, which means, without innovation. even if more transistors can be squeezed in a chip to fulfill Moore's Law, the fraction of those transistors which can be powered on at the same time will decrease. This

is referred as “dark silicon” phenomenon. According to recent studies, researchers from different groups have projected that, at 8nm process node, the amount of dark silicon may reach up to 50%-80% depending on the processor architecture, cooling technology and application workloads [26].

Although dark silicon poses a major challenge for consistently delivering higher performance, it also inspires researchers to rethink how a chip should be designed and managed under a power and thermal constrained environment. The historical way of extracting performance, whether single-thread or multi-thread, by throwing complicated and power-hungry hardwares is no longer desirable. Researchers have proposed various approaches to fight dark silicon[89]. These proposals have largely focused on reducing thermal design power by lowering chip activity, either by selectively powering dedicated special function units, reducing die size, or reducing voltage and frequency. Although these approaches partially address the dark silicon problem, they forsake the opportunity to harness the peak performance potential enabled by Moore’s law.

1.1 Organization and Overview of the Dissertation

This thesis aims to improve computer systems’ performance under the constraint of limited power and thermal budget. We present three pieces of work on software and hardware co-design to demonstrate how designing softwares like compilers and runtimes with underlying hardware support in mind can help boosting performance in a power-efficient way.

In the first work, observing that in-order processors have the potential to achieve higher energy efficiency than out-of-order ones and thus could be more appealing in a power and thermal constrained system, we look into methods to bring a step closer to OoO performance on IO design. We propose **Decoupled Load**, an ISA extension that decouples the data access and register write operations in a load

instruction. With modest system and hardware support, we design a compiler that exploit decoupled load to produce better instruction scheduling.

In the second work, observing that sprinting is a promising way of delivering high performance in future chip designs that are likely to be constrained by power and thermal budget. We propose **UTAR**, a software runtime framework that manages sprints by dynamically predicting utility and modeling thermal headroom. Moreover, we propose a new sprint mechanism for caches, increasing capacity briefly for enhanced performance.

In the third work, observing that applications often exhibit different resource demands at different phases, we propose the idea of multi-resource sprinting and **UTAR+**, an extended version of **UTAR** that is capable of coordinating between different sprinting mechanisms like frequency sprints and cache sprints.

Together, we show the need and the benefit of co-designing softwares and hardwares in the dark silicon era and pave the way for continuing pursuing performance given limited power and thermal budget in future computer systems.

1.1.1 Decoupled Load for Nano-Instruction Set Computers

Architects who design out-of-order (OoO) processors accept higher costs in return for performance. By aggressively scheduling instructions out of program order during execution, OoO processors exploit instruction level parallelism to a greater degree than in-order (IO) ones. However, dynamic scheduling requires sophisticated control and numerous bookkeeping structures—*e.g.*, reorder buffer, load-store queue, register alias table—that increase complexity, area, and most importantly power.

In Chapter 2, we note that even RISC bundles multiple basic operations into an instruction and breaking these bundles could further improve static scheduling – a strategy we call Nano Instruction Set Computing (NISC). The most promising candidates for NISC include branches and loads, instructions that disproportion-

ately impact performance and are amenable to separation into nano-instructions. Indeed, prior work decouples branches into prediction and resolution to improve schedules [56]. In contrast, we decouple functionality in loads and perform code motion to hide their long latencies.

1.1.2 *Utility and Thermal Aware Runtime for Online Sprinting Management*

Computational sprinting [71, 70] is a mechanism for dark silicon, which describes systems that can only power a fraction of its peak resources [27, 90]. Sprints provide a short but significant performance boost and draw extra power and temporarily increase chip temperature beyond its thermal design point (TDP). To manage sprints, prior work has either relied on an obvious trigger (*e.g.*, parallel code region activates additional cores [71]) or offline profiles to estimate utility [28]. However, such approaches are limited to certain types of sprint and cannot adapt to workload dynamics.

In Chapter 3, we propose UTAR, a utility and thermal aware run-time that determines when to initiate and terminate a sprint. Based on hardware support available in current processors, we present a software framework that evaluates utility and makes sprinting decisions. We demonstrate UTAR for a new class of sprints that increase microarchitectural capacity. Specifically, we propose *cache sprints* that expand last-level cache capacity and exceed TDP when that capacity is most useful.

1.1.3 *Coordinated Multi-resource Sprinting*

Applications often exhibit phase behaviors that demands for different types of system resources. As a result, management frameworks need to coordinate between different sprinting mechanisms to realize the full performance potential.

In Chapter 4, we propose UTAR+, an extended version of UTAR that not only determines when to sprint, but also the type of resource as well as the sprinting

intensity. Building upon UTAR’s phase predictor and utility and thermal-aware policy, UTAR+ quickly identifies the most profitable sprinting option to maximize performance/watt.

1.1.4 *Key Contributions of the Thesis*

In summary, we make the following contributions in this thesis:

- We propose decoupled loads, which separate a load instruction into data access and register writeback. We describe instruction semantics and support required from the operating system and microarchitecture. We modify a compiler to exploit decoupled loads and produce high-quality, static instruction schedules. The compiler hoists loads’ data accesses and schedules independent instructions to hide latency. We evaluate performance gains using cycle-level simulations for varied IO processor configurations and SPEC2006 benchmarks.
- We propose UTAR, a utility and thermal aware run-time framework that is able to manage sprints dynamically based on accurate utility prediction and thermal modeling. We propose a new class of sprints that increases the capacity of microarchitectural resources (e.g., caches) and exceed TDP when that capacity is most useful. We implement UTAR in software and evaluate it using a set of cache-sensitive applications. We prototype the run-time system on an Intel Xeon Broadwell that supports last-level cache allocation via Intel Cache Allocation Technology [31]. We show that UTAR-guided cache sprints improve performance significantly and performs very close to oracular policies that use perfect knowledge of sprint utility and thermal conditions to control sprints.
- We propose the idea of multi-resource sprinting and show the extended benefits of coordinating between different sprinting mechanisms. We propose UTAR+,

an enhanced software runtime to manage multi-resource sprints. UTAR+ further improves performance by matching program phases' resource demands with the proper sprinting mechanisms and intensities. We show that UTAR+-guided multi-resource sprints outperform UTAR-guided single-resource sprints across a variety of applications.

Decoupling Loads for Nano-Instruction Set Computers

Architects who design out-of-order (OoO) processors accept higher costs in return for performance. By aggressively scheduling instructions out of program order during execution, OoO processors exploit instruction level parallelism to a greater degree than in-order (IO) ones. However, dynamic scheduling requires sophisticated control and numerous bookkeeping structures—*e.g.*, reorder buffer, load-store queue, register alias table—that increase complexity, area, and power. Faced with these costs, designers may ask whether alternative (micro)architectures could deliver a significant portion of OoO’s performance without its hardware overheads.

OoO derives most of its performance from a better instruction schedule, not the ability to react to dynamic events. One insightful study finds that better schedules account for 88% of OoO’s advantage over IO [55]. Large windows permit aggressive re-ordering and register renaming ensures that schedules are constrained only by true dependencies. In contrast, compilers are constrained by memory aliasing and anti-/output dependencies when scheduling statically. Static and dynamic scheduling

have been studied extensively [13, 50]. However, the discovery that dynamic schedules are the key to OoO performance [55], not reactive mechanisms that mitigate variable-latency operations and wrong-path execution, motivates new architectures that permit better static schedules and produce OoO-competitive performance on IO hardware.

A compiler produces better static schedules when the instruction set defines simple operations. As argued in the case for RISC [66], simplicity benefits code generation since the compiler need not find the rare opportunities to use complex instructions that bundle multiple operations. Moreover, simplicity benefits code motion since the compiler has more flexibility when re-ordering finer-grained computation and pursuing good static schedules. Thus, RISC gives the compiler fewer challenges and more opportunities.

We note that even RISC bundles multiple basic operations into an instruction and breaking these bundles could further improve static scheduling – a strategy we call Nano Instruction Set Computing (NISC). The most promising candidates for NISC include branches and loads, instructions that disproportionately impact performance and are amenable to separation into nano-instructions. Indeed, prior work decouples branches into prediction and resolution to improve schedules [56]. In contrast, we decouple functionality in loads and perform code motion to hide their long latencies.

Scheduling decoupled loads is complicated by stores and branches. Loads cannot be hoisted before a may-alias store without jeopardizing program correctness. Loads cannot be moved across basic block boundaries without ensuring exception safety, which is frequently quite difficult. Existing techniques such as data speculation [16, 47, 17] and superblock formation [36, 53] mitigate constraints imposed by stores and branches. However, such speculative techniques incur overheads when recovering from misspeculation.

We study non-speculative approaches to circumvent the constraints posed by

stores and branches, and show how decoupled loads produce better schedules. Specifically, we make the following contributions:

- We propose decoupled loads, which separate a load instruction into data access and register writeback. We describe instruction semantics and support required from the operating system and microarchitecture.
- We modify a compiler to exploit decoupled loads and produce high-quality, static instruction schedules. The compiler hoists loads' data accesses and schedules independent instructions to hide latency.
- We evaluate performance gains using cycle-level simulations for varied IO processor configurations and SPEC2006 benchmarks. Decoupled loads offer a geometric mean speedup of 8.4%.

Collectively, our results show the potential of NISC architectures. Decoupling an instruction's constituent operations produces better static schedules and brings us a step closer to OoO performance on IO design.

2.1 Decoupled Loads

Architects have long known that load instructions present performance challenges. In this paper, we seek performance by extending a RISC instruction set with decoupled loads that separate data accesses and register writebacks. To support the instruction extension, the microarchitecture must hold new state – data supplied by the cache hierarchy but not yet written to registers. Moreover, the system must accommodate new load semantics for consistency/coherence, exception handling, and context switches. With such support, decoupled loads meet key design objectives such as hiding load-to-use latency without expensive software speculation or hardware overhead.

2.1.1 Design Objectives

Hiding Latency. Decoupled loads help a compiler hide load-to-use latency. Loads, with their long and variable latencies, often occupy the critical path. However, if a load is scheduled well before the first instruction that uses its data, computation for intervening instructions hides the load latency. By separating data access and register write, a decoupled load increases scheduling flexibility. Because the register is written separately, the compiler can hoist a load’s data access higher and more often than it could have hoisted a conventional load.

Although superficially similar, decoupling and hoisting a load’s data access is orthogonal to data prefetching. Prefetchers bring potentially useful data up the memory hierarchy and into the data cache before a load is executed. Thus, they reduce the probability of cache misses and average load latency. Once data resides in the L1 data cache, however, decoupled loads are needed to hide load-to-use latency.

Foreshadowing our experimental findings, decoupled loads improve performance even when prefetching is perfect. An ideal prefetcher would ensure that every data cache access hits. Even in this optimistic scenario, the compiler could hoist a decoupled load’s cache access, find computation to hide its modest latency (*e.g.*, 4 cycles), and improve performance by up to **[[10%]]**. A less accurate prefetcher increases average load latency, producing a baseline for the compiler to produce even greater performance gains.

Avoiding Speculation. We decouple load functionality such that the compiler can hoist a load’s data access safely and non-speculatively. When hoisting above an aliasing store, the microarchitecture supplies the correct value to the load. When hoisting across basic block boundaries, no exception is generated if the branch resolves such that the original load should not be executed.

Speculation permits aggressive code motion but requires fix-up code to correct

the effects of misspeculation. For example, the Itanium employs two types of loads, speculative and advanced, that allow the compiler to schedule loads earlier [77]. Speculative loads can be moved before one or more branches. Advanced loads can be moved before stores even when the alias analysis is inconclusive. When either of these loads are used and hoisted, the compiler inserts check and branch instructions at the load’s original location to detect misspeculation, and it inserts fix-up code that recovers from misspeculation when necessary.

Correction and recovery code for misspeculation introduces overheads. When the compiler misspeculates about the code path or memory aliasing, the fix-up code re-executes the computation correctly. Even when misspeculation is rare, fix-up code increases register pressure as the compiler allocates architected registers for the additional instructions. The Itanium’s 128 registers might accommodate this pressure. However, a RISC architect designing an IO core has a more difficult choice – use 32 registers and spill to memory, or use 128+ registers and incur hardware costs like those for OoO physical register files.

Minimizing Hardware Overhead. We decouple loads to improve the performance of static schedules on inexpensive IO cores. Decoupled loads require modest microarchitectural support – a small structure to hold values after data access and before register write. Foreshadowing our experimental findings, a small table is sufficient to realize the performance potential for decoupled loads.

In contrast, other approaches to mitigate load latency require far more hardware. OoO’s dynamic schedules perform well but require a physical register file, re-order buffer, and load/store queue. Itanium’s speculative and advanced loads require many architected registers to support re-execution and fix-up code [77]. Sophisticated microarchitectures produce a load value earlier in the pipeline by enhancing the front-end with instruction pre-decode, base register caching, and fast address calculation [7]. Each of these approaches require bookkeeping, control, and recovery

mechanisms with larger cost-benefit ratios than those for decoupled loads.

Ensuring Compatibility. Decoupled loads extend an existing RISC instruction set, giving the compiler additional options for code generation. In some scenarios, hoisting a decoupled load’s data access across basic blocks can harm performance—the destination block may execute the load even when the source block is not executed. The compiler should judiciously decouple loads and we implement several, illustrative heuristics to demonstrate this capability.

Decoupled loads are generated by specifying a compiler flag when targeting a processor with the extension. The compiler extracts performance from decoupled loads using static analysis. Although decoupled loads do not require run-time profiles to generate code, they benefit from dynamic binary translation (DBT) frameworks, which permit transparent innovation in the instruction set [17, 56].

2.1.2 *Instruction Semantics*

A conventional load instruction – `load rD, I(rA)` – has three significant pieces:

- **Memory Hierarchy Access:** The first piece of a load instruction is the computation of its effective address ($I + rA$), and the memory hierarchy access. Access includes translation from virtual to physical address, the L1 data cache read, etc.
- **Ordering:** The second piece of a load instruction is its ordering relative to other memory operations. Ordering specifies whether the load comes logically before or after a store from the same or another thread.
- **Register Write:** The third piece of a load instruction is writing the value to destination register `rD`.

Conventional loads constrain scheduling as all three pieces are bundled together in a single instruction. In contrast, the separation of these pieces into multiple

instructions gives greater flexibility. We focus on separating memory hierarchy access from ordering. Doing so allows a load to perform parts of its work with longer latencies (*i.e.*, load-to-use latency of a data cache hit or the even longer latencies of a miss) prior to the point at which it must be ordered relative to other instructions. These separated instructions are linked by a new architectural identifier that we call the *load tag*.

We consider splitting the load into two instructions. Note that a load could be divided into three instructions, but the additional split benefits performance only under heavy register pressure:

- **Data Access** [`load.D$ ltD, I(rA)`]: Data access behaves much like a conventional load except that it places the contents at memory address `I(rA)` into the destination load tag `ltD`. If the instruction faults, the exception is raised only when the load is ordered by a subsequent instruction that uses the same load tag.
- **Order and Write** [`load.wb rD, ltA`]: This instruction *orders* the load and writes the result back to the register file. Writing the result copies the value from the load tag `ltA` into the destination register `rD`. If this value is not yet available, the instruction will stall. Order and write preserves load semantics, causing the decoupled load to behave as if the conventional load were written at this point in program order.

Translating a conventional load into a decoupled one is trivial.¹ The compiler can freely hoist the data access instruction above stores and branches to hide load-to-use latency. However, the compiler must ensure that the ordering instruction for the load remains at the conventional load’s original position or at some place the compiler could have safely moved the conventional load within the program.

¹ Loads from location declared `volatile` cannot be split.

```

load r2, 0(r1) → load.D$ lt0, 0(r1)
                load.wb r2, lt0

```

2.1.3 *Microarchitectural Support*

We present an overview of a microarchitecture that supports decoupled loads. The primary microarchitectural addition is a table to maintain the state of decoupled loads between its `load.D$` and the `load.wb` instructions. We call this structure the Load Tag Table (LTT) as it has one entry per load tag. Each entry includes status bits, exception information, addresses, and values.

Load Tag Table. Two status bits encode the state of an entry. 00 indicates that the entry is invalid, which means that no `load.D$` has been executed since the `load.wb` completed for the previous load that used the entry. 01 indicates that the memory hierarchy access is in progress. 10 indicates that the memory hierarchy access has been completed, but subsequently invalidated by coherence. Finally, 11 indicates that the entry holds valid data.

Each entry holds exception information that tracks any exception detected for a load that has not yet been ordered. For example, `load.D$` may have encountered an invalid virtual address. Each entry tracks the virtual and physical addresses specified and read by `load.D$`, respectively. Finally, the entry holds the value retrieved by `load.D$`.

Figure 2.1 shows a 7-stage, in-order pipeline extended with an LTT. The LTT is physically divided into three parts: the virtual address (VA) in the decode stage; the status, exception and value (S,E,V) in the execute stage; and the physical address (PA) in the second memory stage. Decoupled load instructions, `load.D$` and `load.wb`, index into the table with their load tag. The structure that holds phys-

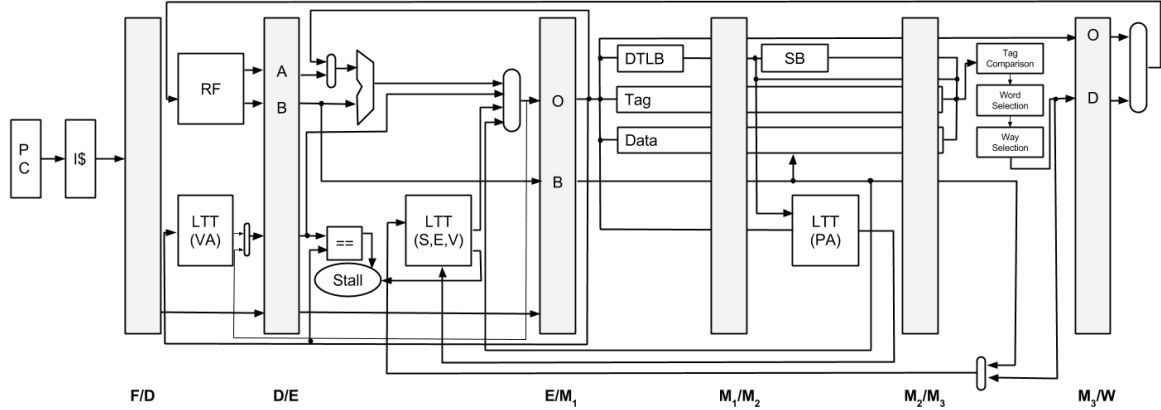


FIGURE 2.1: In-order pipeline extended with load tag table (LTT).

ical addresses is a content-addressable memory (CAM), which is searched by store instructions and coherence invalidations.

Operation and Example. We describe LTT operation with a simple example. The compiler schedules a store between a load’s data access and register writeback, highlighting the load’s interaction with a may-alias store.

```
load.D$ lt0, 4(r2)
st 4(r4), r3
load.wb r3, lt0
```

When `load.D$` enters the execute stage (E), it sets the corresponding LTT entry’s status to 01. Then, `load.D$` updates the LTT entry with its virtual address in the first memory stage (M_1) and with its physical address in the second memory stage (M_2). After retrieving its data from the cache hierarchy, `load.D$` updates the LTT entry with its value in the third and last memory stage (M_3). Finally, `load.D$` sets status to 11 during normal execution and to 10 if an exception arises.

When `store` enters the second memory stage (M_2), it searches the LTT’s CAM. If it finds an LTT entry with the same address, `store` updates the entry’s value in

the third memory stage (M_3). Finally, when `load.wb` enters the execute stage (E), it retrieves the entry's value and sets status to 00. Thus, the LTT ensures correctness in the presence of aliasing stores while potentially shrinking load-to-use latency from 4 cycles to 1.

Suppose `load.wb` executes and reads an entry with status other than 11. If status is 00, the entry is invalid and an invalid-instruction exception is raised. If status is 01, the data access is in progress and `load.wb` stalls until the value returns from the memory hierarchy. If status is 10, coherence invalidation requires `load.wb` to re-access the memory hierarchy, as if it were a conventional load, using the virtual address stored in the LTT.

Note that `load.wb` reads the LTT value two stages earlier than `store` searches the LTT for possible aliases. Even with bypassing, a `load.wb` that immediately follows a `store` risks reading an outdated value from the LTT in the execute stage (E). A conservative microarchitecture could stall if any store is in flight.

An efficient alternative compares page offsets in virtual addresses. The `store`'s offset is available after address generation and the `load.wb`'s is read from the LTT during decode. If offsets differ, `load.wb` and `store` may proceed in parallel. An aggressive alternative compares virtual addresses early to bypass from `store` to `load.wb`. In the event of synonyms (*i.e.*, same physical but different virtual addresses), the `load.wb` is squashed and re-executed.

Discussion. The LTT has some similarity to the load queue in an OoO processor. Both structures hold physical addresses in a CAM. Stores and invalidations must search both. However, LTTs differ in several significant ways.

First, in the LTT, a CAM match either updates the entry's value to reflect the value stored or changes its status to reflect an invalidation. In a load queue, a CAM match flushes instructions because a load executed at the wrong time. Second, the LTT holds only entries for decoupled loads and is much smaller than a load queue.

Third, the compiler assigns LTT entries during register allocation whereas the load queue is a FIFO structure.

Although one might view the LTT as additional “registers” to mitigate register pressure, the performance benefits of decoupled loads cannot be achieved by simply increasing the register file size by the number of LTT entries. From the compiler’s perspective, more registers reduces spilling and mitigates anti/output dependencies. But they cannot help loads circumvent constraints from stores and branches.

Note that this high-level description elides the micro-architectural complexities of a pipelined processor: the `load.wb` may enter the pipeline before the `store` knows its physical address, or even before the `load.D$` writes the LTT. Each of these issues has solutions (*e.g.*, the pipeline can track which load tags have in-flight writers, and act accordingly). However, a compelling evaluation of the details of all of these requires synthesizing a core with the modifications, and is thus beyond the scope of this work, which introduces the ISA changes and focuses on the compiler support.

