System Design for Heterogeneous Architectures

By

Sankaralingam Panneerselvam

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 12^{th} August 2016

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

Michael Swift, Associate Professor, Computer Sciences
David Wood, Professor, Computer Sciences
Remzi Arpaci-Dusseau, Professor, Computer Sciences
Aditya Akella, Associate Professor, Computer Sciences
Christopher Rossbach, Assistant Professor, University of Texas Austin and Senior Researcher, VMWare Research
Jing Li, Assistant Professor, Electrical and Computer Engineering

Dedicated to my loving family and great friends whose everlasting support and encouragement made me realize my dream.

SYSTEM DESIGN FOR HETEROGENEOUS ARCHITECTURES

Sankaralingam Panneerselvam

Under the supervision of
Professor Michael Swift
At the University of Wisconsin–Madison

Heterogeneity in processors designs has been embraced widely in most computing environments from data centers to personal devices and across different business environments from finance to entertainment industries. These designs are provisioned with compute units of different power-performance characteristics including asymmetric cores, programmable accelerators and custom logic. In addition to enhancing the individual application performance, heterogeneity can also improve the power and energy efficiency of the system.

There have been many works in the research community to provide better systems support and to improve the programmability of such architectures. However, most works have often overlooked dynamic heterogeneity where the amount of heterogeneity as seen by the system can vary at runtime. This can be caused by various reasons such as scaling cores through resource aggregation in dynamic processors, performance variability due to sharing, inability to use compute units at full performance due to power constrains and faults in hardware resources. Not acknowledging dynamic heterogeneity can prevent current systems from achieving better performance by not leveraging these architectures efficiently.

In this dissertation, we present three systems targeting the different causes of dynamic heterogeneity and introduce new resource management techniques to properly leverage such architectures.

First, we present *Chameleon*, an operating system extension to support dynamic processors that can scale CPU cores at runtime by pooling re-

sources from neighboring cores. The techniques designed as part of the system enable applications to reconfigure cores quickly based on their characteristics (sequential or parallel) without any added complexity. The system also introduces new solutions to balance the conflicting requirements of multiple applications.

In the second part, we discuss *Rinnegan*, a system spanning operating system and user-mode runtime layers to target the problem of task placement in shared heterogeneous architectures. If high performance accelerators (e.g., GPU) are used by multiple applications, they can benefit by running on alternate compute units (e.g., CPUs). The system combines per-application stand-alone performance information with current system state such as waiting time for accelerators to make better runtime placement decisions for applications.

The final part talks about *Firestorm*, an operating system extension to support power-constrained architectures where compute units are over-provisioned in processors with respect to the power limit. The system introduces the notion of power and thermal awareness in operating systems to help the right set of applications to get their desired performance.

Our dissertation presents various resource management techniques such as resource abstractions to hide hardware complexities, interfaces for better information exchange between system components, mechanisms that enable better use of hardware features and policies to ensure fair use of heterogeneous resources among applications. These techniques prepare the software systems to leverage future architectures and also enable applications to achieve their goals in any heterogeneous environment.

Acknowledgments

Though I enjoy the fruition of earning a PhD, this was possible only through the support of many people. I would like to thank them all for being there for me and helping me to successfully complete my PhD.

I would first like to thank my high school teacher, Dr. Joseph Xavier, without whom I would not have reached the current stature in my life. He believed in me more than I ever did. He was always encouraging, motivating and instilled the habit of aiming higher than what one feels achievable. The wisdom imparted by him has played a major role in all the success that I have enjoyed so far. I am always indebted to him for his teachings and his faith on me.

I couldn't have asked for a better PhD advisor than Michael Swift. He is always open to suggestions and gave me complete freedom to work on any interesting projects. My interests toward reading and analyzing research papers was actually inspired from his breadth of knowledge and his ability to provide a totally different perception on problems during our discussions. During my initial years of research, I always drift towards low level implementation details of the project work. Mike always sets a constant reminder to remember the big picture while doing research. He has had a great influence on my writing and presentation skills that has greatly improved compared to when I started my PhD. He has been a great mentor and have guided me towards becoming a better researcher.

I am grateful to David Wood for his invaluable feedback on my research projects. I have always enjoyed the discussions with him in this

regard. His course on parallel programming is one of the best that I have taken at UW-Madison. I would also like to thank Chris Rossbach for the technical discussions on heterogeneous systems during my internships and in conferences. His comments on Rinnegan during the initial stages helped to shape the system better. I would like to express my gratitude to other members of my thesis committee: Remzi Arpaci-Dusseau, Aditya Akella and Jing Li. Their feedback during the prelim exam helped improve the quality of my research and comments on the dissertation made it significantly better.

I would like to thank the previous members of SONAR group (Matt Renzelmann, Haris Volos, Asim Kadav and Mohit Saxena) for their guidance while I was ramping up as a PhD candidate. I got their help on Linux kernel development, kernel debugging, qualifier exam preparation, and feedback on paper drafts and my presentations. In particular, I am grateful to Haris for letting me to work on the Aerie project.

I am thankful to Mark Hill (Chair) and Michael Swift (Associate Chair) for providing me the opportunity to teach 537 (Introduction to Operating Systems) during Spring 2015. Though I didn't accomplish much in my research during that semester, teaching was a wonderful experience that was totally worth it. Thanks to all the students for participating in discussions and making the classes interesting. I would like to specifically thank Perry Kivolowitz, who handled other section of this course, for being a great mentor and providing me great support throughout the course. Also, thanks to the TA's - Vijay Kumar, Michael O'Neill and Jyotiprakash Mishra - for all their help.

My family has always been supportive of my decisions even if it means quitting my position at Microsoft and pursuing a PhD degree. Appa and Amma always encourage me to do things until complete satisfaction even if it means taking more time. Though I have been away from home for a long time now, their blessings always stay with me. I was fortunate to

have my siblings reside in the mid-west (Akka in Minneapolis and Anna in Milwaukee). Spending time with them makes me feel at home and I always feel refreshed after a visit to their places. I am forever grateful to Appa, Amma, Akka, Mama, Anna and Anni for all their sacrifices, care and support. Special thanks to all the kutties (my nephews) - Aditya, Prithish and Arjun - for making me remember the kid inside me.

PhD would have been a difficult undertaking without the company of great friends. Many thanks to leader (Srinath) and maami (Uthra) for being like a brother/sister to me. Thanks to Sandeep, Ram, Ambi (Venkat), Suri (Surya), Sibin, VC (Vijay), Thanu, Rago, Ragu, Madhav, Theva, Sachin, Sarang, Kuhu, Chembi, Swarna, Andril, Raajay, Vinitha and many others for all your support and all the fun times that we enjoyed together. I have learned so much from every one of you. Thanks again for making me feel at home during my stay in Madison and for all the great memories that will stay with me forever. I am thankful to all my friends in Guindy Vaalibar Saangam and also to Niranjan, Nishanth, Prad and Ramanathan for their advices and motivating me to pursue my interests. I would also like to thank Rohit, Raja, Vinod, Ranga and Sriram for their help and support during my initial years in Madison.

Contents

List of Figures and Tables

1 Introduction

Processor designs have undergone several transformations over the decades. The early designs, based on single core processors, focused on improving the single threaded performance through microarchitectural features [52, 95, 183, 234, 253–255] and also increased frequency [34] due to transistor scaling [64]. Processor models then shifted towards multi-core designs [63, 89, 139] due to limitations posed by the power wall [177, 290], where the focus was turned towards improving system throughput through parallelism. Currently, processor designs have begun to embrace heterogeneity [35, 39, 112, 145, 274] for higher performance as well as better energy efficiency.

Heterogeneous architectures provision compute units of different characteristics in the same processor package or in the same system. The benefits come from different power-performance characteristics offered by these compute units for different set of computations. For example, parallel programs can achieve higher performance on a GPU [187, 232, 236] compared to running the same program on CPUs. Task-specific compute units such as media encoder ASICs can provide 250 times better performance with a power reduction of 500x [103] when compared to encoding on CPUs.

Heterogeneity has proved to be beneficial in terms of performance and in most cases energy efficiency as well. This has made it prevalent in all forms of computing from personal devices such as mobile phones [16, 57] and laptops [39, 97] to enterprise servers [6, 252] and data centers [129, 225].

The benefits are made possible by trading off generality for specialization by designing accelerators for commonly used tasks or important applications. Compute units that do not cater to all forms of computations but target a specific set of tasks such as parallel tasks or an encryption task are generally called accelerators. A few examples of accelerators include GPUs [152], FPGAs [216], crypto accelerator units [120] and digital signal processors [57].

Heterogeneity is not limited to specific forms but can assume several forms from coprocessors that are closely tied to CPU cores such as DySER [96] and SSE SIMD units [157] to independent accelerator devices such as GPU [152]. Current and future systems will be provisioned with different type of compute units including accelerators. Moreover, same task can be run on different compute units however with different efficiency. For example, a parallel program can be run on GPUs or CPUs but it might run better on GPU. Similarly, an encryption task can be run on crypto accelerators or using Intel AES instructions or even using regular instructions available in the ISA. Operating systems and applications will be faced with more compute options to choose from in order to run their programs or tasks.

1.1 Dynamic Heterogeneity

With wide varieties of heterogeneous architectures and also the great benefits it has to offer, the main question is how should such hardware designs be leveraged in software to ensure better efficiency for applications in the system. The rule of thumb to ensure maximum efficiency with a heterogeneous architecture is to schedule programs on the right set of compute units by matching program characteristics with that of the available hardware characteristics. Now consider *the case where the magnitude of heterogeneity as observed by the software or as*

exposed by the hardware can vary at runtime. This behavior is commonly referred as *dynamic heterogeneity* [36, 293] and it can occur for various reasons:

Hardware scaling. *Dynamic processors* [100, 116, 125, 136, 302] support processor-level reconfiguration where performance of CPU cores can be scaled dynamically by fusing multiple neighboring cores.

Hardware sharing. Performance achieved by programs can be highly variable in a *shared environment* [51, 101, 133, 155, 232] where contention for compute units leads to dynamically variable system.

Hardware constraints. Compute capacity in processors are over-provisioned with respect to the power limit (to limit the heat dissipated by the processor). The performance of any compute unit depends on the power available for it to use which can vary due to power usage by other compute units running other applications in the system. Effectively all compute units cannot be used at full performance at all times in such *power-constrained architectures* [59, 74, 85, 173, 226, 274].

Hardware faults. Device failures or transient errors [248, 293] can make certain chip resources unreliable or unusable at runtime and thereby affecting the amount of resources available for applications.

It becomes hard to achieve efficiency for applications in such a dynamic heterogeneous environment. The compute unit that once delivered the highest performance for an application might provide low performance for the same application since the characteristics of the compute unit changed. For example, GPU might be delivering lower performance due to sharing. This requires the system to constantly monitor available hardware resources and their characteristics along with application properties to

perform the right placement decisions.

Furthermore, multi-programming is becoming common in all forms of systems. This resonates with one of the reasons of dynamic heterogeneity caused by hardware sharing. Google's datacenter share the same machines for both latency critical applications and batch applications [155]. With the prevalence of technologies like virtualization [24, 40, 262], multiple virtual machines share the same physical hardware. The bigger challenge is that different applications sharing the system can have varying requirements [51, 133, 155]. It is a daunting task to address the varying needs of multiple applications - throughput for servers, minimum resource usage for cloud applications - in such a dynamic environment. For example, sequential applications prefer to run at the highest frequency possible whereas a parallel application in the same system prefers a high number of cores rather than high frequency. The compute units need to be reconfigured differently (for example, resource pooling or turn on/off resources) based on the applications being scheduled to satisfy their requirements.

Not only does it become hard to achieve efficient execution but also difficult to achieve a predictable or guaranteed performance for applications in a dynamic heterogeneous environment. Operating systems are designed for homogeneous architectures that do not exhibit most of these challenges. The abstractions in current systems are not enough to capture the hardware intricacies or physical constraints, and application frameworks are not adaptable enough to sustain such an environment.

This dissertation is centered around helping programs achieve their performance goals irrespective of the heterogeneous environment in which they run, without much added complexity in application development. Specifically, this dissertation targets the following questions in the context of a dynamic heterogeneous environment:

- What are the right software abstractions to hide the complexities of the heterogeneous hardware from applications as well as to major

parts of the operating system?

- What are the right interfaces needed for information (application requirements and current resource state) exchange between layers of the software execution stack?

- What are the right mechanism and policies surrounding resource management to ensure that the goals of applications are met?

This dissertation focuses on providing the necessary software support including support from operating systems and application runtimes to help applications achieve their desired goals. We focus more on the aspect of resource management with problems related to resource allocation and resource scheduling. This dissertation spans three parts where each part targets the challenges posed by a cause of dynamic heterogeneity and discusses the system built to tackle them.

The first part focuses on dynamic processors that can scale CPU cores at runtime by pooling resources from multiple cores. The new abstractions and mechanisms from the system, *Chameleon* [208], hide the hardware complexities and enable parallel and sequential applications to benefits from dynamic processors easily.

The second part targets the problem of task placement in a shared accelerator-rich environment where same task can be run on different compute units (e.g., CPU or GPU). The system, *Rinnegan* [210], combines per-application stand-alone performance information with current system state to make better runtime placement decisions for applications.

The third and the final part looks into power-constrained architectures, where processors are over-provisioned with respect to the power limit. *Firestorm* [207], introduces the notion of power and thermal awareness in operating systems to help the right set of applications to get their desired performance.

The following sections in this chapter introduces the three major pieces of this dissertation, briefly talks about the challenges faced in current system and gives an overview of the system built to address them.

1.2 Hardware Scaling

Multi-core processors require parallel code to achieve high performance. However, parallel programming is hard, and legacy code may never be rewritten to take advantage of extra processing cores. Asymmetric chip multi-processors (ACMP) provision a chip with a few powerful processors for sequential code and simpler processors for parallel code [98, 145]. These processors also improve power efficiency for code that sees little benefit from powerful processors. Commercial examples of the ACMP architecture include combined CPU/GPU chips such as AMD's Fusion processor [39], and the TI OMAP 5 [275]. However, high performance on sequential code sacrifices parallel performance: the chip area and power dedicated to powerful cores could provide many more simple cores for parallel execution.

Hardware techniques that combine multiple cores or hardware threads into a more powerful execution engine have the potential to provide high performance on both parallel and sequential code. On parallel code, the processor can be configured into a large set of less-powerful cores, providing performance through parallelism. On sequential code, such a processor can combine several of the cores to improve the performance of a single thread. Such a *dynamic processor* can provide high performance over a wide range of workloads by switching between parallel and sequential modes [112]. For example, Core Fusion [125] pools execution resources such as caches and functional units to improve performance. Speculative multi-threading [105, 143, 163] executes different portions of a single thread in parallel on different cores, and Intel's Turbo boost

increases the speed of one core when others are disabled [48]. Such processors may be able to reconfigure between sequential and parallel modes in microseconds.

Current operating systems are ill equipped for processors that can be reconfigured at runtime. They must know at all times which processors are available for cross-processor communication and global operations, which occur tens to hundreds of times per second. Furthermore, they require expensive global operations to reconfigure when changes occur. For example, the hotplug mechanism in Linux can take more than 200ms to add a new processor, far longer than the projected reconfiguration time for dynamic processors. In addition, processors may have mutually exclusive configurations, such as borrowing resources from one core to improve performance of another, that existing schedulers do not support.

In this dissertation, we present *Chameleon*, an operating system extension to Linux that supports dynamic processors with three new capabilities. First, *processor proxies* enable rapid reconfiguration by removing global operations. Instead, another processor takes the place of an offline processor in any communication or global operation. Second, *execution objects* abstract physical cores and hardware threads into logical objects against which threads are scheduled, so the scheduler need not be aware of physical hardware details. Third, Chameleon's *cluster scheduler* decides when and what to reconfigure and provides a *taxation* mechanism that allows a program or administrator to balance the benefit of faster sequential execution against reduced performance for other threads.

Through emulation of dynamic processors on conventional hardware, we show that: (i) processor proxies reduce the latency of reconfiguration from 150ms to 2.5µs, (ii) Chameleon can leverage idle cores to achieve maximum performance for either parallel or sequential tasks via reconfiguration, (iii) fast reconfiguration allows productive use of a configuration for even a single scheduling quantum, and (iv) under contention, Chameleon's

taxation allows flexible control over whether parallel or sequential code is favored. This can either allow high-priority sequential code to preempt other processors or prevent important parallel programs from having processors borrowed for sequential programs.

1.3 Hardware Sharing

With more mature programming models [42, 196, 197], more programs will be able to easily leverage specialized processing units and compete for access to them. Contention for accelerators [101, 232] alters the performance benefits for programs. Virtualization can also lead to contention, as processing units may by shared not just by multiple programs but also by multiple guest operating systems [101]. Furthermore, the data copy and dispatching overhead can greatly diminish performance benefits offered by certain accelerators [249].

In such an environment, applications that decide statically where to run code may perform poorly by waiting for a specific processing unit while there are idle resources in the system. For example, a parallel program that offloads work to the GPU may wait if the GPU is used by other programs while there are idle CPUs available. Ideally, applications should select the processing unit offering the best performance, considering both the speedup it offers as well as the overhead of moving data, dispatching a task, waiting time for the unit to be available, and the application's share of the processing unit's time. With modern programming languages that can generate code for multiple targets, such as OpenCL [197], a programmer should be able to write the code once and let the system decide where it should execute.

However, in today's systems, it is difficult for applications to predict the utilization they can achieve on a processing unit: OS kernels generally do not tell applications how much CPU time or how many threads they

can use. GPUs and other accelerator are often treated as external devices, and may lack any support for fairly sharing them between applications. It may be up to the device driver for an accelerator to make scheduling decisions, which may not be coordinated with CPU scheduling. While there have been research systems such as PTask and others [101, 170, 232] that perform scheduling for tasks on a single processing unit, these systems are unable to select between multiple possible units for a task. Conversely, application runtimes for heterogeneous systems can run a task on different processing units [21, 164], but support only static heterogeneity, where it assumes performance of a processing unit does not vary over time.

Ultimately, the programs should be able to efficiently execute on any heterogeneous platform. In this dissertation, we built *Rinnegan*, a system that (1) provides system-level scheduling support for non-CPU processing units such as GPUs and (2) extends heterogeneous runtimes with kernel support to make informed placement decisions. Rinnegan separates resource management, which is performed by the kernel, from task placement decisions, which are performed by the application runtime.

To support high-quality placement decisions, Rinnegan provides information to applications about their expected utilization of a compute unit: how much time they will receive, and with what delay. This allows an application runtime to make an informed decision about whether to use an accelerator; under contention, it may be better to run the task immediately on the CPU rather than waiting for a small share of the accelerator.

However, without control over how processing units are used, Rinnegan cannot accurately predict the utilization an application will receive: another application could monopolize a processing unit. Rinnegan therefore enforces a scheduling discipline on all processing units. For the CPU, the scheduling is already performed by the Linux kernel scheduler. For other processing units such as GPUs, Rinnegan adds an agent, a resource manager specific to a compute unit (GPU in this case), that can selectively

delay or reorder tasks from different application to achieve desired policies, such as priorities, or proportional sharing.

Complementing this kernel support, Rinnegan provides a runtime library that automatically makes task placement decisions for applications. Rinnegan builds a performance model for the system that incorporates the time to transfer data to a processing unit (e.g., data copy to GPU memory) and the overhead of dispatching a task. For each of an application's tasks, Rinnegan builds a simple model predicting its runtime on each type of processing unit. At runtime, Rinnegan combines the utilization information from the OS with this model to predict how an application task will perform on each processing unit and selects the best place for it to execute.

A key advantage of Rinnegan's architecture is that it easily supports applications with different goals: the decision of what is "best" is determined differently for each application, so some may choose highest throughput for batch tasks, while others may choose a processing unit that offers the lowest latency or best energy efficiency. Rinnegan also supports adaptable applications that can vary the behavior based on available resources; for example, by reducing fidelity or accuracy under heavy contention. In such cases, Rinnegan can call back into the application notifying it when performance goals are not met or when there are idle resources, allowing it to adapt its behavior. Through experiments, we show that Rinnegan performs 1.5-2x better than StarPU [21], a heterogeneous programming model unaware of a shared environment.

1.4 Hardware Constraints

Modern processors cannot use all parts of the processor simultaneously without exceeding the power limit due to the phenomena of *dark silicon* [74, 274]. In other words, the compute capacity of current and future processors is and will be over-provisioned with respect to the available

power. However, in many systems, the power limit comes not from the ability to acquire power, but instead from the ability to dissipate power as *heat* once it has been used. Thermal limits are dictated by the physical properties of the processor materials and also comfort of the user. Thus, power is limited to prevent processor chips from over-heating, which can lead to thermal breakdown. As a result, the maximum performance of a system is limited by its *cooling capacity*, which determines its ability to dissipate heat. Cooling capacity varies across the computing landscape, from servers with external chilled air to desktops with large fans to laptops to fan-less mobile devices. Furthermore, cooling capacity can change dynamically with software-controlled fans [265] or physically reconfigurable systems, such as dockable tablets [273].

Most processors have a safeguard mechanism that throttles the processor by reducing the frequency or duty cycle (fraction of cycles where work happens) on reaching a critical temperature. In software, the thermal daemon [277] aims to increase performance by deploying increasing cooling (e.g., increasing fan speed) if possible before resorting to throttling.

The drawback with current hardware and software mechanisms that enforce power and thermal limits are that they only offer system-wide guarantees but do not enable application-level guarantees.

Power distribution: When power is limited, current systems (hardware and software) throttle all applications equally. However, this approach ignores users' scheduling priorities: power should be distributed among applications based on their importance, so that high-priority applications can use more power to run faster, while low-priority applications have their power reduced and bear most of the performance lost.

Thermal interference: An application that makes heavy use of a processor can trigger temperature throttling that reduces CPU frequency or duty cycle. This can affect all cores, and hence all running applications. Furthermore, throttling stays in effect for a while as the processor cools. As a

result, a low priority or malicious application can trigger throttling that reduces the performance of high-priority applications.

Performance Knobs: Though current hardware supports mechanisms for different kinds of performance (sequential and parallel), neither the hardware nor the software support the right set to polices to balance the varied performance requirements of applications. For example, Turbo Boost [59] is entirely controlled by the processor hardware. Since the hardware is oblivious to application semantics, its decision to activate boosting might not be always beneficial.

Just as the OS actively manages resources such as CPU and memory, we argue that operating systems should treat power and thermal capacity as primary resources in the system. In this dissertation, we propose Firestorm, which is an operating system extension supporting power and thermal awareness in the system in the system. Firestorm introduces new abstractions to manage power and thermal capacity; new interfaces are added (a) for applications to gather power and thermal resources, and allow application-specific use of power and thermal capacity for guaranteed performance; and (b) to support policies to balance the varied performance requirements of different applications. Our system takes a holistic view on both power and thermal capacity by allowing applications to co-allocate them for better performance.

Firestorm requires all applications to gather power before executing their tasks on any compute unit (e.g., CPU or GPU). This requirement is enforced through agents (similar to the agents used in Rinnegan) that act as resource managers for each type of compute unit. The agent predicts the power requirement of a task using a compute-unit-specific power model and tries to acquire the required power from a centralized power center. The center employs a proportional share policy to distribute power, which isolates applications from each other. When sufficient power is not available, the agent runs a task at lower power and hence lower performance.

In order to support the thermal requirements of applications, Firestorm introduces new policies - *thermal conserve* and *selective throttling* - that allows applications to reserve thermal capacity (i.e., keep the processor cooler) needed during execution. Thus, the system does not allow other applications to exhaust the thermal capacity by raising the temperature too high. Firestorm incorporates a system-specific thermal model to predict the required thermal capacity based on the work to be performed by an application. Firestorm also extends Chameleon's [208] execution object abstraction to create thermal headroom for sequential applications. This is done by creating execution object over multiple cores where few cores are forced to an idle state and the resultant power savings is used to boost the performance of the active cores.

Through experiments we show that Firestorm is able to assign more power to applications with higher shares and prevent interference from lower priority programs. Firestorm also balances the performance of multiple applications under a power budget compared to the native Linux RAPL mechanism that prefers parallel programs over sequential programs. In a case where thermal interference from background application costs performance, Firestorm allows high-priority tasks to run at full speed, as compared to 19% slower under native Linux, while background tasks run 28% slower to reduce heat production. Finally, Firestorm supports creating thermal headroom for high power-density applications by making neighboring cores act as heat sinks.

1.5 Hardware Faults

Transistors are shrunk with every generation that enables more number of them to be packed in the same die area compared to the previous generation. Though this allows new processor designs to add more logic into the processors, it comes with the downside of increased unreliability [33, 248].

This can result in transient faults [293] or even core failures [244].

There has been lot of research works to handle and overcome the issue of unreliability in processors. Few works focus on improving the reliability of the overall chip by introducing redundancy to test for erroneous behavior either at the circuit granularity [33] or even at the core granularity by running the program on multiple cores [258]. Hardware virtualization techniques [293] have been proposed to isolate the faulty core and thus migrate the virtual core that had been running on the faulty physical core to an active physical core. Operating systems support techniques such as Hotplug [184] to logically remove a core from the OS. Such techniques are used to remove faulty cores from the OS and avoid scheduling any application thread on those cores. The work on Core Surprise Removal (CSR) [244] can isolate the faulty core with minimal impact on the work happening on active cores.

The hardware resources exposed by the processors can vary over time as the faulty components are made unavailable to the software. Though this dissertation does not focus on the reliability aspects of the system, the techniques proposed in this dissertation can help the overall system cope with the varying amount of hardware resources.

1.6 Contributions

We describe some of the major contributions of this dissertation below:

- We design a new processor proxy mechanism to minimize the software overhead of processor reconfiguration by selectively virtualizing certain capabilities of the offline CPU through the proxy and ensure normal progress in the system. The proxy was the first mechanism to aim for reconfiguration at the granularity of a schedule quantum. The principles employed by the proxy mechanism are also used in other works [211, 298] in a similar context.

- We introduce a new execution object abstraction to represent multiple hardware context as a single execution context and cluster scheduling policy for an application thread to be able to pool resources from multiple cores. Though the designs were originally used for dynamic processors to acquire and use multiple cores, the designs are generic enough to be used for other shared resources such as power.

- We design a decentralized system design targeting shared heterogeneous architecture to enable program task placement decisions to be made at runtime. The mechanisms in the system allows combining application characteristics with system state to make the placement decision. At the same time, the system provides stronger guarantees like isolation among applications and performance guarantees for applications. We also show how the new system can be integrated with existing runtimes such as StarPU [21].

- We present how to expose system metrics for an accelerator such as maximum usage time for an application based on scheduling policy, average task runtime and average number of waiting tasks. Also, we show what are the metrics to use for different types of application - short or long runnning - to make an optimal task placement decision. We also present a simple performance model to predict application performance on different compute units and show why a simple model is sufficient in the presence of accelerators.

- We design new abstractions to represent power and thermal headroom as resources in the system. This enables applications to allocate or reserve them as other resources in the system which effectively translates into guaranteed performance for applications.

- We present power and thermal models that are fast and simple enough to be deployed practically. We also introduce a set of inter-

faces that need to be implemented by any external power or thermal model to be able to work with our system. This allows easier integration with more sophisticated models.

1.7 Overview

The rest of this dissertation is organized into the following sections:

- **Background.** In Chapter 2, we provide a brief background on heterogeneity by discussing the factors that made heterogeneity common in processor designs, benefits of heterogeneity compared to homogeneous architectures, classification of heterogeneous designs and the trade-offs involved in each design, and the support available in current systems to leverage these architectures. We also define the terminologies that we use throughout this dissertation.

- **Systems for Dynamic Heterogeneous Architectures.** The core of this dissertation focuses on the systems built to support architectures that exhibit the properties of dynamic heterogeneity. First, we present the system Chameleon in Chapter 3 that provides system software support to leverage resource aggregation techniques in hardware. Second, we discuss the system Rinnegan in Chapter 4 that presents a software architecture to make better scheduling decisions in heterogeneous architectures. Finally, we we talk about the system Firestorm in Chapter 5 that proposes new resource abstractions and system interfaces to introduce power and thermal awareness in systems to handle power-constrained architectures. We present the evaluation of all three systems in their respective chapters.

- **Related Work.** In Chapter 6, we compare against prior works including operating systems and runtimes built to support heterogeneity. We also discuss works and techniques that inspired the systems'

(Chameleon, Rinnegan or Firestorm) overall architecture or individual components in this same chapter.

- **Lessons Learned, Future Work and Conclusion.** In Chapter 7, we conclude by summarizing our work on how they offer support to handle heterogeneity in future architectures. Finally, we discuss some of the future works and also present the lessons learned during the work on this dissertation.

2 Background

Processor designs had been predominantly homogeneous until a decade back. Even when the CPU designs moved from single core to multi-core, multiple compute units of the same capability were provisioned in the processor package. Though embedded systems like digital cameras or calculators used specialized processors [121, 304] as the primary compute unit, they were not provisioned with variety of compute units. Graphics units were available although they were used only for rendering purposes [247] but not for running general purpose programs unlike now. Heterogeneous architectures still existed but homogeneous designs were common. However in the past decade, many forms of compute units along with CPUs such as GPUs, FPGAs and other accelerators became more common and even general purpose processors follow heterogeneous design principles. This raise several interesting questions:

- What are the benefits of heterogeneous designs over homogeneous architectures?
- Though heterogeneity has been in existence, why did it become more common now? What are the main factors that led to the growth of heterogeneity?
- What are the different kinds of heterogeneity available and what are the dominant form of architectures used in different environments?

- What is the support available in current operating systems or programming models to leverage heterogeneity?

This chapter provides information to help answer the above questions. To begin with, the chapter lists the terminologies used throughout the dissertation.

2.1 Terminology

Applications. Every individual program is referred to as an application that can consist of one or multiple processes, such as a web server. We use the terms applications and programs interchangeably in this dissertation.

Compute unit. A hardware processing element that can perform computations on behalf of the user is referred as a compute unit. It encompasses all forms of processing unit including CPUs and accelerators like GPU, FPGA or ASICs. We use the term processing unit and compute units interchangeably in this dissertation.

Task. A *task* refers to a coarse-grained functional block such as a parallel kernel or a function, such as encrypting a block of data, that executes independently and to completion.

Scheduling. We use the term *scheduling* to mean selecting the next task to execute on a processing unit, and for deciding how long that task may run. The scheduling can be performed by operating system components [46], applications or runtimes [10, 73] or the device hardware itself [146].

Placement. Task placement refers to the process of choosing among different compute units on where to run the task. The scheduling is done after the task placement. The assumption here is that different implementations of the task for different compute units are available.

Accelerators. Compute units that can run certain set of tasks or computations really well. The specialization allows them to offer better perfor-

mance than regular CPU cores. Also, these compute units do not operate independently but tasks are offloaded to them from the OS or applications running on the processor cores.

Dark Silicon. Compute units in current and future processors are over-provisioned with respect to the power limit. Even if the required power can be made available, current cooling techniques cannot dissipate that much heat. So, a portion of the processor chip (or a subset of compute units) remains unused that is referred as Dark Silicon.

Power Limit. The maximum power usable by all resources on a processor chip combined should not exceed the limit. This can be enforced for different reasons such as to limit the heat dissipation or based on the power supply. The power limit value is set by processor vendors in case of the former reason or by the system administrator for the latter reason.

2.2 Rise in Heterogeneity

Heterogeneity in processor design might be a common trend now but it is not a recent innovation. Earlier systems have been provisioned with variety of compute units. There has been exploration of asymmetric processor designs in supercomputing [169] decades ago. FPGAs had been used in high speed router designs [279]. A variety of specialized units (individual hardware unit to improve capability of the main processor) had been used in earlier processors: floating point units [205] have been the most widely used co-processor which is now integrated into the primary processor; memory management unit co-processors [26] which again is integrated into main processor; co-processors for specific applications [55, 56].

However, the primary focus of processor designs had always been to satisfy general purpose workloads [53] rather than a selected few. As a result, improvements achieved through microarchitectural innovations [281] and technology scaling [64] were tuned towards general purpose CPU

(primary processor) rather than incorporating heterogeneity. Most chip area was thus devoted to provisioning hardware resources to improve the majority of workloads [53] rather than a specific application unless the hardware is designed for that application. Also, to some extent the software cost (programming heterogeneous processor or program compatibility with other processor designs) in terms of programmability also posed as a challenge for heterogeneous designs [230]. However, a combination of the following reasons resulted in wide adoption of heterogeneity despite these challenges.

Power Limits. Moore's law [180] paved the way for doubling transistors within the same chip area with every generation while also scaling voltage down. The latter was dictated by Dennard's scaling [64], where the reduction in transistor gate size resulted in less voltage requited to switch transistors. As a result, double the amount of transistors can effectively be used at the same power as the earlier generation. However, given the breakdown of Dennard's scaling [166], voltage and hence the power draw of transistors is no longer dropping proportionally to size. As a result, modern processors cannot use all parts of the processor simultaneously without exceeding the power limit. This manifests as an increasing proportion of *dark silicon* [74, 274].

Power limitations can drive processors towards heterogeneity. First, existing power limits provide a surfeit of transistors that cannot be fully powered [47], creating an opportunity to add specialized features when previously they may have been too specialized. For example, adding support for AES encryption instructions may have been too expensive when transistors were better targeted at improving general purpose workloads. Now however, processors have enough transistors to devote them to minor workloads if speedups are big enough [54].

Second, power limits are driving computation away from general purpose processors. For example, a custom h.264 encoder can be 500 times

more efficient than a CPU [103]. General-purpose processors are not efficient enough to sustain past performance increases simply by adding more cores [74]. Thus, accelerators or specialized compute units that are more efficient for a subset of workloads provide a promising path to improved performance within an energy budget.

High Performance. The free performance boost that comes with processor scaling will be hard to achieved due to physical constraints [74]. However, the demand for compute has been ever increasing that is hard to satisfy by general purpose processors. The focus on exascale computing [20, 228, 259] to address challenges such as prediction of climate changes or understanding human brain demand huge processing capability. Companies have turned towards heterogeneous architecture designs [107, 242] to enable such compute capacity within power-delivery capabilities of a data center. Server workloads such as machine learning process huge amount of data during its training phase. This requires computations to be fast and also efficient in terms of cost (fewer machines). Heterogeneous designs can satisfy such demands. The new area of in-memory computing has exposed the performance bottleneck with CPUs [69]. Yet to release game consoles [287] already demand huge computations to support high frame rate and high resolution media services.

Rather than relying on general purpose processors, the trend has been towards embracing special purpose hardware accelerators for achieving higher performance. Designing hardware for common tasks across applications [80] or implementing widely used application on hardware [130, 225] can result in higher performance boost for applications.

Around 25% of the top 500 supercomputers already make use of programmable accelerators such as GPUs or Intel MICs. Server designs are provisioning accelerators for various purposes such as processing data at line rate [82], performing analytics over high volume of data [252] and support for in-memory query operations [6]. Major vendors have begun

to shift towards specialized compute units for their big data systems: Microsoft is focusing on FPGAs to accelerate their Bing search engine [225]; Google has designed its own hardware for their deep learning training workloads [130]; Baidu has been using GPUs for their training phases [109].

Energy Efficiency. Data centers power costs are already in the order tens of millions of dollars [278]. This cost is going to further increase with the amount of data flowing [2, 282] into data centers since it demands more processing capacity [297]. On the other hand in the consumer world, smaller form factor devices [41, 118, 188] demand more computations with a limited amount of power supply. In addition to performance, energy efficiency has become a major necessity in many computing environments. The focus here is to improve the performance per watt in order to minimize the cost (monetary cost or battery lifetime).

Though various improvements like power gating and deep sleep states have been implemented in general purpose processors, their nature of generality limits the efficiency that can be obtained. Specialized hardware gives up on generality to run only few tasks efficiently. Employing custom hardware design not only results in higher performance but can also improve the energy efficiency of overall data center [204]. Similarly, current mobile systems are provisioned with several accelerators [245] for commonly used tasks such as media encoding for the same purpose of improving energy efficiency.

