Adaptive Mechanisms for Self-Aware Multicore Systems

by

Eric Lau

B. A. Sc., University of Toronto (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2012

Author _____
Department of Electrical Engineering and Computer Science
August 13, 2012

Certified by _____
Srini Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by _____
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by _____
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Adaptive Mechanisms for Self-Aware Multicore Systems
by
Eric Lau

Submitted to the Department of Electrical Engineering and Computer Science
on August 13, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

As the push for extreme scale performance continues to make computer architectures increasingly complex, there has been a call for better programming models, and the systems to support them. Todays microprocessors now expose more system resources than ever to software, leaving it up to the application programmer to manage them. Studies have shown that the energy efficiency of future technologies may eventually affect the ultimate performance of multicore processors, and so programmers are forced to optimize systems for both performance and energy in the midst of countless configurable parameters - an extremely difficult task.

Self-aware systems can configure themselves through introspection, providing performance and energy optimization without pressing an unrealistic burden on the programmer. However, to build effective self-aware systems, we must identify useful sources of adaptivity. This thesis will show the effectiveness of a number of adaptive mechanisms for self-aware multicore systems. We show that adding these mechanisms improves efficiency, and then make a case for coordinated adaptive systems. Coordinated systems treat adaptivity as a first-class object, and can outperform all non-adaptive, statically configured, and uncoordinated adaptive systems that do not possess a general view of system-wide adaptivity.

Thesis Supervisor: Srini Devadas
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

How do you write something like this without loads of help?

You don't. At least, if there is a way I don't know it, because I have a lot of thank you's to distribute.

First, I would like to thank Anant Agarwal for taking me into his group, and giving me a chance to work here at MIT. However much has changed as I've grown up, coming to MIT has ever been a dream of mine. His passion and leadership have been a constant motivation, and his ideas were invaluable to this work.

Thank you to Srini Devadas, who took me under his wing after my first year. It was almost by chance that we pinballed into his group, but he welcomed us with open arms anyway, and provided me with inestimable wisdom and guidance.

To Jason Miller and Hank Hoffmann, who would have been listed as co-advisors if they would let me. Thanks for being consummate leaders and teachers, and being two of the smartest people I have ever met. It still has not ceased to amaze me how much these guys know, and on what seem like virtually every existing topic. For better or for worse, I've never felt more incompetent than when I am with them. I'm not even sure they realize this, but without them this thesis would probably have no content at all.

Thanks to George Kurian, who helped me solve almost every Graphite issue I had to misfortune of facing, which is saying something because almost all of my work has been on Graphite. Also thanks for being an awesome programmer. I always felt a happy sense of security when I knew he was helping out on anything, for it was basically a guarantee that it would be working at the end of it.

Thank you to the test chip family: Sabrina, Yildiz, and Mahmut. We hit some snags and a couple of false starts, but we made it. Now on to newer and better chips! And to the Carbon Group, you guys are awesome. I've worked in a lot of research groups in my academic career, but this group has been by far the smartest and most awesome. I realize I have used the word awesome twice, but really it is the only word that is appropriate here.

On the flip side, my family and friends will always be the reason I do any work whatsoever. The past couple years have been tough, even by my dismal standards, but somehow you all got me through it again. I will not use names, but all of you know how important you are to me.

Do words ever suffice? Thank you so much.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As we continue to push the limits of microprocessor technology, the diminishing returns of transistor scaling have made it increasingly difficult to maintain performance gains. In response, architects have shifted their approach to multicore architectures, where several cores on a chip can provide speedup by exploiting application parallelism. Systems with as many as 64 cores have already been fabricated and tested [3], and at the current scaling pace our systems will contain 1000s of cores within the next decade [4].

The development of these systems introduces a critical problem for architects. As transistor technology continues to scale, the number of transistors on a chip increases exponentially. Historically, this has been a boon for architects, whom have happily benefited from the extra budget for logic. However, while scaling does allow us to fit more transistors per unit area - which, to be sure, allows for the development of 1000-core chips in the first place - it also increases the power density across the chip. In fact, if every transistor in

a 1000-core chip were to be active simultaneously, it would threaten to melt the silicon substrate itself.