2.1.4 System Support

Decoupled load semantics are defined naturally by its two instructions for data access (`load.D$`) and register writeback (`load.wb`). Together, these instructions provide the system enough information to order loads, handle exceptions, and switch contexts.

Load Ordering. The system must order decoupled loads relative to loads and stores from other cores. The `load.wb` is the ordering point of the decoupled load. Because the compiler ensures that `load.wb` occupies the original load’s location in the program, the decoupled load completes at the same point in time as a conventional load would have. Thus, `load.D$` is not visible to other cores until `load.wb` commits, guaranteeing the decoupled load’s correctness under any consistency model.

However, the core responds to invalidation differently to accommodate interaction between decoupled loads and stores from other cores. When a load is decoupled,

another core's store could be ordered after its data access but before its register write. Thus, invalidation needs to associatively search the LTT just as invalidation is required to search the load queue in OoO cores. If a matching address is found, invalidation updates the LTT entry's status, forcing the subsequent `load.wb` to re-access the memory hierarchy.

Exception Handling. When a decoupled load's `load.D$` produces an exception, the system defers handling until its ordering point at `load.wb`. With deferred exception handling, decoupled loads provide the same semantics as conventional loads. Exceptions are precise with respect to the register write. Moreover, the compiler can hoist `load.D$` above branches without risk of handling exceptions from unexecuted code paths unnecessarily.

Context Switching. Load tags, like registers, are part of a process or thread's context. Recall that the LTT contains valid bits, addresses, and values. When a thread is context switched out, the operating system (OS) saves each valid LTT entry's virtual address, but not value, to memory.

When the thread is context switched in, the OS restores each valid LTT entry by re-loading the value from its saved address. To accomplish this reload, the OS re-executes `load.D$` for each decoupled load in flight. Re-execution is necessary after a context switch because other threads may have modified values residing at LTT-held addresses. Re-execution may encounter page faults if the context switch paged out LTT-held addresses. Page faults, like exceptions, are deferred to the decoupled load's ordering point.

2.2 Compiler Support

Compiler support is essential when using decoupled loads to produce high-performance schedules. First, the compiler must determine which loads to decouple into data access and register write. Naïvely decoupling every load would increase pressure on

the load tag table, which is small to ensure single-cycle access, and constrain performance. Furthermore, naïve approaches would increase instruction cache pressure and cause the program to execute more dynamic instructions, harming performance and power efficiency.

Second, the compiler must hoist data access instructions, often above may-alias stores and branches, to hide load-to-use latency. We first describe scenarios in which scheduling benefits from decoupled loads. Then, we propose two compiler scheduling policies that exploit decoupled loads to improve performance.

2.2.1 *Hoisting Over (May-)Aliasing Stores*

Alias analysis attempts to determine whether two pointers to memory refer to the same address. Given a query with two pointers, the analysis responds with one of three possible answers—must, may, or no alias. The compiler uses alias analysis conservatively and does not schedule conventional loads above may-alias stores. With decoupled loads, the compiler circumvents the constraints posed by memory aliasing. We show how data access in decoupled loads can be hoisted above may-alias stores for representative functions.

Scheduling Example. The compiler makes assumptions about machine parameters during instruction scheduling. In our example, the compiler considers a two-wide, in-order machine with two ALUs and one load/store unit. The load-to-use latency is four cycles, which optimistically assumes loads hit in the L1 cache and provides a challenging scenario for decoupled loads. Multiplication requires four cycles and all other instructions require one cycle. Each functional unit is fully pipelined.

We consider single-precision, scalar multiplication and vector addition (SAXPY). Figure 2.2 and 2.3 present the source code and resulting static schedule. Instructions within the same loop iteration are serialized by true data dependencies such that these instructions cannot be reordered or scheduled in the same cycle. In contrast,

```

void saxpy (float *dest, float *x,
            float *y, float a, int n) {
    for (int i=0; i < n; i+=2) {
        dest[i] = a*x[i]+y[i];
        dest[i+1] = a*x[i+1]+y[i+1];
    }
}

```

FIGURE 2.2: SAXPY code example

instructions across loop iterations are not constrained by true dependencies.

Modern compilers extract instruction-level parallelism across iterations by unrolling the loop. However, loop unrolling by a factor of two fails to improve SAXPY performance. The compiler fails to find parallelism in the unrolled loop because it cannot determine whether the store instruction at line 5 aliases the loads in lines 6-7. As a result, the compiler conservatively specifies a may-alias dependence between the store and its following loads, producing a static schedule that spans 21 cycles.

Decoupled loads separate data access from register write, allowing the compiler to hoist the data access. When splitting a conventional load instruction, which it assumes requires four cycles, the compiler generates a three-cycle `load.D$` and a one-cycle `load.wb`; most of the load's latency is attributed to data access. The compiler uses idle issue slots to schedule data accesses in lines 6-7 earlier, which reduces load-to-use latency from four cycles to one. Because stores check and update the load tag table, the schedule ensures program correctness regardless of aliasing. The improved new static schedule spans only 18 cycles, as shown in Figure 2.4.

SAXPY represents a broader class of codes for which alias analysis must be conservative and thus restricts the compiler's ability to schedule code efficiently. We take SAXPY as an example because it is simple and easy to understand. More generally, SAXPY illustrates code in which read-modify-write instructions are performed on data structures indexed with variables. These codes are prevalent and Figure 2.5

Loop :		Cycle	Issue-1	Issue-2
1	ld r9,0(r4)	1	1	14
2	ld r10,0(r5)	2	2	
3	mul r9,r9,r6	3-4		
4	add r9,r9,r10	5	3	
5	st 0(r3),r9	6-8		
6	ld r9,4(r4)	9	4	
7	ld r10,4(r5)	10	5	
8	mul r9,r9,r6	11	6	12
9	add r9,r9,r10	12	7	13
10	st 4(r3),r9	13-14		
11	addi r3,r3,8	15	8	
12	addi r4,r4,8	16-18		
13	addi r5,r5,8	19	9	
14	addi r8,r8,2	20	10	11
15	bne r8,r7,Loop	21	15	

FIGURE 2.3: SAXPY schedule with conventional loads

Loop :		Cycle	Issue-1	Issue-2
1	ld r9,0(r4)	1	1	16
2	ld r10,0(r5)	2	2	
3	mul r9,r9,r6	3	6	
4	add r9,r9,r10	4	7	
5	st 0(r3),r9	5	3	
6	ld.D\$ lt0,4(r4)	6-8		
7	ld.D\$ lt1,4(r5)	9	4	
8	ld.wb r9,lt0	10	5	
9	ld.wb r10,lt1	11	8	9
10	mul r9,r9,r6	12	10	14
11	add r9,r9,r10	16	11	15
12	st 4(r3),r9	17	12	13
13	addi r3,r3,8	18	17	
14	addi r4,r4,8			
15	addi r5,r5,8			
16	addi r8,r8,2			
17	bne r8,r7,Loop			

FIGURE 2.4: SAXPY schedule with decoupled loads

shows a similar loop pattern from `hmm` in the SPEC2006 benchmark suite.

```
void P7EmitterPosterior(
    int L, struct plan7_s *hmm,
    struct dpmatrix_s * forward,
    struct dpmatrix_s * backward,
    struct dpmatrix_s *mx) {
    ...
    for (i = L; i>=1; i++) {
        mx->xmx[i][XMC]=
        forward->xmx[i-1][XMC]
        + hmm->xsc[XTC][LOOP]
        + backward->xmx[i][XMC]
        -sc;
        ...
    }
}
```

FIGURE 2.5: SPEC Hmmer Code Example.

2.2.2 Hoisting Over Branches

Compilers have difficulty hoisting a conventional load instruction over a branch, primarily because the branch could resolve in another direction. If a load is hoisted from a basic block in an unexecuted code path, the load's destination register would be written with the wrong value. Moreover, the load may cause an exception and trigger a handler unnecessarily. For these reasons, Itanium's advanced loads require fix-up code and propagate a token that tracks deferred exceptions to each instruction that depends on the advanced load's value [77]. We show how decoupled loads can be hoisted above branches non-speculatively and without fix-up code.

Scheduling and Examples. Figure 2.6 presents a simplified control flow graph from the `spec_random_load` function in `bzip2`, a benchmark from the SPEC2006 suite. Basic block A has two successors, B and C, both of which start with a load. The compiler cannot hoist these conventional loads above A's branch because it cannot determine the branch direction.

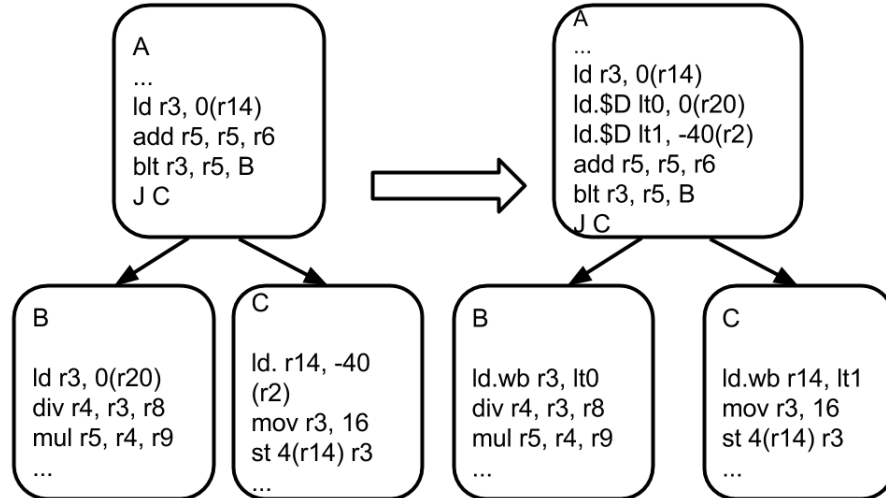


FIGURE 2.6: Hoisting decoupled loads over branch (shared predecessor).

Decoupled loads enable a new schedule. The compiler decouples each load into `load.D$` and `load.wb` instructions. It hoists both `load.D$` instructions over A’s branch while ensuring that both `load.wb` instructions remain in their respective positions. Because `load.D$` places data in load tags and does not fault until ordered by a `load.wb`, the transformation ensures program correctness regardless of branch direction. By hoisting `load.D$` above branches, the compiler overlaps latency of B and C’s loads with the latency of A’s load.

Naïvely decoupling loads and hoisting data accesses above branches may harm performance. Figure 2.7 presents a scenario in which basic block C has two predecessors, A and B. The compiler may wish to decouple C’s load and hoist its data access into B to overlap the latency with the load in B. However, it must hoist data access into both predecessors to ensure a valid value for the writeback regardless of control flow. Unfortunately, A is a loop body and its branch into C has a very low bias. Hoisting C’s data access into A may cause A to execute more slowly due to more dynamic instructions. Since A is executed multiple times, overall performance

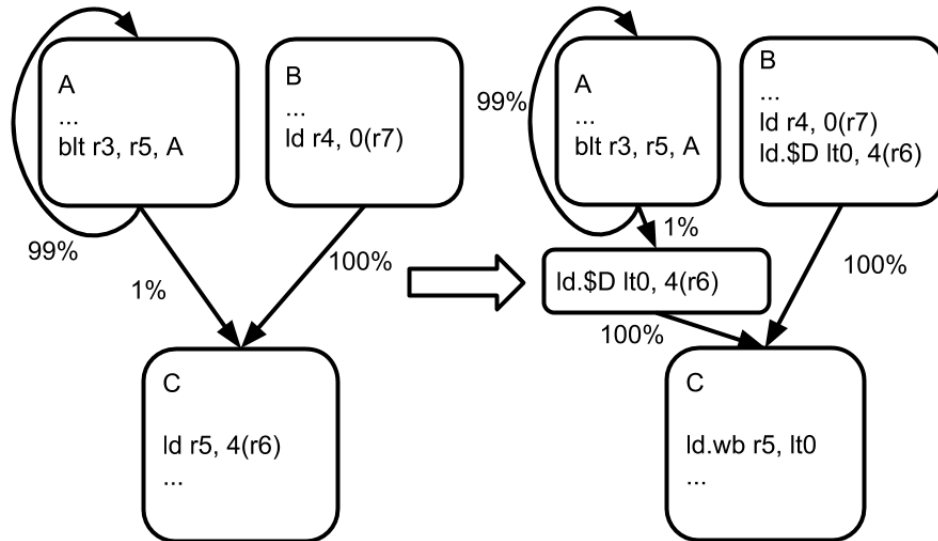


FIGURE 2.7: Hoisting decoupled load over branch (multiple predecessors).

may suffer.

The compiler can implement one of two solutions to this scenario. First, the compiler can be conservative and refrain from decoupling C’s load instruction. Alternatively, the compiler can create a new basic block between A and C that contains only the `load.D$` instruction, which has several advantages. When control flows from A to C, computation is correct and `load.D$` does not affect A’s performance. When control flows from B to C, the hoisted `load.D$` improves performance.

In our implementation, the compiler determines its handling of decoupled loads and branches based on static analysis. Estimating bias is easier in some cases (*e.g.* loops) and more difficult in others. In the future, decoupled loads could be generated and scheduled with profiles and dynamic binary translation. Dynamic frameworks might enable a panoply of more precise optimizations.

2.2.3 Scheduling Policy

Instruction scheduling is a critical stage in the compiler’s code generation pipeline. The compiler tries to reorder instructions to increase instruction-level parallelism and reduce structural hazards. Even when the compiler has a holistic view of the program, it typically reorders instructions within a basic block and not across them. Scheduling instructions across basic blocks is difficult because it risks executing instructions from the wrong code path.

Scheduling for Conventional Loads. List scheduling is a common algorithm used by most modern compilers for scheduling instructions within a basic block. The scheduler constructs a data dependency graph (DDG) in which nodes represent instructions, edges represent instruction dependencies, and edge weights represent instruction latencies.

For a given basic block, Algorithm 1 shows a simplified procedure for list scheduling. The algorithm first builds DDG and then uses the graph to identify instructions that are ready to execute. Whether an instruction is ready at a given cycle depends on when its predecessors were scheduled and their latencies. If no instruction is ready, the algorithm simply increments the cycle count and checks again. Thus, the algorithm repeatedly selects ready instructions, inserts them into the schedule, and updates the graph. The process continues until all instructions are scheduled and the graph is empty.

Extensions for Decoupled Loads. We extend list scheduling to use decoupled loads in Algorithm 2. The new scheduling algorithm discovers new opportunities to exploit instruction- and memory-level parallelism. When the algorithm cannot find a ready instruction in a given cycle, it will expand its search by decoupling a load and determining whether its data access instruction can be scheduled.

The scheduler has several strategies for decoupling loads and hoisting data ac-

Algorithm 1 Scheduling with Conventional Loads

```
1: procedure SCHEDULE(BASICBLOCK *B)
2:   build DDG
3:   while B has unscheduled instructions do
4:     if no instruction ready at this cycle then
5:       goto next
6:     pick instructions
7:     release successor instructions
8:   next:
9:     cycle++
```

Algorithm 2 Scheduling with Decoupled Loads – Bubble

```
1: procedure SCHEDULE(BASICBLOCK *B)
2:   build DDG
3:   while B has unscheduled instructions do
4:     if no instruction ready at this cycle then
5:       if available Ld/St Unit then
6:         if exist load only constrained by may-alias then
7:           goto decouple
8:         if exist independent load in successor blocks then
9:           goto decouple
10:        goto next
11:     decouple:
12:       decouple load into load.D$ and load.wb
13:       mark load.D$ as ready
14:       continue
15:     pick instructions
16:     release successor instructions
17:   next:
18:     cycle++
```

cesses. For any given cycle, the scheduler considers decoupled loads only when no other instruction is ready for scheduling and the load/store unit is idle. To find an instruction for this “bubble” cycle, the scheduler seeks to decouple loads from the current basic block and, if that fails, to decouple loads from the block’s successors.

First, within the current block, the scheduler searches for loads that are constrained only by a may-alias dependence. The scheduler can decouple the load, hoist its data access instruction above the may-alias store, and rely on the load tag table to

supply the value for register write. Second, the scheduler searches for independent loads in the current block’s successors. The scheduler can decouple the load and hoist its data access instruction above the branch.

When a candidate load is found, the compiler decouples the conventional load into `load.D$` and `load.wb` instructions. Then, the compiler updates the graph with dependencies and latencies for the new instructions. To ensure that the ordering point remains in the original load’s position, `load.wb` inherits all dependency information from the original load plus an additional predecessor – `load.D$`. If the scheduler assumes an n -cycle latency for the original load, it assumes $n - 1$ -cycle latency for `load.D$` and one-cycle latency for `load.wb`. With the new graph, the scheduler attempts to fill bubbles in the schedule.

Algorithm 3 Scheduling with Decoupled Loads – Early

```

1: procedure SCHEDULE(BASICBLOCK *B)
2:   build DDG
3:   for each instruction do
4:     if is load instruction then
5:       if no predecessor instruction then
6:         decouple into load.D$ and load.wb
7:         for each of B’s predecessor block P do
8:           hoist load.D$ to the end of P
9:           mark P to be rescheduled
10:      if constrained by may-alias then
11:        decouple into load.D$ and load.wb
12:        add load.wb’s data dependency edges to load.D$
13:      (rest same as Algorithm 1)

```

Algorithm 3 offers a more aggressive approach to scheduling decoupled loads. Instead of decoupling loads only in the presence of bubble cycles during scheduling, the algorithm seeks candidate loads for decoupling immediately after building the data dependence graph. Doing so adds flexibility because the compiler may schedule the entire basic block differently and, for example, prioritize the `load.D$` over other low-latency instructions. However, such an aggressive policy requires new, comprehensive

compiler heuristics to determine when decoupling loads is worthwhile.

2.2.4 Load Tag Allocation

The compiler allocates and frees load tags, which are required when decoupling loads, in a process that closely resembles register allocation. When the compiler decides to decouple a load in the scheduling stage, it assigns a virtual load tag to the pair of `load.D$` and `load.wb` instructions, much like a virtual register number for traditional register allocation. The `load.D$` opens the liveness of the corresponding load tag whereas the `load.wb` kills it.

When the compiler enters the register allocation stage, it allocates load tags just as it allocates registers. Because load tags look very much like registers, we can take advantage of the liveness analysis and register allocation frameworks that already exist in the compiler to support decoupled loads.

The allocator handles load tag pressure differently than register pressure. When the allocator encounters register pressure, it spills register contents to memory. In contrast, when the allocator encounters load tag pressure, it reverses the decision to decouple the load and couples the `load.D$` and `load.wb` to produce a conventional load.

Reverting to a conventional load is preferable to spilling, which generates a new pair of store/load instructions and defeats the purpose of decoupling loads (*i.e.*, hiding load latency). Register allocation follows scheduling in a conventional compiler pipeline, but the compiler must re-schedule a basic block if allocation reverts a decoupled load into a conventional one. Rescheduling incurs no additional overhead since the compiler requires a post-register-allocation scheduling pass to accommodate normal register spills.

2.3 Experimental Evaluation

We extend the OpenRISC instruction set with decoupled loads. To generate code with decoupled loads, we extend the Machine Instruction Scheduler in LLVM 3.5 [45] with Algorithms 2-3. The scheduler searches for two types of conventional loads that could be decoupled to improve performance. First, it seeks loads in the same basic block that have been constrained by may-aliasing stores. Second, it seeks independent loads in successor blocks that could be hoisted above branches depending on the static analysis of branch biases.

Machine Model. To understand the performance of decoupled loads, we perform cycle-level simulation by extending the OpenRISC architectural simulator Or1ksim with an in-order timing model that includes dependency checking, superscalar support, a three-level cache hierarchy and a branch predictor.

Table 2.1 summarizes machine parameters that are used in the simulations. LLVM also uses some of these parameters – for example, the number of functional units, instruction latency – to guide static scheduling. Load latency varies dramatically across applications and even across specific load instructions within the same application. Yet the scheduler requires a single static value that estimates load latency. Our compiler schedules loads using L1 hit latency.

Benchmarks. We evaluate eight integer benchmarks in SPEC2006. We also experiment decoupled loads with floating-point benchmarks in SPEC2006, but are unable to produce meaningful results because OpenRISC does not support double-precision arithmetic in its 32-bit architecture. Double-precision arithmetic is implemented in software, which generates many library calls that restrict code motion for decoupled loads; the compiler cannot hoist loads above a function call.

The standard approaches in performance measurement sample workloads and run a limited number of representative instructions. Yet sampling is difficult when com-

Structure	Configuration
Branch Predictor	GShare, 8KB table, 13 history bits, 4K-entry BTB, 64-entry RAS
Machine Width	Varied – 2/4
Functional Units	LD/ST varied – 1/2, INT ALU 2×, FP-ALU 1×
Load Tag Table	Varied – 8/16/32 entries
L1 Caches	8-way 32KB L1-D\$, 4-way 32KB L1-I\$, 64B lines, 4-cycle latency
L2 Cache	16-way 256KB, 12-cycle latency
L3 Cache	32-way 4MB, 25-cycle latency
Miss Handling	8-entry MSHR
DRAM	140-cycle latency

Table 2.1: Machine Model.

paring different ISAs. Simulating X instructions after fast-forwarding Y instructions could measure performance in very different parts of the workload as code generation and scheduling could produce very different dynamic instruction counts. For this reason, we run each benchmark to completion with its TRAIN input set and measure end-to-end performance speedup.