2.3 Benefits of Heterogeneity

Heterogeneity has become prevalent in most computing environments from hand-held devices to cloud computing. Consumer devices including laptops, mobile devices, wearables and gaming consoles are packed with different type of compute units including accelerators. Server appliances such as IBM Netezza [252] and Oracle M7 SPARC [6] servers offload part

of the database query operations onto accelerators such as FPGAs and fixed-function accelerators. As mentioned earlier, supercomputing environments make use of GPUs or Intel MICs based accelerators. Major vendors such as Microsoft and Google are using custom design accelerators in their data centers for search engine or machine learning workloads. Finally, GPU based accelerators are also being rented by cloud vendors such as Amazon [8]. Such wide-spread acceptance of heterogeneity would not have been possible unless it had to offer great benefits. The benefits are primarily achieved by trading-off generality for specialization.

Performance. Custom hardware can provide several orders of magnitude performance higher than its software counterpart running on CPU cores. Performance here can represent both lower latency or higher throughput. Accelerators designed for compression [142] and media encoding [103] can perform up to 200 times better than CPU cores. It should be noted that this does not represent the speedup of an entire application but speedup achieved for a portion of an application.

In case of high frequency trading, accelerators are designed to carry out major portion of the application logic [216] for low latency reasons. Similarly, Microsoft has improved its search engine throughput performance up to 2x by porting the search logic rules onto FPGA. The same result can be interpreted as requiring half the number of machines to provide same performance. The use of accelerator can thus reduce the operating cost of the data centers as well. The custom designed accelerator (Tensor Processing Unit) from Google can achieve an order of magnitude more performance per watt [129] than regular CPUs for machine learning workload.

Accelerators are provisioned in processors to keep up with the performance demanded by high speed external devices such as NIC cards. Wirespeed processors provision accelerators to process incoming data at line rate [82] which would otherwise be processed by the slower software

stack. Also, programmable accelerators are provisioned closer to the external devices [134, 243] that allows part of application logic to be offloaded to the accelerator and thus improve application performance.

The secondary benefit of a heterogeneous architecture is that offloading work to the accelerator frees up the processor core to run other applications and thus improve the overall system throughput.

Energy Efficiency. Heterogeneous designs can offer better energy efficiency by completing tasks at lower energy. The efficiency is made possible by specialization where additional overhead to support generality is removed. Accelerators such as media encoder require 500 times lesser energy [103] compared to CPU cores to accomplish the same task. Many such accelerators including as digital signal processors and image signal processors are provisioned in mobile devices for the same purpose targeting widely used applications such as media services and photo editing applications.

The big.LITTLE processors [16] can achieve up to 4x energy efficiency which equates to longer battery life. The energy savings can be achieved by running applications with low performance requirements on LITTLE CPU cluster. Thus energy reduction is achieved without any performance degradation. Similar techniques [220] has been used in data center environments where energy is saved by processing incoming traffic on power efficient CPUs during times of low traffic. However, powerful cores are used when the traffic is higher. Thus, energy efficiency is achieved without any loss in performance.

Finally, certain compute units such as GPU accelerator consume more power than the regular processor cores. However, they are still energy efficient than CPUs for certain workloads since the performance per watt provided by the accelerator is higher than that offered by CPU cores. This is similar to sprinting [226] where more power is consumed for a short period of time to finish the task faster.

2.4 Forms of Heterogeneity

Several examples of heterogeneous architectures were mentioned in the previous sections and these heterogeneous designs can assume various different forms based on how different compute units are interfaced with each other or the type of workloads they support. For example, CPU cores employing different microarchitectural techniques in the same package as in ARM's big.LITTLE architecture [16]; programmable accelerators such as Intel MIC [260], GPUs [152], FPGAs [216] and Micron's Automata processor [67]; fixed-function accelerators such as crypto accelerator [120], media accelerator [87, 102]; co-processors such as SIMD unit [157].

Though all these hardware units are nothing but a compute unit, they might not be suitable for all kinds of task. For example, crypto accelerator cannot do anything except encryption/decryption algorithms; GPUs are good for parallel programs but less suitable for control flow intensive programs; task with small runtime in the order of few microseconds might not receive any speedup with off-chip accelerators due to the high cost of data copy over the interconnect. Current and future systems can have any combinations of these compute units provisioned in the same system. It is necessary to understand the trade-offs involved with different types of compute units to be able to use them effectively. To help with this, we have classified them based on communication interfaces between compute units and further extended the classification based on their memory systems and task granularity support. The overview of the classification is shown in Table 2.1.

2.4.1 Communication Interfaces

Heterogeneous architectures are provisioned with compute units of different characteristics and each compute unit can be of various type such as asymmetric cores, accelerators (programmable, fixed-function or recon-

figurable) or co-processors. Inspired from the previous work that built a taxonomy on accelerators [43], we have categorized the heterogeneous architectures into three major classes based on the mode of communication between compute units including accelerators in the system.

Most of these heterogeneous designs are provisioned with multi-core processors along with specialized compute units such as accelerators since the latter cannot operate stand-alone. The classification is based on how closely the specialized compute units are integrated with the processor cores. This integration is necessary to understand the trade-offs involved in offloading or scheduling or migrating computations across different compute units.

Acceleration devices. Shared accelerators are commonly implemented as devices and accessed through memory-mapped or port I/O instructions from a kernel device driver. For example, the Oracle/Sun Niagara cryptography accelerator [120] requires a kernel device driver for access, as do GPUs for accelerating data-parallel computations. The off-chip accelerator devices are connected to the main processors through an interconnect such as PCI [215]. Though Intel MICs [260] are referred to as co-processors by Intel, we classify them as an acceleration device since the currently available Knight's corner is treated as an external device with respect to the OS.

These accelerators can have the greatest power in terms of the compute capacity, as they are freed from the design constraints of executing in the processor pipeline. There are also accelerator devices that are not physically external to the processor but are present on-chip such as GPUs in desktop class processors [97]. These accelerators are still accessed via device drivers but are slowly transitioning towards the co-processor style accelerators [146]. These on-chip compute units might not be as powerful as a stand-alone GPU card since they are limited by the processor chip area.

Access through the kernel and I/O instructions increases the latency of access and limits these accelerators to coarse-grained computations or larger tasks. Furthermore, as acceleration devices execute outside the processor, they may not have access to virtual addressing. Thus, invoking the accelerator may require pinning data in memory and translating virtual addresses in advance of launching the accelerated computation. The off-chip accelerators are provisioned with their own local memory whereas the on-chip accelerators share the same physical memory with the main processor cores.

Co-processors. Several accelerator designs augment the processor pipeline with acceleration logic to offload program logic. A simple example is vector SIMD instructions, which provide data-parallel execution for greater performance and efficiency. Even if these specialized units are provisioned in all general purpose cores, we still consider them as part of heterogeneous design. The reason is that tasks which run using the vector units can also run using general purpose registers although at a lower performance. Also, multiple thread contexts running on the same core can contend for the vector units (mmx registers) during when few threads might be better off running using general purpose registers. More recently, c-cores executes specific application logic with greater efficiency [284], and DySER provides a specialized data path [96].

There are also shared accelerators such as accelerators in IBM wire-speed processors [82] or on-chip GPU in AMD APU [13] that might not be in the processor pipeline but still directly accessible from the CPU cores. New instructions are added as part of the ISA to access them. They are thus accessible directly to application threads as well bypassing the OS completely. The main processors and the accelerators share the same memory bus which acts as a communication channel between the compute units. Co-processor style accelerators might not be as powerful in terms of offered performance as the acceleration devices since their size is limited

by the chip area which is also shared among other resources such as CPUs and caches.

These accelerators provide low-latency access directly from registers or virtual memory. However, co-processor designs may still contend for power if not all co-processors or cores can be active simultaneously. In addition, heterogeneous system with a variety of accelerators attached to different cores may also experience contention. Finally, programmable accelerators, such as DySER, require a configuration step that may limit its ability to accelerate short code fragments.

Independent cores. Finally, asymmetric or heterogeneous processors can also be considered accelerator-based systems. Rather than specializing hardware to a specific computation, such a system provides a variety of general-purpose cores with different performance and power characteristics. For example, ARM's big.LITTLE processor design provides different CPU clusters targeting performance and power efficiency individually [16]. On such a system, a program may execute faster or with lower power if it switches to a specific core for phases of its execution. Similar to other designs, these systems can experience contention if many processes desire a limited set of cores.

Current processors belonging to this type exhibit two main properties: over-provisioning and scalability. They are provisioned with compute units of different properties but all of them cannot be used at the same time to stay within the power limits [74]. Few examples include Mediatek's decacore [84] and Samsung Exynos 8890 [85] where all CPU cores cannot be used at the same time. The second property enables the individual core characteristics to be changed at runtime either by increasing the frequency [111], pooling resources from neighboring cores [125] or scale local resources based on program characteristics [90, 291].

Intel Turbo Boost [59] is a result of both these properties. CPU cores can run at higher frequency only when sufficient thermal headroom is

available. The frequency boosting is achieved when other cores are in idle state and dissipate less heat. It should also be noted that these properties are also applicable to more mature accelerators such as GPUs where turbo boost [256] and voltage frequency scaling [168] techniques are supported. The problem of over-provisioning applies to all types of processors due to dark silicon as discussed before.

Properties	**Acceleration Devices**	**Co-Processor**	**Independent Core**
Systems	GPUs connected through PCI interconnect such as NVIDIA GPUs [92] Accelerated Processing Unit (APU) like AMD Kaveri processors [13, 97] Crypto accelerators in Sun Niagara chips [120]	Consevation Cores [284] DySER [96] SIMD units like SSE [80] IBM wire-speed processor [82]	Processors based on big.LITTLE design [16] Over Provisioned Multi-core System [47] Scalable Cores like WiDGET [291], ForwardFlow [90]
Capability	Similar to other external devices accepting instructions from OS and returning back the result. Most of the current devices are not able to run OS code on them	Special functional unit that makes sense in context of some core. It cannot function independently	Regular core that can execute threads, handle interrupts, access system memory

Granularity	Suitable for coarse-grained acceleration	Fine-grained acceleration is possible since the instructions are avaiable in ISA to access the special units	Thread is migrated to the special core for performance or to conserve energy
Usage Latency	Overhead of transferring data to device memory	Latency of reconfiguring the hardware logic to fit the computational loop. But this can be avoided if reconfiguration is done during compile time	Time taken for migrating thread onto the newer core and additional latency caused by cache misses due to cold cache effect
Accessibility	Device driver is used to communicate with the device and the required data is transferred to the device's memory through techniques like DMA or zero copy	Special functional units are integrated with the core's pipeline and can be accessed through instructions in ISA. So, no special support is needed from OS	Thread needs to be migrated to the special core and all states are readily available due to cache coherency

Resource Contention	Multiple applications might want to make use of the devices at any instant	Per-core resource will not incur contention whereas sharing of resource like DySER block, c-cores by multiple cores can incur contention	Multiple threads might want to access the powerful core

Table 2.1: Classification of compute units in heterogeneous architectures.

2.4.2 Task Offload

As mentioned earlier, maximum efficiency can be achieved in heterogeneous architectures by running programs on the right compute unit by matching hardware and software characteristics. In addition, it is also important to understand the lifecycle of a task to learn about the additional overheads involved during task offloads on to compute units including accelerators. These overheads can prevent the best compute unit suitable for a task to offer higher performance. The life cycle of a task begins with launching a task (queuing the task for execution on a compute unit), copy input data (if required), executing the task and copy output data (if required). Each of these steps except the task execution causes additional overhead based on how the compute units are integrated. We continue the discussion on top of the previous classification.

Memory Systems. All computations on regular processor cores access their data from the main memory. However, this might not be the case for certain compute units such as accelerators. They provision their own local memory for several possible reasons: different memory type to support the compute capabilities (provisioning GDDR to support the bandwidth

requirement of GPUs), different memory consistency models or to make a stand-alone unit compatible with any general purpose processor design. In order to run a task on a compute unit, the required data should be made available. The question is what is the overhead involved in getting the data in/out of the compute unit and how does it affect the overall task execution.

Compute units in heterogeneous processors can be classified as tightly or loosely coupled based on their integration with the main processor in terms of memory. The former design shares the same memory between all compute units and provides a notion of global memory across the system. The latter design provisions local memory for every compute unit with no coherency across local memories.

Loosely coupled compute units (most off-chip accelerators) require the input/output data associated with the task to be copied between different memory locations (main memory and accelerator memory) by the application. The overall data copy process is costly since it also involves pinning data pages onto the memory in addition to the actual copy over the interconnect. As a result, short running tasks might not benefit much in such designs. Accelerators such as on-chip GPUs that share the physical memory with the main processor cores are still classified as loosely coupled. The main memory is statically partitioned between the accelerator and the processor cores making each partition their own local memory.

Asymmetric processor cores [16, 84, 85] within a chip share the same memory and the data access is coherent across all compute units thus making it a tightly coupled design. As a result, a task or thread can be migrated to any compute units without having to explicitly copying the data associated with the computation. Data access across compute units is taken care by the coherent memory supported by the processor. However, cache misses need to be taken into account while migrating threads or offloading that can result in lowered performance. This also obviates the

need for the application (or the developer) to manage data across different compute units.

The co-processor style compute units also fall under the tightly coupled design. Specialized compute units such as the SIMD unit [157] that are part of the processor pipeline executes in the context of a general purpose core. There are also shared accelerators that are not part of the pipeline [82] but coherent with the CPU cores. The coherency is achieved by using a coherent interconnect across the CPU cores and the accelerators which also enables accelerators to access the application's virtual address space.

Finally, the accelerator device class fall under the loosely coupled category but there seems to be a transition towards the tightly coupled model. External accelerators are predominantly connected to the CPU cores through PCI interconnect [215]. Other form of coherent interconnects such as IBM CAPI [271], NVIDIA NVLink [195] and AMD global interconnect [1] moves away from the loosely coupled design but the coherency might not be as cheap as observed in asymmetric cores. These interconnects will transform the system in to a NUMA style design. Similarly, Intel has shown that accelerators can be put into processor socket and can exercise the QuickPath interconnect to stay coherent with the processor cores.

Computation Launching. Applications (or runtimes) or the operating system orchestrate the execution of tasks by offloading/migrating them on to appropriate compute units. Each type of compute unit supports different method of accepting tasks and each of them comes with an associated cost that can affect the overall task execution.

Co-processor style compute units can be accessed through instructions available in the ISA. Accelerators closer to the processor pipeline can operate at the granularity of a few instructions to a functional block. For example, the streaming SIMD extensions from Intel [157] and works such as DySER [96] and conservation cores [284] are tightly integrated with the

processors and so they are suitable for fine grained task sizes. Since these accelerators are part of the CPU core, applications get exclusive access to them whenever the thread is scheduled on the CPU.

Similarly, shared accelerators of co-processor style can also be accessed through instructions but tasks once offloaded are not immediately dispatched for execution but rather queued onto a task queue which is then later picked for execution by accelerator unit when available. Such accelerators also support for asynchronous task offload [82] where the program that offloaded the task need not be blocked on the result unlike the previous co-processor type accelerators.

The acceleration devices on the other hand involves high setup cost including task launch that goes via the device driver in the kernel mode and memory transfers involving pinning memory pages [161]. As a result, small tasks are not sufficient to amortize these setup overheads. For example, the cost to copy 64 bytes of data over PCIe in our experimental setup takes around 6 µs and any single task that runs less than that time does not really benefit from the accelerator. The tasks offloaded are required to run for a longer period of time to amortize the setup cost. To offset these high communication costs, techniques such as double buffering and overlapping compute and data copy can be employed although it increases the complexity of application development.

Asymmetric cores differs from the rest in that application threads can directly execute on them without waiting for tasks to be offloaded. Although, operating system or the application has to be map the region of the program code to the type of processor core. The cost of migration is in the granularity of a context switch and additional overhead in terms of cache misses as discussed earlier.

Though asymmetric cores and shared on-chip accelerators are not capable of handling fine grained tasks as co-processors, they can still support smaller tasks. However, the overhead involved in migrating a thread,

queuing into a task queue, sending a command packet to the accelerator or cache misses should be considered while offloading tasks. The work on LogCA [249] gives an analytical model that helping developers to understand the overall speedup by considering the different cost involved while involving accelerators.

2.5 Support for Heterogeneous Architectures

Given that heterogeneity is becoming prevalent and various forms of heterogeneity are available, it is important to understand the different ways to program applications on such architectures. The ultimate goal of any application developer would be to develop applications once and it should be able to run on any hardware configuration. Though techniques like just-in-time compilation generates code based on available hardware features can help, programming for new compute units such as GPUs, FPGAs or other accelerators still pose a challenge. Also, when multiple applications share the same system, access to various compute resources should be virtualized to ensure safe and fair performance for all applications. In this section, we discuss the support available for programming as well as virtualizing the heterogeneous architectures.

2.5.1 Programming Accelerators

The challenge in developing applications over a heterogeneous architecture is similar to that of the transition from single thread to multi-threaded programming. There are variety of compute units and programming them is no easier feat for developers. Compute unit have different architectures that requires compute unit specific implementation to leverage the hardware features and to yield a better performance. However, different levels of abstraction are provided at different layers of the software stack that can simplify the application development.

User Libraries. The simpler way to write programs and achieve high performance is reusing existing modules that are already optimized for a certain task or algorithms over some input data. Similar principle applies for programming heterogeneous compute units as well. The libraries offer abstraction at functional level where the user supplies input and obtains the output without any focus on how computation is to be performed. Open source libraries such as OpenSSL [199] contain task implementations of crypto functionalities leveraging accelerators in current processors. Applications can leverage such libraries through the interfaces exported by the libraries. NVIDIA has released cuDNN [60], a GPU-accelerated library containing primitives useful for deep learning neural networks. Similarly, OpenCV [198] is an open source library that implements computer vision related functions on top of GPU accelerator.

Instructions. Processors hide the microarchitectural details behind the abstraction of instruction set architecture (ISA). Another way to leverage accelerators are through instructions support exposed by the processor vendors. Co-processors style accelerators are accessible through instructions available in the ISA. The Intel x86-64 ISA includes new streaming instructions [80] to access the SIMD units in the processor. Similarly, the shared on-chip accelerators in IBM's wirespeed processor and AMD Kaveri APU are accessible through new instruction support in the ISA. The instruction takes as input a pointer to a command packet that includes the input parameters to the accelerator. Applications can use standard libraries such as OpenSSL and LibZ to leverage the accelerators or the applications can directly make use of the instructions in the ISA.

Programming Models. With variety of compute units to program for, the goal for programming models targeting heterogeneous architectures is to abstract the presence of different compute units. Programming models such as OpenCL [197], AMD HSA [146] and Microsoft's C++ AMP [42] use the same high level language code to generate programs suitable to

multiple compute units such as multi-core CPUs, GPUs and FPGA [61, 81]. OpenACC [196] is a set of directives similar to OpenMP annotations that can be used in programs to represent the portion of the code to be accelerated. Models like StarPU [21] can handle multiple tasks from an application and distribute them across multiple compute units in the system for better utilization. NVIDIA CUDA [186] and StarPU support data object abstraction that hides the physical location of the data and thus obviating the need for explicit data management from the programmers.

2.5.2 Virtualizing Accelerators

Specialized compute units such as accelerators like any other resources in the system can be accessed by multiple programs simultaneously and it is important to understand the software or the hardware component responsible for virtualizing accelerators and also ensuring guaranteed access to accelerators.

Accelerator devices are accessed through device drivers that serves as the bridge between applications and the accelerator unit. Irrespective of the programming model used by applications, all requests to reach the accelerator have to go through the device driver. This makes it a central entity to enforce resource management activities such as virtualization and arbitrating resource usage over accelerators. However, current device drivers employ very simple policy such as FIFO and do not support other policies. This does not allow fair use of accelerators across multiple applications and any faulty or malicious application can deny access to the hardware from other applications. Research works such as PTask [232], Pegasus [101] and Timegraph [133] have proposed solutions to make accelerators such as GPU as a primary resource in the system and ensure fair access to them based on application importance.

For asymmetric cores, the resource management is handled by OS scheduler rather than a device driver since they are compute units that

can run OS and application threads. However, the OS need to know the capability of different cores to perform effective scheduling. Earlier work on scheduler for asymmetric cores [237] tackles the same issue where the powerful cores are shared among high priority applications. Recent works from Linaro have been addressing the problem of identifying hardware capabilities and scheduling threads based on performance and energy needs.

The co-processor style compute units provide user-mode access to accelerator devices [45, 82, 261]. As mentioned before, these systems provide new instructions that enqueue requests to a shared accelerator that reduces communication latency to a great extent since the path through driver/system is completely avoided. However, direct access from user mode means that resource management has to be taken care by the hardware. Recent work [171] propose software based resource management for such systems by throttling the requests queued to accelerators and thus limiting their access. Resource management support is not needed for per-core compute units such as SIMD units [157] but contention across multiple thread contexts sharing the same core can happen. Applications can choose to use an alternate implementation based on normal registers rather than vector units during contention.

2.6 Summary

In this chapter, we presented background materials on heterogeneous architectures essential for this dissertation. We talked about the factors that led to the growth of heterogeneity and why it is here to stay going forward. We described the various benefits of heterogeneity and how different computing environments are leveraging heterogeneity. We classified heterogeneous designs into few categories to help understand the trade-off involved with each design. Finally, we discussed the state of

current systems and the extent of support available to work with heterogeneous architectures.

3

Operating System Support for Dynamic Processors

Architects proposed dynamic processors like Core Fusion [125], TRIPS [239], and WiDGET [291] that can suit the needs of both sequential and parallel programs by changing core characteristics at runtime. These processors in their native state (referred as split or defused state) expose a large number of smaller cores suitable for parallel programs. At runtime, multiple smaller cores can be combined - pooling microarchitectural resources like the reorder buffer or caches from neighboring cores - to form a powerful core (referred as fused state) suitable for sequential programs. Resource pooling can be observed even in current processors where single threaded performance can be achieved by borrowing power through Intel's Turbo Boost or by forcing a neighboring hyper-thread to idle state.

Current operating systems assume that number of processor cores remains same throughout the system execution runtime and also consider every processor core to operate stand-alone without much sharing with neighboring cores. Also, the OS does not tune the CPU resources based on application characteristics (sequential programs benefit from powerful cores whereas parallel programs can make use of many small cores). The execution context (or CPU abstraction as seen by OS), mechanism and policies for CPU resource management in current systems are based on these above assumptions. However, the reconfiguration property of dynamic processors violates these assumption making current operating systems

unsuitable to leverage these kind of processors.

In this chapter, we present *Chameleon*, an operating system extension to support dynamic processors. Chameleon considers CPUs to be in any of these two states: *fused* when multiple cores aggregated as a single core and *split (or defused)* when core acts individually. The foundations of Chameleon are based on supporting resource aggregation in hardware to leverage dynamic processors and ensure better performance for different kinds of applications in the system.

Chameleon introduces three new resource management capabilities in this regard. First, the low-cost *processor proxy* software mechanism allows fine-grain reconfiguration to switch between fused and split states or vice versa as often as a scheduling quantum. Second, the *execution object* abstraction represents an execution context, which may be a stand-alone core or formed after aggregation from multiple cores. Finally, the *cluster scheduler* enables fair use of compute resources for applications preferring either fused or split state even when fused state allows a thread to use more than one CPU core at once.

We begin by reviewing dynamic processor technology in Section 3.1. We follow with the design of Chameleon in Section 3.2 and implementation details in Section 3.3. Finally, we show the evaluation results for Chameleon in Section 3.4.

3.1 Dynamic Processors

While most computers have a static set of processors, hardware trends indicate that future computers may support a dynamically variable set of processors, either for performance, reliability, or power efficiency.

3.1.1 Hardware Mechanisms

We see at least four reasons why the number of execution contexts exposed to an operating system may vary at runtime.

Performance techniques. Many researchers have demonstrated single-thread performance increases by combining several cores into a single more powerful processing element. Core Fusion and TRIPS increase performance by combining resources, such as functional units, into a larger execution engine that can achieve higher ILP [125, 239]. Core Fusion, for example, claims nearly 80% speedup for some programs and requires only 400 cycles to reconfigure. Speculative multi-threading executes loop iterations or function calls in parallel [105, 143, 163, 200]. Sun's canceled Rock processor had SMT contexts that could switch to automatically prefetch data into the cache [49] for improved sequential performance.

Figure 3.1 shows an example of these architectures. Part (a) shows the native cores on a system, and part (b) shows how cores can be fused together to act as more powerful cores. This example is representative of Core Fusion and speculative multithreading.

A common feature of all these mechanisms is that the performance gain is less than linear in the number of cores: executing two threads on two cores accomplishes more work than executing a single thread on a fused core. Thus, a system must balance the need for sequential performance against other uses of the cores.

Power management. Processors may disable cores to save power or to transfer power to the remaining cores, as in Intel's Nehalem Turbo Boost feature [59]. In addition, processors may be *over provisioned*, in that they contain more processing elements than can be used simultaneously [74]. As a result, a system may switch between a single large, fast core and a smaller number of more efficient and slower cores when parallelism is available or better cores provide little benefit [179]. This also enables a

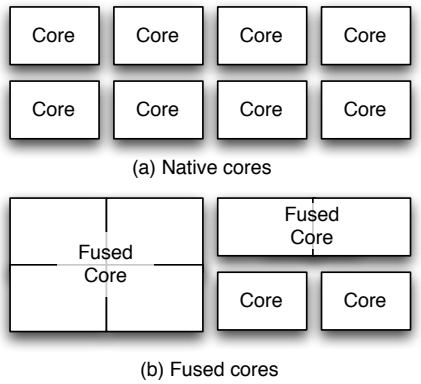

Figure 3.1: The native cores shown in (a) can be reconfigured, for example into a 4-core unit and a 2-core unit shown in (b).

variety of specialized processors for specific tasks, such as encryption.

Reliability techniques. Processing cores may be combined to improve reliability. For example, redundant execution techniques run a thread simultaneously on multiple cores and automatically recover from failures when outputs differ [5, 229, 294]. When surplus cores are available, these techniques promise inexpensive error detection and fault tolerance.

Virtualization. When hosting a website at a cloud provider, a VM can drop to a single processor when workloads are light and use more virtual processors when workloads are heavy.

3.1.2 Operating System Impact of Reconfiguration

These mechanisms all vary the set of processors available to the operating system. However, most OSes must know exactly which CPUs are available at all times. For example, interprocessor communication, whether through inter-processor interrupts or lightweight RPC [27], requires that the OS know if the destination processor is available. In addition, operating systems maintain *per-CPU data structures*, such as run-queues or packet receive queues, to avoid lock and memory contention. Finally, operating

systems perform *all-processor operations* that require the cooperation of all processors, such as read-copy-update (RCU) operations on Linux [167]. Thus, the OS must reconfigure internal data structures whenever the set of available processors change.

We examined the Linux 2.6.31-4 kernel to discover the extent and frequency of these operations. We measure inter-processor communication running `pmake` to build the Linux kernel on a 24-CPU machine with 24 `pmake` processes.

- *Inter-processor interrupts.* Running `pmake` resulted in delivery of 40 IPIs/second/CPU for rescheduling and TLB shootdowns.
- *All-processor operations.* Global RCU callbacks were invoked 140 times per second per CPU when running `pmake`.
- *Per-CPU structures.* We found 446 separate variables in Linux that are defined as per-CPU. The *arch* subdirectory defines 294 variables, which mostly refer to hardware structures. If the set of processors change, these per-CPU data structures must be updated or initialized to reflect the change: 15 subsystems register 35 callbacks to update their per-CPU structures when the set of CPUs change.

Based on these observations, we find that the Linux kernel is intimately aware of the set of available processors. If this set were to change rapidly, numerous subsystems would have to be notified and many per-CPU structures updated. Furthermore, operations that require all processors can only execute when the set of CPUs is stable, either delaying these operations or blocking frequent reconfiguration. While virtual machines can implement reconfiguration, frequent cross-processor communication demonstrates that operating systems must know the set of available processors.

3.2 Design

In this dissertation, we present Chameleon, an operating system extension to support dynamic processors.

We have three design goals for Chameleon:

1. *Rapid adaptation* allows the use of a processor configuration for short periods with low overhead.

2. *Abstracted hardware* provides the operating system with a set of hardware configurations for scheduling threads.

3. *Flexible, intuitive scheduling* allows existing scheduling paradigms/-controls, such as fairness and priority, to apply to dynamic processors.

We seek maximum flexibility in the use of dynamic processors, and thus want to minimize the overhead introduced by the OS when reconfiguring hardware. Low cost allows use of fine-grain reconfiguration, as often as every scheduling quantum. In addition, we want to maintain the operating system's abstract view of hardware so it need not be aware of the details of how a dynamic processor reconfigures, just that other configurations are available. Finally, we seek to extend existing thread schedulers to use dynamic processors, so that existing scheduling policies are naturally extended to cover reconfiguration.

We target Chameleon at dynamic processors with a single instruction set that offer increased performance by disabling some execution contexts and reusing the hardware or power from those contexts. The rapid adaptation targets processors that cannot receive interrupts at all cores in some configurations, such as Core Fusion. In contrast, with Intel's Turbo Boost disabled cores can still receive interrupts, and hence do not benefit from this mechanism.

Our design follows a best-effort approach: the system makes no real-time guarantees, but strives to execute programs as fast as possible. Furthermore, we designed Chameleon to improve performance rather than manage power: the same mechanisms should apply, but must be driven by different policies because fusing CPUs is likely to be less efficient than running on a native CPU. Finally, Chameleon does not address uses of dynamic processors for reliability [5, 294]. Such systems place mandatory requirements on the OS like certain threads must always run in a reliable configuration.

3.2.1 Rapid Adaptation

Future dynamic processors may be able to change the set of available processors rapidly. Operating systems currently use a *hotplug mechanism* or *power management* to adapt to the changes in the set of processors.

Hotplug. Processor hotplug mechanisms [147, 184, 267] allow an operating system to accommodate the addition, removal, or replacement of a processor. They are designed for two uses: maintenance, to remove a failing processor or dynamically add capacity; and virtualization, to change the allocation of processors to a virtual machine. These are both infrequent events, so hotplug implementations optimize for low overhead in the common (no reconfiguration) case, rather than for frequent changes. Reconfigurations that leave a processor able to receive interrupts, such as low-power states [7], can be done with power management rather than hotplug as we describe below.

We measured the performance of hotplug in Linux, and found that it takes 150ms to take a processor offline and 220ms to bring it online. In comparison, the hardware latency of starting a processor is only 10ms. Much of the software overhead in hotplug comes from constructing and distributing per-CPU data structures and quiescing the system with a

global barrier so that the mask of available processors can change. The extra delay when bringing a processor online is largely due to initializing architecture-related registers.

Power management. Operating systems also support power management, which can take a processor offline for short periods to conserve energy. The latency of entering a sleep state is on the order of microseconds, of which very little is spent in software. However, in a sleep state the processor can still receive interrupts, so it is still available to the operating system when needed and global state updates are not necessary.

Thus, power management can only be used for reconfiguration if a processor can still receive interrupts. While this is possible for current architectures, such as Turbo Boost, it may not be possible for architectures that reconfigure hardware, such as Core Fusion.

Thus, we find that both hotplug and power management are inadequate for dynamic processors: hotplug is too slow for frequent reconfiguration, and power management places requirements on the hardware such as receiving interrupts.

Processor proxies. Chameleon provides rapid reconfiguration through *processor proxies*, which are agents running on a separate processor that act on behalf of an offline processor. The proxy can access the private per-CPU data structures of the unavailable CPU. When a physical CPU is temporarily offline due to reconfiguration, Chameleon creates a processor proxy for it on another CPU. The kernel moves its communication endpoints, such as interrupts, to that CPU. Operations that require the presence of a core, such as a TLB shootdown or a read-copy-update (RCU) operation, invoke the proxy and therefore can continue without waiting for the unavailable CPU.

Processor proxies are similar to multiplexing virtual CPUs on a single processor with a hypervisor. However, processor proxies only virtualize the external interface to a processor, such as interrupts and RCU operations.

Thus, a processor proxy does not schedule or run threads. In contrast, a VCPU may run any code and forces the hypervisor to schedule or timeslice multiple CPUs on a single physical CPU.

3.2.2 Abstracted Hardware

Increasingly, operating systems must know the topology of the processor on which they run: whether two execution contexts are hyperthreads on the same core, whether they are cores that share a cache, and whether they share an interface to memory as in a NUMA configuration. The topology allows the operating system to make intelligent scheduling and memory-management decisions, such as to schedule threads from the same process on hyperthreads that share a core, and to allocate memory from a region attached to the thread's core. Asymmetric multiprocessors similarly require informing the OS of a core's increased (or decreased) capabilities.

With dynamic processors, this need for hardware information increases: in addition to the static configuration of hardware, the OS must know about possible dynamic configurations. In Figure 3.1, for example, the OS must know that the four cores on the left and the pair on the right can be fused into more powerful processors.

Execution objects. Chameleon abstracts the physical capabilities of the machine with an indirection layer called *execution objects*. We call the hardware state needed to run a thread (registers, program counter) an *execution context*, which may be a core, a hyperthread, or a fused set of cores. An execution object is a kernel structure that represents a possible execution context, and contains information about the capabilities of the context along with `activate` and `deactivate` methods for reconfiguring the hardware. The smallest native context (*i.e.*, no combining of processors) is a *CPU* (also referred to as a thread context). A *fused execution object* (or

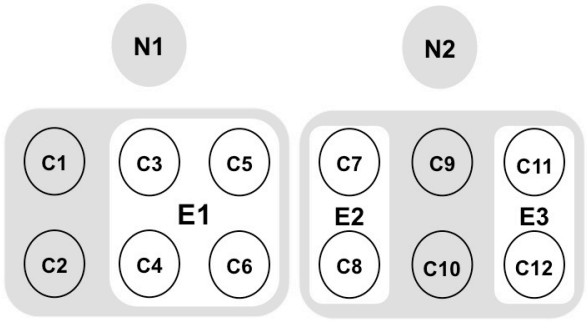

Figure 3.2: System with 12 CPUs (C1-12) managed through nodes.. E2 and E3 are execution objects with two CPUs and E1 with four CPUs.

fused object) represents an object that requires more than one native CPU. For example, in Figure 3.1(b), Chameleon creates a fused object for each fused core.

At any time, the current hardware configuration can be represented as a set of *active* execution objects. Chameleon creates an object when needed by a thread, and the OS invokes `activate` when dispatching a thread to the hardware configuration represented by the object. Each execution object identifies a *representative CPU*, which is the CPU to which interrupts must be delivered when the object is in use.

Execution objects expose *properties* to aid in scheduling decisions. A property is a characteristic of the underlying hardware, such as the relative performance of the configuration or whether additional features/instructions are available (*e.g.*, if only some configurations support SIMD or floating-point operations).

Nodes. Chameleon separates processors into *nodes*, which are groups of CPUs managed together. Each node has a *node manager*, which is responsible for selecting native CPUs within the node to merge into an execution object. The node manager knows the constraints of the hardware, such as which CPUs can be fused or which CPUs can shift power. In addition, it knows the properties of every execution object it can create. Figure 3.2 shows an example of 2 nodes.

When a thread requests an execution object with a property (described in Section 3.2.3), the scheduler selects a node and invokes its manager to request an execution object. The manager makes its best effort to assign the requested resources to the application from the CPUs it manages. As long as a thread is scheduled on an execution object, the node manager keeps the execution object alive. When the thread terminates or requests a different object, the resulting free CPUs can now be assigned to another execution object or left available. The manager prevents fragmentation, which can occur when idle cores are available but hardware constraints prevent them from being fused, by reshuffling the assignment of physical resources to execution objects.

The node manager assigns the CPUs comprising an execution object when creating the object, and does not choose from available CPUs when dispatching a thread. However, a rebalancing mechanism, described in Section 3.3 can relocate threads and create new execution objects if needed.

Node managers are similar to the system knowledge base (SKB) in Barrelfish [27] and Linux and Windows scheduling domains used for NUMA processors [175, 250]. Unlike the SKB, node managers have a restricted focus on processing. Compared to scheduling domains, nodes add constraints on which execution objects can be used simultaneously.