The *dark silicon* challenge describes the phenomenon where, regardless of chip organization and topology, power dissipation constraints will limit the number of transistors that can be active at one time [5]. As a result, much of the chip must be powered off, and is unable to do any useful work; hence the term *dark* silicon. For architects, this means that energy efficiency is no longer a secondary design constraint that is only relevant for mobile and low-power devices. For future technologies, minimizing energy consumption actually allows for more transistors to be powered on, directly affecting the ultimate goal of performance.

Unfortunately, developing efficient, low-power architectures is non-trivial, and even with a working architecture, it is unrealistic to assume that it would be optimal for every possible application type. Traditionally, the job of configuring a system to optimize the competing goals of high-performance and low-energy consumption has been left to the application developer. This is usually done by extensively profiling the application on different configurations, and identifying the optimal resource distribution over the target architecture. This not only requires the expertise in the application domain, but also a deep understanding of the performance and power characteristics in the system. With the added complexity of varying application workloads, unreliable components, and the sheer number of configurable parameters in modern and future systems, this task is essentially impossible for a massively multicore processor.

One vision for addressing these challenges is embodied by self-aware systems (also known as *autonomic*, *adaptive*, *self-optimizing systems* etc.). These type of systems automatically observes its own state, and optimize performance and energy efficiency by adaptively changing the configuration of the system in real time. In this way, the system is automatically optimized as the application runs, relieving the burdensome task of

optimization from the programmer.

Furthermore, while exposing large amounts of configurability to the application developer can be counter-productive, a self-aware system lets the system expose as much of the hardware as necessary. Where a programmer would be overwhelmed by the sheer number of configuration parameters, an intelligent software-based management system can utilize all the extra parameters to make well-informed decisions.

With this new way of managing configurability, self-aware systems give rise to an opportunity for implementing large amounts of adaptivity in both software and hardware. There has already been some work in the development of adaptive mechanisms, but most were closed systems that optimized a single component: very few studies investigated integrated systems utilizing several adaptive mechanisms. In addition, there has been almost no work on the effectiveness of adaptive mechanisms in massively multicore core systems. As technology scales ever further, the knowledge of which mechanisms perform the best is extremely valuable.

1.2 Previous Work

1.2.1 Self-Aware Systems

Self-aware systems dynamically adapt the behavior of system without human guidance [6,7]. These can be implemented in hardware, as Bitirgen et al. showed by implementing a decision engine using a set of fixed-point multipliers, and observing performance traces to change underlying hardware components [8]; or in software, as the ControlWare runtime observes latency feedback and dynamically partitions web server bandwidth across threads [9]. Many systems in both hardware and software have been implemented and evaluated [7, 10, 11].

Depending on the system, the mechanisms used to enable self-awareness vary widely. In [8], the authors dynamically configured L2-cache sharing, memory bandwidth, and the total power budget for a chip multiprocessor. In another study, the authors in [1] focused on microarchitectural components such as adaptive issue queues load/store queues, reorder buffer, etc., and in yet another project the focus was solely on dynamically reconfigurable caches [12].

A common criticism of this work is that these approaches are limited to an inflexible set of adaptivity. Once these systems are built, it is very difficult to add or remove levels of adaptivity. Furthermore, the systems that are responsible for actuating the adaptivity are closed to a single level of the compute stack. For example, the engine in [8] implements a hardware-based neural network that configures the adaptive mechanisms through hardwired connections. The neural network has no knowledge of other parts of the compute stack - such as application-level algorithms or OS-level mechanisms - even though the decisions made by the neural network will likely affect the performance of components across compute stack boundaries. Without consolidating observations from all levels, it is improbable that such a system would converge to a global optimum.

1.2.2 SEEC (SElf-awarE Computing) Framework

The SEEC framework developed by Hoffmann et al. allows the consolidation of all adaptations in the form of goals, and allows adaptations to be made simultaneously at both the hardware- and OS-level [13]. For SEEC, there is no distinction between types of adaptivity, as long as the appropriate developer registers it into SEEC. For example, an OS-level mechanism such as the total cores allocated to an application would be registered by the systems programmer, whereas an application-level mechanism such as algorithm choice would be registered by the application developer. SEEC only requires knowledge that the adaptivity exists, and an associated function stub to effect the change.