Evaluation Strategy. Graphs show results from a 4-wide, 2-LD/ST, 32-entry load tag table configuration unless otherwise specified. We present the geometric mean of speedups obtained over a baseline in-order machine. We first supply insight into the sources of performance by showing how often loads are decoupled and hoisted. We further differentiate decoupled loads hoisted above may-aliasing stores versus those hoisted above branches. We assess performance sensitivity to machine width, prefetching, scheduling policy and microarchitectural resources. Finally, we

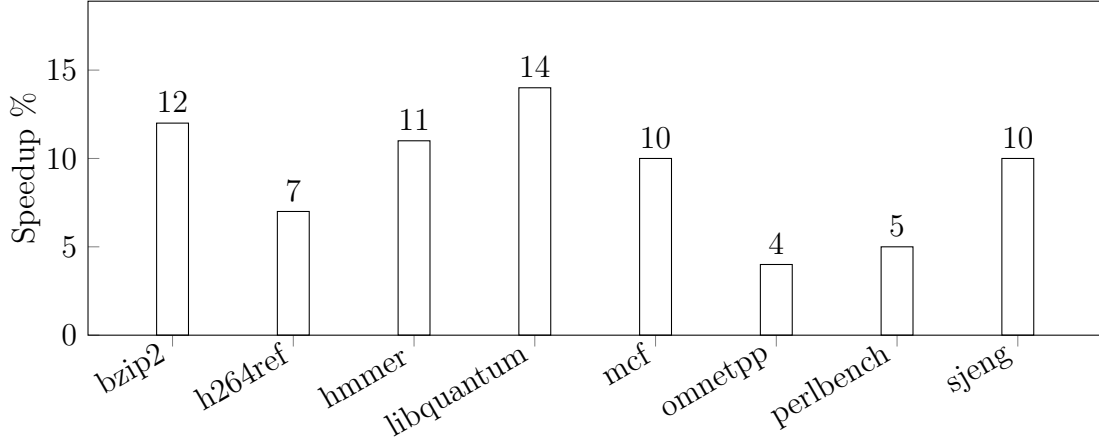


FIGURE 2.8: Performance speedup from decoupled loads over baseline with conventional loads.

discuss side effects of decoupled loads.

2.3.1 Performance Analysis

Figure 2.8 presents performance gains when generating code with decoupled loads instead of conventional loads. Overall, decoupled loads improve performance with geometric mean speedup of 8.4% and a max speedup of 14%. Performance gains vary from benchmarks as six out of the eight benchmarks report substantial speedups, ranging from 7% - 14%, while the other two benefit significantly less.

Instruction Mix. Figure 2.9 presents the number of loads relative to number of dynamic instructions. The black bar denotes the number of useful decoupled loads, measured by the number of completed `load.wb` instructions. The white bar denotes the number of conventional loads that are not decoupled. The gray bar denotes the number of redundant loads, measured by the number of `load.D$` instructions from unexecuted code paths.

Load instructions often comprise about 20% of the total. The compiler can decouple a large percentage of these loads. For example, loads comprise 17% of libquantum’s instruction mix and 70% of these loads are decoupled. Similarly, loads

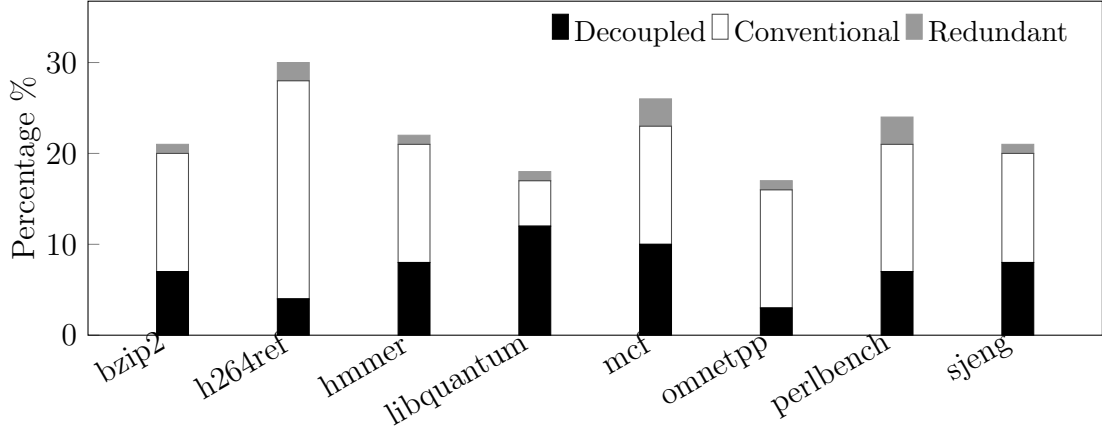


FIGURE 2.9: Load breakdown, relative to number of dynamic instructions.

comprise 23% of *mcf*'s instruction mix and 43% of these are decoupled, which corresponds to a large number of loads in absolute terms. The percentage of decoupled loads indicates how often the compiler uses the new instructions and correlates with performance gains.

Although the compiler decouples loads and hoists data access instructions aggressively, it rarely creates extra work for the application. Unnecessary and redundant data access instructions comprise only 1.4%, on average, of the total. Redundant data access instructions are those that are hoisted from basic blocks in unexecuted code paths. In these scenarios, the application executes a `load.D$` instruction but fails to execute the corresponding `load.wb` instruction. Thus, redundant loads are a measure of wasted work. Figure 2.9 indicates that redundant loads are rare and the compiler generates efficient code.

Stores and Branches. We hoist decoupled loads above may-aliasing stores and branches with two different mechanisms. Hoisting above stores requires a load-tag table that forwards updated values to decoupled loads. Hoisting above branches requires bias analysis and new basic blocks. We find that applications are diverse and require support for both types of instruction re-ordering.

Figure 4.10 illustrates contributions to performance when hoisting a load's data

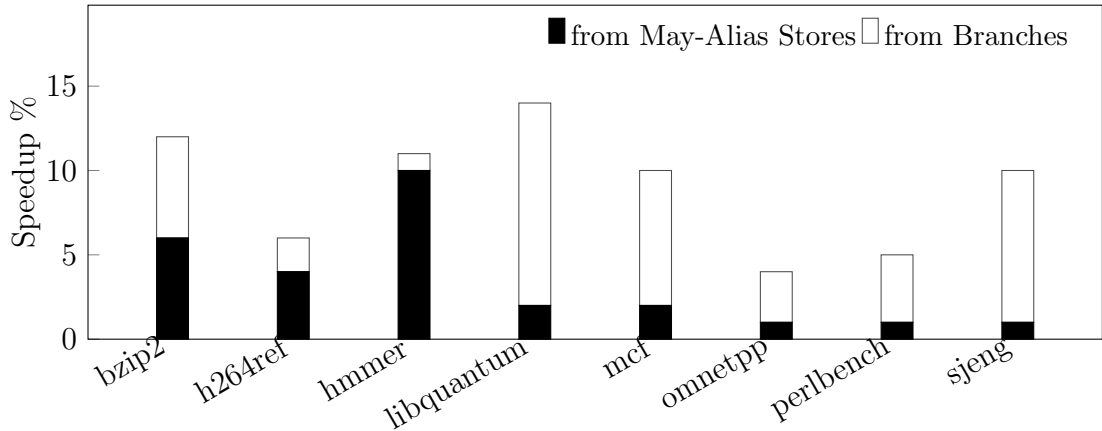


FIGURE 2.10: Contributors to decoupled load performance.

access above stores and branches. Although all benchmarks benefit from both types of code motion permitted when decoupling loads, the extent of these benefits vary. For example, 90% of hmmer’s performance gain comes from hoisting loads over may-aliasing stores whereas benchmarks like libquantum and mcf benefit mostly hoisting loads over branches. In contrast, bzip2 benefits equally from both. Thus, both scheduling techniques are required to realize the full potential of decoupled loads.

2.3.2 Sensitivity Analysis

Machine Width. Figure 2.11 illustrates performance sensitivity to an in-order machine’s issue width and number of load/store units. Decoupled loads perform better in wider machines. Given a wider machine, the compiler is more likely to find idle slots and “bubbles” in the static schedule, which trigger a search for loads to decouple and data access instructions to hoist. Thus, the compiler can decouple loads more aggressively in a wider machine. However, performance differences are modest as the compiler is limited by the number of candidate loads in the application.

Prefetching. Figure 2.12 shows the speedup obtained by decoupled loads over a baseline with no prefetching, with a perfect L2 (every L1 miss served by L2), and with a perfect L1 (every access hits L1). The perfect caches model the effect of ideal

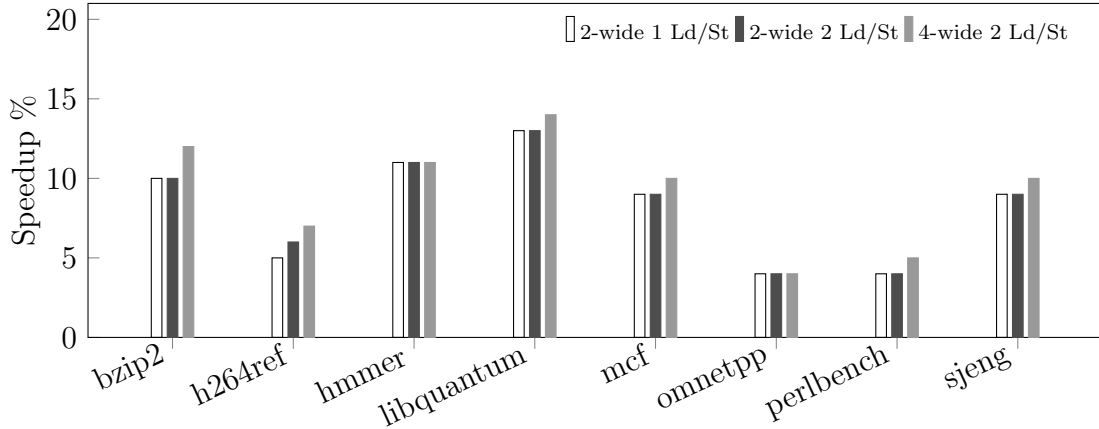


FIGURE 2.11: Performance sensitivity to machine width.

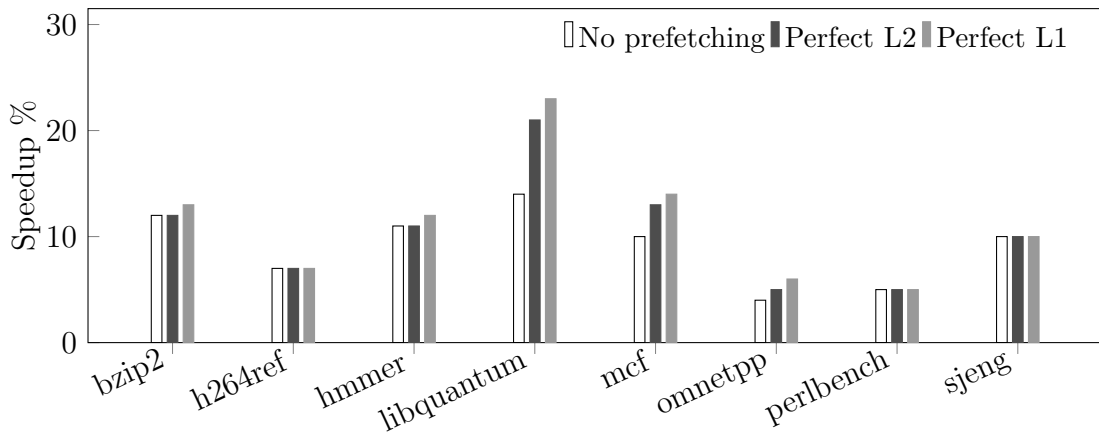


FIGURE 2.12: Performance sensitivity to prefetching.

prefetchers on decoupled loads.

We find that prefetchers do not degrade benefits from decoupled loads. On the contrary, for benchmarks with relatively bad data cache behavior, such as *mcf* and *libquantum*, bundling decoupled loads with prefetching increases speedups. While decoupled loads can potentially overlap cache miss latency, consecutive cache misses are rare because most benchmarks are characterized by regular memory access patterns and good data locality. The majority of decoupled loads' benefits come from hiding the load-to-use latency of a cache hit.

Scheduling Policy. Figure 2.13 compares the intuitive policy in Algorithm 2,

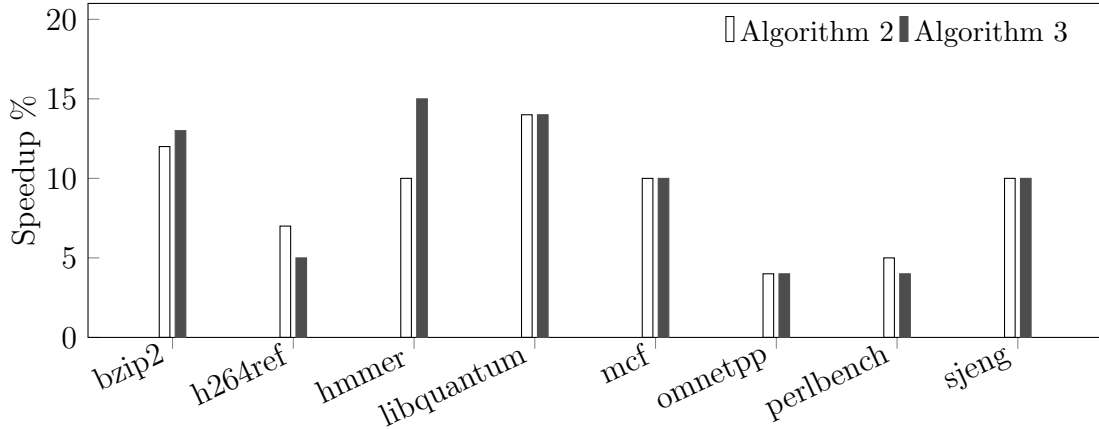


FIGURE 2.13: Performance sensitivity to scheduling policy.

which hides load latency in bubble cycles, with the more aggressive policy in Algorithm 3. For benchmarks with long load latencies and prevalent bubbles, such as `mcf`, decoupling loads in the presence of bubbles or immediately after building the dependence graph makes little difference. A few benchmarks perform better with Algorithm 3. These benchmarks tend to have more instruction-level parallelism, providing the compiler more opportunities to prioritize decoupled loads over low-latency instructions. However, several benchmarks perform worse as the compiler generates more redundant load.D\$ instructions when branches are unbiased.

Load-Tag Table Size. A decoupled load that is in flight requires a load tag, which links the data access instruction to its corresponding register write instruction. Our analysis thus far assumes 32 entries in the load tag table, a conservative configuration that evaluates compiler effectiveness when resources are abundant. In practice, however, we balance performance gains against load-tag table size.

Figure 2.14 evaluates performance for a range of LTT sizes and shows that only a few entries are sufficient. Load tag pressure is usually much smaller than conventional register pressure for two reasons. First, load tags are used only for decoupled loads. Second, load tags have a very small live range. As we vary LTT size from 32 to 8

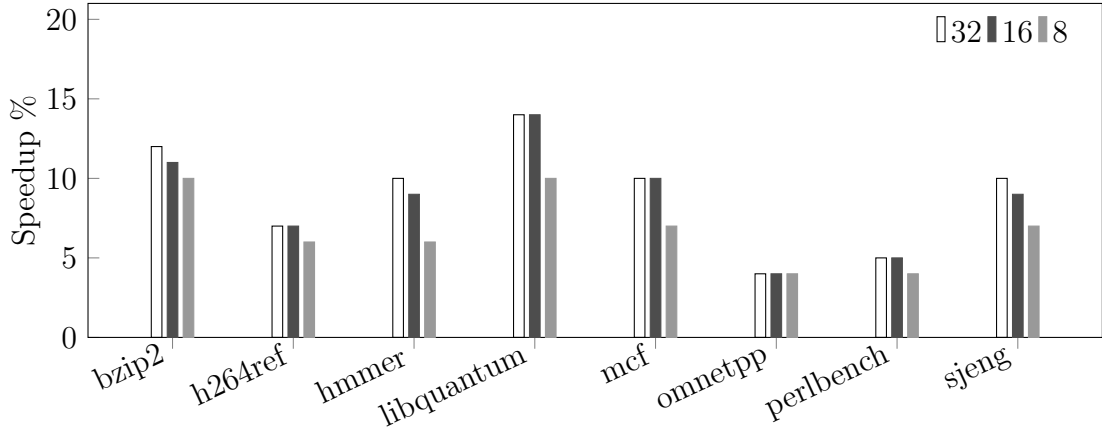


FIGURE 2.14: Performance sensitivity to load tag table (LTT) size.

entries, we find that 16 load tags are sufficient to capture 95% of the performance gains from decoupled loads. When we reduce the number of tags from 16 to 8, hmmer, libquantum and mcf performance suffers. This sensitivity analysis satisfies a key design objective, minimal microarchitectural support, and motivates future work in adapting the processor datapath for decoupled loads.

2.3.3 Side Effects

Figure 2.15 shows instruction overheads from decoupled loads. Decoupled loads separate a load instruction into two parts – data access and register write – and may increase code size. For our benchmarks, decoupled loads increase static and dynamic code size by an average of 5.8% and 7.4%, respectively.

Increases in static code size could reduce the instruction cache’s hit rates, harming performance. However, we observe negligible performance degradation. SPEC2006 benchmarks behave well from the instruction cache’s perspective [39]. Moreover, performance penalties from instruction cache misses are small for in-order cores, which suffer from head-of-line blocking frequently. Misses do not harm performance when instructions are off the critical path or are not immediately ready for execution upon entering the window [56].

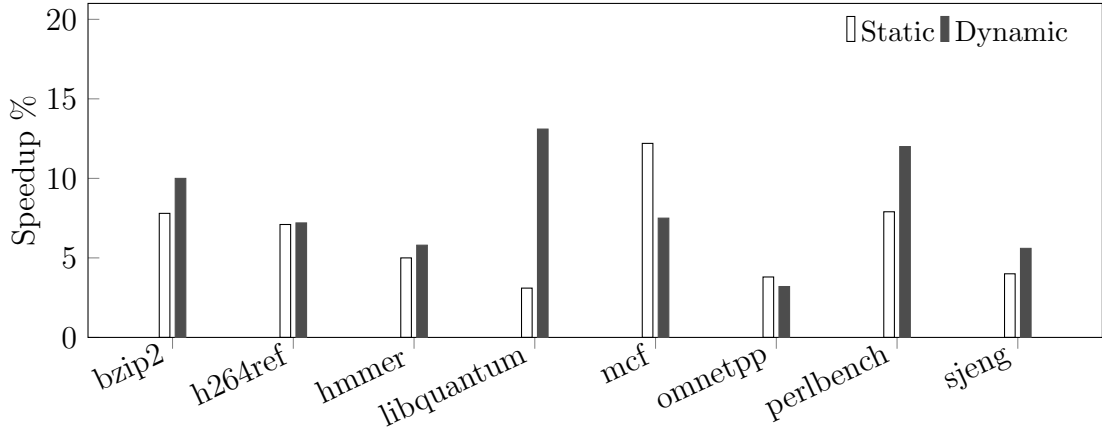


FIGURE 2.15: Code size and overhead.

Increases in dynamic code size could increase contention for issue slots. In practice, however, the compiler decouples loads to fill slots that would otherwise stall the pipeline. Only the data access part of redundant loads contend with useful instructions for issue slots. But we find that the number of redundant loads is very small.

2.4 Related Work

Branches perform multiple operations in a single instruction. By decomposing a branch into prediction and resolution instructions, control flow transfers can be hoisted above branch resolution [56]. Breaking both branches and loads into their constituent parts will produce even better schedules.

Load Latency. Microarchitectural mechanisms have been proposed to tolerate cache miss latencies [46, 15, 87, 32, 33]. CFP uses a slice buffer to drain the missed load and its dependent instructions, freeing issue queue and register file resources [87]. iCFP adapts CFP for in-order pipelines, unblocking latches and allowing independent instructions to execute [32]. SLTP and Multipass Pipelining present other implementations of non-blocking schemes [64, 8]. State-of-the-art iCFP achieves reach 57% of OoO performance with IO design.

Prefetching reduces cache miss latency. Software prefetching inserts explicit prefetch instructions for memory references that are likely to miss in cache [72]. Compilers can detect memory access patterns and tune for varied latencies [61]. Prefetching is especially effective given regular memory access patterns [44, 62], but has also been applied to pointer-based data structures [41, 73]. Prefetching does not hide load-to-use latency once data is in the L1 cache. Indeed, decoupled loads could improve performance even if the system were to use an ideal prefetcher.

The Decoupled Access/Execute Architecture (DAE) [85] decouples operand access and execution with two instruction streams that communicate via queues. DAE is orthogonal to decoupled loads since the former is a microarchitectural implementation and the latter is an architectural extension. However, in a direct comparison with decoupled loads, DAE requires much more complex hardware and its loads remain serialized by stores and branches.

Instruction Scheduling. Dynamic optimization re-schedules code to reflect runtime behavior, adding new code to the application address space [23, 57] or into a hardware cache [40, 63]. rePLay and Region Slip support dynamic optimization and allow region schedules to overlap [65, 86]. Each of these techniques require extensive hardware support (*e.g.*, frame constructor, optimization engine and scheduler, frame cache, recovery mechanism).

Schedulers could perform better when given broader scope. Trace scheduling is an early solution to the global microcode optimization problem [29]. Similar approaches include Superblock and Hyperblock formation [36, 52]. These profile-driven techniques are data dependent. They also require mechanisms to support prediction and recovery after misspeculation, either with fix-up code or checkpoint recovery.

IA-64 provides “advanced loads” and “speculative loads” which allow the compiler to schedule a load before one or more prior stores and branches [77]. These

instructions place a check instruction at the original load's location to detect mis-speculation and branch to fix-up codes. Our scheme differs because the compiler transforms code to execute the data access portion of the load non-speculatively.

Previous work has recognized that branches are often encoded to perform multiple operations in a single instruction. Branch delay slots have been used in several several RISC machines, including the IBM 801[68], RISC II[43], and MIPS as a way to separate the control flow transfer from the target specification and condition computation. Delayed branches are only effective when the condition computation is data independent so that it can be done early. Otherwise, the compiler will have to insert no-ops to preserve the correct semantics of the program. For example, among the three machines mentioned earlier, MIPS was able to achieve a branch cost of 1.3 cycle, meaning the single delay slot could only be used about 70% of the time.