3.2.3 Scheduling

Chameleon extends the OS to schedule threads on execution objects in addition to physical CPUs. The major challenges Chameleon addresses are:

- *Which:* Finding the possible execution objects upon which to schedule a thread.
- *When:* Prioritizing threads relative to each other.

- *Where:* Moving threads to avoid conflicts and minimize reconfiguration overhead.

The Chameleon scheduler leaves scheduling queues attached to native CPUs. When a thread is ready to run, it decides whether to activate an execution object.

Property matching. A key challenge for any asymmetric processor is determining whether and how much a thread benefits from an enhanced execution context. Chameleon does not address this problem, but instead provides a general mechanism to match threads to execution objects. As previously noted, node managers associate properties with a fused object that describe its capabilities, and implement logic to match a request for specific processing features against the properties of different configurations. For example, an execution object providing high instruction-level parallelism for fast sequential execution could have the property `sequential`.

Currently, Chameleon requires that threads specify their desired properties. This could be done explicitly, through a system call, or implicitly by a separate profiling mechanism as in ACMP schedulers (*e.g.*, CAMP [237]). However, Chameleon should work with any mechanism that assigns properties to threads. In the case of a single threaded program, the property might be `sequential`, indicating that the program wants fast execution and does not depend on other threads.

When placing threads in a run queue, the Chameleon scheduler selects a node and invokes its manager to match the desired properties of a thread with the properties offered by the node's execution objects. If there is a match, the node manager creates the object, and Chameleon adds the thread to the run queue of the object's representative CPU and attaches the execution object to the thread; otherwise, it relies on the native OS scheduler to place the thread. The node assignment is similar

to the matchmaking process of the Condor cluster system [227], but with a restricted set of properties.

Cluster scheduling. Chameleon schedules a thread on an execution object as if it is gang scheduling a group of threads on the constituent native CPUs: when all the CPUs are available, the thread will activate the execution object to fuse them. However, Chameleon's scheduler can also elect not to use all the CPUs and instead execute the thread on a single native CPU. Thus, when a thread becomes the next to run, the scheduler determines whether to activate an execution object. We term this *cluster scheduling*.

Chameleon decides when to fuse CPUs together and when to let them run their own threads. Each native CPU in an execution object has its own run queue with threads to execute. When a thread using an execution object becomes ready (*i.e.*, runnable), Chameleon adds it to the run queue for the execution object's representative CPU, and creates *virtual threads* that represent it in the run queues of the CPUs it will borrow. For example, in Figure 3.2, a thread desiring to run on fused context E2 would be placed in its representative, CPU C7, and would have a virtual thread representing it on CPU C8.

Virtual threads represent a thread's ability to borrow CPUs to form a fused object: if a virtual thread has the priority to be dispatched, then its thread can borrow the CPU. Until then, the CPU is available for its own threads. When the scheduler dispatches a thread using an execution object, it checks whether it can activate the execution object. For example, when the scheduler dispatches a sequential thread on C7, it checks to see whether the virtual thread would have been dispatched on the other CPU C8. If so, it preempts any thread on C8, activates the execution object E2, and then dispatches the thread. If not, the thread is directly dispatched on C7.

Cross-CPU contention. Standard CPU scheduling assumes a many-to-one relationship between threads and CPUs, in that many threads share a single CPU and no thread uses more than one CPU. As a result, traditional notions of priority and fair share for threads apply only to a single CPU. However, dynamic processors introduce a new possibility: a thread can preempt multiple CPUs. Thus, threads must have a separate priority or a share for CPUs they borrow. Simply assigning a thread a global share across several CPUs, as in Linux group scheduling [58] is insufficient because it does not reflect the inefficiency of fused objects: under contention, a thread does better on a single CPU than it does running half as long on two CPUs, because speedups are less than linear. With a Linux process group, both configurations would be treated as equivalent.

Chameleon extends the existing notion of priority and share by allowing a thread to have different priorities on its representative CPU and on the CPUs it wants to borrow. We term this mechanism *taxation*: a thread is charged for its use of other CPUs adjusted by a tax rate. Effectively, this means that virtual threads may have different priorities than the thread on the representative CPU. If the tax rate is high, then a thread is charged more for borrowing a CPU than the threads that live on that CPU; thus, its priority is effectively lower and it will not be able to preempt those threads. If the tax rate is low, then it will be able to use other CPUs more cheaply than the threads on those CPUs, and it will be able to preempt them.

3.3 Implementation

Chameleon is implemented as an extension to the Linux 2.6.31-4 kernel. The code changes required by Chameleon were largely concentrated in two Linux subsystems: inter-processor interrupts, to implement proxies; and scheduling, to call into Chameleon during a context switch when activating execution objects. Table 3.3 shows the amount of code comprising

Component	Lines
Processor Proxies	600
Execution Objects	850
Cluster Scheduling	550

Table 3.3: Implementation complexity.

Chameleon's major components.

3.3.1 Processor Proxies

Processor proxies speed reconfiguration because they remove much of the work to change the set of processors. Proxies consist of two elements: (i) methods to create and destroy proxies, (ii) a new execution context for executing interrupts and bottom halves on behalf of the disabled context. The CPU going offline is the *proxied CPU*, and the one that will act as its proxy is the *proxying CPU*.

Proxy creation. The activation of an execution object launches the creation of a proxy by sending a notification to the proxied CPU. The CPU that sends this request will be the proxy. The receiving CPU prepares to go offline by switching to the idle thread, which removes the need to participate in RCU operations, and by ensuring interrupts for the CPU will be delivered to its proxy. These interrupts fall into two categories: device interrupts, which can be redistributed to any online CPUs, and IPIs, which must be sent to the proxying CPU. Each is handled by a separate mechanism.

For device interrupts, the IOAPIC maintains a redirection table indicating the core to which external interrupts should be sent [122]. However, reprogramming the IOAPIC is slow, as we show in Section 3.4. Instead, Chameleon leverages APIC *logical addressing* when possible: device interrupts are broadcast to a logical address and each CPU ANDs a local mask against the interrupt's address and delivers the interrupt if any common bits are set. Chameleon therefore does logical-address renaming by adding the proxied CPU's identifier to the mask for the proxying CPU and

resetting the identifier for the proxied CPU. This causes interrupts, both external and inter-processor, destined for the proxied CPU to be delivered to the proxying CPU automatically.

However, logical address renaming may not be available on all systems because the number of bits representing different CPUs is limited. Thus, we implemented a separate software mechanism to redirect IPIs to the proxying CPU. When creating a proxy, Chameleon records in a *redirection table* that the proxied CPU is being proxied. When a CPU sends an IPI, it consults the redirection table to learn where the IPI should be sent. In this case, Chameleon reprograms the IOAPIC to redirect device interrupts.

When using logical address renaming, redirecting interrupts can cause an IPI to be delivered to two CPUs or not delivered at all, based on the order in which the mask of proxying CPU and proxied CPU are changed. Since IPI handlers in Linux are idempotent and may be called multiple times without harm, Chameleon always updates the local mask of proxying CPU to include proxied CPU's identifier before resetting the mask of proxied CPU. Reversing this order could result in loss of IPIs. Furthermore, when tearing down a proxy, Chameleon invokes IPI handlers on the proxied CPU before invoking the scheduler in case an IPI was lost. These issues do not arise with device interrupts because the hardware ensures they are delivered to only one CPU even if the logical address matches multiple CPUs.

Proxy context. We add a new execution context to the OS, in addition to process context and interrupt context, termed a *proxy context*. A separate proxy context exists on a CPU for each of the processors it proxies and executes only when the proxying CPU receives inter-processor interrupts (IPIs) on behalf of the proxied CPU. We augmented the per-CPU structures with two variables to track proxies: `proxied_by` for a CPU that is being proxied, and `proxying_for` on a CPU that is acting as a proxy.

On receiving any IPIs, the proxying CPU invokes the corresponding

IPI handler natively for the proxying CPU in the proxied CPU's context. Handling IPIs requires access to per-CPU variables, which are normally accessed through the x86 segment registers. When entering proxy context, Chameleon sets the FS register to point to the per-CPU data of the proxied CPU, and resets the register when leaving proxy context. In addition, we modified the `thread_info` macro, which normally uses the stack pointer to find the CPU state of the running task. In proxy context, the macro directs accesses to the data for the proxied CPU.

With proxies, Chameleon ensures that kernel operations requiring the involvement of an offline CPU can proceed, as the role of that CPU is handled by its proxy. This includes inter-processor interrupts for scheduling, read-copy-update operations, and TLB shootdowns, which are dropped because the proxy code flushes the TLB when resuming normal operation.

In some architectures processor proxies may not always be needed: if CPUs can continue to receive interrupts when disabled, the object does not create a proxy but instead halts the CPUs. This occurs when using Chameleon with hyperthreads: the execution object schedules the idle thread on the other hyperthread. Similarly, with Intel's Turbo Boost feature other cores enter a sleep state but can still receive interrupts.

3.3.2 Execution Objects and Node Managers

Chameleon assigns all the CPUs on a socket to a node and instantiates a node manager for each node. The node manager is a kernel component that tracks the execution objects and CPUs on a node. For now, Chameleon uses the existing topology information provided by platform drivers (*e.g.*, , ACPI). We emulate a dynamic processor by informing node managers that they may construct fused execution objects for pairs of hyperthreads on a core and pairs of adjacent cores.

The only property we have implemented is `sequential` with a level, which is the $\log_2$ of the number of hyperthreads in the object. Threads

can request a sequential object of a specified level. Property matching compares the desired level of a thread against available execution objects to find one with the same level. The `activate` method on an execution object configures the hardware to create the desired execution context, and must be called from the object's representative CPU (the lowest numbered CPU in the set). It also creates proxies for the native CPUs borrowed by the object, and then directs the hardware to reconfigure. The `deactivate` method does the reverse of `activate`; it directs the hardware to enable native CPUs and removes proxies for all the CPUs involved.

A thread invokes the node manager to request an execution object. The node manager creates an execution object and assigns unallocated CPUs that fit the object's constraints (contiguous IDs for our emulation) to the execution object. A CPU can only belong to one non-nested execution objects. Thus, in Figure 3.2, CPU C10 could not be part of objects E2 and E3. This constraint may be mandatory for processors that share physical resources, such as Core Fusion, but could be relaxed for systems with more flexible resource sharing.

Any request for change in the type of execution object for a thread also invokes the node manager. If the set of available native CPUs changes, such as when a thread terminates or changes its request for an execution object, the node manager can reassign CPUs between objects to avoid fragmentation that arises when enough idle CPUs are available to create a fused object, but hardware constraints prevent its creation.

3.3.3 Cluster Scheduling

Chameleon's cluster scheduler is built on top of the native Linux scheduler, the Completely Fair Scheduler (CFS) [128]. CFS is similar to Borrowed Virtual Time scheduling [70], and schedules tasks according to the CPU time they have used recently rather than priority. The tasks within a runqueue are ordered by a *virtual runtime* value (`vruntime` in the task

structure), which is a measure of how long the task has run. The task with the lowest virtual runtime value (*i.e.*, is the furthest behind) is selected to run next. The virtual runtime value increases in proportion to the time spent executing.

Chameleon schedules all tasks in native Linux run queues. For threads requesting an execution object, Chameleon adds the thread to the run queue of the object's representative CPU. However, virtual threads, used to track a thread's priority on the borrowed CPUs, are *not* added to run queues. Instead, Chameleon leverages the CFS scheduler design: it records the virtual runtime value a thread *would receive* on another CPU had it been scheduled. This value is set when a thread is added to the queue and does not change, making it fast to record. When seeking to activate an object for a thread, the cluster scheduler is allowed to borrow a CPU, say C, if the thread's recorded virtual runtime value for CPU C is below the virtual runtime value of the thread currently running on CPU C. This indicates that, had it been a real thread, it would have been dispatched already. We describe below how this comparison is performed efficiently.

Activating execution objects

When a thread reaches the head of the run queue, Chameleon consults the thread's task structure to see whether the thread requested an execution object. If so, it checks whether it can activate the object. To make this efficient, Chameleon extends the Linux task structure with a `vruntimes[]` array containing an element for each CPU comprising an execution object. When adding a thread to a run queue, the cluster scheduler updates `vruntimes[]` for the CPUs in the execution object. In addition, each CFS runqueue stores the `vruntime` value a new thread would receive in its `min_vruntime` variable, and Chameleon copies this into the thread when initializing `vruntimes[]`.

When a thread finally becomes the next to run, the cluster scheduler

compares the thread's `vruntimes[CPU]` with the virtual runtime value of the active thread for each CPU it needs to borrow. It only preempts the neighboring CPUs if the thread's `vruntime` is lower on *all* the needed CPUs. This requires preempting other threads, so we enable kernel preemption.

If the scheduler cannot form the execution object desired by a thread, it will try to instantiate an object with just the available CPUs, if one has the thread's requested properties (*e.g.*, , sequential). For example, in Figure 3.2, if a thread wants object E1 but either CPU C5 or C6 is unavailable, the scheduler will see if it can form an object from CPUs C3 and C4. If so, it activates the execution object with just the available CPUs.

Preempting Execution Objects

While a thread executes on a fused object, threads scheduled on its borrowed CPUs may wake up and become runnable at higher priority. In addition, as a thread on a fused object executes, its `vruntime` increases so that it may no longer be the highest priority thread on all the borrowed CPUs.

Chameleon tracks how long a thread on a fused object can run by calculating when the next thread on every borrowed CPU is allowed to run, based on `vruntime` values of the threads it preempted and the running thread's `vruntimes[]` values. During every timer tick, the cluster scheduler checks whether any of the threads on a fused object's constituent CPUs are allowed to run, and if so deactivates the object and reschedules the running thread.

Reconfiguration takes time, so care must be taken to avoid reconfiguring too often. As we show in Section 3.4, fusing an execution object can take up to 8 µs. Chameleon relies on the existing `sysctl_sched_min_granularity` variable from CFS to set how long a thread can run before being preempted when it is no longer the highest priority thread.

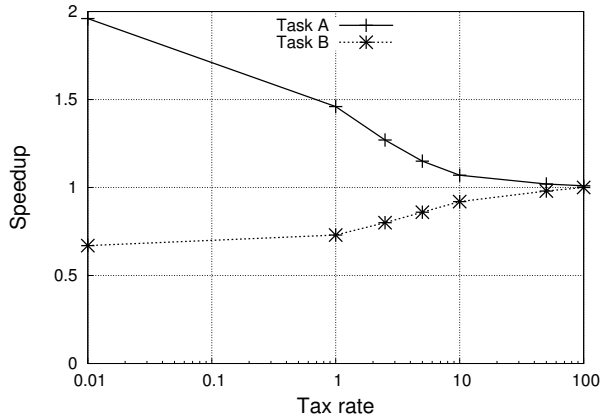

Figure 3.4: The effect of taxation on two identical tasks. Execution time is relative to the thread executing alone on a single native CPU.

Taxation

Taxation controls the relative priority of threads executing natively and threads that want to borrow a CPU for an execution object. CFS already adjusts the rate at which virtual runtime accrues based on priority: lower-priority tasks accumulate virtual runtime faster, so they must wait longer to execute again, while high priority tasks accumulate it more slowly, letting them run sooner. Taxation further adjusts that rate. Much as priority adjusts both scheduling latency and CPU share, taxation can act as a share, so that unimportant tasks only use idle CPUs for fused objects, or as a mechanism to reward threads that achieve high speedups with a fused object.

Chameleon adds an `eo_tax_percent` field to every task structure. When a thread runs on an execution object, the scheduler calculates the delta to its `vruntime` for the representative CPU and all borrowed CPUs, weighted by its priority. For borrowed CPUs, the scheduler multiplies the delta by the tax rate before adding to the `vruntimes[]` array.

Figure 3.4 shows the impact of changing the tax rate when running two

CPU-bound tasks on neighboring CPUs. Task A requests an execution object, while B runs on one of the CPUs assigned to the execution object. If the tax rate is high ($>> 1$), then task A is charged more than task B for using its CPU and thus it sticks with native execution. If the tax rate is 1, then task A receives all of its CPU plus half of task B's CPU. If the tax rate is low ($<< 1$), then task A can preempt task B much of the time because it is charged less than B for using the CPU. Thus, taxation implements a cross-CPU priority mechanism.

Tax rates can be set automatically from the speedup a thread receives on a fused context. We observe that a thread that shows a good speedup deserves more access to a fused object. If progress rates metrics, such as hardware performance counters or application-specific heartbeats [114] are available, a thread's taxation rate can be set according to its speedup. Threads with good speedups deserve lower tax rates, allowing them to borrow neighboring CPUs aggressively, while threads with low speedups should only borrow idle CPUs and deserve high tax rates.

Property Matching

A thread declares its properties of interest by providing the identity of the property (an integer identifier), and a weight indicating the importance of the property. The node manager matches properties by comparing a thread's requested properties against the properties of all possible fused objects in the node. The matching process produces a list of possible execution objects. Currently, all properties are optional, so a thread will still be scheduled on any available CPU if an object with its requested properties is not available.

The current Chameleon implementation relies on a programmer or external agent to specify the properties a thread desires. To prioritize threads that benefit more from a fused object, the node manager uses the weight provided by a thread to identify which thread receives more benefit

from a fused object when there is contention. Existing ACMP scheduling mechanisms or static profiling could provide information about the speedup of fused objects [238].

Rebalancing

The node manager spreads out threads requiring a fused object within a node so they can run concurrently, and to shift resources to threads that benefit from them more. When an execution object is freed, the CPUs it used are given to the active execution objects if they need them, such as an execution object that requested 4 CPUs but received only 2 CPUs. This assignment is prioritized based on the weight assigned to threads' properties.

When the node manager is not able to assign the requested number of native CPUs to an execution object, it tries to reclaim CPUs from low-priority threads and reassign these CPUs to a new fused object. If this is not possible, the node manager breaks the largest execution object and assign half the CPUs to a new execution object. This ensures that native CPUs are not reserved for a single thread when multiple threads request fused objects.

In addition, the node manager defragments execution objects. If the requested number of CPUs for an execution object are available within a node, but physical constraints prevent them from being fused (*e.g.*, , they are not physically contiguous), the node manager detects fragmentation. It will then migrate threads to create a contiguous block of CPUs that can be used to satisfy future requests. Whenever the node manager makes any change to the physical resource assignment to the execution object, it notifies the execution object, which also updates the `vruntimes[]` for any threads scheduled on the object.

3.4 Evaluation

We evaluate Chameleon through emulation to answer these questions:

1. *Cost:* What is the latency of reconfiguring with processor proxies, and the added costs of scheduling with execution objects?

2. *Benefit:* Does Chameleon enable threads needing higher sequential performance to receive it, while allowing parallel programs full use of the CPUs in a system?

3. *Contention:* Does Chameleon behave reasonably and predictably when there are many threads contending for resources?

As dynamic processing hardware is not yet available, we evaluate Chameleon through emulation.

3.4.1 Experimental Platform

We emulate dynamic processors and ACMPs on a standard multi-core system by varying the performance of the different CPUs. We performed our experiments on 32-bit Linux kernel version 2.6.31-4 running on a system with 12 GB RAM and two Intel Xeon X5650 chips, each chip containing six cores and each core with two hyperthreads. We refer hyperthreads as CPUs in this section. We instructed Chameleon to create two nodes of 6 cores (12 CPUs) each, and allow each node to create execution objects with 2 or 4 CPUs. We disabled TurboBoost because it varied the frequency when the system entered the P0 state, leading to widely fluctuating results. We leave the minimum scheduling granularity at the default value, 16ms.

We could not use DVFS to emulate asymmetric performance, as on our platform it applies to an entire socket rather than a single CPU. Instead, we use Intel's *clock-modulation* feature [123, 301] similar to past research on dynamic processors [12]. This mechanism is used for thermal throttling

Architecture Model	CPUs	Speedup in Duty Cycle percent		
	slow, fast	low	med	high
CMP	24, -	50	50	50
ACMP-3	12, 3	75	87.5	100
ACMP-6	12, 6	62.5	75	75
Dynamic	Fuse 2 threads	62.5	75	75
	Fuse 4 threads	75	87.5	100

Table 3.5: CMP, ACMP, and Dynamic configurations.

and controls the processor duty cycle by stopping the clock for short periods (less than 3µs) at regular intervals. There are eight levels available through the `IA32_CLOCK_MODULATION` model specific register (MSR). These levels reduce performance from 100% down to 12.5% of full performance in steps of 12.5%. For emulation, the `activate` method on an execution object creates the processor proxy and raises the duty cycle. Unlike real ACMPs or dynamic processors, the performance impact of clock modulation is independent of the code execution. Thus, a program sees a performance drop of 50% if the duty cycle is cut in half.

Table 3.5 lists the configurations we use in experiments. In all cases, we assume the baseline performance is 50% of maximum, and we emulate a faster CPU or fused object by increasing the duty cycle. For a symmetric CMP, we set all 24 CPUs to the baseline speed. For other models, we try to keep the approximate chip complexity similar. The emulated asymmetric CMP systems have either 3 or 6 fast CPUs with 75% or 100% of native performance. We use the native Linux scheduler in ACMP configuration but pin specific threads to the more powerful CPUs. As different programs see different speedups on asymmetric processors, we assign sequential programs to three performance categories (low, medium, high), with different speedups on the fast CPUs.

For dynamic processors, we follow past work [112] and set the maxi-

Program	Description
Fibonacci	Task-parallel threads
pmake	Task-parallel processes
Blackscholes [31]	Data-parallel threads
Mandelbrot-DB (M-DB)	OpenMP kernel, dyn. binding
Mandelbrot-SB (M-SB)	OpenMP kernel, static binding
gcc [110]	Low-CPU single thread
astar [110]	Low-CPU single thread
dealII [110]	Medium-CPU single thread
lbm [110]	Medium-CPU single thread
sjeng [110]	High-CPU single thread
N-Queen [148]	High-CPU single thread
h264ref [110]	High-CPU single thread

Table 3.6: Workloads.

mum performance of fusing 2 threads to 75% (a 50% improvement over the base CPU's performance), and of 4 threads to 100%, doubling the performance of a base CPU. Again, we set different speedups for each sequential performance category to mirror the ACMP speedups. Achieving the same speedups with ACMP and dynamic processors may be optimistic, but it helps illustrate the benefit of being able to reconfigure.

We do not add additional delay to emulate the hardware cost of reconfiguring. Core Fusion estimates the delay at 400 cycles plus a pipeline flush [125], and other systems do not give any latencies. However, processor proxies flush the TLB when they are torn down, so the flush plus cost of subsequent TLB misses is included in our results.

3.4.2 Workloads

Past work has shown that I/O and memory-bound workloads receive little benefit from faster cores [25], so we largely evaluate with CPU-bound programs. Table 3.6 lists the workloads we use to evaluate Chameleon. As we do not have real hardware, variations in how a program performs

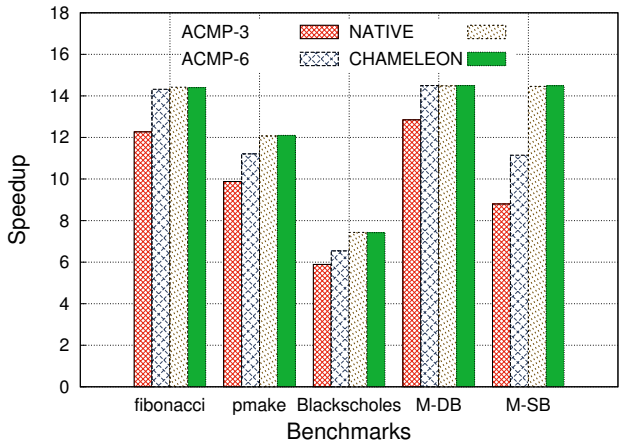

Figure 3.7: Performance of parallel programs.

on specific hardware cannot be evaluated. Thus, we instead evaluate on programs with different styles of a parallelism: task parallel threads on Intel's Thread Building Blocks (Fibonacci), task parallel processes (pmake); data-parallel threads (Blackscholes), OpenMP with a static binding of tasks to threads (Mandelbrot-SB); and OpenMP with a dynamic binding of tasks to threads (Mandelbrot-DB), which can execute more tasks given a more powerful CPU. We ran a variety of SpecCPU benchmarks on our emulator, and found they performed similarly to simple kernel benchmarks, so we also use a simple N-Queen program for sequential programs. We repeat experiments at least three times and report the average results. As there is little variance in the measurements, we do not include error bars.

3.4.3 Baseline Results

We evaluate Chameleon on single workloads to validate the emulator and to evaluate how close Chameleon gets to ideal performance.

Parallel performance. Figure 3.7 shows the performance of parallel workloads on four configurations: the CMP, the two ACMP models, and

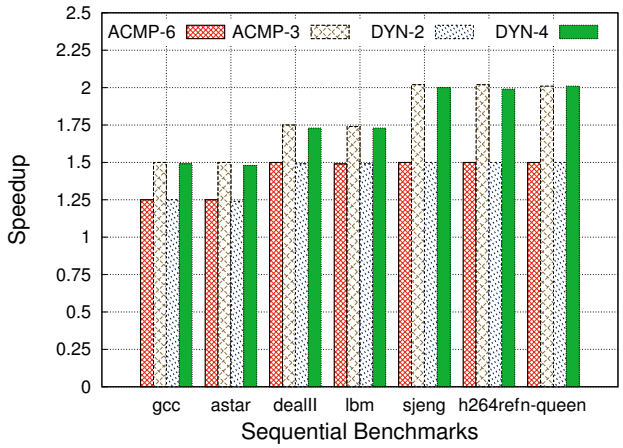

Figure 3.8: Performance of sequential programs.

Chameleon. For all the parallel workloads, the best performance comes from the CMP configuration, which provides 24 CPUs. In the ACMP models, the lost parallelism outweighs the powerful CPUs present in the system except for Fibonacci and Mandelbrot with dynamic scheduling. Programs with static thread scheduling, such as Blackscholes, suffer from load imbalance when the threads on the fast CPUs wait for the threads on slow ones to finish. Chameleon uses all CPUs to achieve performance similar identical to the CMP.

Sequential performance. Figure 3.8 shows the performance of single-threaded workloads with varying benefit from faster CPUs. We execute each program on a single hyperthread, with the rest of the system idle. On ACMP systems we pin the program to a powerful core. For Chameleon, we evaluate both fusing two and four CPUs (DYN-2 and DYN-4). For these workloads, Chameleon is able to achieve the same performance as the asymmetric systems, despite using processor proxies. These results demonstrate that proxying introduces very little overhead and allow programs full access to the hardware's performance. In addition, they demonstrate that our emulation mechanism provides identical speedups across different programs.

Operation	Hotplug	Proxy - APIC	Proxy - Logical
Fuse 2 CPUs	150ms	250µs	2µs
Fuse 4 CPUs	430ms	375µs	8µs
Split 2 CPUs	220ms	20µs	1.5µs
Split 4 CPUs	640ms	60µs	4.2µs

Table 3.9: Latency of reconfiguration.

Overhead. The overhead of Chameleon arises in two places: reconfiguration and scheduling. The latency of fusing and splitting CPUs is shown in Table 3.9. We present two versions of Chameleon's split and fuse operations: **Proxy - APIC** reprograms the IOAPIC to deliver interrupts to the proxying CPU during a fuse, and **Proxy - Logical** uses the logical addressing technique described in Section 3.3.1. When reprogramming the IOAPIC, proxy creation takes between 250-375µs. Logical addressing does not communicate with the slow IOAPIC and is 50-100 times faster. Compared to the native Linux hotplug mechanism, Chameleon is up to 75,000 times faster at fusing (hot *un*plugging a CPU) and 160,000 times faster at splitting an execution object of two CPUs (hot plugging a CPU).

The number of processors that can be addressed with logical addresses may be limited in some systems, and in such cases IOAPIC reprogramming is needed. The fuse case for **Proxy - APIC** is more expensive than splitting because Chameleon reprograms the IOAPIC to remove the proxied CPU. When splitting a proxy, it does not reprogram the IOAPIC to include the proxied CPU (this is the same behavior as Linux hotplug). Fusing and splitting four CPUs is costlier since the proxy creation and destruction phases are carried out sequentially. The major savings compared to hotplug come from avoiding the notification of subsystems that the set of CPUs has changed. The remainder of our experiments uses the IOAPIC reprogramming method, as logical addressing does not currently support 24 CPUs.

We measured the added scheduling work during context switches, and there was no difference between context switching native threads on

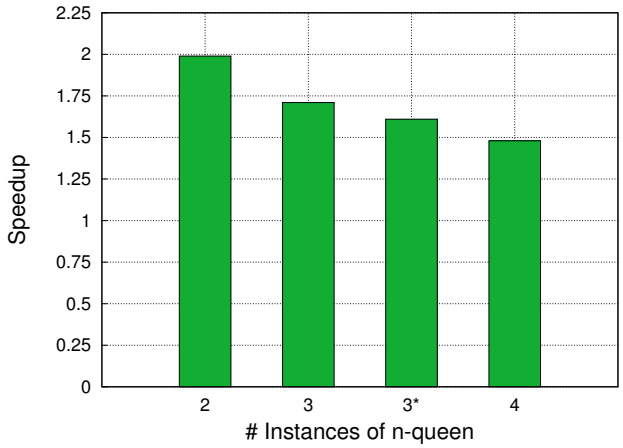

Figure 3.10: Over-provisioned performance of 2-4 sequential programs and a parallel program. The column marked by 3* was run without the node manager notification mechanism.

Chameleon and native Linux: both took between 2.5µs - 3µs. These results demonstrate that the added latency of Chameleon's scheduling techniques is low.

IPIs handled by proxy. We ran the pmake workload alongside a sequential application making use of an execution object with 4 CPUs. The proxy created during the activation of the execution object is responsible for handling the IPIs destined to the CPUs of the execution object and in this case was proxying for 3 CPUs. The proxy handled 10 IPIs per second, mostly TLB shootdowns. This number is lower than the 40 IPIs/second reported in Section 3.1 because the offline CPUs switch to the idle thread, which avoids rescheduling IPIs. However, the proxy still receives TLB shootdowns because the idle thread leaves the previous user-mode address space from `pmake` on the processor.

3.4.4 Scheduling Mixed Workloads

We evaluate Chameleon with a mix of workloads under two situations: *over-provisioned*, when there are more CPUs than threads; and *under-provisioned*, where there are not enough CPUs. In each case, we start a mix of parallel and sequential programs at the same time and measure their completion time. If one program finishes early, the other program can make use of its CPUs. We use the Mandelbrot and N-Queen kernels because they are purely CPU bound and their performance reflects only the added effect of scheduling decisions by Chameleon, as shown in Figure 3.8.

Ideally, when there are idle CPUs, Chameleon will opportunistically use them to execute sequential tasks. When there is contention, Chameleon should only use them if the tax rate allows sequential threads to preempt threads on neighboring CPUs.

Over-provisioned performance. We measure the performance of executing a parallel program requiring 16 CPUs and 2-4 sequential programs simultaneously. The results in Figure 3.10 show that for two sequential programs, Chameleon creates a 4-CPU fused object for each task that achieves almost double the baseline performance. The four sequential threads are scheduled on 2-CPU fused objects, and achieve 50% speedup over baseline. The case of 3 sequential threads is explained more in the following paragraph. Thus, Chameleon is able to effectively place sequential threads when there are idle CPUs, and can balance them to achieve maximum performance.

This experiment also demonstrates Chameleon's load balancing capability. With three sequential threads and eight available CPUs, one of the threads can run on a 4-CPU fused object, while the others run on 2-CPU fused objects. When the faster thread completes, the node manager distributes the physical resource used by the just finished thread to currently active threads, allowing both remaining threads to use 4-CPU fused

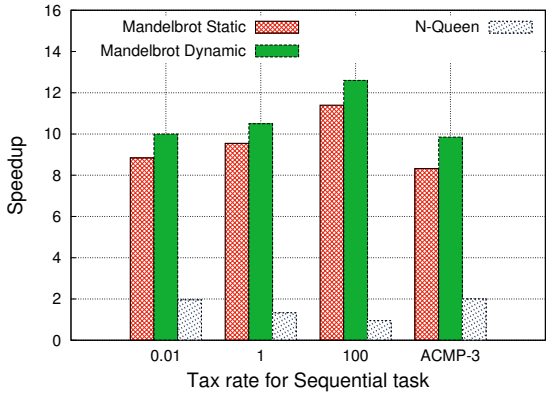

Figure 3.11: Under-provisioned performance of 3 instances of N-Queen program and a parallel program with varying taxation rates.

objects. The column labeled 3* shows the benefit of notifying the node manager when a task completes. When we disable notification, which redistributes idle CPUs, performance suffers because only one thread uses a 4-CPU fused object even when idle CPUs are available.

Under-provisioned performance. We perform a similar test using three sequential programs and a parallel program with 24 threads. As there are no idle CPUs, Chameleon must decide whether to timeshare CPUs between the two programs. For these experiments, we test with both static and dynamic binding of the parallel threads, and vary the tax rate between 0.01, 1, and 100. We also present the result for the ACMP-3 configuration. In the ACMP configuration, the sequential programs were pinned to the fast CPUs while letting Linux schedule 15 parallel threads.

Figure 3.11 shows the speedup for the sequential and parallel programs relative to running on a single baseline CPU. As this is a single hyperthread, the speedup from using more CPUs is less than linear in the number of CPUs available. With a low tax rate, the sequential program is able to borrow the neighboring CPUs most of the time for a speedup of 97% over baseline (the speedup is less than 100% because the parallel thread

occasionally uses the CPU). With a tax rate of 1, the sequential program borrows CPUs approximately half the time. But since the parallel threads also use the CPU the observed speedup is less than 50%. With a high tax rate, the sequential workloads were not able to preempt the parallel threads. As a result they complete at baseline speedup.

The parallel program shows similar variation: for dynamic scheduling where threads can have varying amounts of work, performance drops in direct proportion to the time N-Queen borrows a CPU: with the high tax rate, it gets a 11.4x speedup over baseline, while with a low tax that drops to 10x. With a low tax, N-Queen finishes quickly and the parallel program then uses all 24 CPUs. With static scheduling, where every thread must perform the same amount of work, performance varies from 11.4x speedup with the high tax rate to 8.8x speedup with a low tax rate, because of load imbalance while the N-Queen is running. Thus, the use of tax must consider both the benefit to sequential programs and the cost to the parallel programs that are preempted. The N-Queen program's use of the CPU is similar when run with both the static and dynamic parallel programs, so its performance does not change.

Compared to the ACMP, Chameleon achieves the same sequential performance and parallel performance on the dynamic parallel program with a low tax rate because it has more CPUs. The higher tax rates trade lower sequential performance for parallel performance exceeding the ACMP. These results show how taxation adjusts the priority of sequential and parallel programs. Its benefit may depend on how well parallel programs react to losing a CPU.

Mixed sequential workloads. This experiment demonstrates Chameleon's ability to prioritize the use of CPUs based on the speedup a thread achieves. In this experiment we ran a mix of sequential workloads of different classes: one high CPU (sjeng), one medium CPU (lbm) and two low-CPU workloads (astar) with staggered start times. We

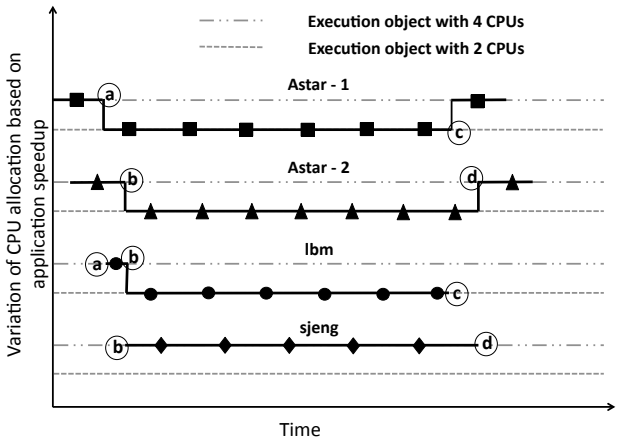

Figure 3.12: CPU allocation to execution object in a workload mix of sequential programs with different speedups.

annotated each program's thread with its speedup as the weight on its sequential property, similar to what might be provided by an ACMP scheduler. We constrained all the programs to a single node with a total of 10 CPUs.