We will describe SEEC in detail in Section 2.3, but we point out now that while SEEC dissolves the barriers between adaptivity at separate levels, there has been little work studying such a framework with specific adaptive mechanisms, especially on future massively multicore processors. While SEEC indeed possesses the advantage of being able to adapt across computing boundaries, it also takes on the burden of managing exponentially more adaptivity. In this thesis, we will study the performance of SEEC on future multicore systems, and show precisely the adaptations that benefit SEEC the most.

1.3 Thesis Scope

This thesis will explore adaptive mechanisms, or *knobs*, that provide useful adaptivity for future massively multicore self-aware systems. To measure the effectiveness of these knobs, we will investigate the implementation details of the SEEC (SElf-awarE Computational) framework using the distributed multicore simulator Graphite [14].

Our main metric for evaluation is the energy efficiency of a system, or the performance per watt, because as discussed future architectures bring the issue of energy consumption into the foreground. Designing an optimal architecture has always been difficult, because the resource demands of different applications can be exceedingly dissimilar. To make matters worse, the traditional approach of over-provisioning functionality to protect against the risk of underperformance is no longer realistic, because the threat of dark silicon makes the waste of energy intolerable. Architects have since allowed parts of the system to be configurable, and exposed the knobs to the application programmer so that the optimal architecture can be configured after fabrication. Unfortunately, even if an optimal system configuration existed, identifying it is a task that takes extreme amounts of time and effort, especially as the number of cores continues to scale.

Self-aware systems dynamically configure themselves on-the-fly, and rely on a runtime

controller to determine the optimal configuration. In this way, there is no tedious process of statically configuring a system, and even an initially over-provisioned system can be made to adapt to the application at hand. Self-aware systems have been an active research topic in the past decade, but with that said, the uncertainty in future computer architecture has left gaps where little work has been done. To wit, even with the failure of Dennard's scaling and the advent of core scaling, there has been little work on adaptivity on massively multicore architectures.

One key challenge is determining whether or not a knob is actually suited for a massively multicore system, because the large number of variables makes this result non-trivial. Therefore, in this thesis, we will focus on potential knobs for self-aware systems, and show results on their effectiveness on a massively multicore architecture.

For this work, we will be using the SEEC framework to enable self-awareness, but we note that many of the points we show will hold true for any self-aware controller. Any self-aware system that provides goal-oriented programming will allow the developer to specify goals instead of configuration parameters, and the otherwise tedious process of system optimization is abstracted away.

This thesis is organized as follows. Chapter 2 provides essential background on self-aware systems. Chapter 3 describes the architecture of the Angstrom project, especially the features that enable self-awareness. Chapter 4 is the focus of the thesis, where the details of the adaptive mechanisms, or knobs, are thoroughly described. Details on the implementation of the knobs will be included, as will the tradeoffs they provide to a self-aware system. Chapter 5 is the experimental evaluation section. It shows that an adaptive system outperforms a static non-adaptive system, and then evaluates the performance of the knobs when used by a self-aware controller. Chapter 6 will conclude this thesis by summarizing the contributions of this work, and describing future avenues of promising research.

Chapter 2

Self-Aware Systems

2.1 Overview

Self-aware systems (sometimes also known as *autonomic*, *adaptive*, or *self-optimizing*) attempt to automatically monitor the behavior of a system, and dynamically optimize its own configuration based on runtime conditions. With the unyielding trend of multicore scaling, and the coming to fore of hard power constraints, these systems address a critical issue: How exactly do we select the optimal system configuration for our application? Self-aware tuning of system parameters allows an application to accomplish its goals for performance and energy efficiency, and it does so without placing the burden solely on the application programmer. However, most modern systems are only proficient at dealing with a single source of adaptivity, and therefore cannot leverage the knowledge of other adaptable parameters in the system, whether synergistic or competitive. We call such systems *uncoordinated* adaptive systems, and introduce the SEEC framework that provides a *coordinated* adaptive solution.

2.1.1 Multicore Scaling and Limitations

Traditionally, managing system resources was a job delegated to the application developer. This is done by extensively profiling the application on numerous different configurations, and identifying the optimal resource distribution over the target architecture. While this was a reasonable task in the past, multicore scaling has made the space of possible resource distributions grow exponentially, and rendered this approach unusable in future systems.