2.5 Conclusion

Current instruction set architectures bundle multiple operations into one instruction, which prevents compilers from aggressively re-ordering instructions. We propose decoupled loads to separate data accesses and register writes. Decoupled loads enable better static schedules by allowing compilers to hoist data access above may-alias stores and branches, two major barriers for code motion on conventional loads. Decoupled loads require modest system and microarchitectural support and improve performance by enabling better static schedules.

UTAR: Utility and Thermal Aware Runtime for Online Sprinting Management

Computational sprinting [71, 70] is a mechanism for dark silicon, which describes systems that can only power a fraction of its peak resources [27, 90]. Sprints provide a short but significant performance boost by activating reserve cores and/or increasing frequency and voltage. Sprints draw extra power and temporarily increase chip power beyond its thermal design point (TDP). Sprints often require thermal packages that deploy phase change material to increase the system’s thermal capacitance, buffer heat during a sprint, and dissipate that heat during normal operation.

Because sprints cannot be sustained, the system needs a mechanism to decide when to start and stop a sprint. Such a mechanism needs to assess both the benefit (*i.e.*, performance gains) and the cost (*i.e.*, thermal capacitance consumed). Moreover, because sprint decisions made in the present affect sprint capability in the future, the mechanism needs to pace its sprints to maximize long-run performance. Prior work has either relied on an obvious trigger (*e.g.*, parallel code region activates additional cores [71]) or offline profiles to estimate utility [28]. However,

such approaches are limited to certain types of sprint and cannot adapt to workload dynamics.

We propose UTAR, a utility and thermal aware run-time that determines when to initiate and terminate a sprint. Based on hardware support available in current processors, we present a software framework that evaluates utility and makes sprinting decisions. This framework integrates new approaches to online phase classification with prior approaches in phase prediction. In every management epoch, the framework predicts the workload’s utility from sprinting and assesses the system’s thermal profile. We show that these predictions are accurate and permit judicious sprints over time.

We demonstrate UTAR for a new class of sprints that increase microarchitectural capacity. Specifically, we propose *cache sprints* that expand last-level cache capacity and exceed TDP when that capacity is most useful. While we use the cache as an example, microarchitectural sprints could apply to other microarchitectural resources such as the reorder buffer, issue queue, etc. Our perspective complements prior work, which focused exclusively sprints that activate additional cores and boosts their frequencies.

Our study of cache sprinting demonstrates its viability. We find that utility from extra cache capacity varies across program execution. Such phase behavior permits judicious sprints that increase capacity when utility is high and cool the system when utility is low. Furthermore, our power models indicate that cache sprinting durations can be long enough to capture lengthy, high-utility phases. Longer durations are possible because increasing cache capacity is less power-intensive than activating cores or scaling voltage and frequency.

We implement UTAR in software and evaluate it using a set of cache-sensitive applications. We prototype the run-time system on an Intel Xeon Broadwell that supports last-level cache allocation via Intel Cache Allocation Technology [31]. We

show that UTAR-guided cache sprints improve performance by 17% on average and by up to 40%. Moreover, UTAR outperforms a greedy policy that sprints at every opportunity and performs within 95% of oracular policies that use perfect knowledge of sprint utility and thermal conditions to control sprints.

3.1 UTAR Design

Computational sprinting is a mechanism that temporarily exceeds a chip’s sustainable thermal budget to boost performance. Specifically, it activates additional cores and/or boosts frequency, exceeding the processor’s thermal design point (TDP) by an order of magnitude or more. Managing these sprints (*i.e.*, determining when to start or stop) can be straight-forward. Additional cores can be activated at the beginning of a parallel code region. Benefits from frequency boosts are relatively easy to estimate.

However, managing these and other types of sprints that activate microarchitectural resources to maximize long-run performance is challenging. Thus, we design a general framework called UTAR that can (1) manage microarchitectural sprints, (2) identify sprint opportunities accurately, and (3) minimize hardware and management overhead.

Broad Application. To find broad application to varied microarchitectural sprint mechanisms, UTAR cannot rely on simple triggers like those proposed for computational sprinting. We instead integrate new approaches to online phase classification with prior work in phase prediction to make sprinting decisions based on the inherent characteristics of different application phases.

High Accuracy. One of the most important design objectives of UTAR is to “sprint when it really matters”. Sprinting without justified benefit in the present not only wastes energy but can also potentially hinder future sprinting opportunities. UTAR uses multiple history-based predictors to ensure each decision to sprint is made

with high confidence.

Low Overhead. Since sprinting is a mechanism that is usually applied under an already power and thermal constrained environment, the management framework should not add additional, power-intensive hardware. UTAR is a software runtime system that utilizes hardware support already available on modern processors. Moreover, UTAR works well with coarse-grained epochs that span 100M instructions, making management overheads negligible.

Figure 3.1 motivates these design goals with a snapshot of program execution split into phases. At “current time,” UTAR’s four tasks are to (1) confirm that the previous epoch was actually “phase B”, (2) predict that the next epoch is “phase A”, (3) predict the utility from sprinting in “phase A”, and (4) determine whether the thermal headroom in the next epoch permits a sprint during phase A. UTAR must complete these tasks accurately and efficiently.

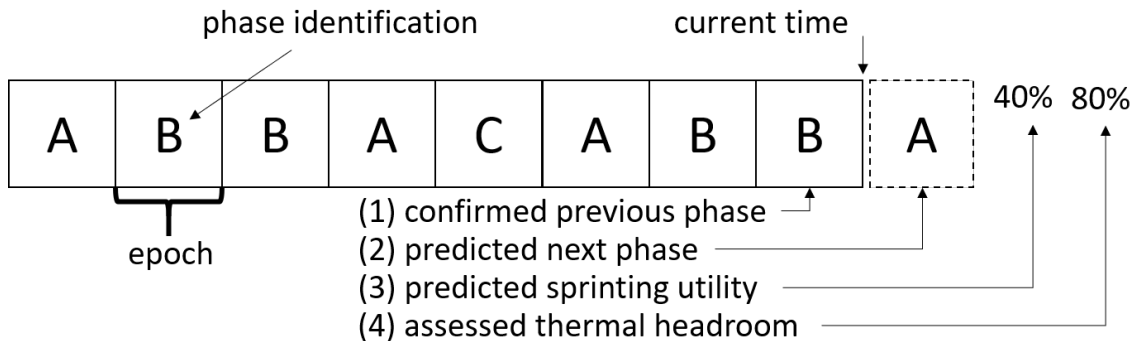


FIGURE 3.1: Example program execution in phases

3.1.1 Management Architecture

Figure 3.2 presents an overview of UTAR. UTAR manages sprinting by collecting data about the previous epoch. When an epoch ends, the phase classifier assesses hardware performance counters, which supply the phase signature, to decide whether the epoch corresponds to a previously observed phase or a new one (Section 3.1.2).

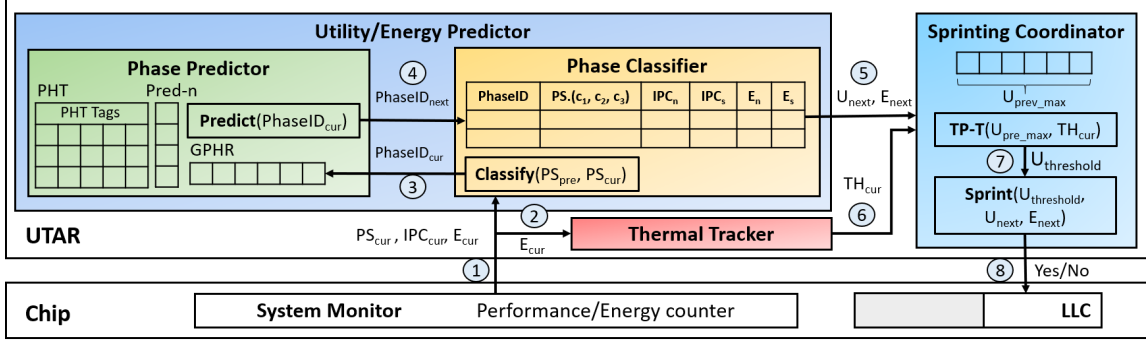


FIGURE 3.2: UTAR overview

UTAR makes a series of predictions about the next epoch to decide whether to sprint. First, the phase predictor uses the most recently completed epoch’s phases to query the phase history pattern table and predict the next epoch’s phase (Section 3.1.3). Second, the utility and energy predictor uses the next epoch’s phase as well as its historical performance and energy measurements to predict the effects of sprinting (Section 3.1.4). Third, the thermal tracker uses the previous epoch’s energy consumption to determine how much thermal headroom remains in the system (Section 3.1.5). Finally, the sprinting coordinator aggregates predictions to decide whether to sprint in the next epoch (Section 3.1.6).

3.1.2 Phase Classifier

UTAR classifies epochs into phases to facilitate the prediction of utility from sprints. Epochs associated with the same phase exhibit similar characteristics such as instruction level parallelism, memory intensity, branch prediction accuracy, etc. UTAR captures these characteristics using hardware performance counters available on most modern processors. UTAR uses a phase signature, defined by data from a few counters, to classify each epoch into a corresponding phase.

Phase Signature. UTAR constructs a phase signature from three performance counters: the number of L1 and L2 cache misses per thousand instructions and the number of branch mispredictions per thousand instructions (abbreviated L1, L2, BR

MPKI).

These hardware counters present a number of advantages. First and foremost, these counters are good indicators of program phase and cache utility. They measure activity in both datapath and caches, allowing them to distinguish between program phases. Moreover, these counters are independent of last-level cache capacity, producing the same phase signatures for the same instructions in both normal and cache sprinting modes. Finally, these counters are available on most processors and programs, making UTAR compatible and portable.

Phase Signature Boundaries. UTAR tracks phases and their corresponding signatures during program execution. For each epoch, UTAR determines whether its signature matches a previously observed phase or differs enough from prior signatures to describe a new phase. UTAR makes this determination by specifying boundaries around phases and their corresponding signatures. When a measured signature lies within an existing phase’s boundaries, the epoch is associated with that phase. When it lies beyond any existing boundary, the epoch is associated with a new phase.

An epoch with a phase signature of N counters can be viewed as a point in a N -dimensional space with each counter being its respective coordinate. Figure 3.3 shows a 2D example in which a phase signature only consists of two counters: L1 MPKI and BR MPKI. In this example, the phase classifier has already identified three phases: A, B and C, represented by the rectangles with the solid lines. The dots within each rectangle represent the program epochs that have been classified into the corresponding phase. Suppose the coordinates of the next epoch “x” are within the rectangle of phase B, then x is classified into phase B, and phase B’s boundary is updated after including x. However, suppose the coordinates of the next epoch “y” are not within any of the existing rectangles, then y forms a new phase D.

Algorithm 4 details the analysis of phase and signature boundaries. When an

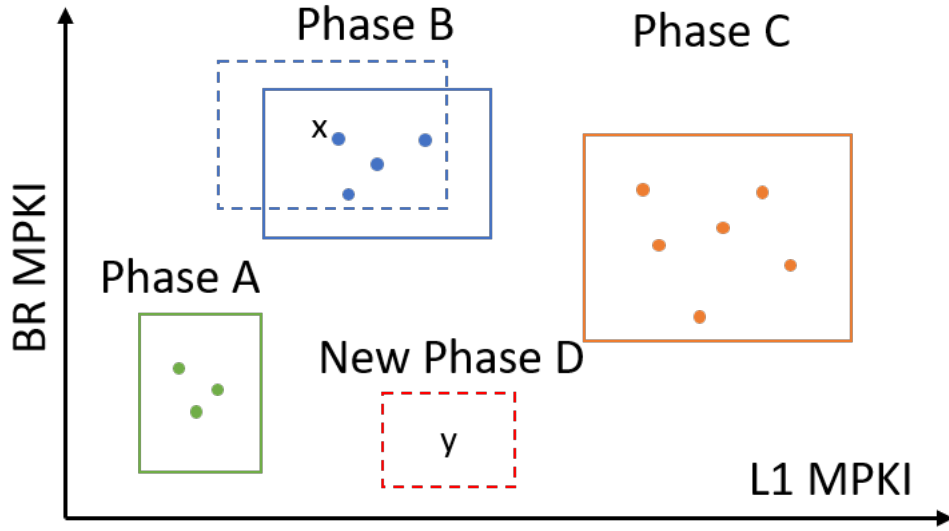


FIGURE 3.3: Phase Classifier Example

epoch ends, UTAR collects a signature for the current epoch PS_{cur} . The algorithm determines whether that signature lies in the neighborhood of signatures for previous observed phases $PS_{pre}[i]$.

Algorithm 4 Boundary-based Phase Classification

```

1: procedure CLASSIFY( $PS_{pre}[\ ]$ ,  $PS_{cur}$ )
2:   PhaseID = -1
3:   closest = INT.MAX
4:   for  $PS_i \in PS_{pre}$  do
5:     if for every  $C_j \in PS$ :  $\frac{|PS_i.C_j - PS_{cur}.C_j|}{PS_i.C_j} < B_{rd}$  or
6:        $|PS_i.C_j - PS_{cur}.C_j| < B_{ad}$  then
7:       dist = ED( $PS_{cur}$ ,  $PS_i$ )
8:       if dist  $\downarrow$  closest then
9:         closest = dist
10:      PhaseID =  $PS_i$ .PhaseID
11:  if PhaseID == -1 then
12:     $PS_{cur}$  forms a new phase

```

Phase signature PS_1 is within the boundary of signature PS_2 if, for every counter C_j in the signature, the relative difference between $PS_1.C_j$ and $PS_2.C_j$ is smaller than boundary parameter B_{rd} or the absolute difference between $PS_1.C_j$ and $PS_2.C_j$ is smaller than parameter B_{ad} . We must assess both absolute and relative differences

because comparisons between small values (e.g., those less than one) can produce large relative differences. Finally, UTAR associates the current epoch with the nearest signatures and phase based on the Euclidean distance.

Boundary Parameters. The values of B_{rd} and B_{ad} determine phase granularity, which ultimately determines the total number of phases that describe the program. The bigger the boundaries, the smaller the number of phases. On one hand, fewer phases mitigate classifier overheads, which are linear in the number of phases. On the other hand, more phases increase analysis granularity and permit more accurate predictions of utility and energy. Finer granularities lower variance and improve prediction for utility and energy for each phase. We set boundary parameters to $B_{rd} = 0.30$ and $B_{ad} = 2$ and show, in Section 3.4.3, that these parameters require a small number of phases yet predict utility and energy accurately.

Example. We illustrate phase classification using a simple example in Figure 3.4. In this snapshot of the program execution, the phase classifier has already identified 3 different phases and recorded their corresponding phase signatures. At the end of the current epoch, the phase classifier reads the phase signature PS_{cur} , compares it with the phase signature of each of the 3 phases and identifies that it is within the boundary of phase 2. The process is repeated for the next epoch. However, this time, the phase classifier finds PS_{next} is not within the boundary of any existing phase. As a result, the next epoch is assigned to a new phase with phaseID 3.

Note that UTAR does not use a standard classification algorithm like K-means because that requires prior knowledge of k (i.e., the total number of phases in an application) and it is impractical to know this value for every different applications. Instead, the boundary-based classification approach allows UTAR to discover the number of phase clusters dynamically and thus is easy to adapt to different applications. Moreover, invoking k-means at every epoch is computationally expensive thus less desirable in a software runtime framework.

Phase Classifier					
PhaseID	PS.(c ₁ , c ₂ , c ₃)	IPC _n	IPC _s	E _n	E _s
0	(2.52, 1.56, 1.84)
1	(22.37, 16.34, 0.02)
2	(9.76, 9.53, 0.01)

$PS_{cur} (9.76, 9.78, 0.01) \rightarrow \text{Phase 2}$
 $PS_{next} (2.01, 1.72, 3.59) \rightarrow \text{New Phase: 3}$

FIGURE 3.4: Phase Classifier Example

3.1.3 Phase Predictor

UTAR implements phase prediction using a Global Phase History Table (GPHT) predictor [38]. First, the classifier assigns the just-completed epoch to a phase. Then, it uses the classified phase to index and update the history table. Finally, it uses the table to predict the next epoch’s phase.

The green box in Figure 3.2 details the Global Phase History Table (GPHT). Its main structure consists of a global shift register, called the Global Phase History Register (GPHR), that tracks the last few observed phases. GPHR contents are used to index into a Pattern History Table (PHT), which caches several previously observed phase patterns (PHT Tags), their corresponding predictions for next phase (PHT Pred-n), as well as recency information (Age/Invalid). When GPHR and PHT tags do not match, the predictor falls back to Last Value Prediction, predicting the last observed phase, stored in GPHR[0], as the next phase. Such table-based history predictors are much more effective than simple statistical predictors, particularly for highly variable programs [22, 38].

This phase prediction strategy exploits the fact that many programs consist of phases that have similar characteristics and behaviors. Accurately predicting these phases dynamically, as the program runs, benefits various online optimizations such as hardware reconfiguration, voltage and frequency scaling (DVFS), thermal management, and hotcode optimization [9, 35, 20, 34, 84, 5, 94, 38, 51]. Phase behaviors are repetitive and prior work has proposed table-based history predictors to capture past phase patterns for future prediction [22, 38].

3.1.4 *Utility and Energy Predictor*

UTAR uses historical data, recorded in the phase classifier, to predict the utility from sprinting in the next epoch. The classifier tracks, for each phase, four measures—instruction throughput and processor energy when running in normal and sprinting modes (IPC_n , IPC_s , E_n , E_s).

Updating the Predictor. When an epoch is classified and assigned to a phase, its performance and energy profile updates the classifier’s data corresponding to its phase and mode. Initially, when the classifier encounters the first epoch observed for a phase, it will record the performance and energy profile for either the normal or sprinting mode. Later, when the classifier encounters the second epoch for the same phase, the sprinting coordinator collects data for the other mode by initiating or halting a sprint.

This mechanism ensures the classifier collects data for normal and sprinting modes at least once for each phase, thereby exploring the utility from sprinting. After the classifier has initialized a phase’s entry with its first two epochs, performance and energy histories are updated automatically as the sprinting coordinator makes decisions at run-time.

Recovering from Misprediction. The case of misprediction, in which two epochs’ phases are classified incorrectly and thus share the wrong utility and energy

history, causing the predictor to make incorrect sprinting decisions, (e.g., not sprinting in high-utility epoch) is possible. We show in later section that UTAR achieves very high accuracy. However, even in the rare case of misprediction, UTAR can recover by accessing predictor confidence (based on age of table history) and exploring sprints give low-confidence predictions of low-utility epochs.

Invoking the Predictor. The phase predictor forecasts the next epoch’s phase, producing a phase ID. This ID indexes into the phase classifier’s table to produce corresponding historical data for instruction throughput and processor energy, which serve as the predicted utility and cost from sprinting in the next epoch.

3.1.5 Thermal Tracker

The thermal tracker monitors power dissipation and thermal headroom to constrain sprint decisions. The tracker measures power dissipated by the system with either live measurements or power models [2]. Power measurements drive a thermal model that calculates the thermal headroom, which quantifies the amount of heat (measured in Joules) that can be expended before exceeding the processor package’s thermal capacitance. When the package uses an engineered phase change material [71], the thermal model includes the following parameters:

- C_{pcm} : The phase change material’s thermal capacitance, which determines the amount of heat that can be buffered.
- R_{pcm} : The phase change material’s thermal resistance, which determines the maximum power that can be dissipated during a sprint.
- R_{package} : The processor package’s thermal resistance, which determines how quickly heat transfers from the phase change material into the ambient after a sprint.

- R_{total} : The system’s total thermal resistance, which determines the processor’s maximum sustained power.

Let us denote the system’s nominal power as P_n , its sprinting power as P_s , and its thermal design point as P_t . Furthermore, denote the system’s thermal headroom as H_t at time t . Suppose the system sprints from t_1 to t_2 without exhausting the thermal headroom and then operates in normal mode from t_2 to t_3 . We can estimate the thermal headroom at these points in time.

$$\begin{aligned}
 H_2 &= H_1 - (P_s - P_t) \times (t_2 - t_1) \\
 H_3 &= H_2 + (P_t - P_n) \times (t_3 - t_2)
 \end{aligned}$$

3.1.6 *Sprinting Coordinator*

The sprinting coordinator initiates and terminates sprints. The decision to sprint must balance utility in the present and the future, a trade-off encapsulated by two questions. First, how significant is utility in the next epoch U_{next} compared to potential utilities further into the future? Second, if the system sprints and reduces thermal headroom in the present, will it be able to sprint and exploit high-utility epochs in the near future? Answers to these questions would permit the system to use thermal headroom judiciously and enhance performance over the long run.

The coordinator pursues these objectives given predictions of utility and models of thermal headroom for the next epoch. If the coordinator is aggressive and sprints when utility is low, it may deplete the system’s thermal headroom that might have been helpful further in the future. If the coordinator is conservative and does not sprint when utility is low, it may lose an opportunity to translate thermal capacitance into performance.

Thermal Proportional Threshold (TP-T). We propose a policy, TP-T, that determines whether the coordinator initiates a sprint in the next epoch. Shown in

Equation 3.1, TP-T sets a threshold for initiating a sprint $U_{\text{threshold}}$ by multiplying the fraction of the thermal headroom already consumed by the maximum utility U_{max} derived from recent sprints (*e.g.*, over the last ten epochs). The coordinator initiates a sprint if predicted utility exceeds the threshold and the system possesses sufficient thermal headroom.

$$U_{\text{threshold}} = U_{\text{max}} \times \frac{H_{\text{consumed}}}{H_{\text{total}}} \quad (3.1)$$

In effect, the coordinator treats thermal headroom as a resource whose value varies depending on the amount available and it treats utility as the return from spending that resource. As the amount of thermal headroom decreases, it becomes more valuable and the coordinator requires a higher return when consuming it.