Ideally, the rebalancer will spread the available CPUs across the sequential programs and prioritize remaining CPUs to the programs with the best speedup. Figure 3.12 shows the results of this experiment. The two low CPU (astar) workloads were started first and were each given a 4-CPU fused object by the node manager. When lbm – a medium speedup application – was started subsequently, the manager borrowed 2 CPUs from one of the low-speedup astar applications and gave them to lbm at event ⓐ. Similarly, when sjeng – a high speedup application – started, it borrowed 2 CPUs, one each from lbm the other astar, at event ⓑ.

In short, when the remaining two workloads started, the node manager noted that the demand for sequential performance (a total of 16 CPUs forming 4 fused objects) exceeded the supply (10 CPUs), and split the objects in use by the astar instances so the two remaining programs could

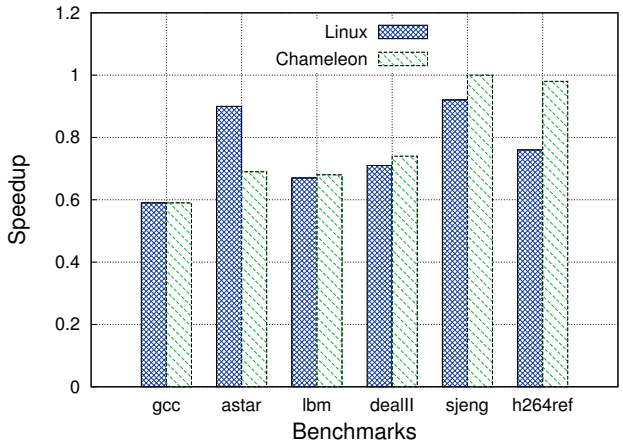

Figure 3.13: Chameleon on SMT.

each receive a fused object. As described in Section 3.3.3, the node manager uses the speedups of the programs to allocate the remaining CPUs to the program receiving the most benefit, sjeng, which then runs on a 4-CPU fused object. The other three programs all used a 2-CPU fused object. Upon exit of any workload, the node manager distributes the released CPUs to the active workloads based on their speedup. In this case, lbm completes first at event ⓒ and its resources are given to one astar instance. After sjeng exits at event ⓓ, its resources are given to the second astar.

Thus, Chameleon's node manager and rebalancer are able to allocate CPUs across a set of workloads with different speedups, when notified of those speedups.

3.4.5 Chameleon on Real Hardware

The previous results show that Chameleon can exploit the flexibility of dynamic processors to achieve improved performance. Chameleon can also prioritize threads on existing hyperthreaded processors. Since hyperthreads share the cache and processor pipeline, running two hyperthreads on a core may reduce the performance when compared to running a single

thread. As a result, the Linux scheduler tries to schedule threads on separate cores before using two hyperthreads on a single core. However, when there are more threads than cores, it is possible that a thread that benefits from running alone on a core may get scheduled on a shared CPU.

We apply Chameleon's cluster scheduler to this problem by treating each core as a fused execution object, which when activated forces one of the hyperthreads to idle. We ran one instance of six SPEC application annotated by their corresponding speedup values as listed in Table 3.6 over 4 physical CPUs (8 hyperthreads) with the native Linux scheduler and Chameleon. The baseline is the application running on a non-shared CPU. Figure 3.13 compares the speedup achieved on the two systems. Linux has no knowledge each application's speedup and thus any application may be scheduled on a non-shared core. In this case, sjeng and h264 achieve a good speedup. Chameleon therefore prioritizes these two programs and schedules them on non-shared cores. Thus, Chameleon is able to prioritize resources for threads that benefit more from running alone, while Linux treats all threads equally.

3.5 Conclusion

Dynamic processors will lead to new opportunities for improving performance, reliability, and power consumption by reconfiguring the set of running processors. Existing operating systems cannot react to changes fast enough to fully utilize reconfiguration, and do not have scheduling mechanisms to take advantage of them. In this chapter, we discussed Chameleon that extends Linux to enable rapid reconfiguration through processors proxies, allowing use of reconfiguration even for short periods. It abstracts the reconfiguration abilities of the hardware with execution objects and nodes, which expose the new capabilities of the hardware to programmers and the scheduler. Chameleon's cluster scheduling with

taxation allows sequential code to use idle cores and provides a flexible trade-off between single-thread performance and parallel performance.

We have also extended Chameleon for other forms of dynamic processors such as processors with dark silicon where all cores cannot be used at full power the same time. The discussion on this extension is presented in Chapter 5. We also plan to extend Chameleon for other uses of dynamic processors. Chameleon focuses on performance benefits of dynamic processors, but they should also promise power efficiency and reliability, which demand different scheduling policies.

4

Efficient Resource Use in Heterogeneous Architectures

The provisioning of accelerators such as GPUs, DSPs, and video encoders is becoming common in architectural designs. There are two major trends that make task placement harder in accelerator rich systems. First, a single task can run on different compute units like a parallel task running on CPUs or GPU. Second, mature programming models such as OpenCL [197] and C++ AMP [42] allow applications to automatically make use of accelerators. When more applications are accessing accelerators, contention for accelerators can arise.

The task placement decision has to consider the performance trade-off on different compute units where the task can run. However, a key challenge is that the trade-offs does not remain the same due to sharing. Current systems and runtimes use stand-alone performance information to make task placement decisions assuming a single application makes the most use of accelerators. However, it is clear from the above trend that static decisions are not sufficient in a shared environment. A contended accelerator might perform worse than an idle CPU.

In this chapter, we present *Rinnegan* [210], a system with a kernel extension and a runtime library, to perform scheduling and task placement in a shared heterogeneous environment. Rather than relying on static performance information alone, Rinnegan combines them with current system state information including the load on accelerators while making

task placement decisions. *OpenKernel*, the kernel component in Rinnegan, allows resource managers (e.g. GPU driver, CPU scheduler) to share the system state information with applications. *Libadept*, user-mode runtime, estimates the stand-alone performance of application tasks on different compute units. The runtime combines the performance estimate with system state information during task offloads to make better task placement decisions.

Rinnegan also decouples task placement from scheduling contrary to current systems and perform task placement in applications rather than in the kernel. This enables adaptability in applications (e.g. trade-off quality for performance due to lack of sufficient resources) and making custom task placement decisions based on the performance goals of applications. Resource management, including scheduling and resource allocation, is still performed in the kernel to ensure proper isolation among applications.

We start with describing the design of Rinnegan in Section 4.1 that includes the design of OpenKernel and the libadept as well. We follow with the implementation of Rinnegan in Section 4.2 and then finally we show the evaluation results for Rinnegan in Section 4.3.

4.1 Design

We designed Rinnegan as a decentralized architecture separating *resource management* (how much access a process gets to a resource), from *task placement* (where tasks run) decisions as shown in Figure 4.1. The former is left to the OS kernel, which is responsible for isolation, while the latter is pushed closer to applications. To support high-quality placement, the OS publishes information allowing an application to accurately predict its expected performance on all processing units in the system.

Compared to prior systems that perform both resource management and task placement in the kernel [101, 232, 237], this architecture offers

several benefits:

- Application frameworks for heterogeneous systems already perform task placement, so this design fits well into existing systems.
- Applications can easily specify their own performance goals or modify their behavior in response to contention, as the placement policy is tightly coupled to the application.
- Performing placement in applications avoid adding complexity to the OS kernel and makes the Rinnegan more easily portable to other operating systems.

4.1.1 Platform Requirements

We designed Rinnegan to target heterogeneous systems with multiprogrammed workloads such as desktops and laptops, mobile devices [85], and shared clusters [131].

Heterogeneous. Rinnegan target architectures that provide heterogeneous processing units in the form of discrete or on-chip accelerators, and both single-ISA and multi-ISA heterogeneous processors. Target systems include unified CPU/GPU chip, such as AMD's APUs [13], systems with discrete GPUs, or with on-chip accelerator devices [82, 120].

Multiprogrammed. Without multiprogramming, a single application can take complete control of the hardware and thus assume the heterogeneity is static and predictable. In contrast, multiprogramming leads to contention and variable performance as applications come and go.

4.1.2 Resource Management

Rinnegan promotes shared resources like accelerators as first-class entities through its resource management support in the kernel. The major responsibilities of the kernel layer in Rinnegan are resource allocation, enforcing

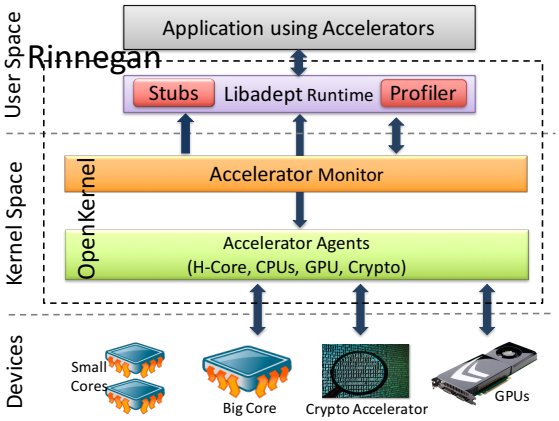

Figure 4.1: Architecture of Rinnegan.

isolation and exposing usage information about processing units. We call this kernel component the *OpenKernel* because of the transparency it provides to applications. The OpenKernel consists of *Accelerator Agents* which act as resource managers and the *Accelerator monitor*, which publishes usage information from agents.

Accelerator agents. In order to support a wide variety of devices with different characteristics, Rinnegan attaches an agent to each unique type of processing unit, such as CPUs, GPUs, and each type of accelerator. Agents perform two key tasks. First, agents act as the scheduler for a type of processing unit and enforce global policy about how much and when each process can use a processing unit. Second, agents gather utilization information about processing units.

Agents implement scheduling policies by controlling when tasks are dispatched to processing units; by delaying tasks from one application, a unit can be used by others. This allow agents to implement global scheduling policies. For example, agents can use shares to provide proportional-share scheduling that is consistent across all processing units, which can provide performance isolation and also guarantee performance for critical applications. Alternatively, shares can also be allocated separately for each

unit for finer-grained control. Maximum shares for an application is set by the administrator.

Agents gather and share usage information about the processing units they manage. The agent tracks information needed to predict performance, such as utilization by every application, average size of tasks, and number of applications. This information is provided by the agents to the accelerator monitor (described next) which publishes it to applications.

Accelerator monitor. The accelerator monitor is a kernel service that publishes usage information from various agents. Some information, such as total utilization of a processing unit, is published globally to all applications. Other information, such as an application's expected share of a unit, is specific to a process; the monitor calculates and shares this information separately with each process. The per-application information reflects the priority, share, or scheduling guarantees of an application. For example, processing units that have been reserved for exclusive use by one application will show up as busy for all other applications.

4.1.3 Task Placement

Rinnegan follows other runtimes for heterogeneous architectures [21, 42, 186, 197] and employs a task-based programming interface. The programmer or compiler generates task implementations for multiple processing units (not necessary for single-ISA systems). There are several runtimes [42, 197], research systems [65, 151] and initiatives like HSAIL [117] that allow code to be written once and then compiled into binaries optimized for different compute devices. For more varied architectures, such as fixed-function accelerators, a programmer may have to code multiple implementations.

The user-mode runtime layer of Rinnegan is contained in the *libadept* library and provides three key services. First, the runtime builds a performance model for every program task on each processing unit, which

allows it to predict performance. Second, it provides a simple interface for placing tasks on the best available processing unit. Finally, it allows applications to specify performance goals that guide the placement decision.

Performance model. The runtime builds a model to predict the performance of program tasks on different processing units. Rinnegan measures basic system performance, such as the overhead to dispatch a task and to copy data to processing units that lack coherent memory. For each application, the runtime measures the performance of an application's tasks on different processing units; this allows it to calculate processing time per unit data from the task execution time and the amount of data operated on. These measurements form a model of uncontended (stand alone) performance that can be used to calculate turnaround time of a task on any processing unit in the system, including dispatch overhead, data copying, and actual execution. The stand-alone performance from the model is later combined with utilization information from the accelerator monitor to predict the expected turnaround time even in the presence of contention.

Accelerator stubs. Rinnegan abstracts different processing units into a single procedural interface, so application developers reason about invoking a task but not *where* the task should run. Applications invoke a *stub* interface to launch a task. The overall launch process involves two stages: (a) choosing the right processing unit (b) dispatching task to the chosen unit. The stub considers a processing unit for task offload only if an implementation of the task for that processing unit is available. Stubs use information from the OpenKernel and the performance model from the profiler to predict the performance of the task on (runnable) processing units. The processing unit that yields the best performance is chosen based on these predictions, and the task is offloaded to that unit by the stub. The second stage deals with the actual dispatch of the task. The Rinnegan runtime contains dispatch and data transfer routines specific to an agent

(and thus specific to a processing unit). The stub handles transferring data to the unit if necessary (e.g., it lacks coherent memory and the data is not already at the unit) and then dispatching the task using these routines.

Rinnegan applications can specify performance goals to libadept, which are used by the stub in selecting the best placement for a task. For example, an application may specify maximum throughput, indicating it is willing to wait for an accelerator that provides a high speedup, while others may seek minimum latency and prefer a lower performing CPU that is available immediately.

Optimizer. For tasks that have specific performance targets, such as a desired long-term throughput, Rinnegan provides an *optimizer* that ensures these goals are met. The optimizer is part of the runtime and runs as a thread within the application. It monitors application performance by invoking a *tracker* function provided by the application, which returns the current performance of the application. If performance is below the desired level, the optimizer can either request more resources from the OS (e.g., increase priority or share), or invoke a callback registered by the application to reduce the amount of work (trade-off quality of the results for performance) so performance is acceptable. For example, a compression application may decrease compression to sustain a throughput when insufficient CPU is available. The optimizer can also detect that performance is above what is desired, and again either invoke the OS to release resources or notify the application that it can do more work.

4.2 Implementation

We implemented Rinnegan as an extension to the Linux 3.4.4 kernel. The code consists of accelerator agents for CPU and GPU, the accelerator monitor, and the libadept shared library linked to user-mode applications. The OpenKernel and the libadept runtime consists of around 2000 and 3400

lines of code respectively. In this section, we describe the implementation of Rinnegan, beginning with how agents schedule tasks and then discuss about user-mode task placement.

4.2.1 Accelerator Agents

We implemented accelerator agents for GPUs and single-ISA heterogeneous CPUs (with standard and fast cores). These agents enforce a scheduling policy by delaying requests, and pass processor usage information to the accelerator monitor. Table 4.2 shows the statistics generated by the agents. Rinnegan does not yet handle directly accessible accelerators, but disengaged schedulers [170] can be extended to implement the agent functionality for such devices.

GPU agent. The GPU agent manages any GPUs present. As GPU drivers are usually closed binaries that we cannot extend with agent functionality, we therefore implement the GPU agent functionality in a separate kernel-mode component that receives scheduling information (policy, priorities/shares) from an administrator and enforces scheduling policy. The libadept runtime invokes the GPU agent before offloading tasks, which can stall tasks to enforce the scheduling policy. To track device usage (e.g., task size), libadept passes task execution time to the agent after it completes.

The agent calculates GPU utilization by each application from the information passed by the runtime. As short-lived processes do not accumulate much utilization (e.g., *current utilization* is zero), agents also report the average task size on each GPU and the number of applications using the device. We implement three different policies in the agent: *FIFO*, *Priority-based* and *Share-based*.

The FIFO policy is the default policy in GPU drivers and exposes a simple FIFO queue for scheduling tasks from different applications. However, this policy cannot enforce performance isolation if tasks have

Agent	Details Published	Visibility
CPU	# active threads at each priority level	Global
	Schedule time ratio for each priority	Global
	# standard cores per application	Private
GPU	Current utilization by an application	Private
	Maximum share allowed for an application	Private
	Average task size	Global
	# active applications	Global
	Runqueue status of each priority queue	Global

Table 4.2: Details published by agents.

different execution times: a long task can monopolize the device because current GPU drivers only support context switching at kernel (i.e., task) granularity.

The *Priority-based* policy prevents monopolization by reordering requests to allow higher-priority applications to run before lower-priority ones. However, without preemption support that is not supported in current GPUs, low latency cannot be guaranteed if a long task is running.

The default policy in Rinnegan is *Share-based*. This policy provides soft performance isolation by guaranteeing execution time to each application. The share distribution is maintained by Rinnegan over coarse time scales on the order of hundreds of milliseconds. The coarse nature is due to the lack of preemption support in GPUs, wherein long-running tasks with low shares can temporarily exceed their share distribution. This policy provides strong performance isolation by guaranteeing execution time to each application. This policy defines three application classes: CUSTOM, NORMAL and BACKGROUND. In the CUSTOM class, designed for applications with strict performance requirements, applications reserve an absolute share of one or more GPUs, and the total shares on a device cannot exceed 100. Possession of 50 shares on a GPU guarantees the application a minimum of 50% time on the device. The shares unused by the CUSTOM class are given to applications in the NORMAL class, where the remaining shares are distributed proportionally among applications

according to the shares they possess. Finally, BACKGROUND applications use the GPUs only when applications of the other two classes are not using the device. To implement this policy, the agent monitors the utilization by each process, and throttles applications using more than their share. Thus, after executing a long task, an application will be blocked from running more tasks to let other applications use the GPU.

CPU agent. The CPU agent supports two classes of CPU cores, *standard* and *fast*, under the assumption that parallel code runs best by having as many cores as possible, while mostly sequential code prefers one or a few fast cores. Rather than providing a new scheduler, the agent uses Linux's native CFS scheduler [128], which already provides priorities and shares. The CPU agent publishes usage information including the number of threads and fraction of time given to threads at each priority level. This information allows the monitor to predict how much CPU time a thread will get on each class of cores. The share based policies (described above) for strong performance guarantees can be implemented by dynamically mapping the amount of shares to a `nice priority` value. This leverages the fact that every priority level has a time slice ratio over other levels.

Additionally, for standard cores, the agent provides hints on the number of standard cores a program can use exclusively, similar to scheduler activations. The calculation is based on a min-funding [286] policy: all cores are allocated to applications based on their priority or share, and unused cores are redistributed. For example, a single-threaded application can use only a single core and other cores it *could use* are instead given to parallel applications. This enables parallel applications to avoid time shares cores in most cases.

Accelerator monitor. The accelerator monitor aggregates information from all agents and publishes it to applications. While Linux already publishes CPU utilization information under `/proc`, we need a higher-performance mechanism given the frequency of access. The monitor shares

two pages of data with application using Rinnegan, which have read-only access to the pages. The *global data page* is mapped in all Rinnegan processes and has information about the whole system, such as the average task size on a GPU. The *private page* has process-specific information, such as its maximum allowed utilization (e.g., what fraction of the time it can expect) for each processing unit.

Agents invoke the monitor with the visibility of information, an identifier for the information, and a new value (e.g., <public, GPU average task size, 40 ms>). Agents calculate utilization information at periodic intervals of 10ms, which they immediately push to the monitor. All information calculated by the agents are moving averages. We currently do not synchronize access between the monitor and applications. However, the data is entirely scalar values used as scheduling hints, so races are benign.

4.2.2 libadept Adaptive Runtime

Every Rinnegan application links to libadept, which consists of the performance model, stubs, and the optimizer.

Performance model. The runtime builds and maintains a performance model for each of the application's tasks. The profiler predicts the task execution time by the amount of data processed by the task; we assume that task execution time is proportional to the amount of data processed, which is true for many applications. For applications where this is not the case, a more sophisticated performance model should be used; we experienced this with the *Sphyraena* workload (Table 5.5) that executes SQL select queries. The execution time is dependent on the selectivity of the operators which our model did not consider. Rinnegan also allow individual applications to provide their own performance model. During task dispatch, libadept invokes the application to get the predicted task performance on different processing units, which it then combines with information from the monitor. This allows the flexibility of using a custom

performance model for every application, such as more mature models [99, 138, 178].

During application startup, we execute an initial set of offloaded tasks from the application on all processing units, similar to Qilin [159]. This generates a set of parameters—the processing time per unit data—that are later used to predict execution time. The model is saved for use across program invocations. For short-lived applications, the developer can request that the model be generated incrementally over multiple invocations of the program, rather than as one step the first time the program runs. A program with a fewer number of tasks is considered short-lived and the profiling stage should not adversely impact such applications.

With this initial data, the performance model generates predictions for task execution time. libadept continuously monitors the accuracy of predictions; if it is off by more than 25% for more than 10 predictions, libadept recalibrates the model by running tasks on different processing units to generate a new set of measurement for the model. Rinnegan does not expect the model prediction to be 100% accurate and accommodates some error in prediction. The amount of error that can be handled depends on the speedup ratio between processing units. More details on the impact of errors in performance model is analyzed in Section 5.4.5.

libadept runs a generic profiling task during system startup to measure system performance, such as the latency and bandwidth of copying data to/from a GPU and latency of dispatching tasks. For processing units with coherent memory, such as CPUs, the performance model assumes no copy is required. The current implementation of libadept does not account for the contention in I/O bus. This assumption is approximated by a fixed overhead for transfers in addition to the per-byte cost.

Accelerator stubs. Rinnegan stubs are implemented as a single `Accelerate` function that can launch any task on any processing unit. Applications register a task by providing a set of implementations (one per process-

ing unit type). They then invoke `Accelerate` with the task object and its parameters. The `Accelerate` function invokes the performance model to determine where to run the task, moves data to the selected processing unit if necessary, and launches the task on the unit. While we use OpenCL [197] to generate implementations for CPU and GPU, they can also be hand written for better performance

`Accelerate` invokes the performance model to predict the task execution time, and then computes the expected turnaround time as follows:

$$\begin{aligned} \mathit{Latency} \;=\; & \mathit{Overhead} + \mathit{Data\ Copy} \;+\; (\mathit{Predicted} \\ & \mathit{Task\ Execution} \;*\; (100\ /\ \mathit{Utilization})) \end{aligned}$$

The term utilization comes from the agent and is the fraction of time the process can expect to receive on the processing unit. It captures both expected wait time, as wait time is inversely proportional to utilization, as well as its share.

The stub dispatches tasks to a GPU by informing the GPU agent about the task start, managing any data movement and finally invoking the OpenCL or CUDA runtime to submit the task to the kernel driver. For processing units with coherent memory, no movement is required. For those without coherence, such as discrete GPUs, stubs explicitly migrate data to and from the GPU's memory when necessary.

To dispatch tasks onto CPU cores, libadept starts worker threads affinitized to each core in each CPU class (e.g., standard and fast) during program startup. At runtime, it adds tasks to workqueues serviced by the desired worker threads.

The `Accelerate` function also takes a flag to indicate whether the task should run synchronously or asynchronously. Asynchronous tasks return an event handle that can be used to wait for their completion. The tasks are queued to an internal scheduler that maintains a central queue. Tasks are processed (the prediction process) in-order from the central queue and then pushed onto processing unit-specific task queues. The application

Name	System Configuration	Accelerators
asym_config	12 cores in 2 Intel Xeon X5650 chips, 12 GB RAM	2 fast cores–one per socket–at 62.5% duty cycle Remaining 10 cores at 37.5% duty cycle
simd_config	12 cores in 2 Intel Xeon X5650 chips, 12 GB RAM	*GPU-Bulky* (GPU-B): NVIDIA GeForce 670 with 2 GB GDDR5 RAM *GPU-Wimpy* (GPU-W): NVIDIA GeForce 650 with 512 MB GDDR5 RAM

Table 4.3: System configurations.

must enforce any data dependencies between tasks. Task execution may not happen immediately upon queuing to the central queue. The internal scheduler keeps the length of the processing unit specific queues low to maintain predictor accuracy by avoiding a long latency between selecting a unit and executing a task.

Optimizer. Rinnegan enables applications to target different performance goals, such as throughput guarantees or minimum execution time (best performance). The optimizer expects two inputs: the performance goal (e.g., 100 MB/Sec or 10 Frames/Sec) and, a tracker function that measures and returns current application performance. The optimizer, part of the libadept runtime, spawns a thread that periodically (every 500ms) invokes the tracker function. The tracker function is to be implemented by every application and it is registered as a callback with the runtime during application start-up. The function is responsible for returning the current application performance (same performance metric — throughput or latency — as that of the target performance) on invocation. Applications without a performance goal always seek maximum performance (minimum execution time) and the optimizer does not run in this case.

The optimizer operates in three phases when the tracker function indicates that an application is not meeting its goal. Each phase is carried out

only when the preceding phase fails to improve performance to meet the target. First, the optimizer tries to spread tasks across more processing units. Applications can thus offload multiple (asynchronous) tasks at once to different units. This is not the default behavior, because the libadept tries to minimize resource cost by ensuring performance on minimum resources. Second, for applications with appropriate privileges, it invokes the operating system to increase priority for desired processing units. This phase targets the application to be in CUSTOM class share-based scheduler, and is important if applications need to meet strict performance goals (e.g., throughput or soft real-time). If the second phase does not succeed, for example if the application lacks the privilege to increase its share or all the shares are already allocated, the optimizer notifies the application via a registered callback function. The application uses this callback function, which provides achieved performance, to modify its workload. For example, a game could reduce its frame rate or resolution. Upon availability of more resources (when other applications exit), the application automatically gets additional resources since the shares are proportional.

When an application performs above its goal or the optimizer observes that an application is not using its maximum utilization, it performs the same phases in reverse. This allows the application to modify its workload to increase its workload. e.g., frame rate, or to release resources to the OS by relinquishing its shares. Applications without a performance goal always seek maximum performance (minimum execution time) and the optimizer does not run in this case.

Listing 4.1: Program using libadept

```
        /*** Libadept ***/

void Accelerate(Task *task, bool sync) {
  /*
   * 1. Iterate through all processing units
   * 2. Calculate actual task latency on all units
   * 3. Assume punit offers minimum latency
   */
  Schedule(task, punit);
}
        /*** Application ***/

void TaskGpu(void *args) {/* Task logic on GPU */}
void TaskCPU(void *args) {/* Task logic on CPU */}

int main() {
    ...
  Task *task = IntializeAcceleratorTask
               (SIMD, TaskGPU, TaskCPU, args);
  Accelerate(task, ASYNC);
  WaitForTask(task);
    ...
}
```

Data movement. When launching tasks, the stub may need to migrate the task's data to the selected GPU if it lacks coherent access to memory. Moving a task back and forth between processing units can hurt due to excess data copies. This can occur when a task is at the break-even point between two processing units, so a small change in utilization can push the

task to switch processors. It can also occur when there are rapid changes in the utilization of a unit and predictions of performance are inaccurate.

Rinnegan implements mechanisms to avoid both causes of task migration. For tasks near the break-even point, Rinnegan dampens movement with a *speedup threshold*: tasks only migrate if the expected speedup is high enough to quickly amortize the cost of data movement. Thus, small performance improvements that require expensive copies are avoided. In addition, for subsequent task offloads, stubs make an aggregate decision that determines where to dispatch the next group of tasks, rather than making the dispatch decision for each task before switching to the new unit. In this *task aggregation*, group size grows with the data movement overhead.

Stability. Rapid changes in utilization can occur when multiple applications simultaneously make an offload decision: they may all run tasks on the same processing unit, leading to poor performance and then all migrate away from the unit. This *ping-pong problem* is common to distributed control systems, such as in routing [135]. Rinnegan takes a two-step approach to resolve the ping-pong problem. First, libadept detects when the predicted runtime is different than the actual runtime. This indicates that the utilization obtained from the monitor was wrong. Second, libadept temporarily uses the *actual* turnaround time of the last task instead of predicted turnaround time. Thus, the first applications to use the unit runs quickly and observe high performance, while those arriving later experience queuing delay from congestion and hence lower performance. Applications with higher delays will tend to choose a different processing unit, while those with less delay stay put.

Programmer effort. Rinnegan exposes new interfaces for programmers. We created a simple task-based programming model to help prototype a complete system from programming model to system software. Though writing a program using the new interfaces can be done without much

Applications	Task Size Ratio	Task Speedup Ratio	
		GPU-B	GPU-W
AES	**1**	32	22
LBM	3.33	1.4	0.5
DXT	5.6	16	4.5
lavaMD	20	27	7.5
Grep	33	10	3.3
Histogram	83	12	4

Table 4.4: GPU application characteristics. Speedups are relative to the CPU alone, and size is relative to *AES*.

effort as shown in Listing 4.1, it still requires program modifications for the new interface. To avoid this extra work, we integrated the libadept runtime with the StarPU [21] heterogeneous runtime system. libadept is used as an execution platform; we invoke `Accelerate` from StarPU's dispatch function. However, we did not need to make any changes to the StarPU's interfaces meant for application development. It already incorporates a profiling stage, that we use for Rinnegan' performance model. Thus programs already written for the StarPU model can leverage the Rinnegan's dynamic task placement without any code changes.

4.3 Evaluation

We address four major questions in our evaluation: (a) How beneficial is adaptation in a shared environment? (b) Can Rinnegan satisfy application-specific performance goals? (c) Can Rinnegan isolate applications from each other? (d) How well a decentralized scheduler performs? We also evaluate the overhead and accuracy of individual components of Rinnegan.

The experiments in this section focus on understanding the behavior of Rinnegan when applications of different characteristics (speedup, size of the task, long running/short lived) are run together. We evaluate the performance of Rinnegan against other systems such as StarPU and c-sched (a centralized system we wrote to be similar to an Oracle) and also with

Name	Description
Blackscholes [31]	Mathematical model for a financial market
Dedup [31]	Deduplication
Pbzip [214]	file compression
AES [3]	AES-128-ECB mode encryption, OpenCL
LBM [270]	Fluid dynamics simulation, OpenCL
DXT [194]	Image Compression, OpenCL
lavaMD [50]	Particle simulation, OpenCL
Grep [251]	Search for a string in a set of files, OpenCL and OMP for CPU
Histogram [251]	Finding the frequency of dictionary words in a list of files, OpenCL and OMP for CPU
Sphyraena [23]	Select queries on a sqlite database, CUDA
EncFS [232]	FUSE based encrypted file system, CUDA
Truecrack [280]	Password cracker, CUDA
x264 [165]	Video Encoder, OpenCL

Table 4.5: Workloads.

different scheduling policies. The primary goal is to show how Rinnegan handles resource contention, which can occur in real systems.

4.3.1 Experimental Methods

Table 4.3 lists the configurations used in our experiments. We disable Turbo Boost and hyper-threading to avoid performance variability.

Platform. We emulate an asymmetric CPU (*asym_config*) using Intel's clock-modulation feature [123] (in our platforms DVFS cannot be used for a single core) to slow down all cores but the *fast* cores. Similar mechanisms have been used in past research to emulate asymmetric processors [12]. We wanted a single infrastructure with different forms of heterogeneity (asymmetric cores and several accelerators). So, we chose to emulate the asymmetry rather than using a separate real hardware such as big.LITTLE processors [16]. Though none of our experiments make use of both forms of heterogeneity, we believe this infrastructure will serve best for the extension of this work. We emulate powerful cores by setting ten cores to run at

62.5% duty cycle and the remaining two at 37.5%. So, the speedup of fast core is 1.6x over slow core [16]. Where noted, we run some applications at 100% to emulate applications-specific speedups. The *simd_config* configuration comprises two GPUs of different performance and 12 CPU cores. Applications access these processing units through OpenCL, which uses the NVIDIA OpenCL SDK for GPUs and Intel OpenCL SDK for CPUs.

The GPU workloads are run simultaneously for all experiments performed on *simd_config*. We run workloads continuously, and present the relative throughput (tasks/second) for applications on the system under test compared to running all the applications on GPU-B. For a few experiments, those shown in Figure 4.7 and Figure 4.10, we allow some applications to finish in order to show how the system reconfigures. The application properties are shown in Table 4.4.

Workloads. We run the workloads listed in Table 4.5 in a multi-programmed environment to create contention for hardware resources. We select workloads with specific characteristics, such as tasks suitable for fast cores and varying task sizes to demonstrate Rinnegan's capabilities and to exercise different forms of heterogeneity.

Dedup, *Pbzip* and *Blackscholes* are parallel applications with tasks that can benefit from running on powerful CPU cores. We use six workloads for GPUs, described in Table 4.5. These workloads demonstrate the effect of task size and implementation style. For OpenCL programs, we compile to both GPU and CPU code. The same set of applications was also ported on to StarPU runtime. For CUDA programs, we use a separate implementation for the CPU. Table 4.4 shows the speedup for these applications when running on GPUs and their average task sizes (relative execution time). The speedup shown is relative to using all 12 CPUs at full performance. We report the average of five runs and variation was below 2% unless stated explicitly.

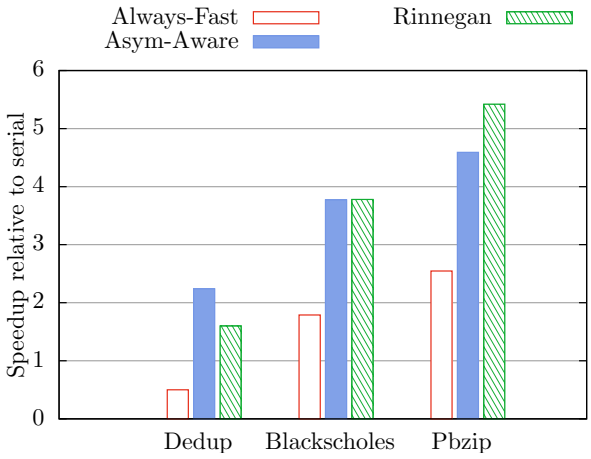

Figure 4.6: Contention for fast cores.

4.3.2 Adaptation

A major benefit of Rinnegan compared to existing heterogeneous runtimes is its ability to use application-specific performance models to select the best placement for a task, but at the same time dynamically adapting to runtime conditions. We evaluate how well applications adapt to a set of common contention scenarios.

Contention for fast cores. Rinnegan allows applications to accelerate important code regions on a small number of fast cores. We run multiple parallel applications concurrently that have tasks suitable for a fast CPU: the compression phase in *Dedup* and *Pbzip*, and the financial calculation in *Blackscholes*. The speedup of fast core over regular core is 1.6x. Using *asym_config*, we compare three configurations: (a) *Always-Fast* is a compile-time static policy that always runs all tasks on the fast cores, (b) *Asym-Aware* is similar to Global Task Scheduling (GTS [126]) in which it modifies the Linux scheduler to execute tasks on normal cores but migrate them to a powerful core if it becomes idle, and (c) *Rinnegan*, where the stub decides where to run tasks based on speedup achievable. To demonstrate the ability of Rinnegan to prioritize fast cores for applications with better

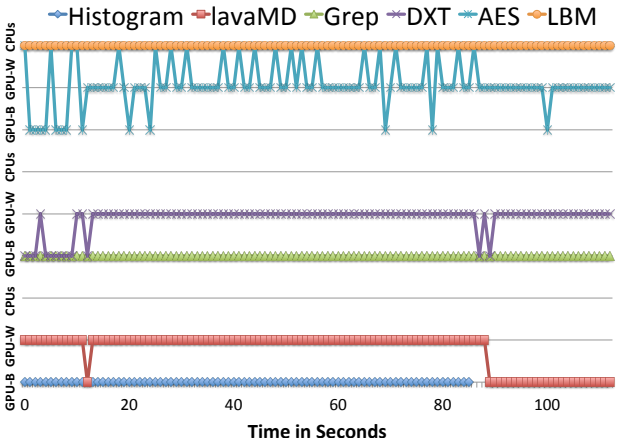

Figure 4.7: Rinnegan task placement with FIFO policy.

speedup, we made *Pbzip* receive a speedup of 2.65x on the faster cores. This speedup boost was applied for all three configurations. This was done by modifying the scheduler to increase the duty cycle to 100% only when *Pbzip* runs on the faster core.

Figure 4.6 shows performance normalized to serial versions of the applications running on a regular core. The overall performance of Rinnegan and *Asym-Aware* is comparable though both configurations perform differently on *Dedup* and *Pbzip*. Rinnegan runs the application with the best speedup, *Pbzip*, on the fast cores, which gives that application an 18% speedup compared to *Asym-Aware*. The *Asym-Aware* greedily schedules task on faster cores as they become available without any regard to their speedup. Rinnegan trades-off the performance of other applications for the high speedup application *Pbzip*. The *Always-Fast* performs poorly because it under-utilizes the regular CPU cores while waiting for the fast CPUs. Overall, the system throughput is 3% higher in Rinnegan than *Asym-Aware*.