Consider a single core with O(M) configurable components, each with O(N) configurations. The total space of possible configurations for this system is on the order of O(N^M). If these variables are small, as is the case for current low core-count systems, an intelligent programmer can explore a reasonable part of the search space and determine a configuration that is close to - if not exactly - optimal. However, in a system with Q cores, the total number of configurable components grows exponentially, producing a space of configurations on the order of O(N^{QM})! With the added challenge coming from the fact that massively multicore processors will likely have several applications running simultaneously, even the best of programmers cannot determine which reasonable part of the configuration space to search.

Furthermore, the challenges of multicore scaling are not limited to finding a single, globally optimal configuration. Even assuming that a programmer can determine the optimal configuration of a massively multicore system at one time, there is no guarantee that it is optimal during the entire lifetime of the application. Many applications go through several phases, and each of those phases exerts specific demands on the system, likely requiring several time variant configurations for optimality.

Consider a Fast-Fourier Transform application, where the convolution of large matrices is usually split into transpose and multiplication phases. We use the example of the *fft* application from the SPLASH2 benchmark, which uses an algorithm which performs each

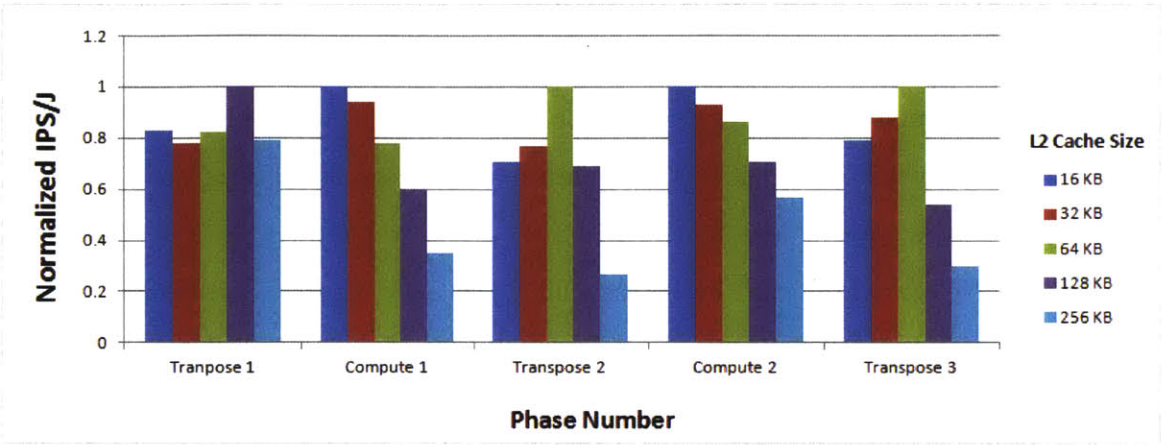


Figure 2.1: Normalized energy efficiency results for *fft* using different L2 caches, divided across phases. The target architecture has 64 cores and an 8KB L1 Cache.

FFT with three transpose phases interspersed with two compute phases [2]. Figure 2.1 is generated by the experimental methodology described in Section 5.1.1, and shows the energy efficiency of different L2 cache sizes on a 64-core architecture.

During the transpose phases of *fft*, the application is dominated by memory operations, and during the compute phases the application is dominated by compute operations. As a result, the demands on the cache are greater in the transpose phases than the compute phases, but it is evident in the figure that this relationship is non-trivial. Depending on how warm the caches are before each phase, the actual efficiency of the cache can vary. In fact, the first transpose phase shows a larger cache demand than the pursuant transpose phases. Small caches are predictably efficient during the compute phases. Due to the conflicting nature of these phases, there is no single resource distribution that can be optimal for both: if a 64KB cache was implemented, then the *Compute 1* phase is 78% optimal, whereas if a 16KB cache was implemented, the *Transpose 2* phase is 70% optimal. An ideal scenario would feature a cache that can adapt its size to the appropriate phase, so that energy consumption is optimized for the necessary performance.