3.2 Cache Sprinting Architecture

We highlight the advantages of UTAR for cache sprinting, an instance of a new class of sprinting techniques we call “microarchitectural capacity sprinting.” Cache sprinting briefly expands the processor’s last-level cache (LLC) capacity beyond what is dictated by the thermal design point. During normal operation, half of the LLC ways are powered off and unused. During a sprint, the remaining ways are powered on. When sprinting, the LLC dissipates more dynamic power as the cache supplies more data in response to hits. Moreover, the LLC’s additional cache ways dissipate more dynamic power as each access must check twice the tags. Finally, larger caches dissipate more static power, independent of the number of accesses.

3.2.1 Case for Cache Sprinting

Cache sprinting complements prior mechanisms in computational sprinting that activate additional cores and boosts their frequencies. Sprinting with the last-level cache serves memory-bound applications that do not benefit from boosted processor cores.

Figure 3.5 shows the performance gains for two alternative sprint mechanisms that consume 2W of extra power—scaling frequency and increasing LLC capacity—on an Intel Xeon Broadwell processor. Scaling frequency from 2.1GHz to 2.3GHz improves performance for compute-intensive applications by 10% on average but has little effect for memory-intensive applications. In contrast, increasing cache capacity from 2MB to 4MB improves performance for memory-intensive applications significantly. Although the potential benefits of LLC sprints are significant, the management mechanism must exploit that potential.

Sprints make sense only if the utility of increased cache capacity varies across program execution, permitting sprints when utility is high and cooling when utility is low. We find that this is indeed the case for most workloads and illustrate with gcc as an example in Figure 3.6. The x-axis shows epochs of 100M instructions and the y-axis shows gcc speedup with a 4MB LLC over 2MB LLC in that epoch. Utility varies from as little as 0% (an excellent time to cool) to as much as 80%, with sustained periods of 70% (excellent times to sprint).

LLC has much lower power density than processor cores used in prior approaches to computational sprinting [90]. Given the sprint’s smaller power delta due to the cache’s lower power density and smaller difference between max and average power, there is an opportunity to sprint longer and more aggressively before hitting thermal constraints. Note that our framework generalizes beyond LLC sprints to other memory microarchitectures or technologies. For example, 3D-DRAM could offer hundreds of MB of LLC but would benefit from a sprint framework that exploits its huge capacity while managing its limited cooling capabilities.

3.2.2 System Architecture

UTAR is a software runtime system that controls cache sprints. Software decides when to sprint and communicates that decision to hardware via a control register.

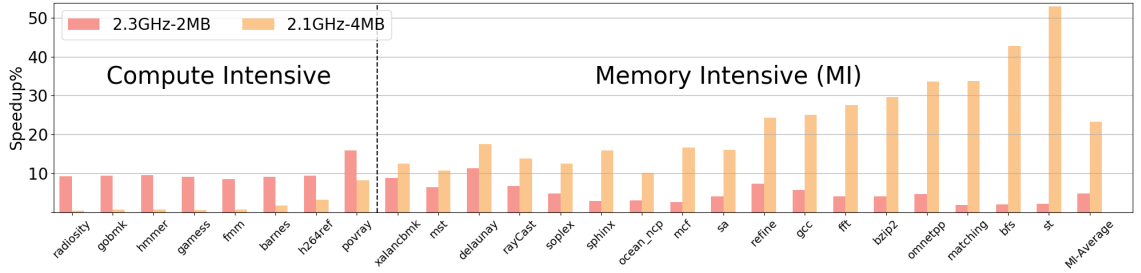


FIGURE 3.5: Performance comparison: scaling frequency vs increasing LLC capacity, baseline 2.1GHz with 2MB LLC

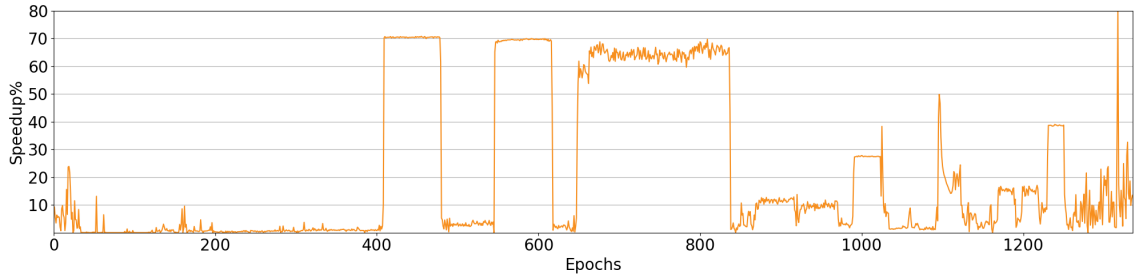


FIGURE 3.6: Performance gain: gcc-s04 running with 4MB LLC over baseline 2MB

Although full hardware control is possible and permits decisions at finer granularities, software control permits more sophisticated management policies. The operating system can track information about each process’s behavior and perform complex computation at coarse granularities to make informed sprinting decisions that reflect program behavior.

When a sprint starts, the additional cache ways are empty and require time to fill. If the sprint is timed well, the ways fill with useful data, the hit rate increases, and workload performance improves. As the sprint progresses, it consumes the chip’s thermal headroom by raising the processor package’s temperature toward safety tolerances or transforming the phase change material into an amorphous state. The sprint ends when the software (i.e., operating system) finds little utility from extra cache capacity or the hardware (i.e., thermal monitors) finds that the thermal headroom has been exhausted.

Sprints must end with sufficient time and thermal headroom to write dirty data

to memory before powering down. In an extreme case, every block in the additional ways is dirty and block addresses are distributed poorly across memory pages. For example, writing back 16MB of dirty data would require less than 500 microseconds, a negligible transition delay given sprints that span several seconds.

Although software can make decisions about utility, it cannot be trusted with the chip’s physical safety. Hardware must monitor thermal headroom and halt sprints, even if it means overriding the operating system, when headroom is exhausted. Specifically, hardware must set the machine state register’s (MSR’s) sprint bit to zero and ignore writes to that register until thermal headroom has been sufficiently restored so that the cache can safely sprint without jeopardizing thermal budgets. Finally, in the event of a thermal emergency, the hardware controller can halt the processor cores until cache ways are powered off.

3.3 Experimental Methodology

System Setup. We emulate LLC sprints on an off-the-shelf chip multiprocessor, restricting its LLC capacity in nominal mode and restoring its capacity in sprint mode. We focus our sprint evaluation on serial workloads running on a single core and size LLC capacity accordingly.

Table 3.1 summarizes the system configuration. We conduct experiments on physical hardware using an 8-core Intel Broadwell Xeon E5-2620 v4 processor. The Xeon processor has a 20MB, 20-way last-level cache, which corresponds to 2.5MB per core. In normal mode, we configure Intel’s Cache Allocation Technology(CAT) [31] to restrict a core and its single-threaded program to use 2MB (not 2.5MB because CAT uses way-partitioning to allocates cache capacity at one-way, 1MB granularity). In sprinting mode, we permit a core to use 4MB of cache. Note that CAT maintains the victim buffer when shrinking the last-level cache, mitigating the conflict misses that would have normally occurred with a low-associative LLC.

We configure the system to minimize interference and ensure deterministic results. First, we disable C-states and DVFS, fixing all cores’ frequencies to 2.1GHz. Second, we disable the Watchdog hang timer as well as the Address Space Layout Randomization. Third, we stop all non-essential system daemons/services. Fourth, we use cset command for CPU shielding so that other processes will not run on the core targeted for experiments. Finally, we run each application three times and use the average number for all reported performance values. We collect the relevant hardware counters’ values for phase signatures, from Section 3.1.2, using Performance Application Programming Interface (PAPI) [91]. These numbers are collected every 100 million instructions using the PAPI_overflow() function.

Field	Value
Core	8-core Broadwell, 2.1GHz
L1 Caches	32KB, private, split D/I, 8-way
L2 Caches	256KB, private, 8-way
L3 Cache	20MB, shared, 20-way Way-partitioning with Intel CAT [31]
Memory	8GB, RDIMM, DDR4 2400MT/s
OS	Red Hat Enterprise Linux 7.5 Linux kernel version 3.10.0

Table 3.1: System Specification

Applications. We evaluate applications from the SPEC CPU2006 [30], SPLASH2 [93] and PBBS [82] benchmark suites. We focus on the 17 LLC-sensitive applications from Figure 3.5, which exhibit higher performance gains from larger cache capacity than with processor frequency scaling. Applications in SPEC CPU2006 and SPLASH2, are run to completion with the largest input size, which usually takes minutes. Applications in PBBS are much smaller and are run multiple times until the total number of instruction is at least 100 billion, each time with a different input of roughly the same size.

Power Model. We estimate our system’s power consumption using Intel’s Running Average Power Limit (RAPL) driver [2]. Static power is estimated by measuring a microbenchmark that only calls the `sleep()` function while dynamic power is measured during the execution (minus estimated static power).

On our system, the available RAPL domains only provide power for the package (cores + LLC + memory controller) as a whole. We use CACTI and power breakdowns from prior work [14] to estimate the individual power of cores and LLC. In summary, we estimate the nominal and sprinting power of a scaled system (single core + 2MB LLC) to be 6W and 8W, respectively, in which LLC consumes about 1W per MB.

Note that we do not include DRAM power into the model because it does not affect processor sprints. Processors and DRAMs are cooled separately thus off-chip DRAM’s thermal budget is independent from the on-chip cores and caches’ thermal budget. On the other hand, effective sprints should reduce DRAM power because the additional cache capacity helps reduce memory traffic.

Thermal Model. Our thermal model is similar to that in prior work, which integrates PCM close to the chip to increase thermal capacitance [71, 70]. Specifically we consider paraffin wax, which is attractive for its high thermal capacitance and tunable melting point when blended with polyolefins [69]. We consider the amount of PCM that can absorb approximately 10J of heat before melting completely, thus enabling a sprint spanning tens of seconds. Given paraffin wax’s latent heat of 200J/g, the system only needs to deploy 0.05g of PCM to provide such extra thermal capacitance. Although processors exhibit hotspots (e.g., register file), we assume packages deploy thermal interface materials with low thermal resistivity to distribute the heat prior to PCM contact [6].

Note that our experiments are performed on real hardware whose TDP already accommodates 20MB-LLC, so no actual PCM is deployed, but the PCM parameters

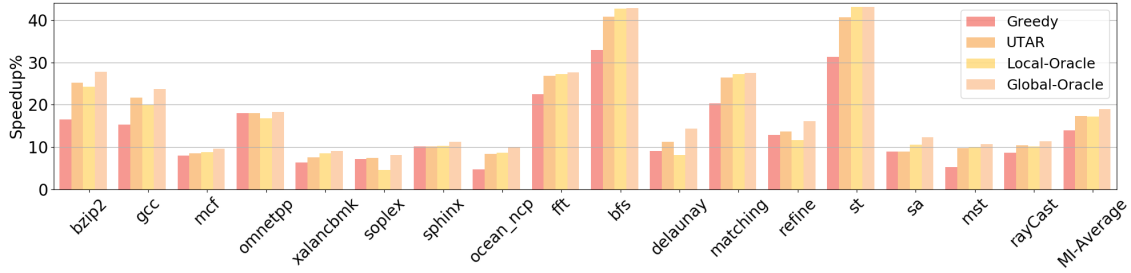


FIGURE 3.7: UTAR Performance: compared against various policies

affect the run-time system’s model of thermal headroom. Finally, the baseline system has a TDP of 7W for 1 core and 2MB LLC. We evaluate performance sensitivity to these system parameters in Section 3.4.4.

3.4 Evaluation

We evaluate UTAR and its thermal-proportional threshold policy against several alternatives. Since there is no prior work in managing LLC sprinting, we compare against three other policies that vary in their knowledge of sprint utility and thermal headroom.

Greedy (G) initiates a sprint whenever thermal headroom is available, ignoring utility. Once thermal headroom is exhausted (i.e., PCM transition to amorphous state completes), it waits for the system to cool and restore thermal headroom before starting the next sprint.

Local-Oracle (LO) has perfect knowledge of sprint utility from all prior epochs ($U_{history}$) as well as the next epoch (U_{next}). It initiates a sprint if U_{next} is greater than the average of $U_{history}$ and stops otherwise. LO is an oracular policy, which represents heuristics that only incorporate local information obtained through online profiling or prediction. The sprint threshold is set based on utility alone.

Global-Oracle (GO) has perfect knowledge of sprint utility from all past and future epochs in the program’s execution. It sorts epochs based on their sprint utili-

ties and iteratively selects the epoch with the highest utility for sprinting execution. If the selected epoch cannot sprint due to lack of thermal headroom, the epoch is marked for normal execution and GO proceeds to the next epoch. GO uses global information obtained through offline profiling and cannot be implemented online. However, this policy provides an upper bound on performance from cache sprinting.

Thermal-Proportional Threshold (TP-T) predicts the sprinting utility of the next epoch (U_{next}) and tracks the maximum utility observed from a sprint in the recent past (U_{max}). Then it sets the sprint utility threshold ($U_{\text{threshold}}$) by multiplying the percentage of the already consumed thermal headroom and U_{max} as shown in Equation 3.1. Finally it initiates a sprint if U_{next} exceeds $U_{\text{threshold}}$.

3.4.1 UTAR Performance

Figure 3.7 shows the performance gains of LLC sprinting mechanisms, over a baseline that operates under nominal power. UTAR-guided LLC sprinting improves performance by 17%, on average, and by up to 40%. When compared with alternatives, UTAR’s TP-T policy outperforms greedy heuristics and is competitive with the oracular policies.

Comparison against Greedy (G). TP-T significantly outperforms G for 8 of 17 benchmarks because its sprints are more judicious and timely. Figure 3.8(a) shows the probability density of sprint utility for three of these applications: gcc, ocean and bfs (one from each benchmark suite). Their sprint utility distribution often appears bimodal, revealing many epochs that benefit greatly from sprinting and many that do not. The large variance in sprint utility offers TP-T an opportunity to spend its limited thermal budget wisely to maximize performance gains. For instance, ocean frequently achieves large performance gains between 50 to 60% and also achieves small performance gain between 0 to 20%. TP-T is more likely to reserve its sprints for the large gains. Greedy, on the other hand, could waste its thermal budget in

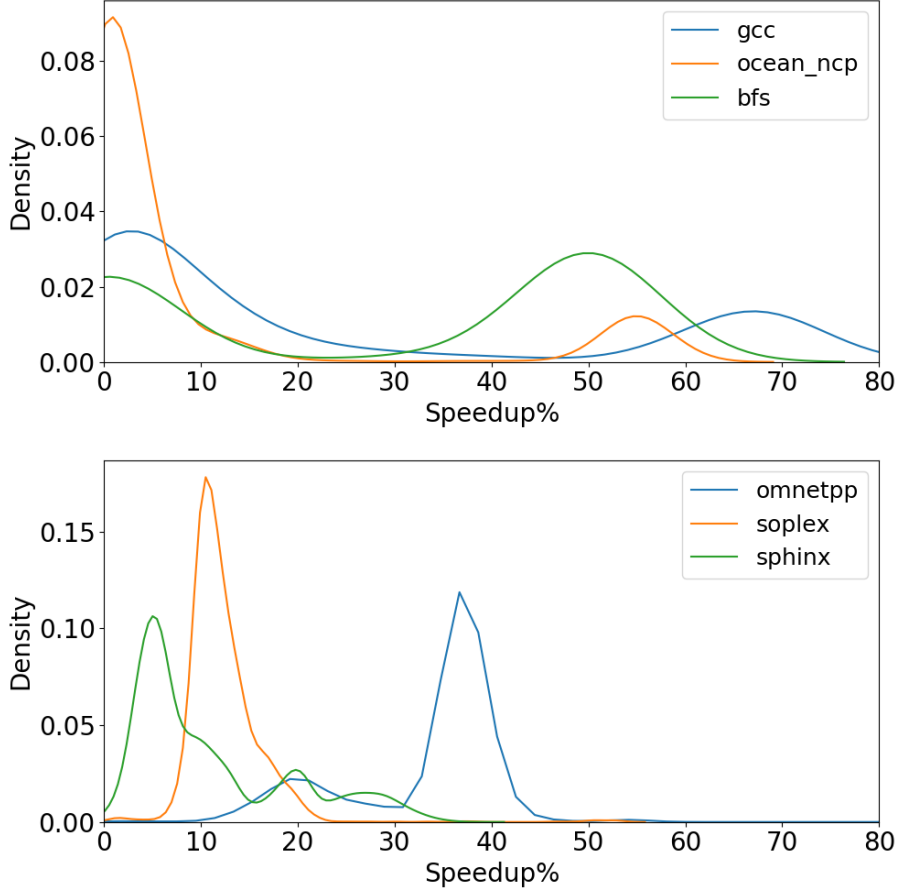


FIGURE 3.8: PDF for utility: (a) bimodal, (b) unimodal

low-utility epochs and have insufficient headroom for high-utility ones.

Greedy performs comparably to TP-T for several benchmarks. Figure 3.8(b) shows the probability density of sprint utility for three of these applications. They often exhibit very small variance in sprint utility and all epochs benefit similarly from sprinting. For example, most of soplex’s epochs report utility between 10 to 20%. There is not much opportunity to prioritize sprints for high-utility epochs. As a result, TP-T’s threshold will be lower than the average sprint utility for most epochs when thermal headroom is available. In effect, TP-T behaves very similarly to greedy.

Comparison against Local Oracle (LO). TP-T’s performance is close to

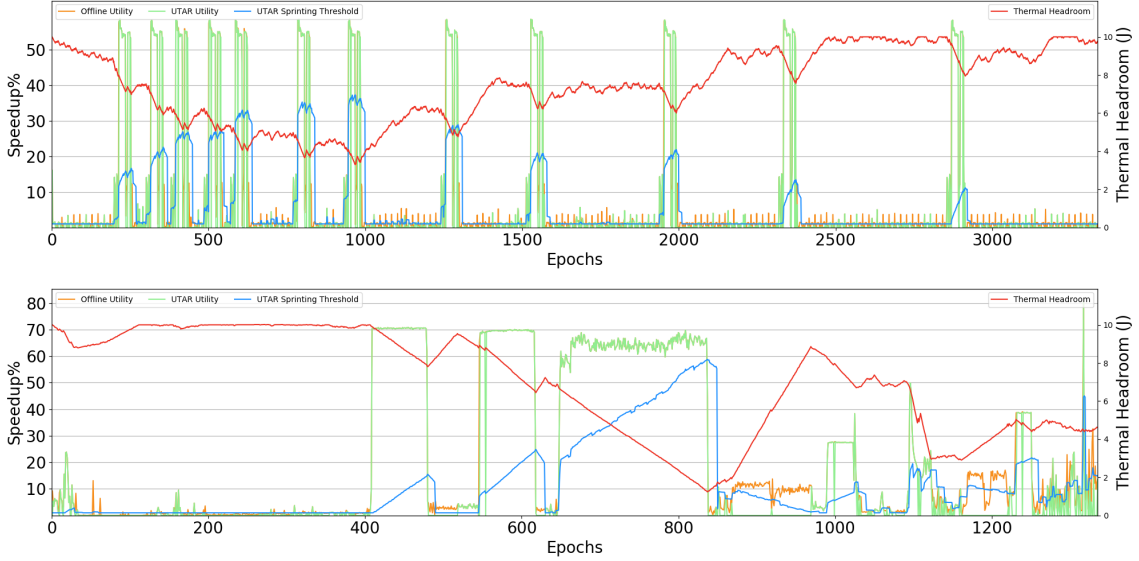


FIGURE 3.9: UTAR Sprinting Behavior: (a) ocean_ncp, (b) gcc-s04

LO's. Indeed, TP-T performs better than LO for 7 of 17 benchmarks. Although LO has perfect knowledge of local utilities, its sprint thresholds neglect the system's current thermal conditions. As a result, LO can be sprint too conservatively when thermal headroom is abundant and too aggressively when thermal headroom is scarce. TP-T is aware of thermal constraints and paces its sprints according to available headroom.

There are cases where LO outperforms TP-T and is close to GO (e.g., bfs, st) due to two reasons. First, because of their bimodal utility distribution, when LO sets sprint thresholds based on average of local utilities, the high-utility epochs will also exhibit utilities greater than the thresholds. Second, these applications tend to have a small number of recurring phases such that local information captures global behavior.

Comparison against Global Oracle (GO). On average, TP-T performs within 95% of the performance upper bound set by GO. GO has perfect knowledge of sprint utility for every epoch and perfect knowledge of the system's thermal condition.