Contention for accelerators. A key goal of Rinnegan is to manage contention for shared accelerators. In current systems, applications cannot

choose between processing units and thus must wait for the contended accelerator. Rinnegan, though, allows applications to use other processing units *if* they are faster than waiting. As a result, applications with large benefit from an accelerator tend to use it preferentially, while applications with lesser benefit will migrate their work to other processing units. Task size also plays a role with the FIFO policy: large tasks dominate the usage on bulky GPU. We ran all the OpenCL GPU workloads concurrently on *simd_config* using the FIFO policy.

In the *Native* system, all applications offload tasks to GPU-B, which leaves GPU-W and the CPUs idle. In contrast, Rinnegan achieves 1.8x better performance because it makes use of GPU-W and the CPU to run tasks as well. To explain these results, Figure 4.7 plots the processing unit used by each applications from a run where we launch all apps at the same time. *LBM* uses the CPU only exclusively because it gets low performance on the GPUs. In contrast, *Histogram* and *Grep* always uses GPU-B, because of their relatively larger tasks. This causes long delays for other applications, and hence *lavaMD* and *DXT* move to GPU-W. When *Histogram* completes, *lavaMD* switches to GPU-B and shares the device with *Grep*. On the other hand, *AES* switches execution between CPU and GPUs since the amount of data to encrypt varies with every task offloaded by the program. For smaller data, *AES* runs on the CPU to avoid costly data copies. The three column stacks labeled *FIFO* in Figure 4.11 shows the percentage of time each application gets on different processing units. The native stack at the left shows the default behavior where all tasks are offloaded to GPU-B alone.

Rinnegan as an execution engine. We integrated Rinnegan as the execution engine for StarPU, a runtime for heterogeneous architectures, to analyze the impact of contention awareness in the runtime. We rewrote all six GPU workloads to run on the StarPU runtime. We compare three different configurations for this experiment. We use two of StarPU's na-

tive policies [21, 268]: *DMDA* (deque model data aware) offloads task to the best performing processing unit taking into account the previously queued tasks from the local application and *Eager* employs a task stealing approach where worker threads of any processing unit can run a task as long as the implementation for that unit is available. The third configuration is StarPU with Rinnegan, where task offload decisions are made by Rinnegan based on input from OpenKernel.

The *DMDA* policy is representative of application runtimes [21, 223] for heterogeneous architectures where tasks are offloaded to the processing unit that performs best for that application (inclusive of all tasks from an application) in isolation. As a result, DMDA always offloads tasks to GPU-B for our workloads since it offers highest speedup. Since the task placement decisions consider only individual task performance but not contention at accelerators (due to other applications), DMDA's performance suffers by not utilizing the lower performing processing units such as GPU-W and CPUs when GPU-B is busy.

The eager policy, similar to runtimes such as Cilk [32] and others [241], employ a task stealing approach for task distribution in heterogeneous architectures. With this policy, the StarPU runtime for each application spawns worker threads for every compute unit in the system (12 workers for every CPU, 2 workers for each GPU). Every worker thread pulls task from the task pool when they are free without any regard to task speedup on that processing unit. As a result, when the GPU is temporarily busy, CPU worker threads pull many tasks that could have run at higher speedup on a GPU. Thus, the eager policy provides the lowest performance of the configurations.

We show the individual applications throughput and the overall system throughput relative to StarPU DMDA in Figure 4.8. Rinnegan is able to utilize the less-loaded processing units and thus provide better throughput than other policies. As a result, the overall system throughput

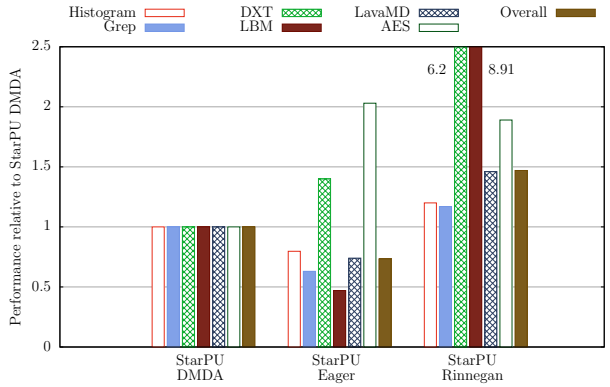

Figure 4.8: Throughput of different StarPU-based systems.

	Truecrack	**Sphyraena**	**EncFS (read)**	**x264**
GPU-B	320 W/S	38 Q/S	340 MB/S	15 F/S
GPU-W	230 W/S	13 Q/S	190 MB/S	-
CPU	15 W/S	0.2 Q/S	13 MB/S	-

Table 4.9: Stand-alone performance of CUDA workloads. W/S - Words per second; Q/S - Queries per second; F/S - Frames per second.

of StarPU+Rinnegan is 2x and 1.5x better than native policies of StarPU (Eager and DMDA) respectively.

4.3.3 Application-Specific Goals

The libadept library allows applications to set their own performance goals through an optimizer that guides offload decisions. We demonstrate Rinnegan's ability to support application-specific performance goals by running the CUDA applications—*x264*, *Sphyraena*, and *EncFS*—with their own goals in CUSTOM class and *Truecrack* in BACKGRND class. Native performance of application is shown in Table 4.9. The goals for each application are:

x264	15 Frames/Sec, can degrade quality for performance.
EncFS	275MB/Sec on sequential reads.
Sphyraena	30 Queries/Sec. minimum
Truecrack	Background: use GPU only when unused.

Of these applications, *x264* can adapt by reducing fidelity and *Sphyraena* uses asynchronous tasks and can spread its work across multiple processing units.

We started the applications in their respective classes where libadept assigns default shares to applications (15 shares on GPU-B each for applications in CUSTOM class and no shares assigned for BACKGRND class applications). We let libadept request shares automatically from agents after the initial assignment. We expect Rinnegan to automatically adapt—find the right set of processing units, or adjust shares, or callback to applications to alter configurations to adjust performance—without any manual intervention. The adaptation of the system is shown in form of a time graph in Figure 4.10. We start *x264*, *Sphyraena* and *Truecrack* at time zero, and *EncFS* at time 60. When applications finish, we do not restart them so as to release resources for other applications' use.

As the application runs, the optimizer detects the lag in performance and attempts to use multiple GPUs. *x264*, as a synchronous application, cannot leverage multiple GPUs. However, *Sphyraena* spreads its tasks across both GPU-B and GPU-W. *x264* receives around 80 shares on GPU-B and *Sphyraena* gets 20 on GPU-B and 100 shares (all) on GPU-W. Around 5th second, the runtime notifies applications to reduce fidelity to improve performance. *x264* adapts by reducing quality and is able to reach 13 FPS. Though *Sphyraena*'s goal was to achieve 30 Queries/Sec, it cannot achieve its goal due to the lack of resources in the system. The libadept notifies the application about this underperformance. Unlike *x264*, *Spyraena* cannot

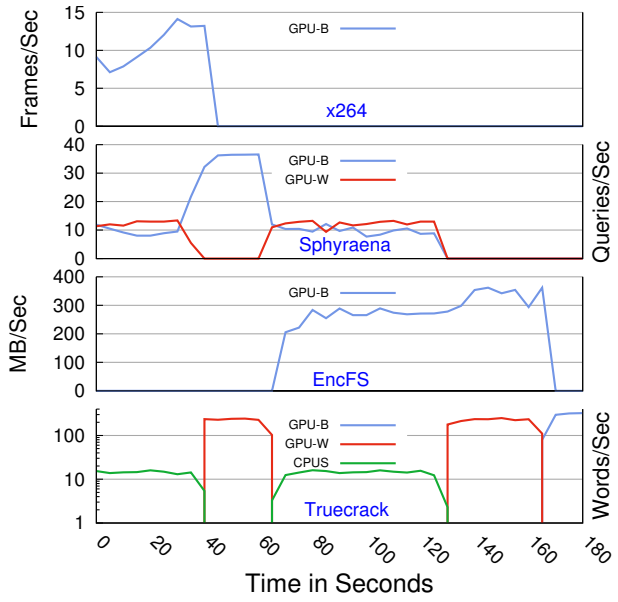

Figure 4.10: CUDA workload adaptation.

adapt its output to improve its performance. *Truecrack*, being a background application, runs on CPU since the accelerators (GPUs) are occupied.

When x264 completes, *Sphyraena* receives the whole GPU-B and achieves its goal of more than 30 queries/sec, and *Truecrack* adapts by moving to GPU-W, which greatly increases its throughput. At 60 seconds, we start *EncFS* and reduce the *Sphyraena* goal to 20 queries/sec. The optimizer adjusts the shares such that both applications achieve their goals by giving 85 shares of GPU-B to *EncFS*, and remaining of GPU-B and the whole GPU-W were given to *Sphyraena*. Truecrack is forced to move to the CPUs at this point, as both GPUs are totally saturated. Only when *Sphyraena* completes around 120 seconds does *Truecrack* move to GPU-W, and then to GPU-B when *EncFS* completes at 160 seconds. These results demonstrate how applications adapt to the changing use of the system, as well as how the optimizer allows applications to achieve their goals by spreading the work, increasing the share of a processing unit, or reducing

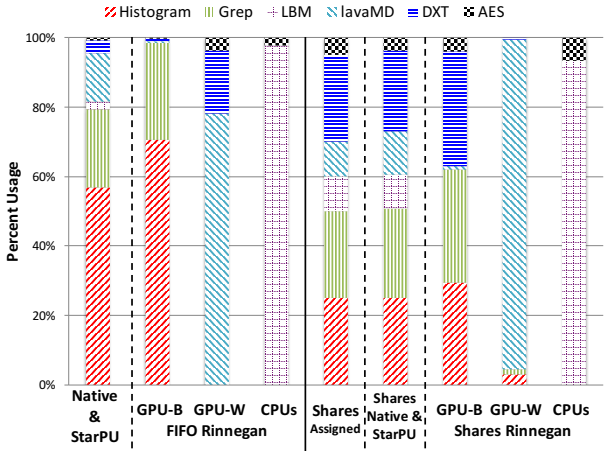

Figure 4.11: Percentage of time spent on different devices with various policies. the columns Native & StarPU and Shares Native & StarPU use only GPU-B. Shares Assigned column is the input.

the workload.

4.3.4 Preserving Isolation

Rinnegan only does placement in user-mode and leaves scheduling and policy enforcement in the kernel to protect against poorly behaved applications. To demonstrate Rinnegan's ability to isolate the performance of different applications, we allotted CUSTOM shares on GPU-B in the ratio of 25:25:25:10:10:5 to *Histogram*, *Grep*, *DXT*, *lavaMD*, *LBM* and *AES*. Such a share ratio was used to show that (a) applications with large tasks (*Histogram*) can be constrained, (b) small tasks (*LBM*) can receive guaranteed share, and (c) isolation is achieved even in the presence of varied task sizes.

Rinnegan applications. The three stacks on the right labeled *Shares Rinnegan* in Figure 4.11 show the portion of each processing unit used by each application. *Histogram* gets major portion of GPU-B because of its large task sizes in FIFO-based policy. With share-based scheduling, *Histogram*,

Grep and *DXT* evenly share GPU-B, while *lava-MD* uses GPU-W since it yields better performance than the guaranteed 10 shares on GPU-B. We observe that three applications (*Histogram*, *Grep* and *DXT*) receive more than their assigned 25% on GPU-B (30% each) because other applications decided to offload tasks on GPU-W or CPUs. So, these three active applications enjoy equal share on GPU-B. Also, applications with lower shares of GPU-B, such as *lavaMD* and *LBM*, offload tasks to GPU-W and CPUs respectively since the performance is better than on their 10% of GPU-B.

StarPU applications. We also show how Rinnegan can isolate applications that are not using Rinnegan by running StarPU version of our GPU workloads. However, we modified the workloads to notify the agents about their task execution time. The leftmost stack in Figure 4.11 shows the utilization received by each application with StarPU's native uses DMDA policy that runs all tasks on GPU-B. The *Shares native & starpu* column in the center-right shows the performance when share-based policy is enforced by the Rinnegan GPU agent: the resulting utilization almost exactly tracks the shares assigned. These results show both that native Linux and StarPU alone cannot or do not enforce performance isolation, while Rinnegan can provide isolation even for applications not actively using it for task placement. We note that without GPU driver support, Rinnegan relies on applications to call into the agent to know start and completion time of the task, and not all applications may do this. However, this functionality can be enforced either through GPU driver modifications or disengaged scheduling [170].

4.3.5 Decentralized vs. Centralized

Task placement in Rinnegan is decentralized: individual applications make placement decisions based on the state information exposed by the monitor. However, the information can be stale due to the delay between when it is generated and when applications place a task based on the

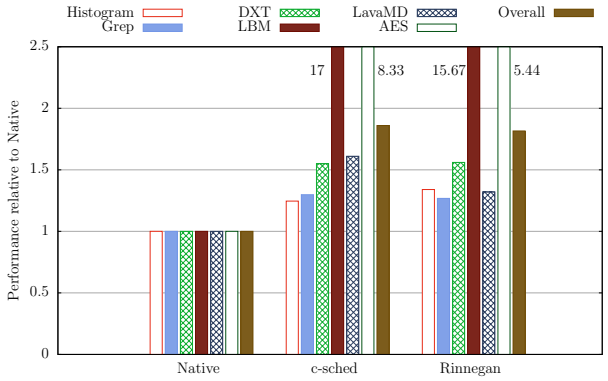

Figure 4.12: Centralized vs. Decentralized systems.

information. In contrast, task placement in a centralized architecture is performed by a single central entity for all applications. It has global knowledge of all tasks dispatched so far and can make accurate decisions for future tasks. We compare the behavior of Rinnegan against a centralized system to understand how much performance is lost due to a decentralized architecture.

Centralized scheduler. We built a simple centralized scheduler, *c-sched*, that behaves similarly to PTask [232] or Pegasus [101] but without the complexity and kernel modifications of those systems. libadept maintains task queues per processing unit for each application to track all dispatched tasks from that application. With c-sched, these queues are made global by implementing them on a shared page accessible by all applications, so the per-applications task queues becomes system-wide task queues. This enables c-sched to have a global view of the system and to predict the wait time for every processing unit accurately. Thus the c-sched scheduler is aware of tasks from all applications and can make perfect task placement decisions. However, it requires applications to be cooperative and can be easily gamed by providing erroneous speedup values.

Contention results. Using the same six GPU applications as used in pre-

vious experiments, we compare the individual applications and the total system throughput of c-sched against Rinnegan. We also consider a native system that offloads task to GPU-B, the highest speedup processing unit. The results, shown as the c-sched and Rinnegan bars in Figure 4.12, demonstrate that despite being decentralized Rinnegan still performed within 2.5% of a perfect centralized scheduler. Rinnegan performs close to the optimal scheduler is less prone to gaming and supports application-level adaptation, which is difficult in a centralized system. To achieve adaption, application-specific constraints and adaptation techniques would have to be conveyed to the centralized service, making it more complex.

Short-lived applications. Task placement for short-lived applications is hard without the global knowledge of the system since the utilization information can fluctuate. We generated such applications that start, run a small task for 0.5 – 2ms and then exit. We generated small tasks with varying inputs for *AES* and *DXT* workloads. We ran an experiment running multiple such applications periodically to test the overall throughput of the system in terms of tasks processed per second. The c-sched system should provide the best performance since it has a global knowledge of the system. The performance of Rinnegan is only 3.5% lower than c-sched whereas the native system that offloads all tasks to GPU-B is 20% slower than c-sched. Rinnegan performs well even without global knowledge of all workloads by using the average task size and average number of applications metrics rather than utilization information.

Varying mix of workloads. We want to compare Rinnegan on a more varied set of workloads. We created workloads by varying three different parameters: Task sizes (large and small), speedup(high and low), and longevity of applications (long running or short-lived). We generated eight synthetic workloads with manually generated performance model for different configurations. All eight workloads were ran and the system throughput was measured. We compared the results from c-sched, which

could see all tasks and select the best ones for each workload against Rinnegan. Overall, we found that Rinnegan performed similar to c-sched whereas the native system performed 40% lower than c-sched. Despite the lack of global knowledge, Rinnegan achieves performance equal to an omniscient global scheduler.

Performance. To evaluate the decentralized design, we compare Rinnegan against a centralized scheduler that has a complete awareness of all tasks in the system. We built a centralized scheduler (c-sched) that is aware of tasks from all applications and can make the perfect task placement decisions. For long running applications, like the set of workloads in the above case, Rinnegan achieved the same throughput as c-sched. For short-lived applications, Rinnegan performed 6% worse than c-sched since it does not track the arrival rate of tasks a processing unit. As a result, the utilization information is not precise.

4.3.6 Overhead and Accuracy

We separately measured the overhead of Rinnegan's mechanisms and the accuracy of its profiler.

Overheads. The primary overhead in Rinnegan comes from stubs, which must decide where to dispatch tasks. The overhead of stubs ranged between 1µs when choosing between fast and regular CPU cores to 2µs for selecting among different GPUs. Aggregation can reduce this by changing the dispatch decision less often.

Task profiler accuracy. We measure the difference between the task profiler's predicted run time and the actual task latency including the wait time. Across all our experiments, the prediction error is between 8–16%. The error came from two sources. First, Rinnegan predicts that task size is a linear function of input data size, which is not true for all applications (e.g., *Sphyraena*). Second, we found that the data copy latency to the GPU

varied due to contention for the PCIe bus.

To understand the importance of accurate performance predictions, we built an analysis tool to observe the impact of profiler error on task placement. For applications with 10x speedup, when the error rate increases from 5–100% the probability of making an incorrect decision increases by only 0.5%–5%. For the same range of error with applications getting only a 2x speedup, the error probability varies from 2.3%–25%. This shows that error in profiler prediction does not impact placement for tasks with better speedups.

Reducing data movement. Rinnegan reduces the amount of data movement by limiting the frequency with which tasks move between processing units with task aggregation and the speedup threshold. To measure the benefit of Rinnegan's mechanisms for limiting data movement, we disabled the mechanisms and randomly varied *LBM*'s maximum utilization for GPU-B between 65–75%. Its preference flips between GPU-B and the CPUs and performance suffers as a result. Compared to a system without the threshold and aggregation mechanisms, the speedup threshold improves performance by 5%, and task aggregation improves performance by an additional 60%.

Ping-pong problem. We also investigated the impact of the ping-pong problem by comparing Rinnegan against our c-sched centralized scheduler. Because it has perfect knowledge of the utilization of all processing units, c-sched does not have the ping-pong problem. For the contention experiments described in Section 4.3.2, we compared the task movements of Rinnegan to c-sched. We found that both systems had similar amounts of task movement. Because of varied task size, applications saw different processing unit utilization and hence made different scheduling decisions. To force a ping-pong problem, we ran five copies of the same *Grep* workload, which offloaded tasks of the same size. Rinnegan's ping-pong avoidance mechanism helped in stabilizing task placement sooner, resulting in the

same task throughput same as the c-sched. Without the mechanism, the system took 8-10 task offloads to stabilize, as compared to 2-3 with ping-pong prevention.

4.4 Conclusion

Heterogeneity in various forms will be prevalent in future architectures and the maturity in programming models is going to make it easier to program accelerators. Rinnegan exposes available heterogeneity with the OpenKernel, and uses libadept to conceal it from application programmers. The kernel retains control over resource management but provides application-level code with the flexibility to execute wherever is best for the application. The decentralized design employed by Rinnegan performs well and at the same time does not require extensive changes to the kernel.

5 Operating Systems for Power-Constrained Architectures

Due to the phenomena of Dark Silicon, the compute capacity of current processors is over-provisioned in that the compute units – CPUs, GPU – cannot be used at full performance without exceeding the power limit. The maximum performance of all compute units are capped to stay within the power limit. The major reason for enforcing power limits is to control heat dissipation. Regular cooling techniques such as heat sinks, fans or passive cooling in smaller form factor devices are not sufficient to dissipate the heat generated by processors when they run at full capacity. If power limit is not enforced, it can result in thermal meltdown (or overheating) resulting in damaging the processor substrate.

Current processors support mechanisms to prevent them from exceeding power limit (power consumption of the processor cannot exceed the limit) or thermal limit (the maximum processor chip temperature cannot exceed the limit). However these mechanisms can be used to implement a processor-wide (and thus system-wide) policy but cannot be used to provide application-level guarantees. Intel's Turbo Boost like techniques that are supported in processors are opportunistic and they increase core frequency only when extra thermal headroom is available but not based on application importance. Also, malicious or background applications

can increase the chip temperature and thereby throttling the entire processor along with all applications running in the system. Under such physical limits, an ideal system should multiplex power and thermal capacity among compute units – by turning on/off or increasing/decreasing p-state – based on the importance of applications (e.g. run GPU at higher performance than CPUs when GPU-bound program has more priority than CPU-bound programs).

In this chapter, we present *Firestorm* [207], an operating system extension that promotes power and thermal capacity as primary resources in the system. It operates with the goal of choosing the right processor configuration to meet the application goals within the physical limits. Firestorm abstracts power in terms of power tokens and effective power distribution is achieved by forcing applications to acquire tokens before they run. Token collection is dictated through a policy based on application priority. Thus, performance achievable is directly proportional to the amount of tokens collected by the application. Firestorm also models thermal usage of each application based on the compute unit(s) and the p-state at which they are used. The policy caps the tokens obtainable by background applications to limit their thermal usage. The capping is done to run the primary application without getting throttled. Firestorm thus introduces power and thermal awareness and enables the right set of applications to get more performance in such architectures.

We begin with a brief background on mechanisms available in current systems to enforce power and thermal limits, and then discuss the limitations of those mechanisms on why they are not sufficient to achieve process level guarantees in Section 5.1. We follow with the design of Firestorm in Section 5.2 and implementation details in Section 5.3. Finally, we present the evaluation results for Firestorm in Section 5.4.

5.1 Background and Motivation

5.1.1 Background

The power consumption of processors is usually dependent on the speed (frequency) at which it operates. Higher performance can be achieved by feeding more power to the compute units in the processor. However, the increase in power consumption results in increased heat dissipation.

Power controls. Power limits exist at a processor level to either control heat production or to limit system power to what is available from the power infrastructure (e.g., PDU capacity). Power limits can be enforced in two ways. First, by fixing the maximum frequency for the processor, a maximum bound on the power consumption can be established, although actual power can vary widely based on the workload. Second, by fixing the actual power consumption, processors can be run at any frequency as long as they stay within the power limit.

Most current processors support both mechanisms. In Intel processors, the RAPL mechanism runs cores at maximum core frequency possible and then throttles the frequency when power consumption exceeds a specified limit. Also, as their processors do not support per-core DVFS, the frequency of all cores will be reduced uniformly when throttled.

The DVFS (Dynamic Voltage Frequency Scaling) mechanism increases/decreases performance by altering the voltage and frequency level of the processor. Since the dynamic power consumption is proportional to square of voltage and frequency, power savings by reducing performance level is higher. Intel and AMD processors support different power planes for CPU and on-chip GPU, and thus the chip offers two voltage regulators for each of them. This is also why per-core DVFS is not possible, since a single regulator controls the voltage/frequency controls of all cores. On the other hand, duty cycle modulation (stopping the clock for short periods of 3µs at regular intervals) can operate at individual cores but it

does not change the voltage. As a result, the power savings achieved by duty cycle modulation is below DVFS for the same performance.

Thermal Controls. Thermal limits ensure protection of the processor from thermal breakdown due to overheating or to stay within the user comfort zone. Though power controls can control heat dissipation, a conservative power limit can result in reduced performance. Sequential applications prefer high frequency cores resulting in high power density. This can result in thermal hotspot although the processor is within its power limit.

Similar to the RAPL mechanism, processors throttle compute units when the chip temperature reaches the critical temperature (temperature beyond which processors can breakdown). Throttling can either be reducing the frequency or the duty cycle (stopping the clock for short periods of 3µs at regular intervals) of the cores or idle thread injection [224]. All compute units are throttled to get the chip back to safer thermal zone. Systems with variable fan speeds can step up fan speeds as they nears critical temperature. These policies are implemented either in BIOS or by the operating system. Processor stay in the throttled state (lower frequency or higher fan speed) for an extended period of time before moving to a normal state to avoid oscillations in and out of the throttling state.

5.1.2 Motivation

Though processors expose mechanisms to support power and thermal management, they are not sufficient or too coarse-grained to provide process-wide performance guarantees. Current mechanisms are only sufficient to enforce system-level power or thermal limits.

Heterogeneity in Demands

We see at least three reasons on why unequal power distribution among applications is important. First, applications may use power differently to

accomplish their goals. Sequential programs may want to power a single core as high as possible, while parallel programs may be faster when spreading power across as many cores as possible. Batch programs may run for long periods at a lower power level, while interactive applications do best with a high-power burst of activity.

Second, within a single system users may have different performance goals for programs. For example, on servers an administrator may want to dedicate power to latency-critical applications at the expense of background batch jobs [155]. On mobile systems, an interactive application may be prioritized for more power and performance than background applications.

Third, hardware itself is becoming heterogeneous, and may have a mix of CPU cores (e.g., ARM's big.LITTLE architecture with in-order and out-of-order cores) or both CPU and GPU cores as in AMD's APUs and Intel's recent processors. The power demand of each type of processing unit can be very different: applications may need more power to use a GPU or big CPU, but receive a super-proportional speedup by doing so. It should be noted that the power requirements of each compute unit varies.

However, the mechanisms supported in current processors for power distribution across different compute units are not sufficient as discussed below.

CPU-GPU Power Distribution. On processors with integrated GPUs, current hardware and software cannot adequately control power across both the GPU and CPU cores. For power management, Intel places them on different power planes thereby enabling individual voltage regulators, and hence frequencies, for each compute units. Through a machine-specific-register (MSR), software can configure the power distribution between CPU and GPU. However, the mechanism does not directly control the power consumption of the two planes: for the same MSR value, the power distribution ratio varies with the number of active CPU cores as well as

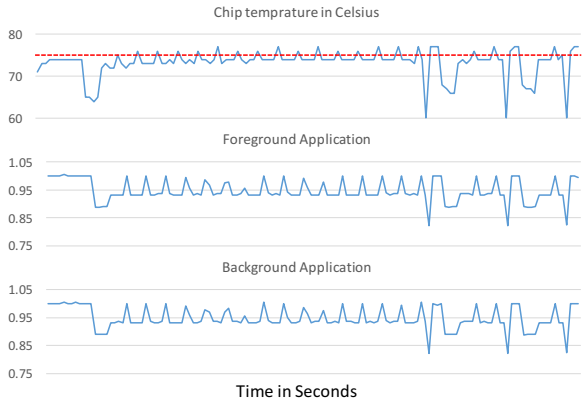

Figure 5.1: Thermal Interference.

the total power limit for the processor. Thus, actually controlling power use across both CPU and GPU requires modifying the power distribution (MSR input) when either of the impacting factor changes.

Turbo Boost. Current processors have mechanisms that automatically boost frequencies when there is power/temperature headroom, but this may not always improve performance. Processors from Intel [59], AMD [9] and Samsung [85] boost single-core frequency when neighboring cores are in idle state, indicating there is thermal headroom. However, prior work [156] has shown that such aggressive approach does not always equate to higher performance because not all applications (memory bound) benefit from high frequency. This opportunistic mechanism activates boosting whenever thermal headroom is available rather than using it when needed. So, applications that can potentially benefit from the additional frequency cannot leverage boosting technique if no thermal headroom is available.

Reactive Throttling

Current processors enforce thermal limits by throttling the entire processor when the critical temperature is reached. However, activity from

low-priority tasks on one core can cause throttling of high-priority tasks on others if the former raise the temperature too high [155]. Figure 5.1 shows a simple example plotting the performance of a primary and a background application over time when run together. It can be noted that the performance of both applications follow similar pattern of drops over time due to throttling. As the processor temperature reaches the critical temperature, the thermal daemon [277] reduces the frequency of the entire processor. The expected behavior is that primary application performance should not be impacted whereas the background application can be throttled. However, current systems only seek system-wide guarantees and hence throttle the entire system uniformly.

Low priority application can impact other applications even when it is not running alongside them by exhausting the thermal capacity. In cases where a user wants to run a high-priority task overclocked for a short period [226], a background task that already raised the processor to its thermal limit can prevent overclocking, as their is no thermal headroom to further increase frequency. If thermal capacity is treated as a resource by the OS, it can allocate the required thermal capacity to the primary application, so the background application will be forced to run at lower frequency.

Power and Energy

Power or energy or both are limited in many compute environments and this trend is going to continue in the future. Operating systems should begin to treat them as primary resources for efficient resource usage. Though these two resources are derivations of each other, knowing the differences in their characteristics can help us better understanding their applicability. The characteristics of these two resources are summarized in the Table 5.2.

Properties	Power	Energy
Resource Definition	Power is an instantaneous metric that determines the amount of resources that can be used at any instant in a system.	Energy is a measure of power consumption over time. It should be noted that energy is the same during high power usage for a short duration and low power usage over a long duration.
Resource Constraints	Power usage is limited by the thermal capacity of the system (to prevent it from over heating) or capacity of the power distribution unit.	Energy is limited in mobile devices due to limited battery capacity and energy reduction can greatly help in reducing operational cost in data centers.
Measurement	Intel processors support RAPL counters that reports power consumption of the entire chip based on performance counters. However, it does not expose the power information of individual cores. Power models can be built in software (similar to the one used internally by the processor) to predict the power draw of any compute unit based on performance counters.	RAPL counters also report energy consumption but not at the granularity of individual compute units. Energy models rely on power model to predict the power consumption of the device and on schedulers to track the runtime of the task.

Control Mechanisms	RAPL counters can be used to enforce a power limit for the entire chip. Also, features such as DVFS (frequency voltage scaling), duty cycle throttling and idle sleep states can be used limit the power draw of individual compute units	In addition to the power control mechanisms to limit the energy consumption, schedulers can also control the energy consumption by limiting the application execution.
Current Systems	Modern OS force idle CPU cores to deep sleep states to save on power consumption. It also supports configurable policies such as turning off hardware devices after some duration of inactivity for the same purpose. Current systems only support opportunistic solutions as the above but do not treat power or energy as a proper resource in the system.	
Limitations	It is hard to predict power consumption since power variation in compute units can be caused by different factors such as change in program characteristics and resource unavailability (cache misses).	Applications can deploy power in any form (single core with high frequency or multiple cores at low frequency) but still remain within energy guarantees. However, this can violate power limits or thermal limits or both.

Table 5.2: Properties of Power and Energy as Resources.

There have been past systems such as Cinder [235] and ECOsystem [299] that promote energy as a primary resource and thus enable important applications to receive more energy and can also guarantee a minimum lifetime (in terms of battery usage) for the overall system. Similarly, power based abstractions can enable systems to rightly decide the resources to use and the performance at which they should be used. This can enable over-provisioning of resources with respect to the power limit which is shown to be beneficial in data center environments [76, 289]. Over-provisioning can also be caused by physical constraints in processors [74, 274].

Energy based abstractions have the advantage of hiding any power draw variations in compute units (and other resources) during applications execution. Energy accounting is done periodically and program execution can be limited (or throttled) if the applications has exceeded its quota. Also, it is easier to capture energy consumption through RAPL like counters exposed in current processors. However, a separate power model is required if energy sensors are not exposed by the hardware resource.

The challenge is that energy abstraction is not suitable to capture power or thermal limits. These limits exist due to the limited capacity of the power distribution unit or to prevent the system from over-heating. The energy abstraction provides the flexibility for applications to deploy power in any form (high power for shorter time or low power for longer duration). It is possible for multiple applications to use high power and stay within energy limits but violate the power or thermal limits. In such cases, power based abstractions can help predict the power needs of all active applications and throttle individual compute units (through DVFS like techniques) if the overall consumption exceeds the power limit. The same abstraction can be used to understand the thermal needs of applications beforehand and thus prevent the system from over-heating.

Though energy abstraction has its own usefulness, we believe the basic

power abstraction is more suitable for a power-constrained environment that Firestorm targets. We also believe power and energy abstractions are not mutually exclusive but can complement each other.

5.2 Design

Firestorm is an extension to Linux that introduces power and thermal awareness in the operating system. The system associates performance requirements of applications with power and thermal requirements. Firestorm enables OS to manage them as primary resources in the system and thus allowing applications to allocate/reserve power and thermal capacity from the system.

Every application in the system receives shares based on their importance (Section 5.2.1). They get access to resources such as power and thermal capacity based on their shares (Section 5.2.2 and Section 5.2.3). Any request for computation goes through agents (Section 5.2.2), compute unit-specific resource managers, that controls performance based on the power and thermal capacity available to the application (based on its shares). The overall flow diagram of Firestorm is shown in figure 5.3.

The major design goals for Firestorm are:

1. *Power and Thermal capacity as resources.* Just as processors allocate memory and processor time, Firestorm should explicitly control allocation of power and thermal capacity to applications.

2. *Isolation.* Low-priority or malicious applications should not impact other applications due to lack of energy or thermal capacity.

3. *Performance guarantees.* Applications with strong guarantees, such as latency or soft real-time constraints should be guaranteed power thermal capacity to meet deadlines.

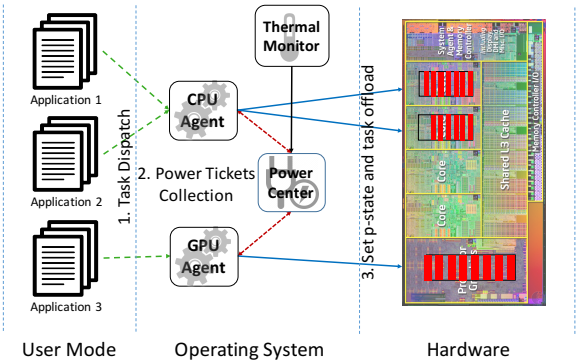

Figure 5.3: Control flow in Firestorm.

A quick note on terminology: we use the words application and program interchangeably, and the term processing units or compute units refers to units such as CPUs, GPUs, and accelerators. A task is a coarse-grained functional block such as a parallel region or function that executes to completion.

5.2.1 Application Classes

Inspired by the different scheduling classes in Linux, Firestorm divide applications into three different classes. The classification helps to understand the importance of the applications and thus how stringent their requirements are. The policy decisions — power distribution and preserving thermal capacity — are devised based on the applications classes as can be seen in later sections.

- *Soft Real-Time.* Applications with performance guarantees in terms of SLAs, either periodic or latency limits.
- *Best Effort.* Applications without guarantees that still desire the highest performance based on available resources.
- *Background.* Applications where performance is not a primary goal.

Soft real-time applications are guaranteed power and thermal headroom to meet their performance needs, but require admission control from the system to avoid overcommitting resources. Best-effort applications can fully utilize systems resources as long as soft real-time applications are ensured guaranteed performance. Background or low priority applications are typically not user facing (e.g,. data scrubbing) and do not have stringent performance requirements and should not interfere with applications in other classes. We follow the design principles of previous systems [155] where we give up on background performance to ensure guaranteed or high performance for applications belonging to other classes.

5.2.2 Power as a Resource

Firestorm focuses on power-limited architectures where all compute units cannot be used at maximum performance within the power budget. The system enforces control over the total power consumed by processing units as well as the power consumed by individual applications. To keep the system within power limits, Firestorm ensures that there is sufficient power *before* allowing an activity to proceed. Note that our focus is power and not energy, although energy-efficient computations are complementary to using Firestorm' mechanisms.

Abstraction. Firestorm abstracts the notion of power in the form of *power tickets* [235, 299] to quantify power as a resource in the system. A power ticket represents the ability to use power: for example, a ticket represents the ability to use one-hundredth of a watt. The difference in power consumption between neighboring frequency levels and duty cycle levels is less than one tenth of a watt for some frequency ranges. We therefore designed the power tickets to be fine grained (rather than a full watt) to capture these differences. The power limit of a processor or system is expressed as a limit on the total number of tickets in use. Thus, dynamically reducing the power limit reduces the number of tickets available. Tickets

are managed by one or more *power centers* (one per power socket) that acts as a power source(s) of the system and *agents* request tickets from power center on behalf of the application (discussed under mechanism).