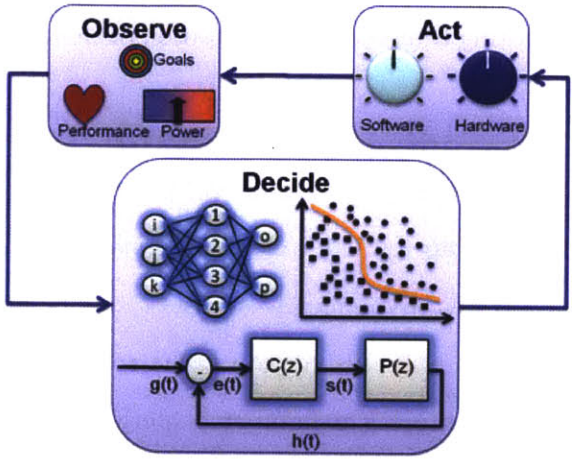


Figure 2.2: Graphical representation of an Observe-Decide-Act loop in a self-aware system.

A final challenge with multicore scaling is that it also increases the chances of unforeseen events during the execution of the application. This could be a thermal emergency that may throttle the speed of a core, or an error that restarts a part of the application. These are cases that a statically configured system has no means of resolving, but an intelligent self-aware system can deal with in stride.

2.1.2 Observation-Decision-Action Loops

The common denominator of all self-aware systems is the negative feedback loop that corrects the error between the current and target state. At a high level, this feedback loop is driven by continuously *observing* system behavior to determine how close it is to its goals, *deciding* based on these observations what it can do to achieve the goal, and then *acting* out its decision by tuning the available knobs. Hence, these loops are known as *observe-decide-act (ODA) loops* [6, 15], and are illustrated in Figure 2.2.

Regardless of the type of adaptivity available in a system, every self-aware system

contains an ODA loop. In this subsection we will show how the ODA model fits in two existing self-aware systems, noting that the implementation of the ODA loop varies drastically in literature [9, 16–18].

In the adaptive processing system described by Albonesi et al. in [1], a number of microarchitectural mechanisms are built for configurability. Depending on statistics gathered by the controller, the system dynamically adjusts the issue queue, load/store queues, reorder buffer, register files, and the instruction cache to match the utilization demand. Observations are made by monitoring a number of performance counters built into the hardware, and decisions are made to adapt the knobs when the counters trigger specific events. For example, the average occupancy statistics of the issue queue are tracked by a counter, and if a smaller queue is sufficient to hold the average number of instructions, an event is triggered and the queue downsizes; if an overflow occurs, the adaptive issue queue upsizes. This sort of ODA loop is present in many parts of the system. Figure 2.3 illustrates the adaptive processing system of [1] superimposed with the associated ODA loops. Note that, in this case, the ODA loops are built directly into hardware, and they all function independently.

Intel's Turboboost feature controls the frequency of its cores to maximize performance [19]. If a core is heavily used, Turboboost can tune up the frequency of that core. On the other hand, if a core is lightly used, Turboboost can tune down the frequency, or even power gate the core to yield even larger power savings. Observations are made by hardware counters implemented as MSRs (Machine State Registers), and the decision for the frequencies of the cores is ultimately constrained by power delivery limits, current consumption, and temperature. When Turboboost decides that the statistics are under the constraints, it steps up the frequency of the active cores. Figure 2.4 shows the associated ODA loops on Turboboost.