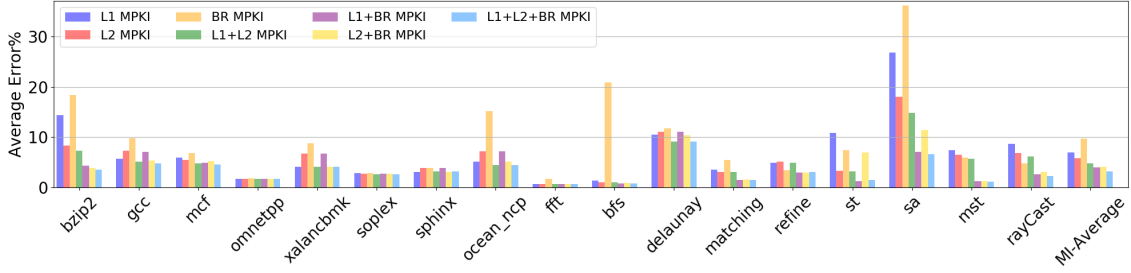


FIGURE 3.10: Impact of phase signature definition on phase classification accuracy

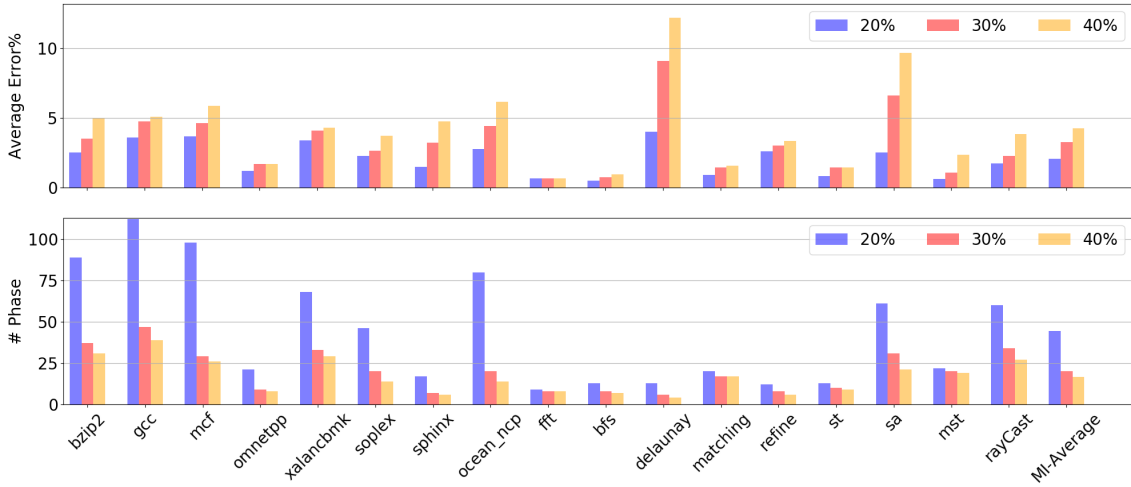


FIGURE 3.11: Impact of signature boundary on: (a) phase classification accuracy, (b) total phase number

This oracular knowledge persists even as epochs are selected for normal or sprinting computation. The small performance gap between TP-T and GO indicates that UTAR accurately predicts sprint utility and that the TP-T thresholds are effective for pacing sprints to maximize performance gains over the long run.

3.4.2 Sprinting and System Dynamics

Figure 3.9(a) shows system dynamics during the complete execution of `ocean_nrp`. The orange line plots offline profiles of sprint utility. The green line plots online measurements of sprint utility achieved from UTAR’s TP-T policy. Note that the orange and green lines align in most cases. The red line shows the available thermal

headroom. The blue line shows UTAR’s threshold for deciding to sprint.

The figure illustrates two major system behaviors. First, UTAR accurately identifies high-utility epochs and executes sprints during these times. For epochs that benefit most from sprints, the orange line overlaps with the green line. Second, UTAR dynamically adjusts its sprint threshold according to recently observed performance data and the available thermal headroom according to the TP-T policy.

`Ocean_ncp` is representative of applications that have a few highly-repetitive phases and have a bimodal utility distribution. UTAR can manage sprinting effectively for these applications because its phase classifier and predict can capture phase behaviors effectively. Moreover, its TP-T policy can easily identify high-utility epochs and trigger timely sprints.

Figure 3.9(b) shows the same system dynamics for a different type of application, `gcc`. Compared to `ocean_ncp`, `gcc` has many more phases and those phases are much less repetitive. `Gcc` is representative of applications that go through many different stages during their execution. This type of application is harder to manage as its less predictable. However, we see that UTAR still performs well, sprinting at most of the high-utility epochs and dynamically adapting the threshold to phases and thermal budget. The gap between TP-T and GO is mainly caused by phase and utility mispredictions.

3.4.3 *UTAR Prediction Accuracy*

Phase Classification. UTAR assigns epochs to phases as the first step in predicting utility, which is critical to the effectiveness of the TP-T policy. We measure accuracy by assigning an epoch to a phase tracked by the classifier and determining how closely that phase’s performance predicts the epoch’s performance. Error is the percentage difference between the phase’s instruction throughput and the epoch’s actual throughput. Two design elements affect accuracy: the definition of the phase

signature and the boundaries that define neighborhoods around phase signatures.

Figure 3.10 shows classification error when including varied hardware counters in the phase signature. Including more performance counters in the phase signature improves classification accuracy. A single counter can perform poorly (e.g., bzip2, sa) and a second counter significantly improves accuracy. Moreover, using diverse measures of activity improves accuracy. Among signatures defined by two counters, combining either L1 or L2 MPKI with BR MPKI performs better than combining L1 and L2 MPKI. L1 and L2 MPKI both characterize memory intensity while BR MPKI characterizes datapath activity. UTAR uses all three counters to define phase signatures that achieve high accuracy across all applications.

Figure 3.11(a) shows classification error when using varied boundaries (B_{rd}) around phase signatures. Smaller boundaries capture finer-grained phases and improve accuracy. However, Figure 3.11(b) shows that finer-grained phases increases the number of phases the classifier tracks, which increases overhead. UTAR sets $B_{rd} = 0.3$ to limit the number of phases yet accurately classify performance to within 97% of the actual value. Although narrowing boundaries ($B_{rd} = 0.20$) improves accuracy slightly, it significantly increases the number of phases for a few applications. On the other hand, broadening boundaries ($B_{rd} = 0.40$) noticeably reduces accuracy without much reducing the number of phases.

Utility Prediction. Figure 3.12 assesses how the phase classifier’s accuracy translates into utility predictor’s accuracy. Error is the percentage difference between predicted and actual speedups from a cache sprint where speedup is predicted from instruction throughputs reported by the classifier. Overall, UTAR accurately predicts utility and sprint speedups to within 95% of actual values.

Utility prediction accuracy exhibits larger variance across applications than phase classification accuracy. Some applications achieve higher accuracy because of their relatively stable phases and sprint utility (e.g., omnetpp and fft). But some appli-

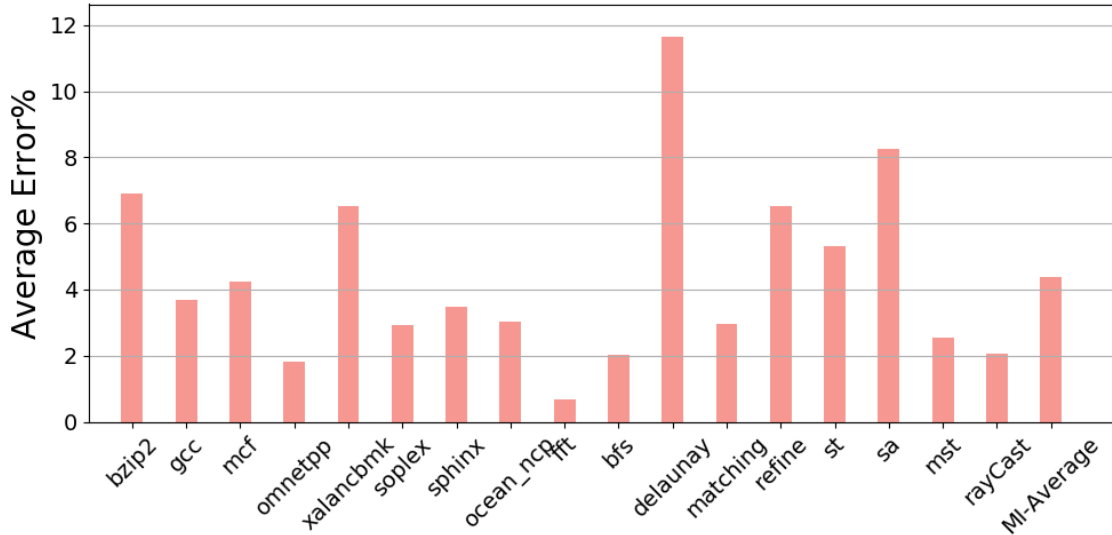


FIGURE 3.12: Utility prediction accuracy

cations are harder to predict because of high variance in their phases (e.g., bzip2) and sprint utility (e.g., xalanbmk). The delaunay and sa benchmarks report the largest errors, for both phase classification and utility prediction, because they have many epochs with similar phase signatures but different performance profiles.

3.4.4 Sensitivity Analysis

Thermal Design Power (TDP). We have been evaluating a system that operates nominally at 6W, sprints at 8W, and is constrained by a 7W thermal design point. For this system, the cooling duration that restores the thermal headroom must match the sprint duration. The time spent generating heat at 1W above TDP must be matched by time spent dissipating heat at 1W below TDP; the sprint-to-cool ratio is 1:1.

Figure 3.13(a) presents performance under tighter thermal constraints where TDP is set to 6.5W, closer to nominal power. As the gap between nominal and thermal design power narrows, the system requires more time to cool after sprint. For our parameters, the cooling duration must be 3× longer than the sprinting duration;

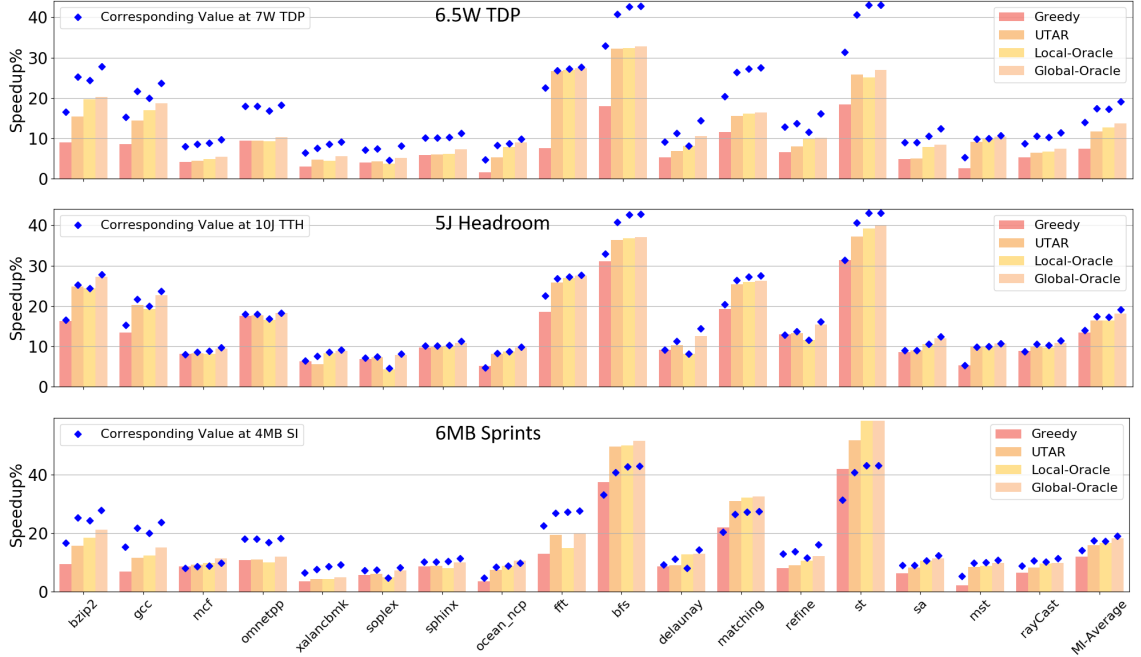


FIGURE 3.13: UTAR performance sensitivity to (a) TDP (6.5W), (b) total thermal headroom (5J), (c) sprinting intensity (6MB)

the sprint-to-cool ratio is 1:3. In effect, sprints become more expensive, degrading performance gains from sprinting across all management policies.

Costly sprints reduce the system’s tolerance to poor decisions. TP-T’s advantage over G grows under tighter thermal constraints (e.g., fft, bfs, mst). TP-T identifies high-utility epochs and sprints judiciously whereas G luckily sprints during high-utility epochs when thermal headroom was generous but suffers from poor decisions when headroom becomes scarce. As sprints become more expensive, prediction accuracy becomes more important and TP-T cannot compete with oracular policies. TP-T underperforms LO, which benefits from perfect knowledge of sprint utility (e.g., bzip2, gcc).

Total Thermal Headroom (TTH). We have been evaluating a system with PCM that provides 10J of thermal headroom. Figure 3.13(b) presents performance when headroom is halved to 5J. Performance is unaffected for most applications.

Headroom determines the duration of a full sprint. For example, 10J permits our system to sprint for tens of seconds. But when applications prefer sprints that exceed the full duration, headroom impacts performance less than TDP because it does not change the maximum sustainable sprint-to-cool ratio. As long as the application derives varied utilities from sprints across time, TP-T will exploit low-utility epochs for cooling and judiciously exploit thermal headroom by dynamically setting the thresholds for sprints.

Sprinting Intensity (SI). We have been evaluating a sprint that doubles last-level cache capacity from 2MB to 4MB. Figure 3.13(c) presents performance when sprints triple capacity to 6MB. More intense sprints do not necessarily translate into long-run performance and, in fact, only three applications benefit. As sprint intensity increases to 6MB, sprint power increases to 10W and the maximum sustainable sprint-to-cool ratio falls to 1:3.

Figure 3.14 shows the time spent sprinting when a sprint expands cache capacity to 4MB or 6MB. Unless performance from 6MB is much greater than that from 4MB, the more intense sprint does not justify the extra power. Cache capacity suffers from diminishing marginal returns and most applications in our study do not benefit significantly more at 6MB compared to 4MB.

When designing a system for bimodal operation, sprint or normal, architects must understand applications' performance sensitivities to resource allocations. Ideally, the sprinting and normal modes are configured to maximize marginal performance gains given a marginal resource allocation. If applications' sensitivities vary, the system could support multiple sprint intensities. UTAR could easily track performance histories for each sprint mode and application phase, deciding which mode improves performance most efficiently. Indeed, designing and managing multiple sprint modes is an avenue for future work.

Comparison to Sustained Operation at TDP. We set our baseline, nominal

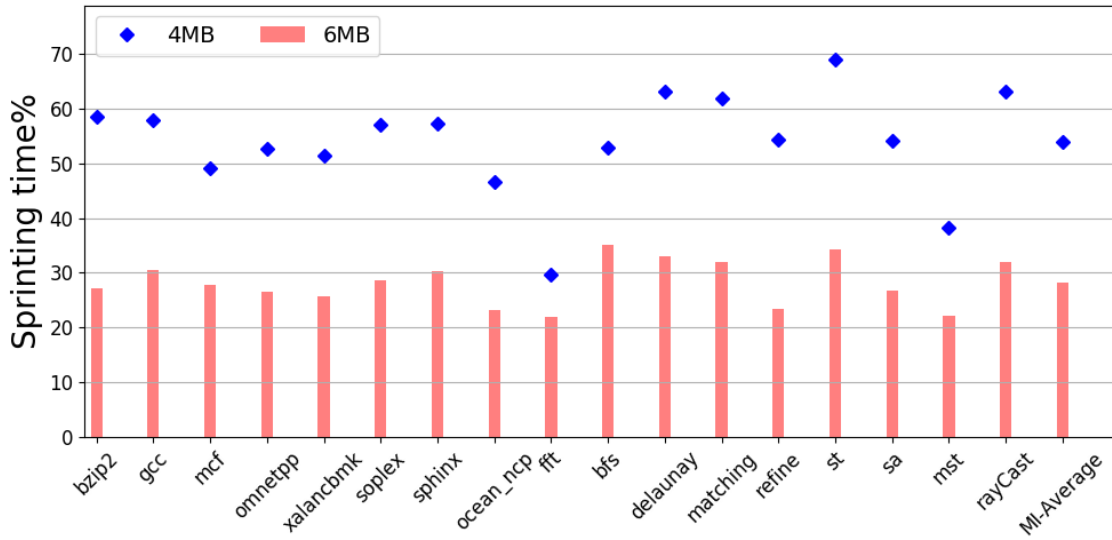


FIGURE 3.14: Impact of sprinting intensity on sprinting time

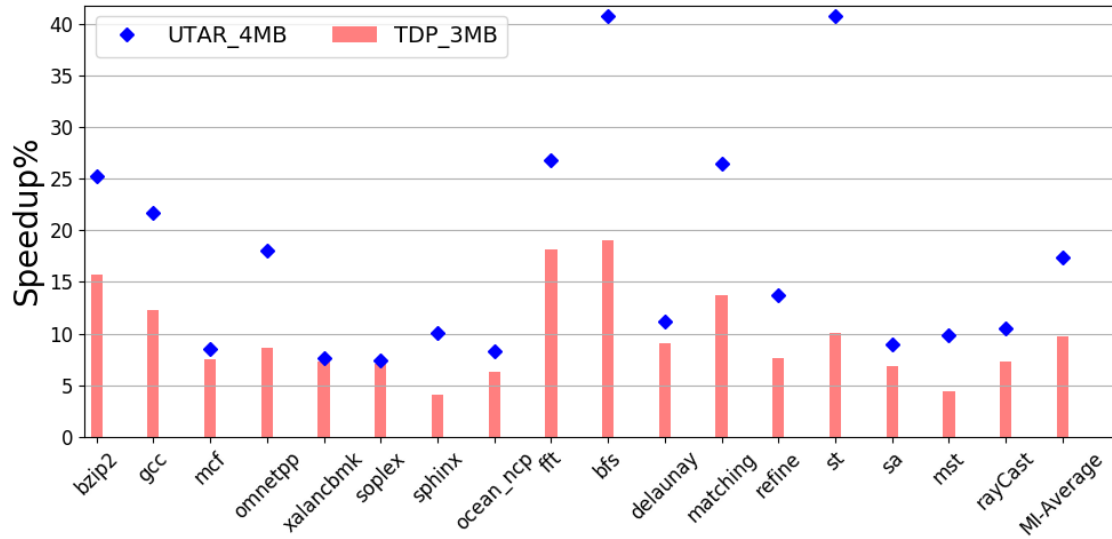


FIGURE 3.15: UTAR performance against perfect TDP

power at 6W, which is 1W below the 7W thermal design point that dictates sustainable power draw. However, we also compare UTAR against an operating point that dissipates 7W, assuming that the processor uses power to operate exactly at the thermal design point. Because our models indicate that the system requires 1W of power per 1MB of cache capacity, sustained operation at 7W permits the sustained use of a 3MB LLC.

Figure 3.15 compares performance from sprinting (dynamic 2MB and 4MB modes) and sustained operation at TDP (static 3MB mode) relative to a baseline 2MB mode. For applications that exhibit high variance in sprint utility, UTAR significantly outperforms sustained operation at TDP. For applications that exhibit low variance, performance depends on the application sensitivity to cache size. Some applications (*e.g.*, `omnetpp`) gain significantly more performance from a cache size much larger than permissible under TDP and prefer an approach that alternates between sprint and nominal modes. Other applications (*e.g.*, `xalancbmk` and `soplex`) are insensitive to cache sizes between 2MB and 4MB such that sustained operation with 3MB matches the performance of sprint-and-cool.

3.5 Related Work

Computational sprinting has been applied to datacenters [83, 96, 28]. At such scale, power constrains not only the processor chip but also the servers and clusters that share a power supply. Uncoordinated sprints risk tripping the circuit breakers. Although batteries can supply power to complete computation during power emergencies, future sprints are forbidden until batteries recharge.

Management policies have been proposed to partition the last-level cache to minimize interference [49, 98], maximize system throughput [12, 67, 25], or improve fairness [48, 60, 92, 95]. Performance-centric partitioning mechanisms often construct miss rate curves (MRC), which characterize the miss rate as a function of the

cache allocation. Utility-based Cache Partitioning (UCP) [67] use simple hardware monitors to estimate the MRC, relying on the stack property of the LRU replacement policy [54]. Prior work has also estimated the MRC in software [10, 74, 75, 88], tracing memory addresses and using analytical models [10, 24]. Unfortunately, tracing memory addresses is expensive and introduce significant slowdowns over native execution. Ubik [42] predicts and exploits the transient behavior of latency-critical workloads to maintain their tail latency. However, transient behavior is not always analyzable for all partitioning schemes (e.g., way-partitioning). Ubik also makes assumptions about future cache microarchitectures (i.e., Vantage).

Most dynamic phase analysis techniques observe that performance is strongly correlated with executed code [79, 20, 80, 21, 76]. These techniques often collect some form of *execution frequency vectors* (EFV) that identify the code executed at some point in time. Prior works have constructed EFVs with instructions [79, 20, 76] or basic blocks [80, 21]. Unfortunately, EFVs are expensive because they record multiple samples of instruction addresses for accuracy. Because of performance counter skid [3], recording the precise instruction address requires tools like Intel PEBS [1] which introduce additional overhead [76].

Prior work have used dynamic phase analysis to predict future application behavior [22, 81, 97, 78, 37, 38]. Observing that phases recur, prior studies use a table-based history predictor to capture past phase patterns [22, 38]. These techniques often tailor phase definitions and boundaries for specific optimization. For example, prior work define phases based on the ratio of memory bus transactions to micro-ops and statically determines phase boundaries to guide DVFS [38]. More sophisticated dynamic resource management requires more careful phase definition.

3.6 Conclusion

We propose UTAR, a software runtime system that manages sprints intelligently based on the application’s sprint utility and the system’s thermal headroom. We also propose cache sprinting, which dynamically allocates last-level cache capacity. With a modest amount of thermal headroom provided by phase change materials, UTAR-guided cache sprinting can improve performance by 17% on average and up to 40% over a non-sprinting system. Moreover, the system performs within 95% of a globally optimized oracular policy.

Coordinated Multi-resource Sprinting

Sprinting is a class of mechanisms that temporarily boost performance at the cost of consuming power more than it is allowed by a system's thermal envelope. Prior work has proposed sprinting on processor cores by activating more cores and/or increasing core frequency. In previous chapter, we proposed a new sprinting mechanism by briefly expanding last-level cache capacity. Because these sprints focus on different microarchitecture components, their effectiveness depends on the degree of benefits different applications would obtain given additional amount of those components. Compute-intensive applications tend to benefit more from boosting core frequency while memory-bound applications are more likely to prefer increasing last-level cache capacity. Even within a single application, it is very likely that different phases would stress different resources. As a result, there is a great opportunity for designing a system that is capable of sprinting on multiple types of resources to exploit the full performance potential.