Every power center in the system act as an independent power zone (e.g., one for each socket, one for off-chip GPU). A single centralized power center can become a bottleneck bogged, and having multiple power centers provides the ability to scale with multiple sockets. To use a compute unit, agents contact just the power center hosting unit without affecting other power centers. System-wide power limit can be enforced by ensuring the sum of tickets across all power centers is below the limit. Tickets can also be transferred between centers for long-term power shifting, similar to load balancing of threads across cores and sockets.

Mechanism. In Firestorm, power-consuming portions of the system are controlled by an *agent* that ensures power is available to use the component. An agent is a resource manager for a single type of compute unit and thus every compute unit (CPU, GPU or other accelerators) has its own agent. It also acts as a bridge between applications and compute units similar to a device driver. Applications offload tasks to agent and the agent is responsible for gathering sufficient power tickets from power center on behalf of the application before running the task on the compute unit. The responsibility of an agent is two-fold: (a) It gathers power tickets from the power center on behalf of the application and return the tickets back after task completion. (b) It calculates the number of tickets needed to run a task on the compute unit it manages; more efficient devices require fewer tickets than power-hungry devices, and computations that require less power (e.g., are memory bound) similarly require fewer tickets.

Firestorm employs a pay-before-use model, to ensure the performance of an application is proportional to the power received and also to stay within power limits. Long-running applications in Firestorm can not accumulate power tickets, and applications that are waiting or suspended

use no tickets.

Before executing a task, the agent consults its power model (explained in Section 5.3) to determine how many tickets are needed to execute at highest possible p-state (performance level), and requests those tickets. Based on the number of tickets received, the agent configures the hardware to limit its power draw to the amount allocated with help from the power model, such as by lowering frequency/voltage/duty cycle.

Policy. The initial number of power tickets in the power center is set by the administrator, and indicates the power limit for the system. Soft real-time applications reserve power tickets to ensure they have adequate power to execute. The remaining power tickets are shared by applications in best-effort and background class. Every application in the system is assigned shares, and the power center employs a proportional share policy for power distribution across applications. Firestorm also incorporates an additional admission control policy for background class to minimize interference. Background applications execute either when there are no active applications from other classes or a minimum of 50% of total tickets are totally unused (left after allocation to real time and best-effort class) and available in the power center.

Firestorm uses a proportional min-funding mechanism [286] to allocate excess power capacity. If a task requires fewer tickets than an application possesses, the power center re-allocates excess tickets to other applications that could use more power, proportionally to the share of each application. For example, assume application A has shares sufficient to gather 25 power tickets but wants to use a GPU that can consume only 15 tickets even at highest performance. Rather than under-utilizing the extra 10 shares, the power center reallocates those tickets to other applications proportional to the number of shares they have. Firestorm supports preemption of power tickets when application shares change or a new application (soft real-time or best effort) enters the system. The share allocation is automatically

readjusted by the proportional share policy.

5.2.3 Heat as a Resource

Firestorm takes initial steps toward introducing thermal awareness into operating systems. It avoids or minimizes thermal interference caused by low-priority applications and also allows applications to reserve thermal capacity to ensure guaranteed performance. The above are made possible by promoting thermal capacity as a primary resource in the system.

Abstraction. The thermal capacity of the system is generally defined as the amount of heat needed to raise the system's temperature by one degree [295]. However, to measure the thermal capacity requires knowing the material composition of the heat sink and also properties of the cooling devices (e.g., fan) used. To avoid this complexity, we instead abstract the thermal capacity of the system in terms of the processor chip temperature. The difference between the current chip temperature and the critical temperature is the available thermal capacity of the system. To make it usable, we build a model that predicts the amount of time required for the chip temperature to rise from a start temperature to an end temperature. The model is based on the amount of work done by the processor which is captured in terms of the power consumption of the compute units. More details on the thermal model are discussed later in Section 5.3.2.

Mechanism. Firestorm incorporates a thermal monitor whose goals are to avoid thermal interference and reserve thermal capacity. The first goal is achieved by adding a monitor service that periodically reads the processor temperature sensors to check whether the temperature is nearing the critical temperature. The monitor takes action to reduce temperature through throttling, which lowers the frequency or duty cycle of compute units. Throttling is achieved by notifying the power center to lower the number of power tickets issued to the agents on behalf of applications.

The second goal is achieved by exposing a set of interfaces for applications to reserve thermal capacity based on their workload demands.

Policy. The monitor offers two set of throttling policies each targeting different class of applications.

The objective of the *selective throttling* policy is to minimize thermal interference due to low priority applications and thereby trade off the performance of background applications for other applications classes. Firestorm uses selective throttling for best-effort and background applications classes. This policy employs a *reactive approach* in keeping within thermal limits similar to the Linux thermal daemon service [277]: the system only takes action when it reaches a temperature limit. In order to avoid interference caused by background applications, the policy employs a two-stage throttling mechanism based on two temperature limits — a lower background trip temperature and a higher best-effort trip temperature. The lower-stage throttles background applications when the chip temperature reaches the background trip temperature value. The higher-stage gets activated when the chip temperature exceeds the best effort trip temperature value, and throttles applications from both background and best-effort class.

The monitor differs from normal reactive approach by choosing which applications to throttle and thus trades off their performance for others. Throttling is done by reducing the power tickets given to the applications. The monitor conveys two set of information to the power center: (a) application class to be throttled (b) how much throttling as a percentage reduction in power tickets. The effectiveness of throttling depends on how high the chip temperature is compared to the trip values. A large difference demands more throttling and hence a higher reduction in power tickets. It should be noted that best-effort applications also get throttled if throttling background applications is not sufficient to keep within thermal limits. However, background applications get throttled more than

best-effort applications.

The *thermal conserve* policy is used for soft real time applications that require guaranteed performance. As discussed before, low priority applications can heat up the processor (exhaust thermal capacity) so much that subsequently scheduled applications cannot run at full performance without getting throttled (lack of thermal capacity). In other words, low-priority applications can affect soft real-time applications by making them miss deadlines/guarantees. For example, in data centers, latency-critical jobs and batch jobs are often scheduled in the same hardware for better utilization. These thermal problems have been shown to be possible in such cases [155]. To avoid this problem, Firestorm allows soft real time applications to reserve the required thermal capacity.

The policy requires knowing in advance the amount of work to be done by the soft real-time application to ensure sufficient thermal capacity is available. The work may be either a high-intensity task running quickly for a medium-intensity task for a longer period. The amount of work is captured by knowing how long an application will run for, and how much power it consumes while it is running, as the power is dispersed as heat. The former value can be predicted by the scheduler from past behavior or provided explicitly by the application as a periodic scheduling requirement. The latter value is obtained by predicting the power consumption of the application through the power model. With these numbers, a thermal model can compute the initial chip temperature that should be set for the application to run unthrottled. In other words, the thermal model computes the minimum thermal capacity needed for the application to run unthrottled. This temperature value is set as the best-effort trip temperature and the background trip temperature is also modified accordingly. Thus, the policy makes sure that even while background or best-effort applications are running, the reserved thermal capacity is available.

5.2.4 Support for Power Density

Sequential applications deploy all their gathered power on to a single CPU core to run at a high frequency. However, such increased power density can lead to the CPU core becoming a thermal hotspot. Current processors make sure that sufficient thermal headroom is available (other CPU cores are not dissipating any heat) before actually boosting the CPU frequency [9, 59, 85]. These techniques are complementary to the thermal conserve policy where thermal capacity is created rather than preserved. Firestorm includes a new execution object abstraction for sequential applications to create thermal headroom enabling software controlled turbo boost. The abstraction is in contrast to the current hardware mechanism (Section 5.1) where processors activate turbo boost when possible instead of when needed.

Abstraction. The execution object abstraction can be viewed as the combination of execution context with high power density along with required thermal capacity. Firestorm uses execution object to create thermal headroom for sequential applications where all CPUs constituting the execution object except the active CPU are treated as heat sink for the active CPU. The number of CPUs in the execution object is proportional to the thermal headroom required. Sequential applications need to request the kernel for an execution object with the required amount of thermal headroom during its start time. The abstraction is inspired from Chameleon [208] that uses the abstraction to represent an execution context formed from multiple CPU cores in dynamic processors.

Mechanism. The execution object supports two operations: activate and deactivate. Only after gathering sufficient power credits to run at high frequency can an execution object be activated. Activation involves creating thermal headroom and boosting the frequency of the active CPU. This is translated to forcing the constituent CPUs to idle state (except active CPU), increasing the frequency of the active CPU and allowing the sequential

application to run on the active CPU. This mechanism is compatible with current processors where Turbo Boost is automatically activated by the processor after execution object activation. Deactivation involves the reverse process where the constituent CPUs are no longer forced to idle state but allowed to run other threads.

Policy. The policy is responsible for making a decision on whether to activate an execution object or not when requested. Naively activating an execution object upon request will prevent other applications from executing since idle threads are forced on other CPUs. The policy strives to balance the requirements for sequential and other applications in the system. Firestorm introduces a configurable knob called *turbo_tax* and it allows the system to to favor sequential or parallel applications or even take a middle ground.

5.3 Implementation

We implemented Firestorm as an extension to the Linux-4.3.0 kernel. The code changes can be attributed to two major components.

- Power management, including power tickets, agents, power model, frequency balancer and the power sync.
- Thermal management that includes the thermal model and the thermal monitor service along with its policies.

Most code changes were made in the kernel by adding new functionality with a few minor changes to the Linux scheduler to incorporate CPU agents and operations on execution object (activation and deactivation). The total implementation efforts include around 2400 lines of code added to the kernel.

5.3.1 Power-Aware Scheduling

Firestorm's power-aware scheduling consists of the power center, which manages power tickets; agents that enforce power limits; and a power model for CPUs and GPUs to predict power consumption. We use Intel's clock-modulation feature (also called a duty-cycle mechanism) [123] on CPU cores as a means to reduce performance and thus reducing power, since DVFS cannot be set for each core in our system.

Power Model. We built a simple power model based on linear regression for individual CPU cores through offline analysis. This requires a one-time profiling stage when Firestorm runs for the first time. The profiling stage involves running the SPECCPU 2006 workload suite and measure the power consumption of every workload at different frequencies. We found that integer and floating-point instructions per cycle (IPC and FPC) have high correlation with the CPU power consumption in our test platform. We used the Intel energy performance counter registers to measure the power consumption of the chip. The current model does not support hyper-threading, which introduces additional modeling and control complexities and is left for future work. We built a linear regression model based on these data where the IPC, FPC, frequency and the duty cycle are input to the model and the model predicts the power consumption of the thread running on a CPU core.

The measured chip power includes cores as well as the LLC and other shared structures. To separate out the cost of using a core, we ran a single benchmark on one to four cores and measured the power consumption. We found that power increased linearly, indicating that the per-core cost is the delta in power draw when enabling an additional core. The computed power with zero cores constitutes the LLC and shared structure power.

The CPU agent measures IPC and FPC for every running thread and use the model to predict the maximum power needed to run the application thread. When a thread is context switched in, the agent contacts the power

center to request tickets for this maximum power. If the power center does not return the requested tickets, the agent again uses the power model, but this time to determine the highest duty cycle that can be used with the available power.

The GPU agent works similarly. Since most GPU drivers are closed source, we instrument applications to call the GPU agent before task launch to gather power tickets and after task completion to return the tickets. We are still in the process of building a regression based power model for on-chip GPUs. Instead, we run our workloads on the GPU at different frequencies and record the power consumption. The results populate a table for each application, which is later used during experiments to predict the power draw of the same GPU workloads.

Firestorm can operate with different power models for compute units as long as they expose the required interfaces for agents to use them. The list of interfaces to be supported by a power model is given in Table 5.4.

Frequency Balancer. Intel processors only support a single voltage domain for all CPU cores in a socket. Firestorm must determine an ideal processor frequency to ensure overall system throughput despite supporting heterogeneous performance demands from applications. We observe that most workloads perform better by reducing frequency rather than duty cycle: a thread running at 25% duty cycle at 3 GHz frequency consumes same power as one running at 100% duty cycle at 2.4 GHz frequency, but performs much worse

To avoid over-reliance on duty cycle, Firestorm does not run the processor at the highest frequency requested by an active application. Rather, the balancer of the power center tracks the power tickets given to active threads on all CPU cores. In case of parallel programs, we consider only the request of the primary thread (thread 0) of that program. The balancer identifies the maximum frequency at which each thread could have run with the obtained power tickets. We call this the ideal frequency of the

Type	Interfaces	Description
Power Model	`maxpower_for_task (task metrics, frequency)`	Given the task characteristics, return the maximum number of power tickets needed. CPU model in Firestorm takes IPC, FPC, frequency (current processor frequency) as inputs. GPU model takes SM (Streaming multiprocessor) utilization as task metric predict the power.
	`maximum_pstate (task metrics, power)`	The maximum performance state at which a given task can run with the given number of power tickets. For CPUs, Firestorm returns the duty cycle at which the CPU core can run at current processor frequency. For GPU, returns the frequency at which it can run.
	`maximum_frequency (task metrics, power)`	The maximum frequency at which the task can be run using the given power tickets at 100% duty cycle.
Thermal Model	`calculate_temperature (power, time)`	Returns the processor temperature at a particular time with the given amount of work (power).
	`critical_temperature_duration (current temperature, power)`	Returns the time taken to reach the critical temperature given the power consumed by the application.
	`thermal_capacity_needed (power, time)`	Returns the minimum thermal capacity (starting temperature) needed if the application runs for time interval 'time' without exceeding the critical temperature.

Table 5.4: Power and Thermal Model Interfaces.

thread. We aggregate the ideal frequency of all active threads to calculate the time-average ideal frequency for all running threads, and select that as the running frequency of the processor. A frequency balancing thread runs periodically every 1 second to recalculate the ideal frequency value. The long interval was picked to amortize the cost of voltage-frequency

switching, which is around 50ms.

Applications belonging to the background classes are not considered for these frequency calculation to avoid any interference with other applications. Conversely, if there are applications in the soft real time class, their its frequency request is chosen as the current frequency ignoring any requests from the best-effort class.

Errors in Power Model. In order to cope with the errors in the power model, the power center has a feature called power sync that recalibrates the ratio of power tickets to watts. Errors in power model can either underprice or overprice the power tickets required for a task. The former leads to using fewer power tickets than the actual power consumed, and the latter results in requiring more power tickets for should be needed.

Underpricing can thus cause the system to exceed the power limit even without spending all power tickets. Firestorm avoids this by leveraging the RAPL power limit register. In addition to limiting the total number of power tickets in the system, Firestorm configures the RAPL register to stay within the power budget. Even if power budget is exceeded without consuming all power tickets, the RAPL unit will avoid from exceeding the budget. Overpricing is handled by power sync. For processing units that report power usage, such as Intel Sandy Bridge and later CPUs, the power sync service measures the actual power usage and compares it against the number of tickets in use. If power consumption is below power tickets spent (overpriced), the power sync mechanism creates more power tickets to increase power consumption, and does nothing for the underpriced case since it is taken care by the RAPL budgeting mechanism. It should be noted that the power correction happens for the entire system and not per application.

Interactive services. Interactive threads run for a short period and contribute little to long-term power consumption. Furthermore, the require low latency. Firestorm therefore classifies some threads as interactive ser-

vices and does not require them to gather power tickets before execution nor run at a reduced duty cycle. All system threads are considered interactive and application threads whose execution time is below their sleep time (maintained by the kernel in their task structures) are also considered interactive.

5.3.2 Thermal Isolation

The thermal monitor service in Firestorm ensures that no single application overly consumes thermal capacity (i.e., overheats the system) and triggers throttling that hurts the performance of other applications.

Thermal Model. Similar to the power model, Firestorm incorporates a machine-specific thermal model that depends on the type of cooling available and its current state (e.g., current fan speed) in the system. We modeled the temperature trend using a logarithmic regression model where power consumption and time are inputs. It outputs the maximum temperature of the chip at that particular time.

We use a single profiling stage to measure the thermal performance of the system, which generates a model that can be saved and reused by every application. We assume that the state of the cooling device does not change over the system. A new model has to generated for every state (e.g., different fan speeds) of the cooling device. A model that encompasses all states of the cooling device is left as a future work. The CPU thermal model is generated by running multiple instances of a CPU-intensive workload that heavily uses all functional units. We run these instances at different power limits and record the chip temperature every second using on-chip thermal sensors. The regression model built over the data is

$$temperature = ((a * power) * log(time)) + (b * power)$$

where a and b are constants. The same model can be used for other applications since the power consumed by them captures the intensity

of work done by those applications. While tools like Hotspot [300] offer more accurate models, they require specific details about the processor intricacies, such as the material composition of the heat sink, that make it hard for end-user systems to deploy.

We built a GPU thermal model using similar techniques by running a GPU-intensive workload at different power limits. When multiple workloads run on various compute units (e.g., both CPU and GPU workloads) simultaneously, we observed that the chip temperature is dominated by the highest temperature of the two compute units when the same workloads were run individually. So, we use the maximum of the two model predictions as the chip temperature. Similar to the interfaces for power model, we designed a set of interfaces to be supported by any thermal model integrating with Firestorm, shown in Table 5.4.

Throttling. The thermal monitor service minimizes thermal interference by reducing performance of low priority applications. The monitor service identifies the class of applications to be throttled based on the trip temperature that is exceeded currently. The monitor informs the power center about the applications class to be throttled. The power center throttles an application by reducing the number of tickets given to an application (agent actually requests for tickets on behalf of the application), and the agent decides the optimal how to use those tickets for maximum performance. The extent of reduction is dependent on how long the chip temperature has been above the trip temperature.

5.3.3 Execution Objects

Execution objects supports sequential applications by creating thermal headroom for them to compensate for their increased power density. Our design leverages the Linux CFS scheduler [128], which records the virtual runtime for each thread and runs the thread with the lowest runtime next.

An execution object is formed from multiple CPUs where one of them is the active CPU that runs the sequential program and the others are used as heat sinks. In order to represent execution object as an execution context on all the CPUs, virtual threads are created to represent the execution object on all constituent CPUs. Only if all virtual threads manage to occupy the head position in the corresponding run queue, indicating they are next to run, can the activation of the execution object be carried out.

After every execution, the execution runtime, or virtual time spent executing, of the real program thread (on the active CPU) along with the runtime of the virtual threads are updated. If the runtime is not updated, then the virtual threads always gets to stay at the head of the queue. Conversely, if the runtime is updated by a huge value, it will rarely reach the head of the queue. We implement *turbo_tax* as a multiplier of the actual runtime for virtual threads: the runtime is updated as *program_runtime* * *turbo_tax* where program_runtime is the time an application ran before getting context switched out. If the tax value is less than 1, then sequential applications are charged less than the actual time used, which grants them priority ahead of regular threads from other programs. Hence, they get to use neighboring cores as heat sinks more. A tax value greater than 1 does the opposite and favors using cores to run regular threads.

5.4 Evaluation

We address the following questions in our evaluation: (a) How efficient and flexible is the power distribution infrastructure in Firestorm? (b) Can Firestorm achieve thermal isolation among applications? (c) How does Firestorm manage the power density requirement across applications? (c) What is the overhead and accuracy of individual components?

Name	Description
Truecrack [280]	Password cracker, single threaded
Histogram [251]	Finding the frequency of dictionary words in a list of files, OpenCL
x264 [165]	Video Encoder, parallel program
Pbzip [214]	File compression, parallel program
Spec 2006 [110]	Single threaded workload suite

Table 5.5: Workloads.

5.4.1 Experimental Methods

Platform. We use a Desktop class Intel Ivy Bridge processor i7-6700K with four cores and overclocking enabled. The TDP (Thermal Design Power) for this processor model set by Intel is 91W. It should be noted that the processor does not support per-core DVFS. We use overclocking to show how Firestorm can provide higher sequential performance without being limited by the Intel Turbo Boost mechanism. The native maximum frequency of the CPU is 4 GHz and 4.2 GHZ with Turbo Boost. The GPU has a maximum speed of 1.15 GHz. Overclocking extends the CPU to 4.5 GHz and the GPU to 1.5 GHz. The processor has a single socket and thus runs with a single power center.

We used the RAPL counters for power measurements, on-chip thermal sensors for temperature measurement, Intel's power governor tool [66] for enforcing different power limits on the processor, the Linux thermal daemon service [277] to stay below the critical temperature for the native Linux system. The critical temperature are set to 75°C for the native system. We set the trip temperature values in Firestorm to 70 and 75 for background and best-effort classes respectively, although the trip temperature can be changed dynamically by the thermal conserve policy. We also used the duty cycle mechanism to vary the power of individual cores. We report the average of five runs and variation was below 2% unless stated explicitly.

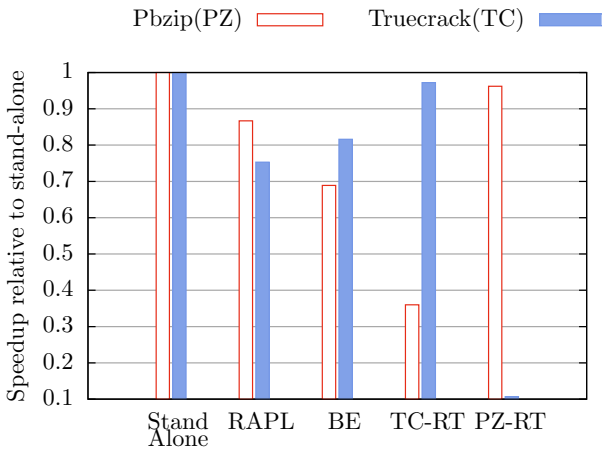

Figure 5.6: Ticket distribution for applications in soft real time class.

Workloads. We use the workloads listed in Table 5.5 for our experiments. The GPU workload are written in OpenCL and built using the Beignet runtime [28] that supports OpenCL interfaces for Intel on-chip GPUs. We measure performance for workloads as throughput: (a) *Truecrack*: words processed per second (b) *x264*: frames processed per second (c) *Pbzip*: data compressed per second (d) *Histogram*: files processed per second.

5.4.2 Power Management

We analyze whether Firestorm can distribute power to the right set of applications, to ensure guaranteed performance for applications and balance the performance of multiple applications. We set a lower power limit of 30 W (TDP limit is 91W) for all the experiments in this section. The lower limit help us understand the behavior of the system when applications contend for limited power tickets. The limit is enforced by limiting the total number of power tickets to 3000, including tickets used for the LLC.

Sequential and Parallel Applications. This experiment demonstrates Firestorm's ability to balance performance across applications based on

	Balancer Frequency	Truecrack (Desired)	Pbzip (Desired)
TC (standalone)	4.5	4.5	-
PZ (standalone)	3.83	-	3.83
RAPL	3.4	-	-
BE	3.7	4.5	2.9
TC-RT	4.5	4.5	-
PZ-RT	3.7 - 4	-	3.7 - 4

Table 5.7: Frequency balancer (Frequencies in GHz)

their power shares. We ran a single threaded *Truecrack* workload and a parallel *Pbzip* workload at the same time for all configurations (except for standalone) of this experiment. Every application thread runs on its own dedicated core (pbzip was limited to 3 threads). We consider five different configurations: (a) *Standalone*: Applications run standalone in the system within the power limit. (b) *RAPL*: Both applications run in native Linux with power limit enforced through the RAPL counter. (c) *BE*: Both applications are assigned to the best-effort application class with equal shares. (d) *TC_RT*: *Truecrack* is assigned to soft real-time class configured with fixed power tickets (e) *PZ_RT*: Pbzip assigned to the soft real-time class configured with fixed tickets.

The results are shown in the Figure 5.6 where the performance of applications is normalized to the standalone case. The *RAPL* and *BE* configurations are comparable since both represent the native configurations of Linux and Firestorm respectively. Under RAPL all cores run at same frequency, so the parallel program achieves higher performance since it uses multiple cores. The sequential program is also made to run at the same frequency (rather than a higher frequency) even though it uses fewer resources (a single core) and therefore performs worse than standalone. Firestorm balances this heterogeneous demand across applications by aggregating individual applications' desired frequency demand. The de-

sired frequency (the standalone frequency given the application's power tickets) and the balancer frequency that Firestorm arrives at are shown in the Table 5.7. Since both applications receive equal shares under *BE*, *Truecrack in BE* performs better than *Truecrack in RAPL* since it gets to use its power to increase the frequency. On the other hand, RAPL always favors parallel programs by choosing the highest optimal frequency for all cores.

Firestorm provides a RT (soft real time) class for applications demanding guaranteed performance. The applications in RT class receive a guaranteed number of power tickets. We ran separate experiments placing each application in separately the RT class for the last two configurations *TC_RT* and *PZ_RT*, while the remaining application placed as best-effort. *Truecrack* was reserved 1400 power tickets and *Pbzip* with 2400 power tickets. These are the number of power tickets required to achieve performance similar to standalone. The remaining power tickets are used by Firestorm for LLC and the other application. While we determined these shares through manual experimentation, systems that monitor application performance like Heartbeats [115] could be used to set the shares automatically.

In these two configurations, the RT applications achieve within 4% of native performance while companion application suffers. The small performance drop is due to error in power model and fluctuations in the application's characteristics (IPC, FPC) where the reserved tickets is not sufficient to run at full performance. For PZ_RT, the processor run at 3.7 GHz as chosen by the RT application *pbzip*. The power tickets available for *Truecrack* were only sufficient to run at or below 25% duty cycle and hence it performs poorly.

CPU-GPU Applications. This experiment demonstrate how Firestorm distributes power based on application shares across CPUs and GPU, and also its ability to redistribute unused tickets. We ran *Histogram*, a

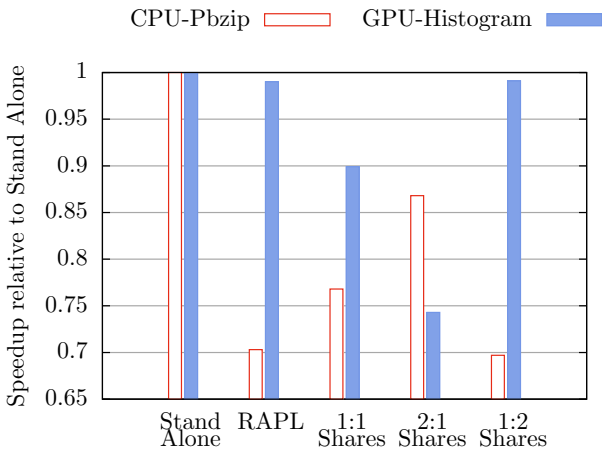

Figure 5.8: Power ticket distribution in the best effort class.

GPU application and *Pbzip*, a CPU application at the same time for all configurations (except for standalone). Both applications belong to the best effort class and so the power distribution is dictated by the proportional share policy. We consider five different configurations: (a) *Standalone*: Applications are run standalone in the system with the power limit. (b) *RAPL*: Native Linux with power limit enforced through RAPL counter and the default power ratio setting favors the GPU plane over the CPU power plane. (c) *1:1-Shares*: Both applications receive equal shares. (d) *2:1-Shares*: *Pbzip* receives twice the shares of *Histogram*. (e) *1:2-Shares*: Complement of the previous configuration.

The results are shown in the Figure 5.8 where the performance of applications are normalized to the standalone case. As with the previous set of experiments, The *RAPL* and *1:1-Shares* configurations are comparable. The default power ratio across power planes in Linux (*RAPL*) favors the GPU more than the CPU, and as a results *Pbzip* performance drops 29% while *Histogram* drops only 2%. In contrast, Firestorm explicitly allocates equal shares to both applications, which better balances their performance. As shown in the *1:1-Shares*, *Pbzip* drops only 27% and *Histogram* drops

Pbzip (CPU) Tickets	Histogram (GPU) Tickets	Input Ratio	Final Ratio
1736	869	2	1.997
1317	1317	1	1
1061	1579	0.5	0.671

Table 5.9: Ticket distribution (Ratio - CPU:GPU)

10%.

The performance of applications can be improved by assigning more shares with respect to other applications in the system. This is shown in the last two configurations *1:2-Shares* and *2:1-Shares*. The ticket distribution as per the policy is shown in table 5.9 in the column *Input ratio*, and the *Output ratio* shows the actual ratio used. The total number of tickets is below the 3000 available since some tickets are spent on the LLC. For *2:1-Shares* and *1:1-Shares*, the ticket ratio follows the input. However, in the case of *1:2-Shares*, *Pbzip* receives more tickets since unused tickets from *Histogram* are reassigned by the power center.

5.4.3 Thermal Isolation

We evaluate Firestorm's ability to avoid thermal interference among applications to guarantee performance for applications by preserving thermal capacity.

Avoiding Thermal Interference. Current systems do not offer thermal isolation in a shared environment. This experiment shows the ability of Firestorm to isolate applications from any thermal interference. We ran the same experiment as shown in Figure 5.1 where two instances of *Truecrack* are run at the same time. One instance is background class and the other is best-effort class. The system was provisioned with sufficient power tickets to run both applications at full performance. However, they cannot be run at full performance without exceeding the critical temperature (75°C).

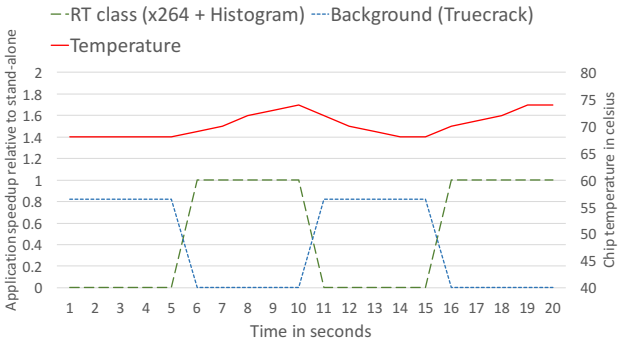

Figure 5.10: Thermal conservation: Temperature is plotted with right-side y-axis and speedup follows the left-side y-axis.

As shown in figure 5.1, native linux throttles all applications uniformly: the Linux thermal daemon safeguards the processor from exceeding the critical temperature by reducing temperature for all applications. This results in the performance of both best-effort and background applications dropped by up to 19%. When we perform the same experiment on Firestorm (not shown in the figure), it selectively reduces the performance of background application up to 39% by lowering its duty cycle, while the foreground application is continues to run at full performance (overclocked frequency of 4.5 GHz). The thermal monitor in Firestorm contacts the power center to reduce the power tickets given to background application thus reducing its performance.

Ensuring thermal capacity. This experiment shows the ability of Firestorm to reserve thermal capacity for RT class applications. *Histogram* and *x264* belong to the soft real time class demanding an SLA of 2 files/sec and 5 frames/sec respectively, and *Truecrack* is run as a background application. We interleave RT applications and background application such that each run for 5 seconds. Both RT applications *Histogram and x264* run for 5 seconds followed by *Truecrack* running for the next 5 seconds. We ran 50 such iterations for each configuration and compare the performance of the RT class applications with and without the thermal conserve policy.

Sufficient power credits are available in the system to run applications at maximum performance (overclocked frequency of 4.5 GHz).

Figure 5.10 captures a snapshot of the system for two iterations when the experiment was run using the thermal conserve policy. In native Linux with RAPL, both RT class applications failed to meet their SLAs for more than 50% of the iterations. This occurs because the thermal capacity was exhausted by background application during its 5 second run. The processor is hot when the RT application runs, and as a result gets throttled to a lower speed.

In contrast, with Firestorm's thermal conserve policy, the RT application achieve their SLA by running at standalone performance. This is possible since the chip does not overheat while the background application runs. Firestorm chooses a throttle limit for the background application based on the thermal requirements of *Histogram* and *x264*. The thermal conserve policy sets a trip temperature of 68°C for the background application and does not allow the background application to exceed that temperature. As a result, the performance of background dropped by 18% in order to satisfy the SLA guarantees of the RT applications.

5.4.4 Support for Power Density

In addition to distributing power across applications, Firestorm also allows applications to decide how to use power most effectively. Parallel applications spread power across more cores, while sequential applications use power to run a single core as fast as possible, leading to high power density. This can result in causing a thermal hotspot unless sufficient thermal capacity available.

This experiment demonstrates the ability of Firestorm to create thermal headroom for applications. We disabled over-clocking support in processor for this experiment and use native Turbo Boost. We want to demonstrate that by creating thermal headroom through execution objects,

the processor can use Turbo Boost to execute sequential code at a higher frequency. We use *Truecrack* as the sequential application and *Pbzip* as the parallel application on three cores. The maximum frequency is 4 GHz and the turbo frequency is 4.2 GHz. We ensure sufficient power tickets are available to run at full performance (all cores at 4 GHz).

The different configurations are: (a) *Standalone*: Applications are run standalone in the system. (b) *Native*: Native Linux with both applications running simultaneously. (c) *Tax**: The variants of the configuration are achieved by configuring the *turbo_tax* knob to different input values. It should be noted that lower values prefer sequential over the parallel applications.

The results are shown in the Figure 5.11 where the performance of applications are normalized to the standalone case. The maximum speedup achievable by the parallel application is 1 because turbo boost will not be activated when multiple cores are active. The *native* case does not show any performance improvement for the sequential application since all cores are active. Tax value of zero would always prioritize the sequential application over others and this would result in sequential application to get same as standalone performance but the parallel performance will be zero. The various *Tax** configurations — 50, 100, 150, 200 — prioritize sequential applications over parallel applications by 2:1, 1:1, 1:1.5, 1:2. The maximum speedup achieved by the sequential application is 5.78% over native. The *turbo_tax* balances the 5.78% of extra sequential performance against 100% of parallel performance. The various tax values represent the trade-off of opportunity provided by Firestorm: with a high tax rate, the system favors parallel programs as it is hard for *Truecrack* to activate an execution object, so it executes on a single core. With very low tax rates, the system favors running *Truecrack* at a higher frequency, which greatly diminishes *Pbzip* performance. Overall, these results demonstrate how the em turbo_tax configuration setting allows an administrator to balance

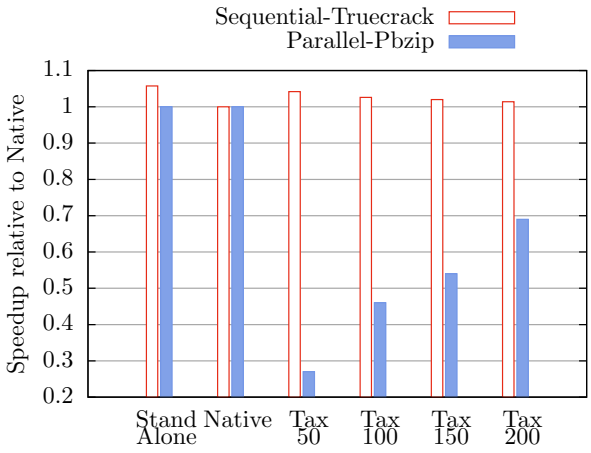

Figure 5.11: Provisioning for Thermal Headroom.

high sequential against high parallel performance.

5.4.5 Overhead and Accuracy

We measured the overhead of Firestorm's mechanisms and the accuracy of our performance and thermal models.

Overheads. The primary overhead in Firestorm comes from gathering power tickets from the power center during task dispatch. The overhead comes to 0.75µs for both the CPU and the GPU agents while collecting tickets and 0.4µs while giving it back. It should be noted that the context switch time without the power center is 2µs. Since the CPU agent hooks into the scheduler, the overhead from CPU agent adds upto the context switching time.

Power model accuracy. We measured the power model accuracy by comparing its predicted power consumption for a task against the actual power derived from the RAPL performance counters. We found that the error was close to 16%, largely due to our simplistic model using only IPC and FPC to predict the power consumption.

Thermal model accuracy. We measured the thermal model accuracy by comparing the (temperature) predictions of the model against the actual chip temperature (as measured using on-chip thermal sensors). We ran real workloads under different power limits and comparing the temperatures at different times. The model prediction had an error rate close to 12%.

Model interfaces latency. Bulk of the work to generate the model is done during the system startup time. The interaction of the agent with the model during the system runtime through its interfaces does not cost much. The latency of every interface call is around 0.05µs.

6 Related Work

In this chapter, we discuss the works that are related and also those that inspired the systems presented in this dissertation. First, we talk about the research projects and existing systems similar to the problem targeted by Chameleon in Section 6.1. Then, we describe the scheduling related works in the context of heterogeneous architectures related to our work in Rinnegan in Section 6.2. Finally, we describe the power and thermal management based works related to Firestorm in 6.3.