In this Chapter, we propose the idea of coordinated multi-resource sprinting and demonstrate this idea with frequency sprinting and cache sprinting. Observing that these two types of sprints complement each other (i.e., an application phase is likely

to only benefit from one of these two sprints), we propose UTAR+, an extension of UTAR which we proposed earlier to manage single-resource sprints, to manage both of these sprints at the same time. At the beginning of each epoch, UTAR+ identifies phases, estimates the utility and cost of each sprints for that epoch and decides whether to sprint as well as which resource to sprint based on the estimated utility and current available thermal headroom. We also show that by treating these resources like knobs that can be tuned both ways, sometimes one can be tuned down to enable the other to be tuned up further (i.e., reduce frequency to allow more cache capacity for cache sprinting). UTAR+ is capable of making such tradeoffs using simple heuristics based on previous sprinting results.

We implement UTAR+ in software and evaluate it using a wide range of applications. We prototype the run-time system on an Intel Xeon Broadwell that supports last-level cache allocation via Intel Cache Allocation Technology and frequency scaling. We show that UTAR+-guided multi-resource sprints improve performance by 22% on average and by up to 83%. In many cases, UTAR+(multi-resource sprints) outperforms UTAR(single-resource sprints) significantly by identifying different sprinting preferences and trading off power budget accordingly.

4.1 Case for Multi-resource Sprinting

4.1.1 *Heterogeneous Resource Demands*

Applications today are becoming increasingly diverse and complicated, with many different functionalities usually encapsulated in a single software product. As different functionalities are invoked by users, applications often demands different type as well as the amount of computer resources. For example, a Google map user might zoom in or zoom out maps on his/her phone to look for a specific location, or might just enter the name of the location and let the Google find the quickest path. The zooming part only needs loading maps thus is very likely to be memory bound, while

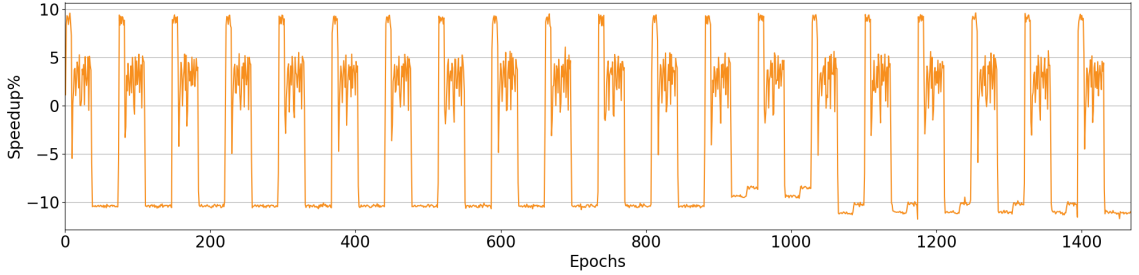


FIGURE 4.1: Performance gain: rayCast running with 2.3GHz + 2MB LLC over 2.1GHz + 4MB LLC (no sprint is: 2.1GHz + 2MB LLC)

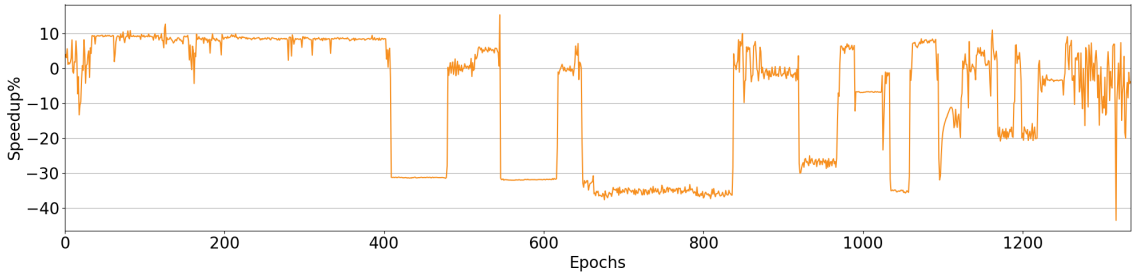


FIGURE 4.2: Performance gain: gcc running with 2.3GHz + 2MB LLC over 2.1GHz + 4MB LLC (no sprint is: 2.1GHz + 2MB LLC)

the navigation part requires lots of computation (e.g., traffic delay) and thus is more compute intensive.

If the system only considers sprints with a single resource (e.g., core frequency or last-level cache capacity), it can only accelerate one of these tasks and forgo the other. With multi-resource sprinting, by identifying the correct resource to sprint on, the system can adapt to applications’ different resource demand and reach the full performance potential.

Figure 4.1 compares the performance gains over time of the application rayCast from PBBS [82] benchmark suite with two alternative sprint mechanism that consume roughly 2W of extra power—scaling frequency versus increasing LLC capacity—on an Intel Xeon Broadwell processor. This figure clearly shows that rayCast repeatedly goes through several different phases that prefers different sprinting resources.

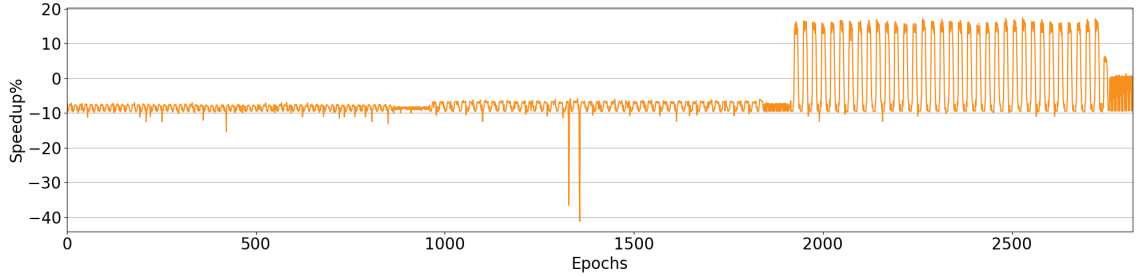


FIGURE 4.3: Performance gain: bzip running with 1.9GHz + 5MB LLC over 2.1GHz + 4MB LLC (no sprint is: 2.1GHz + 2MB LLC)

Another example can be seen in Figure 4.2 with application gcc from SPEC2006 benchmark suite. Gcc differs from rayCast in that most of its parts that benefit from frequency sprinting more than cache sprinting are at the beginning of the program, where thermal headroom is abundant. If the system only considers cache sprinting it would have wasted the opportunity to boost performance for almost one third of the application.

4.1.2 Tradingoff Resources for More Intense and Longer Sprints

Even if an application mostly prefers single-resource sprinting, having multiple resources in the decision making process can still be useful because we can treat resources like knobs that can be tuned both ways. Thus tuning one resource down can free power budget to tune up the other resource further. Figure 4.3 illustrates such an example using the bzip2 application, this time, with only the cache sprinting mechanism but at two different intensity levels: 4MB and 5MB. In order to keep the power consumption roughly the same for fair comparison, the one with larger cache capacity is compensated by turning down the frequency from 2.1GHz to 1.9GHz. This figure shows that in the last one third of the program, bzip achieves better performance if it “trades” frequency for cache capacity. This makes sense because higher clock frequency is less useful if the application is in a phase that frequently misses the last-level cache and waits on memory accesses. Not only can the system

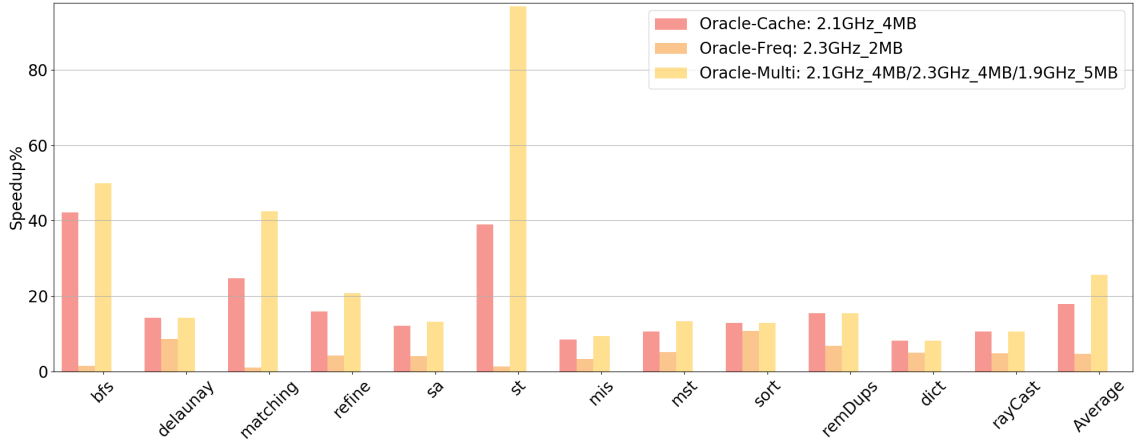


FIGURE 4.4: Performance comparison: single-resource vs multi-resource sprints over 2.1GHz + 2MB LLC

trade one type of resource for the other for a higher-intensity sprint, it can also do so to enable a longer sprinting duration. For instance, suppose the target application is going through a lengthy memory-intensive phase that would need more thermal headroom than what is provided by the deployed phase change material. The system can turn down core frequency when the thermal headroom is about to be exhausted, “stay even” on TDP and still be able to keep sprinting on cache capacity till the end of that phase.

4.1.3 Performance Improvement Opportunity

Figure 4.4 shows the performance gains achieved by three different sprinting mechanisms with oracular policies for all the applications in PBBS benchmark suite. The baseline is a single core running at 2.1GHz with 2MB of last-level cache. “Oracle-cache” sprints by increasing LLC cache capacity to 4MB. “Oracle-freq” sprints by increasing core frequency to 2.3GHz and “Oracle-multi” includes both of these options and can also trade core frequency for cache capacity with a sprinting option at 1.9GHz + 5MB LLC. The oracular policies have perfect knowledge of the utility(speedup) achieved by each sprinting mechanisms for each epoch and work by

prioritizing sprinting on the high-utility epochs. For “Oracle-multi” this means each epoch, if chosen to sprint, will pick the mechanism that gives that epoch the highest utility. Note, all these sprinting configurations consumes the same extra power over the baseline (2W).

4.2 Managing Multi-resource Sprinting

In the previous chapter, we proposed UTAR, a utility and thermal aware software runtime that is capable of dynamically managing sprints with high accuracy and low overhead. UTAR uses several common performance counters to construct “phase signatures” and then use them to discover and predict phases as programs run. UTAR explore the sprinting opportunity for each phase and record utility/energy histories. Together with actively tracked thermal headroom, UTAR is able to assess “whether it is worth to sprint in the next epoch”.

Although UTAR is designed to be applicable to not just one sprinting mechanism (e.g., cache sprinting). It is not designed to manage multiple sprinting mechanisms simultaneously. To manage single-resource sprints, UTAR only needs to make a binary decision at each epoch: sprint or not sprint. With multi-resource sprints, the management framework not only need to decide whether to sprint, but also which type of resource to sprint with. Moreover, if one considers to trade off resources to allow multiple “intensity levels”, then it needs to determine that as well.

Although the decision making process becomes more sophisticated in a multi-resource sprinting setting, the underlying principles to make those decisions stay the same. On the high level, the management framework still needs to access the utility (but for multiple resource types and intensity levels), track the thermal headroom and make decisions with a goal of maximizing performance under thermal constraints over long run. We propose UTAR+, an adapted version of UTAR that is capable of managing multi-resource sprinting, specifically frequency sprinting and cache sprint-

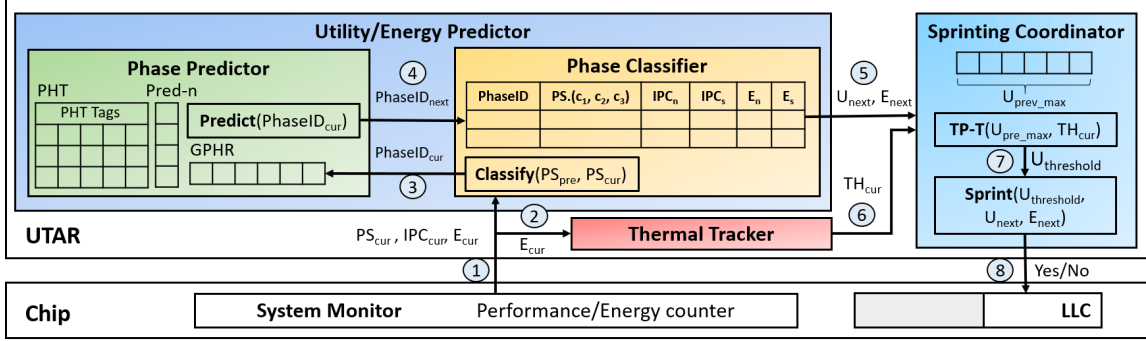


FIGURE 4.5: UTAR overview

ing, at the same time.

4.2.1 UTAR Overview

Figure 4.5 presents an overview of previously proposed UTAR (Section 3.1). UTAR manages single-resource sprinting by collecting data about the previous epoch. When an epoch ends, the phase classifier assesses hardware performance counters, which supply the phase signature, to decide whether the epoch corresponds to a previously observed phase or a new one. UTAR then makes a series of predictions about the next epoch to decide whether to sprint. First, the phase predictor uses the most recently completed epoch’s phases to query the phase history pattern table and predict the next epoch’s phase. Second, the utility and energy predictor uses the next epoch’s phase as well as its historical performance and energy measurements to predict the effects of sprinting. Third, the thermal tracker uses the previous epoch’s energy consumption to determine how much thermal headroom remains in the system. Finally, the sprinting coordinator aggregates predictions to decide whether to sprint in the next epoch.

4.2.2 UTAR+: What’s Unchanged/Changed

Among the different components of UTAR, the phase classifier, the phase predictor and the thermal tracker remains unchanged. These components take inputs from the

performance and energy counters from the system monitor and thus are independent of which sprinting mechanism used. The Thermal-Proportional Threshold (TP-T) policy also does not need to be changed because its principle functionality is to determine the significance of the sprinting utility given a certain available thermal headroom, which still applies in the multi-resource sprinting setting.

To be able to manage multi-resource sprinting, at minimum, UTAR needs to assess and compare the utility of each possible sprinting “configuration”. This means as new phases are being identified by the phase classifier, the sprinting coordinator needs to actively exploring sprinting with each configuration for each phase and record the utility and energy history. Once it has all the history data, the sprinting coordinator then can pick the configuration that has the highest utility to sprint with if the TP-T policy indicates that utility is worthy of sprinting.

4.2.3 UTAR+: Optimizations

The key to an effective and efficient management framework in the multi-resources sprinting setting is to quickly and accurately identify the sprinting mechanism and intensity that offers highest performance/watt. UTAR is already shown to be highly accurate in utility prediction (Figure 3.12). However, it is not efficient to blindly explore every sprinting option for newly discovered phases, especially when the number of options increases. UTAR+ takes several approaches to minimize the number of sprinting exploration.

Filter Using Last-level Cache Misses. The first step is to determine the sprinting mechanism. UTAR+ uses the L3_MPFI (*i.e.*, last-level cache misses per kilo-instruction) performance counter to filter out some easy-to-decide scenarios. L3_MPFI is usually a good indicator of memory intensity of a particular program phase. A low L3_MPFI number indicates a small number of memory accesses. Figure 4.6 shows each program phase’s preference for two different sprinting mechanisms

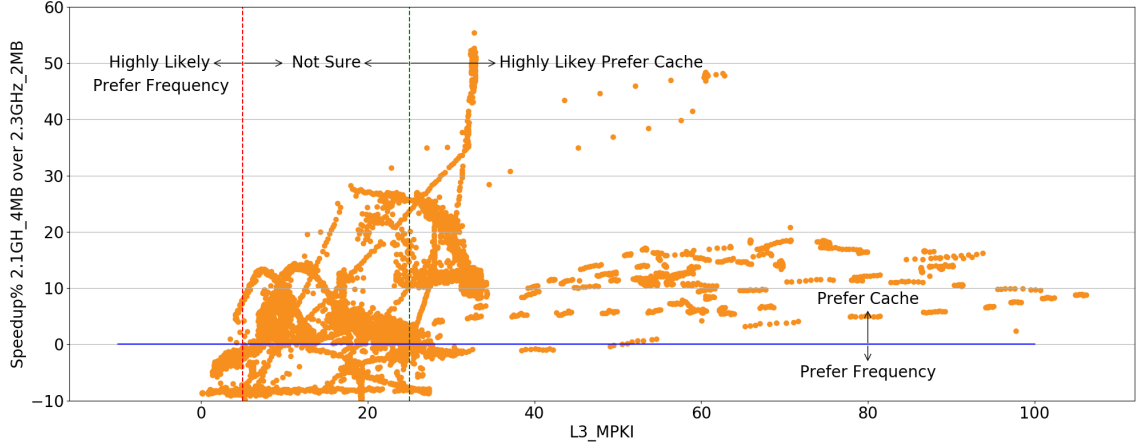


FIGURE 4.6: Sprinting Mechanism Preference over LLC Miss

in relation to the L3_MPki of that phase. Similar to previous setting, the baseline configuration has a 2.1GHz core and 2MB of last-level cache. The system can either sprint by boosting frequency to 2.3GHz or increase LLC capacity to 4MB (both consumes roughly 2W of extra power). Each dot in this figure represent an epoch of 100 million instructions in PBBS benchmark suite and y-axis represents the performance improvement cache sprint has over frequency sprint. If we draw two threshold lines at L3_MPki = 5(red) and L3_MPki = 25(green), we can see that almost all the epochs on the left of the red lines prefer frequency sprinting while almost all the epochs on the right side of the green line prefer frequency sprinting. Using these L3_MPki thresholds, UTAR+ can quickly identify the proper sprinting mechanism for epochs that are either very compute-intensive or very memory-intensive.

Filter Using Cache Sprinting Utilities. For epochs that fall between these two thresholds lines, it is hard to determine their preferable sprinting mechanism just based on the L3_MPki value. However, UTAR further filters out unnecessary sprinting explorations by initiating a cache sprint and assessing the resulting utility. The observation here is that cache sprinting and frequency sprinting are two complementary mechanisms, if UTAR+ sees a high utility result from the cache sprint

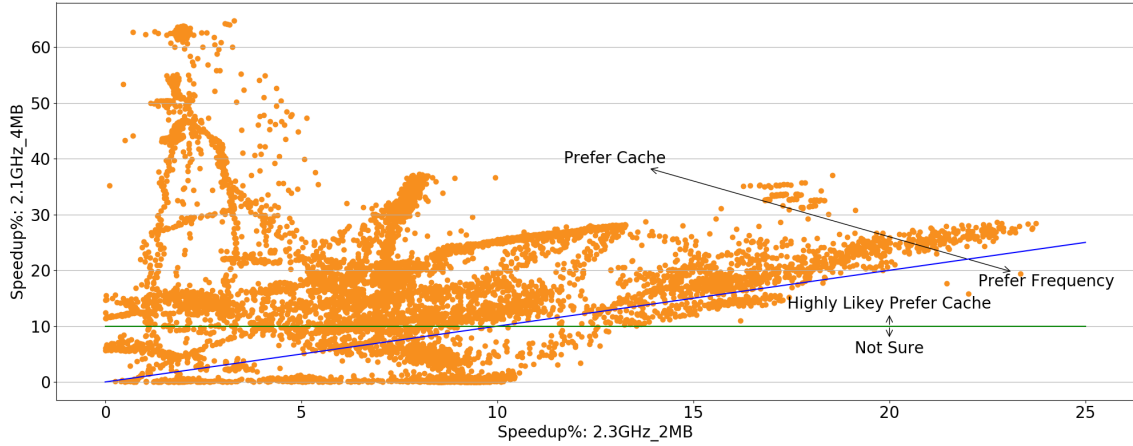


FIGURE 4.7: Sprinting Mechanism Preference over Cache Sprinting Utility

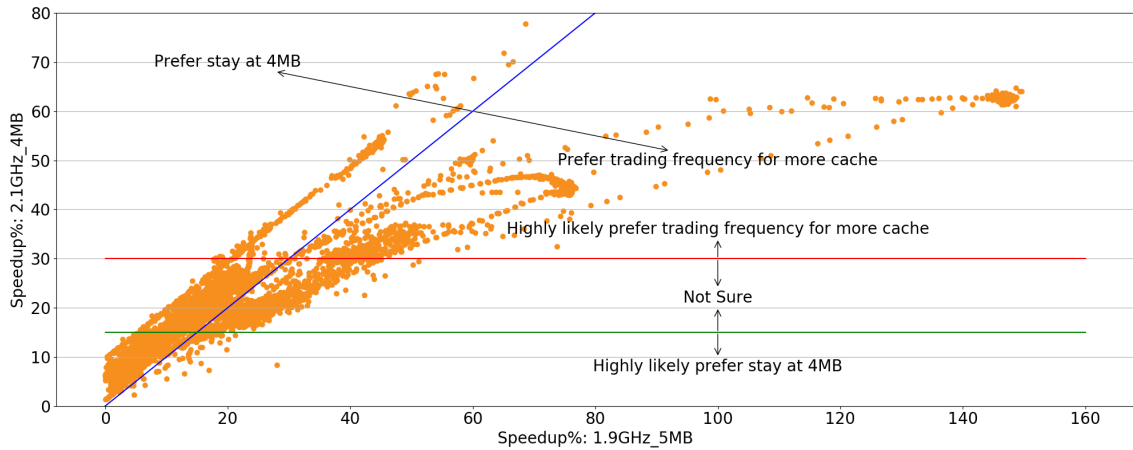


FIGURE 4.8: Cache Sprinting Intensity Preference Cahce Sprinting Utility

for a particular epoch, it is unlikely that a frequency sprint instead will achieve higher utility. Figure 4.7 shows program phases' preference for cache sprinting and frequency sprinting over cache sprinting utility. The blue line is the break-even line so epochs on the upper left side of the blue line prefer cache sprinting and vice versa. If we draw a threshold line at cache sprint utility = 10%, we can see that almost all the epochs above the green line prefer cache sprints. Using this utility threshold, UTAR+ further filters out a large number of epochs that would not need exploration for frequency sprinting.