6.1 Support for Dynamic Processors

Though Chameleon was one of the early systems built to support dynamic processors such as Core Fusion [125], the individual components of Chameleon shares many similarities with previous works.

6.1.1 Asymmetric Scheduling

Several works investigate scheduling parallel and multiprogrammed workloads on asymmetric or heterogeneous processors [104, 141, 150, 237, 238]. Scheduling on such hardware begins with understanding the program characteristics in order to match with the right processor core. These systems include a profiling tool that periodically reads the performance counters to gauge the program characteristics. Many of these systems focus on power efficiency, so they seek to identify which threads gain

the most efficiency. Though Chameleon in its current state relies on administrator or the programmer to provide as input the program type, Chameleon could use these techniques to identify the type of execution most suitable for a given program.

Mogul et al. investigate the use of ACMPs for operating systems [179] by running the OS code on less powerful cores for better energy efficiency. This was achieved by migrating threads to the less powerful core upon entering the kernel mode through system calls. Similar schedule matches based on hardware and software characteristics can also be achieved in Chameleon using its property mechanism, which is similar to the matchmaking [227] support in Condor system in terms of functionality.

Most similar to Chameleon is Luo et al.'s work on the use of helper threads and cache resizing on an ACMP [160]. The work analyzes the use of thread speculation for workloads with different configurations of an ACMP hardware. Similar to Chameleon, this work determines when to allocate resources to speed sequential threads, but focuses again on identifying which threads gain the most benefit. The work is orthogonal to Chameleon, where the property of an execution object can be altered based on the benefits achieved by the workloads.

ACS accelerates critical sections on an ACMP [272]. This acceleration is achieved by migrating the execution of the critical section onto the powerful core. A new instruction is introduced in the ISA, which when invoked transfers the control of the critical section execution to the larger core. The idea is that by completing the critical section faster, the impact on serialization or time spent waiting for the critical section can be reduced and thus improve the overall application performance. When compared to Chameleon, our system relies on core reconfiguration to form a powerful core. The latency of such reconfiguration is still too long to help individual critical sections. Similar work by Wamhoff et al. [288] explores the use of Turbo Boost capability to increased the performance of critical sections.

Chameleon can perform similar operations while activating the execution objects.

6.1.2 Gang Scheduling

Chameleon's cluster scheduling is similar to gang scheduling in that it will try to schedule multiple CPUs simultaneously. Although it should be noted that the cluster scheduling is more flexible in its requirement compared to gang scheduling where Chameleon can still activate an execution object even if all CPU cores are not available for reconfiguration. Past work on gang scheduling [19, 30, 77, 127, 201] focuses primarily on time-sharing gangs of threads. While Chameleon can do this, there is little benefit because performance can still be improved by running time-sliced threads in parallel on separate cores. Thus, Chameleon is most effective when there are idle threads to be used opportunistically, or when a sequential thread has higher priority than competing workloads.

Though gang scheduling has shown to be beneficial for some form of parallel programs [78], it has not been supported in current operating systems since applications in general do not benefit much from it [217]. However, Chameleon's execution object abstraction and the cluster scheduling framework are widely applicable in processors that support resource aggregation [59, 116, 125, 136] and current processors exhibit such a property.

Gang scheduling is supported in research systems such as Barrelfish [27, 218] and Tesselation [154] that are operating systems built for future many core architectures. Tesselation uses a form of gang scheduling to provide a *cell* of processors to an application, which is similar to Chameleon's execution objects [154]. However, it provides cores for software process to use, while Chameleon provides cores to hardware for an enhanced CPU.

6.1.3 Support for Reconfigurable Hardware

Recent work on scheduling for reconfigurable hardware has largely focused on embedded and real-time systems [86, 140, 257]. In these environments, precise models of the transition costs and the execution time of code on different hardware are needed. These systems also place mandatory requirements on scheduling, so flexible trade-offs like Chameleon's taxation are not used. In contrast, Chameleon focuses largely on best-effort workloads and must rely on admission control to meet performance goals.

Several projects discuss OS support for introducing reconfigurable logic onto a processor [62, 158, 269]. However, OS support for these systems focuses on efficiently allocating the reconfigurable logic to specific functions rather than on thread scheduling.

Windows 7's support for core parking [176], which coalesces threads onto a single core to disable the remaining cores, is similar to Chameleon's scheduling of threads on the execution object. It is also used to balance threads between hyperthreads. However, core parking targets all threads at a specific subset of CPUs, rather than context switching between configurations.

6.1.4 Faster Reconfiguration in Operating Systems

Processor reconfiguration is much simpler through virtualization techniques [24, 137, 283] by migrating all virtual CPUs running on that physical CPU core to a different physical CPU core. The process is similar to thread migration in OS and the latency of reconfiguration is really low. However, the downside of virtualization is that it causes significant overhead during the normal execution of the virtual machines due to an additional software layer.

NVIDIA Tegra processors [193] employ an ACMP architecture by provisioning faster cores for higher performance and a companion core for

better energy efficiency [193]. The processor uses a hardware virtualization technique to switch to the companion core when most of the system is in idle state to conserve battery. Similarly, hardware virtualization techniques have been proposed to provide reliable execution in the presence of faulty processor cores [293]. The challenge is the semantic gap that exists between the hardware and the software layer where the hardware is oblivious to application properties or OS goals. However, Chameleon makes scheduling decisions based on application properties and the policies set by the administrator.

Recent work as part of the Barrelfish OS was inspired from Chameleon in providing support for a faster reconfiguration mechanism to support hardware reconfiguration and fast kernel updates. The system employed a solution similar to virtualization where it can migrate the CPU core state to be run on a different physical CPU but without the additional overhead of virtualization. The low overhead CPU virtualization is achieved by re-architecting the OS from scratch. However, Chameleon targets support for faster reconfiguration in current operating systems.

The system Bolt [211] was also built by us with the same goal for reasons of better performance and energy efficiency. Bolt improved on the hotplug infrastructure by classifying the hotplug operations as critical and non-critical where the latter operations can be moved out of the critical path and completed at a later point in time asynchronously. Though Chameleon's proxies were designed for fast reconfiguration, they only virtualize external interfaces like interrupts. They do not migrate per-core states like threads necessary for ensuring progress and thus making them ideal only for shorter time scales.

6.2 Resource Management in Heterogeneous Architectures

There has been a lot of research in designing runtimes and operating systems to perform task management in heterogeneous architectures. We compare Rinnegan to those systems in this section.

6.2.1 Runtimes for Heterogeneous Architectures

Many systems provide runtime layers to aid applications in using heterogeneous units (e.g., StarPU [21], OmpSS [72, 221–223], Merge [151], Harmony [65], Lithe [206]). These runtimes abstract the presence of different processing units present in the system. Given a task, they are responsible for dispatching it to the right processing unit and thus yield better performance for the application. Rinnegan shares the similarity of automatically selecting where to run a task. Unlike these runtimes, Rinnegan supports multi-programmed systems where there may be contention for processing units. In addition, Rinnegan exposes system-wide resource usage to applications to enable them to self-select processing units. Task placement decisions are thus made with awareness to other applications in the system.

There are works [99] that focus on partitioning tasks across different heterogeneous compute units in the system. Better performance is achieved by leveraging all compute units in the system. However, the challenge is that the hardware characteristics of the compute units differ and tasks run at different performance. So, a given task is partitioned into subtasks and distributed across compute units based on the performance they have to offer. Though, Rinnegan does not perform such task partitioning, these works are orthogonal to Rinnegan and can be integrated.

Runtimes for databases help in task scheduling on heterogeneous architectures [38, 108]. Similar to the task partitioning, the query operations

in a query plan are distributed across different compute units or co-located on the same compute unit to avoid data copy. However, they assume the system resources are not shared with other applications. Query operations once mapped to a compute unit are not reassigned to other units based on contention. Though Rinnegan makes task placement decision at runtime, it relies on the programmer to enforce task dependency.

Other systems abstract the presence of heterogeneous processing units. For example, PTask [232] provide an abstraction for GPU programming that aids in scheduling and data movement. However, it does not schedule tasks across different type of compute units such as CPU and GPU. Rinnegan differs by exposing multiple types of processing units, not just GPUs, and by exposing usage to applications to self-select a place to execute. In contrast, PTask manages allocation and scheduling in the kernel.

A few runtimes [223, 241] employ task-stealing for better utilization of all compute units in the system. However, these runtimes perform stealing without regard to the speedup of the stealer destination over the tasks's current placement nor contention at compute units. As a result, the stolen task might perform worse at the destination compute unit. Rinnegan employs a task steering approach in contrast to the stealing approach where tasks are dispatched to an appropriate processing unit based on its standalone performance and also the current load at the processing unit.

The Taurus distributed runtime [162] is built on top of JVM (Java Virtual Machine) based managed runtime. The runtime is responsible for coordinating garbage collection across multiple processes running over different machines. Though the runtime does not target heterogeneous architectures, the principle of a centralized distributed runtime is an interesting design to be applied for these architectures. The design gives a global view of all the applications in the system and enables to make an optimal task placement decision. However, it relies on cooperation from all applications for normal behavior and also it is hard to achieve

application adaptation in such designs.

6.2.2 Resource Scheduling

Many past systems provide resource management for heterogeneous systems (e.g., PTask[232], Barrelfish[27], Helios[189], Pegasus [101], TimeGraph [133]). These systems consider all compute units including accelerators as primary resources in the system. In contrast to these systems, where the OS transparently handles scheduling and resource allocation, Rinnegan cooperatively involves the application. The primary goal of Rinnegan is to enable adaptation in applications. It exposes processing unit utilization via the accelerator monitor to allow applications to predict where to run their code, instead of making the choice for the application.

Other systems provide contention-aware scheduling (Grand central dispatch[88], Scheduler activations[10]), where applications or the system can adjust the degree of parallelism. Rinnegan enables a similar outcome, but over longer time scales. Unlike scheduler activations [10], Rinnegan does not notify applications during change in resource allocation, but allows applications to monitor their utilization.

There were many systems designed to focus on task scheduling in heterogeneous processors. Octopus-Man [220] provides a task scheduling framework by leveraging ARM big.LITTLE processors [16] to provide QoS guarantees for a critical application and also energy reduction. Rinnegan can provide similar guarantees for applications in soft real time class. However, the Octopus-Man system does not model performance of all applications but only that of the single primary application. As a result, it does not support any scheduling policies to arbitrate resource usage across multiple applications. Moreover, the system is designed for a data center environment and assumes application cooperation. It does not perform well in the presence of malicious applications. The work on power management [263] based on price theory is similar to Octopus-

Man with the same goal of application guarantees and energy reduction. However, the system does not support adaptation in applications. The Global Task Scheduling (GTS) [126] aims to schedule tasks based on their demand for compute units. Unlike Rinnegan, the scheduling decision in GTS is based on the runtime of the threads between context switches but this does not consider the application needs.

Baymax [51] is similar to Rinnegan where it offers QoS guarantees for critical applications in a shared heterogeneous environment. Unlike Rinnegan, it employs a centralized architecture where tasks from all applications are sent to a central service that makes policy decisions by reordering tasks to ensure that the critical application tasks are not delayed by other applications. However, it is hard to support application adaptation using a centralized design. Also, Baymax does not make a decision on where to offload tasks but only when to offload tasks. It does not dispatch tasks to alternate processing units but only reorders them on their way to the GPU accelerator. Finally, it assumes cooperation from applications whereas malicious applications can offload to accelerators without dispatching through the centralized service and affect the primary application.

6.2.3 Adaptive Systems

Rinnegan is inspired by many previous works in the area of adaptive systems. These systems try to identify resources as available to applications and then tune applications accordingly for better performance or towards other application goals. Odyssey [191] was built as an adaptation platform for mobile systems. The operating system exposes resource availability to applications which then trade-off output fidelity for performance. Rinnegan employs a similar architecture for shared heterogeneous systems and uses alternate processing units as a mode of adaptation.

Application heartbeats [115] tracks individual application performance

and it either requests more resources to reach the desired performance or adapts configuration parameters to throttle application performance to stay within power/energy limits. Rinnegan employs similar adaptation techniques for heterogeneous architectures with the help of the system kernel. In addition, the support for system level policies in Rinnegan can help to enforce resource limits on multiple applications sharing the same system.

Similar adaptive techniques have been employed in databases [91] where new interfaces between OS and databases are designed for better performance. Varuna [266] tackles a similar problem but targets only parallel programs. Performance is improved by dynamically adapting the parallelism in the applications based on the runtime conditions. However, the system runs strictly at user level and does not offer isolation across applications.

6.2.4 Multi-Level Scheduling

Applications are well aware of their resource needs and can receive higher benefits from application specific resource management rather than a system-wide resource management policy (For example, LRU (Least Recently Used) based cache replacement policy might not be suitable for all applications [18]). Many systems [27, 73, 113, 154, 219] have been built with this principle where the resource management is split into multiple levels. The system layer is simple and is mainly responsible for resource allocation to ensure isolation among applications whereas every application decides for itself on how it wants to manage these allocated resources. Most of these systems are built for generally designed for a homogeneous platform and are concerned with the number/duration of CPUs available but not the type. In most heterogeneous systems, though, applications or runtimes make task placement decisions but are oblivious to the shared environment. Rinnegan integrates the two designs (resource allocation by

system layer and task placement by applications) by allowing the system to manage resources and also enable applications to make informed task placement decisions aware of contention.

6.2.5 Performance Model

Many systems and runtimes for heterogeneous architectures use performance models to make scheduling decisions with the goal of maximizing application or overall system performance. The CAMP system [237] tracks metrics such as instructions per cycle and cache stall time for every application to determine their speedup achievable on asymmetric cores. Rinnegan currently uses a static speedup value between slow and fast cores since the emulation of asymmetric cores through duty-cycle mechanism results in static speedup for all applications. However with the support of application specific model, Rinnegan can easily integrate other performance models.

Runtimes such as StarPU [21] and OmpSS [72] maintain previous task execution times on different processing units for all applications and then estimate the average execution time to determine the processing unit that offers the minimum execution time while making placement decision. With the integration of StarPU and Rinnegan (StarPU+Rinnegan configuration), Rinnegan leveraged the performance model offered by StarPU runtime while making placement decisions.

Other runtimes such as Baymax [51] use linear regression and approximate nearest neighbor based models to determine the execution time of tasks. Profiler tools [14] can predict speedup on different processing units by combining program code characteristics such as loops and hardware characteristics such as number of cores. Though Rinnegan use a simpler model based on linear relation between the input data and the task execution time, it worked well for most of the workloads we considered. For certain workloads such as Sphyraena [23] (database workload), any

of these above models does not work well since it relies on application specific information such as data cardinality. Rinnegan with its support for custom performance model can help in such cases.

6.3 Power and Thermal Management

There has been many works in the area of power and thermal management due to the increase in power density in processors or power constraints in smaller form factor devices. We discuss those works in comparison with Firestorm in this section.

6.3.1 Power Performance Efficiency

Previous works [149, 213, 264, 266] have focused on improving the performance per watt (energy efficiency) for individual applications or the entire system. Better efficiency is achieved by finding the right processor configuration such as the optimal number of cores to be turned on and also the frequency at which the processing units should operate for a particular application or a mix of applications to achieve the best performance at minimal energy. The same goal is also achieved by adapting applications such as altering the degree of parallelism and thus use lesser resources. Firestorm focuses only on mechanisms and policies to distribute power whereas these works focus on finding the optimal power setting for a given application or the system. The above works are complementary to Firestorm where the techniques employed in these works can be integrated to Firestorm to provide guaranteed performance for applications at better energy efficiency.

6.3.2 Thermal Management

Past research works [68] have compared and analyzed the performance and power aspects of different throttling techniques for thermal management by assuming a fixed temperature constraint. Other works [22] propose new mechanism such as idle injection to reduce average processor temperature by trading-off performance. Currently Firestorm only makes use of duty cycle-based throttling but it can incorporate new policies to decide the type of throttling mechanism to deploy based on these works.

Many works [94, 172, 181, 292] have focused on minimizing the system (or chip) temperature through scheduling techniques. A common technique is to migrate computation to a cold resource when the current resource heats up. For example, a floating point intensive thread is moved to a core where the FPU is relatively cold. Also, complementary threads (threads that depend on complementary resources) are co-scheduled in processors that support multiple hardware threads to avoid hotspots. We think these works are complementary to Firestorm where the scheduling based techniques can be employed using Firestorm's features.

There are also works [106] to prevent thermal interference among applications. A malicious thread can create thermal hotspots and thus affecting other thread contexts sharing the same core. The work identifies the application thread that causes the hotspot and throttles that thread without affecting other applications. In contrast to the hardware throttling mechanism that throttles the entire core and thus all thread contexts, this solution provide higher degree of isolation. Firestorm's focus has been to proactively prevent such interference by reserving thermal capacity.

6.3.3 Thermal Modeling

Tools such as Hotspot [300] provides a more accurate thermal model for processors. The accuracy is achieved by including several input factors

such as package material composition, area occupied by microarchitectural resources and cooling solutions. Firestorm abstracts all hardware details and builds a regression based thermal model by tracking the processor temperature over period of time while running different workloads. Since Firestorm published the required interfaces to be exposed by a thermal model, more sophisticated models such as Hotspot can be integrated well with Firestorm.

6.3.4 Power Management

ARM's Intelligent Power Allocation (IPA) [15] combines power and thermal management similar to Firestorm. The IPA system decides the power to be deployed to each processing unit by controlling their p-states based on two inputs: (a) Performance requirements for processing units such as big CPU cluster, little CPU cluster and GPU. (b) Thermal headroom available. The system is built as a control loop where the p-states are adjusted based on the change in the inputs. Though the IPA system is much similar to Firestorm, it does not take into consideration the requirements of individual applications.

The work on cooperative boosting [213] proposes a coordinated power management for applications that use on-chip CPUs and GPU. Given an application, the system chooses the best p-states for both processing units to yield the highest performance by ensuring high utilization of both processing units and also avoid any adverse impact due to thermal throttling. The work is complementary to Firestorm where the cooperative boosting technique can be used to distribute the power obtained an application through Firestorm on different processing unit to yield better performance.

Several systems such as Cinder [235], ECOSytem [299], and Power Containers [246], have been built with a focus on energy management. Firestorm focuses on the similar goal of promoting power as a primary resource and controlling the power use of every application. Firestorm's

power ticket abstraction is inspired from these systems. Since the focus of these works is energy, they allow long-running applications to accumulate energy over time, while Firestorm instead grants the ability to execute with a particular power draw at a moment in time.

6.3.5 Power Modeling

There has been much research in the past decade on power modeling for multi-core processors, e.g., [29, 119, 132, 185, 246]. The models proposed in these works to predict power consumption could be used to improve the accuracy of power prediction in Firestorm. However, Firestorm's focus is not on power modeling alone but rather to promote power as a first class resource in the system.

7

Lessons Learned and Conclusions

Heterogeneity has become the norm in processor designs [35, 112, 274]. These designs can assume different forms such as asymmetric CPU clusters (e.g. big.LITTLE [16]), programmable accelerators (e.g. GPU [187], DSP [57]), fixed function accelerators (e.g. crypto [120]) or custom logic (e.g. FPGA [279]). Higher performance and energy benefits can be achieved from them by matching application characteristics with hardware characteristics and vice versa.

However, certain properties of heterogeneity will not make this any easier. The challenge is that heterogeneity will not be static but can vary at runtime due to physical properties of the hardware or varied requirements (from different applications) in a multi-programmed system. This is referred as dynamic heterogeneity and it can arise for different reasons such as processor core scaling by resource aggregation in dynamic processors, performance variability due to hardware sharing, and the inability to use compute units at full performance in power-constrained architectures.

In this dissertation, we showed why current operating systems (including hardware mechanisms and application runtimes) are not suitable to sustain such dynamic heterogeneous environments. We also presented three systems to handle the properties that results in dynamic heterogeneity and to provide better support to leverage such architectures.

- First, we focused on dynamic processors that can reconfigure CPU

cores at runtime by pooling resources from multiple cores. The new resource management features from the system Chameleon [208] enable parallel and sequential applications to benefit from dynamic processors easily.

- Second, we targeted the problem of task placement in a shared accelerator-rich environment. Rinnegan [210] combines per-application stand-alone performance information with current system state to make better runtime placement decisions for applications.
- Finally, we looked into power-constrained architectures, where processors are over-provisioned with respect to the power limit. With Firestorm [207], we introduced the notion of power and thermal awareness in systems to help the right set of applications to get their desired performance.

The various resource management techniques and features offered by these systems will enable applications to easily leverage and benefit from future heterogeneous architectures. In this chapter, we will summarize these systems in Section 7.1 and then discuss the lessons learnt over the course of this dissertation in Section 7.2.

7.1 Summary

The dissertation focus on three different systems — Chameleon, Rinnegan and Firestorm— where each system focus on a property/challenge/characteristic associated with future heterogeneous architectures. In this section, we try to summarize each system by reiterating a few important points based on the following questions: (a) Why will future architectures exhibit such a property? (b) Why are current operating systems not suitable

to handle dynamic heterogeneity? (c) How does the proposed system provide the required support to leverage such architectures?

7.1.1 Support for Dynamic Processors

The first system discussed in this dissertation is Chameleon [208] to support dynamic processors. These are processors that can dynamically reconfigure (and thus scale) cores by aggregating resources from neighboring cores. Several processor designs such as Core Fusion [125], TRIPS [239], and others suit the needs of both sequential and parallel programs by changing core characteristics at runtime. There is also a startup, SoftMachines [17], that design processors of this type. Intel's turbo boost [59] and hyper-threading [281] are also cases of resource aggregation techniques. With power constraints in future processors, performance via aggregation is going to be more common in future processors.

We showed that the major assumption in current operating systems is to treat every processor core individually without any dependence on other cores. Modern OSes do not support techniques for resource aggregation: They do not support mechanism to borrow resources at a finer time scale, they do not allow a single thread to use more than one core, and finally, they do not allocate and reconfigure CPU cores based on application characteristics. The lack of proper support make current operating systems unsuitable to leverage these kind of processors.

Chameleon introduced three new capabilities to better support dynamic processors. First, the low-cost *processor proxy* software mechanism to allow faster reconfiguration to borrow resources at the scale of a scheduling quantum. Second, the *execution object* abstraction allow applications to use an execution context more tuned to its characteristics. Finally, the *cluster scheduler* enable fair use of compute resources for different kinds of applications. In short, the foundations of Chameleon are based on supporting resource aggregation in hardware to leverage dynamic processors

and ensure better performance for different kinds of applications in the system.

7.1.2 Scheduling in Heterogeneous Architectures

Next, we targeted the problem of ensuring application performance in any form of heterogeneous architecture including accelerators. Provisioning of specialized compute units such as GPUs, DSPs and video encoders is becoming common across all computing environments. The trend here is that same task can be run on different compute units although with different power-performance characteristics. Now the challenge is to decide where to run the task with multiple applications trying to make use of the accelerators.

Current systems and runtimes make task placement decision unaware of a shared environment. As a result, they always offload task to the best performing accelerator based on stand-alone performance information of that task. However, it is clear from the above trend that static decisions are not sufficient in a shared environment. A contended accelerator might perform worse than an idle CPU.

We presented *Rinnegan* [210], a system with a kernel extension and a runtime library, to perform scheduling and task placement in a shared heterogeneous environment. The static performance information of applications alone is not sufficient but combining them with current system state information such as wait time for accelerators can be used to make better placement decisions. Rinnegan has two major components that are responsible for sharing these information. *OpenKernel*, the kernel component in Rinnegan, allows resource managers (e.g. GPU driver, CPU scheduler) to share the system state information with applications. *Libadept*, user-mode runtime, estimates the stand-alone performance of application tasks on different compute units. The runtime combines the performance estimate with system state information during task offloads to make better

task placement decisions. Our system differs from others on how these information are combined.

In summary, Rinnegan separates task placement from resource management contrary to current systems and perform task placement in applications rather than in the kernel. The former, including scheduling and resource allocation, is performed in the kernel to ensure proper isolation among applications. This architecture allow applications to adapt themselves based on the available resources and to make custom task placement decisions based on their performance goals.

7.1.3 Managing Power-Constrained Architectures

Last, we focus on the issue of power constraints in future architectures. This is caused by the phenomena of Dark Silicon where processors are provisioned with abundant transistors but all of them cannot powered on at the same time without over heating the processor. As a result, the compute capacity of current processors is over-provisioned in that the compute units – CPUs, GPU – cannot be used at full performance without exceeding the power limit. The maximum performance of all compute units are capped to stay within the power limit. The major question is what are the compute units to be powered on and how much power should each of them receive. The power distribution is going to be dependent on the importance of applications that are running on those compute units.

Current operating systems make use of all available compute units without any regard to the power limit. Hardware mechanisms are supported in processors to prevent them from exceeding the power limit (power consumption of the processor cannot exceed the limit) or thermal limit (the maximum processor chip temperature cannot exceed the limit). However these mechanisms are useful to implement a processor-wide (and thus system-wide) policy but are not sufficient to understand application semantics such as their priority. As a result, it is also possible for malicious

or background applications to increase the chip temperature and thereby throttle the entire processor along with all applications running in the system.

Given such physical limits, the system should multiplex power and thermal capacity among compute units – by turning on/off or increasing/decreasing p-state – based on the importance of applications (e.g. run GPU at higher performance than CPUs when GPU-bound program has more priority than CPU-bound programs). We presented *Firestorm*, an operating system extension that promotes power and thermal capacity as primary resources in the system. It operates with the goal of choosing the right processor configuration to meet the application goals within the physical limits. Firestorm introduces new abstractions for power and thermal capacity, thereby allowing applications to gather them similar to other resources such as CPU and memory. This also allows the system to enforce a policy to distribute power and thermal capacity to applications based on their importance. To summarize, our system takes a holistic view on both the resources by allowing applications to co-allocate them for better performance.

7.2 Lessons Learned

In this section, we present some of the lessons learned over the course of this dissertation.

7.2.1 Resource Management for Accelerators (GPU)

There exists a semantic gap between operating systems and current GPU drivers where the latter does not expose any interfaces for the OS to control resource management activities. Application runtimes such as CUDA or OpenCL invoke driver interfaces through IOCTL calls [124]. Major vendors such as NVIDIA and AMD provide binaries for these drivers on

different operating systems. The device driver is completely black box where all actions by the driver are invisible and also not manageable by the operating system.

Accelerators are treated similar to any external device in the system. This was sufficient when the accelerator belonged to a niche class and they were used only by a single primary application in the system. However, as more and more applications begin to benefit from GPU, simple resource management techniques for task scheduling and memory management are not sufficient [232]. Also, there is a semantic gap between the device drivers and the operating system where the device driver is unaware of any details about the application. Priority inversion can arise where low priority application get to use the accelerator when important application is waiting. It is not possible to achieve predictable performance for applications in a shared environment with such primitive policies.

There has been an effort called Nouveau [192] to design an open source device driver for NVIDIA GPU cards. However, the design of the driver is best effort and is not completely supported by the vendors. There has been research efforts [171] to reverse engineer the driver functionalities and control their behavior by intercepting the OS calls made by the drivers. This allowed the authors to implement a custom scheduling policy on top of the black box driver. Rinnegan implemented different policies for GPU by instrumenting applications to invoke the GPU agent to perform resource management activities before invoking the driver. Though the above solutions can work, they are not extensible since they involve more manual efforts for every hardware model or for every application.

With accelerators such as GPU becoming a widely used compute unit [232], the closed ecosystem prevents future systems from exploiting the benfits of the accelerator. In contrast, the open nature of Linux allowed quick adaptation of the OS for processor designs such as big.LITTLE [126]. We believe the operating system should manage GPU like accelerators

along with other primary resources in the system [209]. In order to achieve this, the GPU drivers should expose better abstractions and allow OS developers to plug-in new resource management features to let the OS manage GPU as a system resource rather than treat it as an external device. This will enable OS to make resource management decisions based on application properties and allow applications to achieve their goals. This would have allowed us to easily implement agents in Rinnegan rather than instrumenting the applications. This can also enable new use cases such as effective arbitration of accelerators across multiple virtual machines by cloud providers.

7.2.2 Importance of Workloads

When we started working on Rinnegan, we found it difficult to find open source applications that make use of accelerators. Though benchmark suites such as Rodinia [50] and Parboil [270] offer scientific applications that can leverage accelerators, these applications are used to test the microarchitectural features of GPU like accelerators. The program samples provided as part of the OpenCL SDK and NVIDIA CUDA SDK are mostly microbenchmarks and they illustrate the use of programming model features. Few real workloads such as VLC media player that can leverage GPU are closed source.

With heterogeneity becoming mainstream, we feel that there should be a benchmark suite for heterogeneous architectures similar to Mosbench [37] (to measure scalability of operating systems) and Cloudsuite [79] (commonly used cloud services). A variety of applications from different areas such as machine learning, network services or databases with different goals in terms of latency or throughput requirements can be useful to test the overall system performance. It can also be used to evaluate the resource management aspects of the system when multiple workloads are run concurrently.

As part of the Rinnegan work, we gathered variety of application over the period including scientific applications, histogram application hand written over GPU, and few real applications such as database, encryption, media encoding and password cracker. We should mention that these applications still resemble microbenchmarks rather than an end-end application. However, we think this is a move in the right direction and we believe more real and open source applications will be available with maturity of the accelerator ecosystem.

The main challenge for any systems work targeting new processor architecture is the lack of applications. The main issue is that existing applications need to be re-designed to leverage compute units such as accelerators. We feel that finding real workloads for evaluation in such cases is equally important as designing the actual system.

7.2.3 Role of Operating Systems

The role of operating systems seems to be diminishing with advancement in hardware designs. Hardware is getting much faster and also smarter at the same time. These properties can be observed in new accelerator designs [13, 82] where applications can directly access accelerators avoiding the need for an OS layer. New storage media [83] and high capacity network cards [134] have shown that traditional OS are becoming a bottleneck in leveraging these hardware. New hardware designs [303] enable entire application logic to be run on specialized hardware without much help from operating systems. Major vendors such as Microsoft and Google has been adapting such designs [130, 225] in their data centers for big data services to yield better performance and high energy efficiency. In another hardware trend, parts of the OS functionality are being pushed into the hardware. SR-IOV [153] as part of the PCI specification allows external devices to handle virtualization of the hardware by themselves.

These trends suggest that traditional operating systems are not suitable

for new hardware designs but they do not undermine the importancaaaa OS. There has been research [134, 285] proposing new OS designs to leverage new hardware designs. With the new design, the OS operations are split into data plane (mechanisms such as packet dispatch) and control plane (policies such as bandwidth settings) where the data plane is left to the hardware but the policy decisions are still retained in the system software. Such partitioning of functionalities leverages the high performance of the hardware and also provide fairness among applications. Directly accessible accelerators offer low dispatch latency for applications but the hardware cannot efficiently multiplex its resources among applications. The OS knows better about the application characteristics including their requirements and goals. Even with customized accelerators in data centers, the support for multi-programming with proper isolation can only be offered by an operating system. Though OS plays a significant role in the overall execution stack, we believe systems should evolve at a faster pace to keep up with the new hardware trends.

7.2.4 Adaptation in Current Systems

The support for adaptation is one of the important aspect in Rinnegan's design. There has been lot of previous research on adaptive systems both in hardware [75, 144, 182] as well as software fronts [115, 191]. In spite of the various advantages of adaptation such as energy benefits and performance guarantees, we realized that adaptation is not widely supported in current software ecosystem (other than research).

Most commonly known adaptive systems focus on adjusting the resource allocated for applications to satisfy their performance guarantees. If the performance requirements of an application are known beforehand, this enables the system to allocate just enough resources to satisfy the performance goals rather than achieving highest performance possible. It has been shown that frame rate reduction [190] can result in energy reduc-

tions without giving up on user comfortness. In data center environments, workload can be shifted to low power compute unit such as LITTLE cluster to save energy when the incoming traffic load is low. However, most layers of the execution stack including hardware and software are focused only on achieving highest possible performance but not on the required performance. Hardware techniques such as Turbo Boost [59] illustrates such behavior. Frequency boosting is done opportunistically that might not benefit all applications and infact results in energy wastage [156]. This shows that better interfaces need to be introduced for information to be exchanged across execution layers. We try to achieve a similar functionality through accelerator monitor in Rinnegan.

Adaptation can also take the form of approximation. Previous research works [93, 115, 191] have shown that quality of results can be traded for better performance or energy reduction. There also has been focus of approximation in hardware [75, 182] for more energy savings. However, most application designs always target 100% correctness and current runtimes also do not provide support for approximation. This can be attributed to the additional development efforts needed to incorporate adaptation through alternate application logic. Going forward, we hope that better support through runtimes are available to enable adaptation in applications similar to Heartbeats framework [115].

7.2.5 Hardware Interfaces for Resource Management

Certain accelerators [13, 82] can be accessed directly from applications rather than going via the operating systems. This reduces the time taken for task offload since additional software layers are removed from the execution stack and such accelerator designs are more suitable for short tasks as well. Since tasks are directly offloaded, resource management is taken care by the hardware. Simple and fixed policies implemented by the hardware are not sufficient to achieve guaranteed performance in

a shared environment. In order to preserve the low offload latency and also introduce the flexibility in resource management, we believe that future hardware should introduce new interfaces to bridge the semantic gap between the operating systems and the hardware. For example, the operating system can provide as input the application properties (e.g. priority) using which the hardware can make better scheduling decisions.

Other than the task dispatch hardware mechanism, there are few other resource management related mechanisms supported in hardware that are rather difficult to implement in software. For example, power distribution in Intel processors among different compute units such as CPUs and GPU is controlled by the hardware based on power ratio setting done through a MSR register. However, the downside is that this mechanism expose coarse granular interface where the power ratio can be set between CPUs and GPU but not among the individual cores of the CPU. Similarly, thermal management support in hardware is achieved through the thermal safeguard mechanism. However, the limitation is that it operates at a coarse granularity where it throttles the entire processor on reaching the critical temperature. If the processor had access to the information on applications' properties, it can implement a selective throttling policy where low priority applications are throttled more in comparison with importan applications. The take away is that hardware interfaces should capture application properties in a shared enviroment for better performance.

7.3 Future Work

In this section, I describe the possible extensions of current work and also discuss some of the new heterogeneous architecture designs that I would like to focus as part of my future work.

7.3.1 Leveraging Heterogeneity in Systems

The work on Rinnegan targets the problem of task scheduling in heterogeneous environment. I would like to explore the use of resource management features of Rinnegan to handle heterogeneity in distributed systems and also to explore the applicability of Rinnegan for specific applications such as Databases.

Heterogeneity in Distributed Systems. The use of heterogeneous architectures have become common in data centers and other cluster environments. As we have seen already, major vendors such as Microsoft, Google, Facebook and Baidu are using accelerators such as FPGAs [225], GPUs [44], and even custom designed ASICs [130] to accelerate their big data services.

Dandelion [233] is the system built to run applications by automatically scaling them across GPUs provisioned in the cluster machines. However, it is not suitable current cluster environments for the following reasons. First, clusters are likely to be shared [155] among multiple applications for reasons of better utilization. Second, presence of different type of compute units - Intel/AMD processors, on-chip GPUs, NVIDIA/AMD discrete GPU cards or Intel's Xeon Phi - in the cluster machines [51, 220]. Third, different workloads are optimized for different metric such as latency or throughput. Finally, need for low latency cluster scheduling [202].

I would like to design a hierarchical namespace service similar to the functionality of monitors in Rinnegan that expose information about heterogeneous compute units and the waiting time associated with every compute unit. Further, I would like to investigate on distributed low latency scheduling that takes as input the workload with its goals and the information from the namespace service to dispatch tasks to different compute units in the cluster. The system should make task offload decision at runtime to leverage the heterogeneity in the clusters.