After determining the sprinting mechanism, a similar approach can be used to filter out unnecessary explorations when deciding the sprinting intensities. Figure 4.8 shows programs phases’ preference for two cache sprinting intensities. Note that in the higher-intensity configuration (5MB), the frequency is reduced to 1.9GHz to keep the extra power consumption the same. The observation here is that the epochs that already show high utility in the low-intensity sprints are more likely to prefer trading frequency for more cache capacity while those exhibit low utility usually prefer stay with low-intensity cache sprints. Again using the two utility threshold lines indicated in the figure, UTAR+ further reduces the number of exploration when determine the sprinting intensity.

Filters’ Thresholds. The thresholds lines in previous figures are drawn from observations on large amount of empirical data (i.e., tens of thousands of epochs) which can be obtained through offline profiling. However, even without offline profiles, UTAR can still gradually learn these thresholds values online. For instance, for the two L3_MPFI thresholds that filters out “high-likelihood” epochs in Figure 4.6, UTAR+ can start with a relatively small threshold value for the red line (e.g., $l3_MPFI = 1$) and a relatively big threshold for the green line. As UTAR+ gradually collects utility data from the coming epochs, the threshold value can be updated by observing the correlation between the epoch’s $l3_MPFI$ and sprinting preferences (e.g., if it sees many epochs that have $l3_MPFI = 2$ and have very low cache sprinting utility, the red line is moved towards the right).

4.3 Experimental Evaluation

We took a similar experimental setup we used for UTAR in the previous Chapter 3.3. We emulate cache and frequency sprints on a 8-core Intel Broadwell Xeon E5-2620 v4 processor. The baseline is set with a 2.1GHz core and 2MB of last-level cache. We consider three configuration for sprints which cost roughly the same amount of

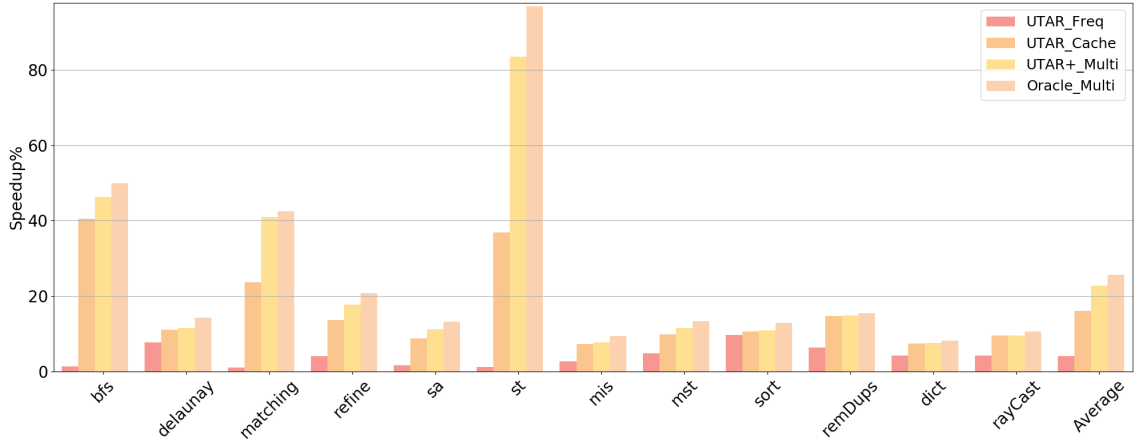


FIGURE 4.9: UTAR+ Performance

extra power: (1) frequency sprint(FS) with 2.3GHz, 2MB (2) low-intensity cache sprint with (LCS) with 2.1GHz, 4MB and (3) high-intensity cache sprint(HCS) with 1.9GHZ, 5MB. We use Intel’s Cache Allocation Technology to dynamically changing last-level cache capacity and use cpupower tool to dynamically change core frequency.

We evaluate UTAR+ and multi-resource sprints against UTAR-guided single-resource sprints and an oracular mechanism. UTAR_Freq only considers frequency sprints while UTAR_Cache only considers cache sprints without trading frequency. UTAR.Multi considers all above three sprints and Oracle.Multi that has perfect global knowledge of utility and thermal information (4.1.3).

UTAR+ Performance. Figure 4.9 shows the performance gains of these four different sprinting mechanisms, over a baseline that operates under nominal power. UTAR+-guided multi-resource sprinting improves performance by 22%, on average, and by up to 83%. It outperforms UTAR-guided frequency/cache sprinting and performs very close to the oracular mechanism.

Sprints Breakdown. To better understand where the performance improvement comes from for multi-resource sprinting, Figure 4.10 breaks down the percentage of epochs in each of the possible mode for all the applications. We see that

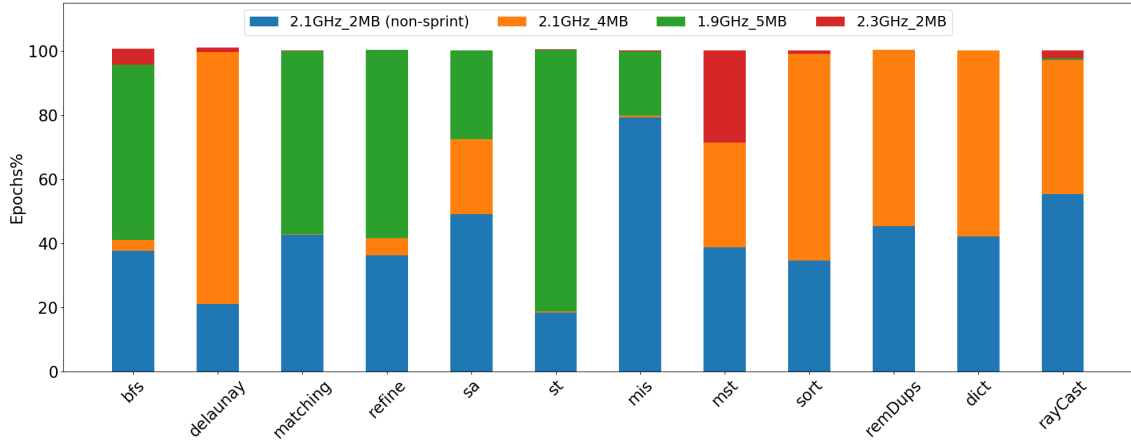


FIGURE 4.10: Epochs breakdown in multi-resource sprints

because this set of applications prefer cache sprinting over frequency sprinting, most of the performance gains of multi-resource sprinting over single-resource sprinting come from trading frequency for more cache capacity, indicated by big size of green bars. Mst is the only application that have a significant number of epochs that prefer frequency sprinting, and UTAR+ is able to capture those epochs to further improve performance. For some applications like remDups and dict, the best option is to sprint on cache without trading frequency, this is probably because they have a smaller working data set that mostly fits into the 4MB cache, and the benefit brought by an additional MB of cache does not outweigh the performance loss caused by reducing frequency. These figure demonstrates UTAR+ is effective in matching application’s resource demands with the most profitable sprinting option.

Prediction Accuracy. Figure 4.11 shows UTAR+ prediction accuracy. Misprediction is counted if the predicted highest sprinting utility and the actual highest sprinting utility of an epoch differs more than 1%. On average, UTAR+ achieves high prediction accuracy, predicting above 94% of the time within 99% of actual utility values. The degree of mispredictions and the resulting performance loss varies across applications. For applications like sa, mis and sort, because their utility differences

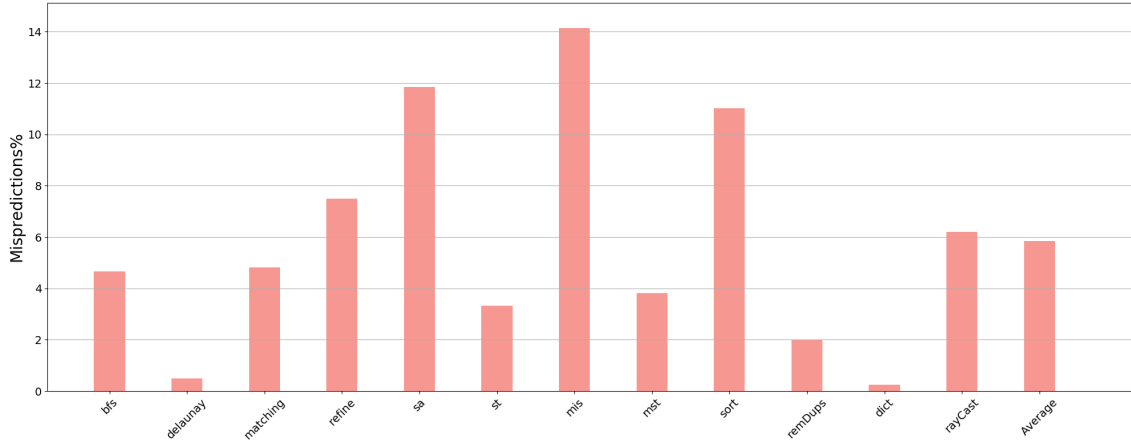


FIGURE 4.11: UTAR+ Prediction Accuracy

between different sprinting options (e.g., LCS and HCS) are relatively small, the chances of mis-predictions are higher. However, for the same reason, mispredictions contribute less to performance loss. On the other hand, applications like `st` have a huge utility difference between LCS and HCS, and thus even a small number of mispredictions can have a bigger impact on the performance loss.

4.4 Conclusion and Future Directions

In this Chapter, we propose the idea of multi-resource sprinting and UTAR+, a software runtime system capable of identifying applications' resource demands and coordinating between different sprinting mechanisms. We show UTAR+-guided multi-resource sprints improve performance over 22% for a variety of applications. We use frequency and cache sprints to demonstrate the effectiveness of UTAR+ and multi-resource sprints, but the general framework applies to other types of microarchitectural resources as well. In the future we plan to look into other microarchitectural components like reorder-buffer, load store queues, etc. These resources are power-hungry but have great impact on system performance and thus are good candidates for multi-resource sprints.

Conclusions

Computer architects used to put most of their efforts on hardware designs. There was a famous quote saying “Anything software can do, hardware can do it better”, where better mainly means better performance. In the era of dark silicon, performance though still very important, is no longer the only design target, other metric like power-efficiency is becoming equally crucial and cracking hardware alone is no longer desirable in achieving all these targets. Instead, in order to improve performance in a power-efficient way, we have to rely more on software, but hardware needs to provide new mechanisms. These can be at instruction set architecture level, and we rely on compiler to do more optimizations, or this can be at system management level, and we rely on operating system to re-allocate resources more efficiently.

This thesis presents three pieces of works that using a software/hardware code-design approach to improve the performance of computer systems under power constraints. The first work looks into approaching out-of-order core performance in an in-order design through the co-design of instruction set architecture, compiler and microarchitecture. The proposed decoupled-load instruction extension together with tailored scheduling algorithms from the compiler and support from hardware, im-

proves performance in a power-efficient way. My second work shift toward designing software runtimes to manage a mechanism (i.e., sprinting) that can be very helpful in post dark-silicon era. The proposed UTAR framework takes advantages of simple and already available hardwares (e.g., performance counters) to analyze applications' behaviors and resource demands and make decisions accordingly, maximizing performance gains over a long period of time. My third work builds upon UTAR with enhancements to support managing multiple type of sprinting resources. The proposed UTAR+ quickly identifies the resource preference for a particular application phase and matches it with the proper sprinting mechanism and intensity, further improving performance for a variety of applications.

Throughout my journey towards this thesis, many lessons have been learned, from identifying interesting research problems, to proposing meaningful solutions, to validating the experimental results. As a computer architect, I often find myself immersed in “hands-on” matters, whether it be debugging a piece of software or setting up hardwares for experiments. At those moments, as I am focusing on very little details, it is easy to lose the big picture and become narrow-minded. Thus I find it very helpful during those moments to take a step back and think about some broad questions of my research. What is the impact of my research ? Does my solution have broader applications ? Can future technology take advantage of my research ? These questions should definitely be asked at the early stage of your research, but they should be revisited throughout the entire process to help you stay focused on your paths and goals. Very often there are more than one way to approach a problem, some are easier and thus might produce results more quickly but less conclusively. Other approaches take more time and efforts but the results can be applicable immediately. Just like computer architecture is all about making tradeoffs, researchers should be aware of the pros and cons of the approaches they are taking and make proper tradeoffs at different stages of their research process.

Finally, communication is one of the, if not the most important skills researchers need to master. You can not make impactful research if you work behind closed doors. Share your ideas and get feedbacks from your professors, colleagues and even students. Don't be afraid of what other people might say or think about your ideas. Positive feedbacks are going to make you more confident and negative feedbacks are going to make you more critical. In the end, you will find them helpful in improving the quality of your research as well as growing as a person.

Bibliography

- [1] Intel 64 and ia-32 architectures software developer's manual.
- [2] Intel running average power limit. [urlhttps://01.org/blogs/2014/running-average-power-limit-rapl](https://01.org/blogs/2014/running-average-power-limit-rapl).
- [3] Intel vtune amplifier 2018 user's guide. [urlhttps://software.intel.com/en-us/vtune-amplifier-help-hardware-event-skid](https://software.intel.com/en-us/vtune-amplifier-help-hardware-event-skid).
- [4] Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.
- [5] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, Dec. 2003.
- [6] and B. Lee, D. Brooks, , and K. Skadron. Cmp design space exploration subject to physical constraints. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, Feb 2006.
- [7] T. Austin and G. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proc. MICRO*. IEEE Computer Society Press, 1995.
- [8] R. Barnes, S. Ryoo, and W.-M. Hwu. "flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense. In *Proc. MICRO*, pages 319–330. IEEE, 2005.
- [9] F. Bellosa. The benefits of event: Driven energy accounting in power-sensitive systems. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, pages 37–42, New York, NY, USA, 2000. ACM.
- [10] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 169–180, New York, NY, USA, 2005. ACM.

- [11] S. Borkar. Major challenge to achieve exascale performance. In *Conf. High-Speed Computing*, 2009.
- [12] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ICPP '15, pages 749–758, Washington, DC, USA, 2015. IEEE Computer Society.
- [13] P. Chang, W. Chen, S. Mahlke, and W. Hwu. Comparing static and dynamic code scheduling for multiple-instruction-issue processors. In *Proc. MICRO*, pages 25–33. ACM, 1991.
- [14] H.-Y. Cheng, J. Zhan, J. Zhao, Y. Xie, J. Sampson, and M. J. Irwin. Core vs. uncore: The heart of darkness. In *Proceedings of the 52Nd Annual Design Automation Conference, DAC '15*, pages 121:1–121:6, New York, NY, USA, 2015. ACM.
- [15] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *Transactions on Architecture and Code Optimization*, 1(4):389–417, 2004.
- [16] X. Dai, A. Zhai, W. Hsu, and P. Yew. A general compiler framework for speculative optimizations using data speculative code motion. In *Proc. CGO*, pages 280–290. IEEE Computer Society, 2005.
- [17] J. Dehnert, B. Grant, J. Banning, R. Johnson, and T. Kistler. The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. CGO*, pages 15–24. IEEE Computer Society, 2003.
- [18] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.
- [19] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.
- [20] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, ISCA '02*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.
- [21] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36*, pages 217–, Washington, DC, USA, 2003. IEEE Computer Society.

- [22] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, PACT '03*, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society.
- [23] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. ISCA*, pages 26–37. ACM, 1997.
- [24] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 55–65, March 2010.
- [25] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multi-cores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, Feb 2018.
- [26] H. Esmaeilzadeh, E. Blem, R. Amart, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. ISCA*, 2011.
- [27] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11*, pages 365–376, New York, NY, USA, 2011. ACM.
- [28] S. Fan, S. M. Zahedi, and B. C. Lee. The computational sprinting game. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16*, pages 561–575, New York, NY, USA, 2016. ACM.
- [29] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Transactions on Computers*, C-30(7):478–490, 1981.
- [30] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.
- [31] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, March 2016.
- [32] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level cache misses in in-order processors. In *Proc. HPCA*, pages 431–442. IEEE, 2009.
- [33] A. Hilton and A. Roth. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *Proc. HPCA*, pages 1–12. IEEE, 2010.

- [34] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA '03*, pages 157–168, New York, NY, USA, 2003. ACM.
- [35] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 250–261, Dec 2001.
- [36] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomputing*, pages 229–248, 1993.
- [37] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-term workload phases: Duration predictions and applications to dvfs. *IEEE Micro*, 25(5):39–51, Sept. 2005.
- [38] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.
- [39] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation: A pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. <http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf>.
- [40] S. Jee and K. Palaniappan. Dynamically scheduling vliw instructions with dependency information. In *Proc. Workshop on Interaction between Compilers and Computer Architectures*, pages 15–23, 2002.
- [41] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proc. HPCA*, pages 206–217. IEEE, 2000.
- [42] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.
- [43] M. G. Katevenis, R. W. Sherburne, Jr., D. A. Patterson, and C. H. Séquin. The RISC II micro-architecture. *Adv. VLSI Comput. Syst.*, 1(2):138–152, 1984.

- [44] A. Klaiber and H. Levy. An architecture for software-controlled data prefetching. In *Proc. ISCA*, pages 43–53, 1991.
- [45] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO*, pages 75–86. IEEE Computer Society, 2004.
- [46] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proc. ISCA*, pages 59–70. ACM, 2002.
- [47] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proc. PLDI*, pages 289–299. ACM, 2003.
- [48] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008.
- [49] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.
- [50] C. Love and H. Jordan. An investigation of static versus dynamic scheduling. In *Proc. ISCA*, pages 192–201. ACM, 1990.
- [51] Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre, and A. Tarkas. Dynamic performance tuning for speculative threads. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 462–473, New York, NY, USA, 2009. ACM.
- [52] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. MICRO*, pages 45–54. IEEE, 1992.
- [53] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicted execution using the hyperblock. In *Proc. MICRO*, pages 45–54. IEEE, 1992.
- [54] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [55] D. McFarlin, C. Tucker, and C. Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proc. ASPLOS*, pages 241–252. ACM, 2013.

- [56] D. McFarlin and C. Zilles. Branch Vanguard: Decomposing branch functionality into prediction and resolution instructions. In *Proc. ISCA*, pages 323–335. ACM, 2015.
- [57] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W.-M. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proc. ISCA*, pages 136–148. ACM, 1999.
- [58] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.
- [59] G. E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
- [60] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. Flexdcp: A qos framework for cmp architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, Apr. 2009.
- [61] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):87–106, 1991.
- [62] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *Transactions on Computer Systems*, 16(1):55–92, 1998.
- [63] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. ISCA*, pages 13–25. ACM, 1997.
- [64] S. Nekkhalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song. A simple latency tolerant processor. In *Proc. ICCD*, pages 384–389, 2008.
- [65] S. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *Transactions on Computers*, 50(6):590–608, 2001.
- [66] D. Patterson and D. Ditzel. The case for the Reduced Instruction Set Computer. *SIGARCH Computer Architecture News*, pages 25–33, 1980.
- [67] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.
- [68] G. Radin. The 801 minicomputer. In *Proc. ASPLOS*, pages 39–47. ACM, 1982.

- [69] A. Raghavan. *Computational sprinting: Exceeding sustainable power in thermal constrained systems*. PhD thesis, University of Pennsylvania, 2013.
- [70] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. Computational sprinting on a hardware/software testbed. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 155–166. ACM, 2013.
- [71] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture, HPCA '12*, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
- [72] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proc. SC*, pages 62–62. ACM, 1995.
- [73] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. *Proc. ASPLOS*, pages 115–126, 1998.
- [74] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10*, pages 53–64, New York, NY, USA, 2010. ACM.
- [75] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase guided profiling for fast cache modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12*, pages 175–185, New York, NY, USA, 2012. ACM.
- [76] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115, Nov 2011.
- [77] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, pages 24–43, 2000.
- [78] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI*, pages 165–176, New York, NY, USA, 2004. ACM.
- [79] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001*

- International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.
- [80] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.
- [81] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 336–349, New York, NY, USA, 2003. ACM.
- [82] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM.
- [83] M. Skach, M. Arora, C.-H. Hsu, Q. Li, D. Tullsen, L. Tang, and J. Mars. Thermal time shifting: Leveraging phase change materials to reduce cooling costs in warehouse-scale computers. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 439–449, New York, NY, USA, 2015. ACM.
- [84] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 2–13, June 2003.
- [85] J. Smith. Decoupled access/execute architectures. In *Proc. ISCA*, pages 112–119. ACM, 1984.
- [86] F. Spadini, B. Fahs, S. Patel, and S. S. Lumetta. Improving quasi-dynamic schedules through region slip. In *Proc. CGO*, pages 149–158. IEEE Computer Society, 2003.
- [87] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proc. ASPLOS*, pages 107–119. ACM, 2004.
- [88] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 121–132, New York, NY, USA, 2009. ACM.
- [89] M. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference*, 2012.

- [90] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136, June 2012.
- [91] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [92] X. Wang and J. F. Martínez. Rebudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16*, pages 19–32, New York, NY, USA, 2016. ACM.
- [93] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture, ISCA '95*, pages 24–36, New York, NY, USA, 1995. ACM.
- [94] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38*, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.
- [95] S. Zahedi and B. Lee. REF: Resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.
- [96] W. Zheng and X. Wang. Data center sprinting: Enabling computational sprinting at the data center level. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 175–184, June 2015.
- [97] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI*, pages 177–188, New York, NY, USA, 2004. ACM.
- [98] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16*, pages 33–47, New York, NY, USA, 2016. ACM.

Biography

Ziqiang Patrick Huang received his Bachelor of Engineering in Information Engineering at East China University of Science and Technology in 2012. In 2014, he received his Master of Science degree from Duke University, in Electrical and Computer Engineering.

He began studying under Prof. Benjamin C. Lee and Prof. Andrew D. Hilton at Duke University System and Architecture Integration Laboratory (SAIL) in January 2014. His research mainly focuses on software and hardware codesign to achieve better performance on systems which are constrained by power consumption. In 2019, he received his Doctor of Philosophy degree from Duke University in Computer Engineering.