Heterogeneity in Database Systems. I am interested in building Rinnegan like system but specific for databases. There has been efforts to de-

sign custom accelerators to perform certain query operations [6, 252, 296] and also works to make use of using programmable accelerators such as GPU [23, 38]. We see two trends that makes databases on future hardware face similar problem as targeted by Rinnegan. First, the database engine is shared across several clients and thus queries from different clients will be run on the same hardware. Similarly in cloud environments such as SQL Azure, multiple virtual machines each running its own database engine share the same physical hardware. Second, the query operations can be run on different compute units (accelerators or CPU cores) at different performance. I am interested in exploring the idea of generating query execution plans based on performance of query operators on different compute units and the load on different compute units at the time of query generation. Also, to dynamically to change query plans if the load on compute units change.

7.3.2 Directly Accessible Accelerators

Processors are beginning to support new interfaces to access on-chip shared accelerators as co-processors. Accelerators like the GPU in AMD Kaveri [13], Crypto accelerator in IBM Power 8 [82] and database accelerators in SPARC M7 [6] can be accessed directly from applications through new instructions in the ISA and virtual memory support in accelerators. Such integration of accelerators enables low latency access since the OS is bypassed and improves programmability by avoiding explicit data copies.

Support for Virtualization. In these type of processor designs, accelerators are virtualized by processors (not software) and it can impact performance of application tasks on accelerators as well as isolation among applications in the system. Memory virtualization is achieved by the use of TLB to cache the virtual to physical address translations. Few designs also provision data caches to avoid memory accesses. I am interested in studying the impact of cache and TLB misses in the context of short tasks

that runs for tens or hundreds of microseconds. I want to explore designs of prefetcher and TLB designs by leveraging the predictable data access patterns for tasks running on accelerators.

Another challenge is that access to such accelerators is not arbitrated by a resource manager due to kernel bypass. This could impact any isolation guarantees in terms of performance in the system. New scheduler interfaces should be supported in the hardware to preserve low latency access and provide isolation support. However, inputs to the scheduler such as policy type and application priorities (or shares) should be set by the operating system. I plan to explore the idea of generic hardware-software co-designed scheduler interfaces applicable for any form of direct accessible devices.

7.3.3 Interfacing Accelerators

CPU cores have been the primary computation engine for most applications. Even with the advent of accelerators, the application thread on the CPU is responsible for co-ordinating computations on accelerators and also transferring data from/to external devices. However, this approach could prove to be costly for applications involving the participation of many accelerators and IO devices to accomplish a single task. Such applications include gesture recognition, augmented reality applications, streaming application over middleboxes, and online deduplication that involves participation of many accelerators and IO devices to accomplish a single task. The legacy co-ordinated approach can lead to redundant data copies, traffic in the interconnect, increased latency and loss in energy efficiency.

There are many hardware improvements that obviate the participation of CPU during data movement. They are direct communication through RDMA support or Intel SCIF interfaces, programmable devices like NetFPGA [303] to perform tasks like inline data filtering, and protection and

virtualization through SR-IOV and IOMMU. Rather than a CPU-centric application design, I want to explore a graph-style design where data can flow through different computation engines without much hand-holding from the main application thread. This raises some interesting challenges such as how to design communication interfaces for accelerators (and IO devices) to interact with each other, data abstraction and flow control mechanism to handle the variation in compute capability of the source and destination devices. This can enable scalable web based services to run independently on accelerators such as Xeon Phi without the involvement of power hungry CPUs.

7.3.4 Near-Data Computation

With the growth in heterogeneity, there has also been focus on programmable devices [134, 203, 243, 252] with compute capability in them. These devices share similar properties with Active Disks [231] and Intelligent RAM [212]. However, the compute capability provisioned in current devices is large enough to offload a major chunk of application logic on to them. Major vendors such as Microsoft and Google have been using similar devices [130, 225] in their data center environments. I am interested in analyzing application performance on such architectures compared to running computations on regular CPU cores. Specifically, I want to look at the improvement in scalability achieved by applications with the programmable hardware. I am also interested in understanding the bottlenecks in current operating systems that prevents application from achieving full performance as compared to running the application completely on the hardware. And finally, I would like to understand the impact of programmability on such hardware designs.

7.4 Conclusion

Processor designs are increasingly becoming heterogeneous and the trend is here to stay because of several factors such as Dark Silicon, demand for performance and energy efficiency, and requirement of power efficiency in smaller form factor devices.

Current systems are designed for homogeneous processor designs and they are ill-suited to handle heterogeneity in architectures. Though few operating systems are aware of asymmetric processor designs such as big.LITTLE and application runtimes such as CUDA and OpenCL help improve the programmability of heterogeneous architectures, they are not aware of the challenges posed by dynamic heterogeneity. This makes these systems less efficient and lose on potential performance that could have been achieved by applications in such architectures.

In this dissertation, we presented three systems — Chameleon, Rinnegan and Firestorm— to tackle dynamic heterogeneity and enable applications to benefit more from heterogeneous architectures. The abstractions introduced simplify application development by hiding hardware complexities, the mechanisms added help applications to leverage heterogeneity better and achieve better performance, the interfaces proposed allow better integration with other systems,runtimes and performance model, and the policies ensure fairness among applications. To conclude, we presented the challenges and opportunities with dynamic heterogeneity and how the systems designed take a step forward in providing better support for future architectures.

Heterogeneity is not limited to compute units but it is becoming common with other hardware resources such as memory [4, 71, 174] and storage [240]. With the advent of software-defined systems [11, 276], the system software (operating systems and runtimes) should enable applications to meet their goals by choosing the right set of hardware resources based on the program requirements and hardware properties. A holistic

solution with complete awareness to heterogeneity is required to ensure better efficiency with future hardware systems.

Bibliography

[1] Fuad Abazovic. AMD uses superfast coherent fabric. http://fudzilla.com/news/processors/38381-amd-s-new-interconnect-tech-is-coherent-fabric, August 2015.

[2] Antony Adshead. Data set to grow 10-fold by 2020 as internet of things takes off. http://www.computerweekly.com/news/2240217788/Data-set-to-grow-10-fold-by-2020-as-internet-of-things-takes-off, April 2014.

[3] Advanced Encryption Standard (AES). http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[4] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 607–618, New York, NY, USA, 2015. ACM.

[5] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-core Processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 470–481, New York, NY, USA, 2007. ACM.

[6] Kathirgamar Aingaran, Sumti Jairath, Georgios Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, et al. M7: Oracle's Next-Generation Sparc Processor. *Micro, IEEE*, 35(2):36–45, 2015.

[7] Jose Allarey, Varghese George, and Sanjeev Jahagirdar. Power management enhancements in the 45nm intel core microarchitecture. *Intel Technical Journal*, 12(3):169–178, oct 2008.

[8] Linux GPU Instances. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html`.

[9] AMD Corporation. AMD Turbo Core Technology. `http://www.amd.com/en-us/innovations/software-technologies/turbo-core`.

[10] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[11] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 233–248, Berkeley, CA, USA, 2014. USENIX Association.

[12] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[13] AMD A-Series Desktop APUs. `http://www.amd.com/us/products/desktop/processors/a-series/Pages/nextgenapu.aspx`.

[14] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 725–737, New York, NY, USA, 2015. ACM.

[15] Intelligent Power Allocation. `https://developer.arm.com/open-source/intelligent-power-allocation`.

[16] ARM Limited. big.LITTLE Technology: The Future of Mobile. `www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`.

[17] Lucian Armasu. Soft Machines' 'Virtual Cores' Promise 2-4x Performance/Watt Advantage Over Competing CPUs. http://www.tomshardware.com/news/soft-machines-virtual-cores-visc,31127.html, February 2016.

[18] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 90–105, New York, NY, USA, 2003. ACM.

[19] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '98/PERFORMANCE '98, pages 233–243, New York, NY, USA, 1998. ACM.

[20] Steve Ashby, Pete Beckman, Jackie Chen andÂ Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, Tony Mezzacappa, Parviz Moin, Mike Norman, Robert Rosner, Vivek Sarkar, Andrew Siegel, Fred Streitz, Andy White, and Margaret Wright. The Opportunities and Challenges of Exascale Computing. `http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf`, 2010.

[21] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference*, pages 863–874, August 2009.

[22] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: Processor-level Preventive Thermal Management via Idle Cycle Injection. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 89–94, New York, NY, USA, 2011. ACM.

[23] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 94–103, New York, NY, USA, 2010. ACM.

[24] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[25] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 282–293, New York, NY, USA, 2000. ACM.

[26] Bill Bateson. Special issue applying the 68000 family xenix and the motorola 68000 family. *Microprocessors and Microsystems*, 8(7):350 – 356, 1984.

[27] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[28] BEIGNET. `https://01.org/beignet`.

[29] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-Driven Energy Accounting for Dynamic Thermal Management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[30] Major Bhadauria and Sally A. McKee. An Approach to Resource-aware Co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[31] Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of 5th Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[32] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 207–216, August 1995.

[33] Shekar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov 2005.

[34] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, Jul 1999.

[35] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[36] Fred A. Bower, Daniel J. Sorin, and Landon P. Cox. The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling. *IEEE Micro*, 28(3):17–25, May 2008.

[37] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[38] Sebastian Breßand Gunter Saake. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proceedings of the Very Large Database Endowment*, 6(12):1398–1403, August 2013.

[39] Nathan Brookwood. Amd fusion. family of apus: Enabling a superior, immersive pc experience. `http://www.amd.com/Documents/48423_fusion_whitepaper_WEB.pdf`, Mar 2010.

[40] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, New York, NY, USA, 1997. ACM.

[41] Daniel Burrus. The internet of things is far bigger than anyone realizes. `http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/`, November 2014.

[42] C++ AMP : Language and Programming Model. `http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf`.

[43] C. Caşcaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A Taxonomy of Accelerator Architectures and Their Programming Models. *IBM J. Res. Dev.*, 54(5):473–482, September 2010.

[44] cade Metz. Facebook Open Sources Its AI Hardware as It Races Google. `http://www.wired.com/2015/12/facebook-open-source-ai-big-sur/`, December 2015.

[45] Cavium Networks. OCTEON II CN68XX Multi-Core MIPS64 Processors. `http://www.cavium.com/OCTEON-II_CN68XX.html`.

[46] CFS Scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[47] Koushik Chakraborty, Philip Wells, and Gurindar Sohi. A case for over-provisioned multicore system. Technical Report UWCS TR1607, University of Wisconsin Technical Report, 2007.

[48] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel®Core™ I7 Turbo Boost Feature. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 188–197, Washington, DC, USA, 2009. IEEE Computer Society.

[49] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's Rock Processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 484–495, New York, NY, USA, 2009. ACM.

[50] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[51] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 681–696, New York, NY, USA, 2016. ACM.

[52] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 223–232, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[53] Andrew A Chien, Allan Snavely, and Mark Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 4:1987–1996, 2011.

[54] Andrew A. Chien, Tung Thanh-Hoang, Dilip Vasudevan, Yuanwei Fang, and Amirali Shambayati. 10x10: A Case Study in Highly-Programmable and Energy-Efficient Heterogeneous Federated Architecture. *SIGARCH Comput. Archit. News*, 43(3):2–9, December 2015.

[55] Yaohan Chu. Application-specific coprocessor computer architecture. In *Application Specific Array Processors, 1990. Proceedings of the International Conference on*, pages 653–664, Sep 1990.

[56] Yaohan Chu and Kozo Itano. A top-down parsing co-processor for compilation. In *System Sciences, 1989. Vol.I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 1, pages 403–413 vol.1, Jan 1989.

[57] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro*, 34(2):34–43, Mar 2014.

[58] Jonathan Corbet. CFS group scheduling. `http://lwn.net/Articles/240474/`, 2007.

[59] Intel Corporation. Intel Turbo Boost max Technology 3.0. `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-max-technology.html`.

[60] NVIDIA CUDNN. GPU Accelerated Deep Learning. `https://developer.nvidia.com/cudnn`.

[61] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Dashanand P. Singh. From opencl to high-performance hardware on FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.

[62] Michael Winston Dales and Câ—‹ Michael Winston Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 10980–10985, 2003.

[63] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[64] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions. *Proceedings of the IEEE*, 87(4):668–678, April 1999.

[65] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[66] Martin Dimitrov. Intel Power Governor. `https://software.intel.com/en-us/articles/intel-power-governor`.

[67] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(12):3088–3098, 2014.

[68] James Donald and Margaret Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 78–88, Washington, DC, USA, 2006. IEEE Computer Society.

[69] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[70] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 261–276, New York, NY, USA, 1999. ACM.

[71] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[72] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[73] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[74] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug 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.

[75] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM.

[76] Songchun Fan, Seyed Majid Zahedi, and Benjamin 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.

[77] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16(4):306 – 318, 1992.

[78] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[79] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[80] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[81] Implementing FPGA Design with the OpenCL Standard. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf`, November 2013.

[82] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the Wire-speed Processor and Architecture. *IBM J. Res. Dev.*, 54(1):27–37, January 2010.

[83] R. F. Freitas and W. W. Wilcke. Storage-class Memory: The Next Storage System Technology. *IBM Journal of Research and Development*, 52(4):439–447, July 2008.

[84] Andrei Frumusanu. MediaTek Unveils Helio X20 Tri-Cluster 10-Core SoC. `http://www.anandtech.com/show/9227/mediatek-helio-x20`, May 2015.

[85] Andrei Frumusanu. Early Exynos 8890 Impressions And Full Specifications. `http://www.anandtech.com/show/10075/early-exynos-8890-impressions`, February 2016.

[86] Wenyin Fu and Katherine Compton. Scheduling Intervals for Reconfigurable Computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 87–96, April 2008.

[87] FV264: FastVDO's hardware H.264 / AVC decoder IP core. `http://www.fastvdo.com/FV264/`.

[88] Grand Central Dispatch. `http://developer.apple.com/library/ios/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html`.

[89] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[90] Dan Gibson and David A. Wood. Forwardflow: A Scalable Core for Power-constrained CMPs. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 14–25, New York, NY, USA, 2010. ACM.

[91] Jana Giceva, Tudor ioan Salomie, Adrian SchÃ¼pbach, Gustavo Alonso, and Timothy Roscoe. COD: Database / Operating System Co-Design, 2013.

[92] Peter N Glaskowsky. NVIDIAâŁ™s Fermi: the first complete GPU computing architecture. *White paper*, 18, 2009.

[93] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 383–397, New York, NY, USA, 2015. ACM.

[94] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 260–270, New York, NY, USA, 2004. ACM.

[95] James R. Goodman. Using Cache Memory to Reduce Processor-memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM.

[96] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 503–514, Washington, DC, USA, 2011. IEEE Computer Society.

[97] Orion Granatir. A Look at Sandy Bridge: Integrating Graphics into the CPU. `http://software.intel.com/en-us/blogs/2011/01/13/a-look-at-sandy-bridge-integrating-graphics-into-the-cpu`, January 2011.

[98] Ryan E. Grant and Ahmad Afsahi. Power-performance Efficiency of Asymmetric Multiprocessors for Multi-threaded Scientific Applications. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 300–300, Washington, DC, USA, 2006. IEEE Computer Society.

[99] Dominik Grewe and Michael F. P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[100] Shantanu Gupta, Shuguang Feng, Amin Ansari, and Scott Mahlke. Erasing Core Boundaries for Robust and Configurable Performance. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 325–336, Washington, DC, USA, 2010. IEEE Computer Society.

[101] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[102] H264 Encoders. https://www.alma-technologies.com/ip-core.H264-Encoders.

[103] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[104] Babak Hamidzadeh, Yacine Atif, and David J. Lilja. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7(7):633–652, 1995.

[105] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March 2000.

[106] Jahangir Hasan, Ankit Jalote, T. N. Vijaykumar, and Carla E. Brodley. Heat stroke: power-density-based denial of service in SMT. In *11th International Symposium on High-Performance Computer Architecture*, pages 166–177, Feb 2005.

[107] Nick Heath. Intel takes its next step towards exascale computing. http://www.zdnet.com/article/intel-takes-its-next-step-towards-exascale-computing/, June 2014.

[108] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious Parallelism for In-memory Column-stores. *Proceedings of the Very Large Database Endowment*, 6(9):709–720, July 2013.

[109] Nicole Hemsoth. Inside the GPU Clusters that Power BaiduâŁ™s Neural Networks. http://www.nextplatform.com/2015/12/11/inside-the-gpu-clusters-that-power-baidus-neural-networks/, December 2015.

[110] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[111] Sebastian Herbert and Diana Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-multiprocessors. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED '07, pages 38–43, New York, NY, USA, 2007. ACM.

[112] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[113] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[114] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM.

[115] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic Knobs for Responsive Power-aware Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM.

[116] Houman Homayoun, Vasileios Kontorinis, Amirali Shayan, Ta-Wei Lin, and Dean M. Tullsen. Dynamically Heterogeneous Cores Through 3D Resource Pooling. 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.

[117] HSA Intermediate Language. `https://hsafoundation.app.box.com/s/m6mrsjv8b7r50kqeyyal`, May 2013.

[118] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B. Nightingale. WearDrive: Fast and Energy-efficient Storage for Wearables. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 613–625, Berkeley, CA, USA, 2015. USENIX Association.

[119] Wei Huang, Charles Lefurgy, William Kuk, Alper Buyuktosunoglu, Michael Floyd, Karthick Rajamani, Malcolm Allen-Ware, and Bishop Brock. Accurate Fine-Grained Processor Power Proxies. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 224–234, Washington, DC, USA, 2012. IEEE Computer Society.

[120] James Hughes, Gary Morton, Jan Pechanec, Christoph Schuba, Lawrence Spracklen, and Bhargava Yenduri. Transparent Multi-core Cryptographic Support on Niagara CMT Processors. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 81–88, Washington, DC, USA, 2009. IEEE Computer Society.

[121] Image processor. `https://en.wikipedia.org/wiki/Image_processor`.

[122] Intel Corporation. 82093AA I/O ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (IOAPIC). `http://www.intel.com/design/chipsets/datashts/29056601.pdf`, 1996.

[123] Intel Corporation. Thermal Protection And Monitoring Features: A Software Perspective. `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/54118.htm`, 2005.

[124] Linux Programmer's Manual. `http://man7.org/linux/man-pages/man2/ioctl.2.html`.

[125] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 186–197, New York, NY, USA, 2007. ACM.

[126] Brian Jeff. big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling. http://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf, November 2013.

[127] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[128] M. Tim Jones. Inside the linux 2.6 completely fair scheduler, Dec 2009. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[129] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, May 2016.

[130] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, May 2016.

[131] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[132] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.

[133] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[134] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. ACM.

[135] A. Khanna and J. Zinky. The Revised ARPANET Routing Metric. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 45–56, New York, NY, USA, 1989. ACM.

[136] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable Lightweight Processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[137] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[138] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[139] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, March 2005.

[140] Hessam Kooti, Elaheh Bozorgzadeh, Shenghui Liao, and Lichun Bao. Transition-aware Real-time Task Scheduling for Reconfigurable Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 232–237, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[141] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[142] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 389–400, New York, NY, USA, 2012. ACM.

[143] Venkata Krishnan and Josep Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-multiprocessor. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 85–92, New York, NY, USA, 1998. ACM.

[144] Logan Kugler. Is "Good Enough" Computing Good Enough? *Commun. ACM*, 58(5):12–14, April 2015.

[145] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[146] George Kyriazis. Heterogeneous System Architecture: A Technical Review. `https://developer.amd.com/wordpress/media/2012/10/hsa10.pdf`, August 2012.

[147] Matthias Laux. Solaris processor sets made easy. `http://developers.sun.com/solaris/articles/solaris_processor.html`, 2001.

[148] Craig Letavec and John Ruggiero. The n-queens problem. *INFORMS Transactions on Education*, 2(3), 2002.

[149] Jian Li and Jose F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 77–87, Feb 2006.

[150] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[151] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM.

[152] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[153] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[154] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[155] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos 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.

[156] David Lo and Christos Kozyrakis. Dynamic management of TurboMode in modern multi-core chips. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–613. IEEE, 2014.

[157] Chris Lomont. `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`, June 2011.

[158] Enno Lübbers and Marco Platzner. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, October 2009.

[159] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[160] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, and Antonia Zhai. Energy Efficient Speculative Threads: Dynamic Thread Allocation in Same-ISA Heterogeneous Multicore Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 453–464, New York, NY, USA, 2010. ACM.

[161] Daniel Lustig and Margaret Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 354–365, Washington, DC, USA, 2013. IEEE Computer Society.

[162] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 457–471, New York, NY, USA, 2016. ACM.

[163] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative Multithreaded Processors. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 77–84, New York, NY, USA, 1998. ACM.

[164] Qualcomm MARE: Enabling Applications for Heterogeneous Mobile Devices. `https://developer.qualcomm.com/downloads/whitepaper-qualcomm-mare-enabling-applications-heterogeneous-mobile-devices`, April 2014.

[165] Marth, Erich and Marcus, Guillermo. Parallelization of the x264 encoder using OpenCL. `http://li5.ziti.uni-heidelberg.de/x264gpu/`.

[166] C Märtin. Post-Dennard Scaling and the final Years of MooreâŁ™s Law Consequences for the Evolution of Multicore-Architectures. *Informatik und Interaktive Systeme*, 2014.

[167] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, , Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Proceedings of the Ottawa Linux Symposium*, July 2001.

[168] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. A Measurement Study of GPU DVFS on Energy Conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 10:1–10:5, New York, NY, USA, 2013. ACM.

[169] Daniel Menascé and Virgílio Almeida. Cost-performance Analysis of Heterogeneity in Supercomputer Architectures. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 169–177, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[170] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM.

[171] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM.

[172] Andreas Merkel and Frank Bellosa. Task Activity Vectors: A New Metric for Temperature-aware Scheduling. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 1–12, New York, NY, USA, 2008. ACM.

[173] Rick Merritt. ARM CTO: power surge could create 'dark silicon'. `http://www.eetimes.com/document.asp?doc_id=1172049`, October 2009.

[174] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, Feb 2015.

[175] Microsoft Corporation. New NUMA Support with Windows Server 2008 R2 and Windows 7. `http://archive.msdn.microsoft.com/64plusLP`, 2008.

[176] Microsoft Corporation. Processor power management in windows 7 windows server 2008 r2. `http://download.microsoft.com/download/3/0/2/3027D574-C433-412A-A8B6-5E0A75D5B237/ProcPowerMgmtWin7.docx`, January 2010.

[177] Christpher Mims. Why CPUs Aren't Getting Any Faster. `https://www.technologyreview.com/s/421186/why-cpus-arent-getting-any-faster/`, October 2010.

[178] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 267–281, New York, NY, USA, 2015. ACM.

[179] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Partha Ranganathan, and Vanish Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, May 2008.

[180] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan 1998.

[181] Justin Moore, Jeff Chase, Parthasarathy Ranganathan, and Ratnesh Sharma. Making Scheduling "Cool": Temperature-aware Workload Placement in Data Centers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[182] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–614, Feb 2015.

[183] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 181–193, New York, NY, USA, 1997. ACM.

[184] Zwane Mwaikambo, Rusty Russell, Ashok Raj, and Joel Schopp. Linux Kernel Hotplug CPU Support. In *Proceedings of the Ottawa Linux Symposium*, pages 181–194, 2004.

[185] Ripal Nathuji and Karsten Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 265–278, New York, NY, USA, 2007. ACM.

[186] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6:40–53, March 2008.

[187] John Nickolls and William J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, March 2010.

[188] David Nield. Microsoft reveals the hololens battery life, and it's not great. http://www.techradar.com/us/news/wearables/microsoft-spills-a-few-more-details-about-the-hololens-1313245, January 2016.

[189] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[190] Kent W. Nixon, Xiang Chen, Hucheng Zhou, Yunxin Liu, and Yiran Chen. Mobile GPU Power Consumption Reduction via Dynamic Resolution and Frame Rate Scaling. In *Proceedings of the 6th USENIX Conference on Power-Aware Computing and Systems*, HotPower'14, pages 5–5, Berkeley, CA, USA, 2014. USENIX Association.

[191] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 276–287, New York, NY, USA, 1997. ACM.

[192] Nouveau: Accelerated Open Source driver for nVidia cards. https://nouveau.freedesktop.org/wiki/.

[193] NVIDIA Corporation. Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf.

[194] NVIDIA OpenCL SDK . http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html.

[195] NVIDIA NVLINK HIGH-SPEED INTERCONNECT. http://www.nvidia.com/object/nvlink.html.

[196] The OpenACC Application Program Interface. http://www.openacc-standard.org/.

[197] The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/.

[198] OpenCV. http://opencv.org/.

[199] OpenSSL. The open source toolkit for SSL/TLS. http://www.openssl.org.

[200] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 303–, Washington, DC, USA, 1999. IEEE Computer Society.

[201] John K Ousterhout. Scheduling Techniques for Concurrent Systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[202] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[203] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[204] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[205] John F. Palmer. The INTEL® 8087 Numeric Data Processo. In *Proceedings of the May 19-22, 1980, National Computer Conference*, AFIPS '80, pages 887–893, New York, NY, USA, 1980. ACM.

[206] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing Parallel Software Efficiently with Lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 376–387, New York, NY, USA, 2010. ACM.

[207] Sankaralingam Panneerselvam and Michael Swift. Firestorm: Operating Systems for Power-Constrained Architectures. Technical Report UWCS TR1837, University of Wisconsin Technical Report, 2016.

[208] Sankaralingam Panneerselvam and Michael M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 99–110, New York, NY, USA, 2012. ACM.

[209] Sankaralingam Panneerselvam and Michael M. Swift. Operating Systems Should Manage Accelerators. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[210] Sankaralingam Panneerselvam and Michael M. Swift. Rinnegan: Efficient Resource Use in Heterogeneous Architectures. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques*, PACT '16, New York, NY, USA, September 2016. ACM.

[211] Sankaralingam Panneerselvam, Michael M. Swift, and Nam Sung Kim. Bolt: Faster reconfiguration in operating systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 511–516, Berkeley, CA, USA, 2015. USENIX Association.

[212] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, March 1997.

[213] Indrani Paul, Srilatha Manne, Manish Arora, W. Lloyd Bircher, and Sudhakar Yalamanchili. Cooperative Boosting: Needy Versus Greedy Power Management. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 285–296, New York, NY, USA, 2013. ACM.

[214] Parallel Implementation of bzip2. `http://compression.ca/pbzip2/`.

[215] PCI Local Bus Specification Revision 3.0. `http://www.xilinx.com/Attachment/PCI_SPEV_V3_0.pdf`, February 2004.

[216] Oliver Pell and Oskar Mencer. Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing. *SIGARCH Comput. Archit. News*, 39(4):60–65, December 2011.

[217] Simon Peter, Andrew Baumann, Zachary Anderson, and Timothy Roscoe. Gang scheduling isnâŁ™t worth it ... yet. `https://www.microsoft.com/en-us/research/publication/gang-scheduling-isnt-worth-it-yet/`, November 2011.

[218] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design Principles for End-to-end Multicore Schedulers. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[219] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design Principles for End-to-end Multicore Schedulers. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[220] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel MossÃ©, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–258, Feb 2015.

[221] J. Planas, R. M. Badia, E. AyguadÃ©, and J. Labarta. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 138–149, May 2013.

[222] Judit Planas, Rosa M Badia, Eduard Ayguade, and Jesus Labarta. AMA: Asynchronous Management of Accelerators for Task-based Programming Models. *Procedia Computer Science*, 51:130–139, 2015.

[223] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. SSMART: Smart Scheduling of Multi-architecture Tasks on Heterogeneous Systems. In *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, WACCPD '15, pages 1:1–1:11, New York, NY, USA, 2015. ACM.

[224] Intel Powerclamp Driver. `https://www.kernel.org/doc/Documentation/thermal/intel_powerclamp.txt`.

[225] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[226] Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin Pipe, Thomas Wenisch, and Milo Martin. Utilizing Dark Silicon to Save Energy with Computational Sprinting. *IEEE Micro*, 33(5):20–28, September 2013.

[227] Rajesh Raman, Miron Livny, and Marv Solomon. Matchmaking: An Extensible Framework for Distributed Resource Management. *Cluster Computing*, 2(2):129–138, April 1999.

[228] Daniel A. Reed and Jack Dongarra. Exascale Computing and Big Data. *Commun. ACM*, 58(7):56–68, June 2015.

[229] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 25–36, New York, NY, USA, 2000. ACM.

[230] Don Reisinger. http://www.cnet.com/news/sony-ps3-is-hard-to-develop-for-on-purpose/. `http://www.cnet.com/news/sony-ps3-is-hard-to-develop-for-on-purpose/`, February 2009.

[231] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active Disks for Large-Scale Data Processing. *Computer*, 34(6):68–74, June 2001.

[232] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[233] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[234] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[235] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy Management in Mobile Devices with the Cinder Operating System. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.

[236] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[237] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 139–152, New York, NY, USA, 2010. ACM.

[238] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging Workload Diversity Through OS Scheduling to Maximize Performance on single-ISA Heterogeneous Multicore Systems. *J. Parallel Distrib. Comput.*, 71(1):114–131, January 2011.

[239] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 422–433, New York, NY, USA, 2003. ACM.

[240] Mohit Saxena and Michael M. Swift. FlashVM: Virtual Memory Management on Flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 14–14, Berkeley, CA, USA, 2010. USENIX Association.

[241] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. Mapping a Data-flow Programming Model Onto Heterogeneous Platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 61–70, New York, NY, USA, 2012. ACM.

[242] Michael J. Schulte, Mike Ignatowski, Gabriel H. Loh, Bradford M. Beckmann, William C. Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K. Reinhardt, and Gregory Rodgers. Achieving Exascale Capabilities through Heterogeneous Computing. *IEEE Micro*, 35(4):26–36, July 2015.

[243] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[244] Noam Shalev, Eran Harpaz, Hagar Porat, Idit Keidar, and Yaron Weinsberg. CSR: Core Surprise Removal in Commodity Operating Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 773–787, New York, NY, USA, 2016. ACM.

[245] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *Proceedings of Sensors to Cloud Architectures Workshop*, SCAW '15, 2015.

[246] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 65–76, New York, NY, USA, 2013. ACM.

[247] Glen Shires. A New VLSI Graphics Coprocessor-The Intel 82786. *IEEE Comput. Graph. Appl.*, 6(10):49–55, October 1986.

[248] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[249] M. Shoaib Bin Altaf and D.A. Wood. LogCA: A Performance Model for Hardware Accelerators. *Computer Architecture Letters*, PP(99):1–1, 2014.

[250] Suresh Siddha and Venkatesh Pallipadi. Chip multi processing aware Linux kernel scheduler. In *Proceedings of the Ottawa Linux Symposium*, pages 337–348, 2006.

[251] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, New York, NY, USA, 2013. ACM.

[252] Malcolm Singh and Ben Leonhardi. Introduction to the IBM Netezza Warehouse Appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '11, pages 385–386, Riverton, NJ, USA, 2011. IBM Corporation.

[253] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[254] James E. Smith and Andrew R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[255] James E. Smith and Gurindar S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, Dec 1995.

[256] Ryan Smith. GPU Boost 2.0: Temperature Based Boosting. `http://www.anandtech.com/show/6760/nvidias-geforce-gtx-titan-part-1/5`, February February.

[257] Steven P. Smith. Dynamic Scheduling and Resource Management in Heterogeneous Computing Environments with Reconfigurable Hardware. In *International Conference on Computer Design*, 2006.

[258] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 223–234, Washington, DC, USA, 2006. IEEE Computer Society.

[259] Avinash Sodani. Race to Exascale: Opportunities and Challenges. MICRO 2011 Keynote, December 2011.

[260] Avinash Sodani. Knights landing: 2nd generation intel xeon phi processor. In *August issue of Proceedings of Hot Chips: A Symposium on High Performance Chips*, 2015.

[261] Solarflare Communications. Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. `http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_OpenOnload_IntroPaper.pdf`.

[262] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM.

[263] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. Price Theory Based Power Management for Heterogeneous Multi-cores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 161–176, New York, NY, USA, 2014. ACM.

[264] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. Price theory based power management for heterogeneous multi-cores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 161–176, New York, NY, USA, 2014. ACM.

[265] SpeedFan. `https://en.wikipedia.org/wiki/SpeedFan`.

[266] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA, 2014. ACM.

[267] William Stanek. Windows Server 2008 R2: A primer. `http://technet.microsoft.com/en-us/magazine/ee677582.aspx`, November 2009.

[268] Task Scheduling Policy. `http://starpu.gforge.inria.fr/doc/html/Scheduling.html`.

[269] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov 2004.

[270] J.A. Stratton, C. Rodrigues, I.J. Sung, N. Obeid, L.W. Chang, N. Anssari, G.D. Liu, and W.W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.

[271] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan 2015.

[272] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 253–264, New York, NY, USA, 2009. ACM.

[273] Surface Book. `https://www.microsoft.com/surface/en-us/devices/surface-book`.

[274] Michael B. Taylor. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA, 2012. ACM.

[275] Texas Instruments. OMAP 5 mobile applications platform. http://www.ti.com/pdfs/wtbu/SWCT010.pdf.

[276] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[277] Linux Thermal Daemon. https://01.org/linux-thermal-daemon.

[278] Patrick Thibodeau. Data centers are the new polluters. http://www.computerworld.com/article/2598562/data-center/data-centers-are-the-new-polluters.html, August 2014.

[279] Stephen M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, March 2015.

[280] Truecrack. https://code.google.com/p/truecrack/.

[281] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[282] Rob van der Meulen and Janessa Rivera. Gartner Forecasts 59 Percent Mobile Data Growth Worldwide in 2015. http://www.gartner.com/newsroom/id/3098617, July 2015.

[283] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[284] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM.

[285] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[286] C. A. Waldspurger and W. E. Weihl. An Object-oriented Framework for Modular Resource Management. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, IWOOOS '96, pages 138–, Washington, DC, USA, 1996. IEEE Computer Society.

[287] Mark Walton. Xbox Project Scorpio: Will it really do 4K? `http://arstechnica.com/gaming/2016/06/xbox-project-scorpio-hardware-specs-can-it-do-4k/`, June 2016.

[288] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 193–204, Berkeley, CA, USA, 2014. USENIX Association.

[289] Guosai Wang, Shuhao Wang, Bing Luo, Weisong Shi, Yinghang Zhu, Wenjun Yang, Dianming Hu, Longbo Huang, Xin Jin, and Wei Xu. Increasing Large-scale Data Center Capacity by Statistical Power Control. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 8:1–8:15, New York, NY, USA, 2016. ACM.

[290] Liang Wang and Kevin Skadron. Implications of the Power Wall: Dim Cores and Reconfigurable Logic. *IEEE Micro*, 33(5):40–48, September 2013.

[291] Yasuko Watanabe, John D. Davis, and David A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 2–13, New York, NY, USA, 2010. ACM.

[292] Andreas Weissel and Frank Bellosa. Dynamic Thermal Management for Distributed Systems. In *IN PROCEEDINGS OF THE FIRST WORKSHOP ON TEMPERATURE-AWARE COMPUTER SYSTEMS (TACSâŁ™04)*, 2004.

[293] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Dynamic Heterogeneity and the Need for Multicore Virtualization. *SIGOPS Oper. Syst. Rev.*, 43(2):5–14, April 2009.

[294] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Mixed-mode Multicore Reliability. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 169–180, New York, NY, USA, 2009. ACM.

[295] Wikipedia. Heat capacity. `https://en.wikipedia.org/wiki/Heat_capacity`.

[296] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[297] Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Andrey Ayupov, Steven Burns, and Ozcan Ozturk. Hardware Accelerator Design for Data Centers. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 770–775, Piscataway, NJ, USA, 2015. IEEE Press.

[298] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 17–31, Berkeley, CA, USA, 2014. USENIX Association.

[299] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: Managing Energy As a First Class Operating System Resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 123–132, New York, NY, USA, 2002. ACM.

[300] Runjie Zhang, Mircae R. Stan, and Kevin Skadron. HotSpot 6.0: Validation, Acceleration and Extension. Technical Report CS-2015-04, University of Virginia, April 2015.

[301] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Hardware Execution Throttling for Multi-core Resource Management. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 23–23, Berkeley, CA, USA, 2009. USENIX Association.

[302] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 25–36, Washington, DC, USA, 2007. IEEE Computer Society.

[303] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, Sept 2014.

[304] Zilog Z80. `https://en.wikipedia.org/wiki/Zilog_Z80`.