The Turboboost ODA loop is different from the aforementioned adaptive processing

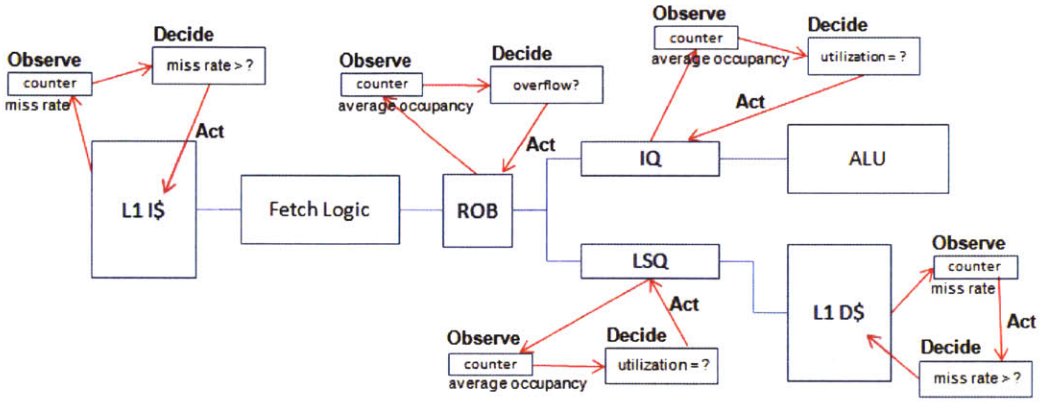


Figure 2.3: Adaptive microarchitecture in [1]. The shaded components are designed for adaptivity, and each adapt to application demands through independent ODA loops. Observations are made by built-in performance counters, and decisions are made by events triggered by the counters.

system: each adaptive component in Turboboost does not have its own independent ODA loop. Instead, there is a single decision engine that collects all the observations from the cores, and makes a decision on the core frequencies based on the state of the global system. Nevertheless, both Albonesi's adaptive processing system and Intel's Turboboost are equally valid self-aware controllers, and both rely heavily on ODA loops.

2.2 Uncoordinated Adaptive Systems

Most self-aware systems, such as those explained above, do very well in their sphere of influence. A major drawback, however, is that because they only have knowledge of their own knobs, they do not work well with other sources of adaptivity. For example, in the adaptive processing system described in the previous section, all of the adaptivity is built into hardware, and so application-level goals are beyond the scope of its observations. Software-based approaches, on the other hand, will assume that the hardware is fixed and make no attempt to coordinate with these low-level adaptations. These systems are closed to

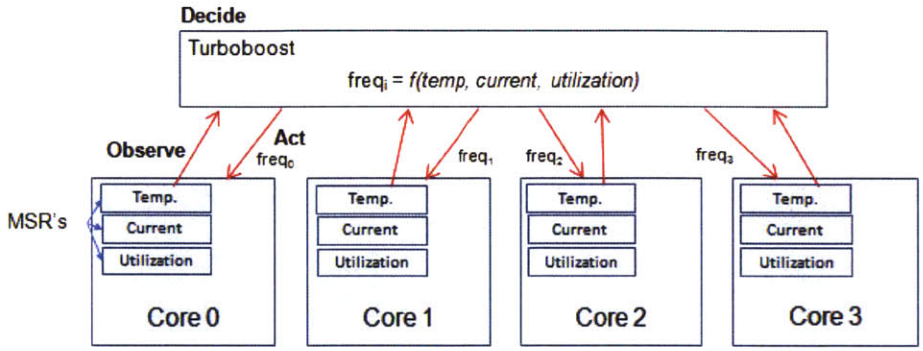


Figure 2.4: Intel Turboboost ODA loop. Core frequencies are determined by a global controller that makes decisions using observations on all cores in the system.

other levels of adaptivity, and cannot coordinate with others; we refer to such systems as *closed/uncoordinated adaptive systems.*

Of course, this is not an issue if there is only one single adaptive mechanism in the system. However, future systems will have many resources that are capable of being adaptive, and the impact of each knob on the other cannot be ignored. Simply running multiple ODA loops in hardware and software simultaneously is not a valid solution either, because even with both loops targeting the same goals, it is almost impossible for each loop to make intelligent decisions without understanding the impact it has on the rest of the system. For example, two closed ODA loops running at the same time and targeting the same application performance goal might realize the system is underperforming, and in response both allocate extra resources. More likely than not, the next observation by the controller will find that the system is over-performing, because both loops independently added enough resources to make up the performance deficit and over-provisioned the system. In response, both controllers remove resources from the application, and the system never converges to an optimum.

The root cause of this problem is the existence of configurations that are sub-Pareto optimal. Consider a system with the competing goals of performance and energy: a Pareto

optimal configuration is a configuration that runs with maximum performance and minimum energy. With the sheer number of configurations available in a massively multicore processor, the fact that some combinations of those knobs are sub-Pareto optimal configurations is not surprising. However, in general, Pareto-optimal points are the only ones that a controller in a self-aware system should ever consider. Unfortunately, without a sense of how other knobs behave, an uncoordinated controller has no way of gauging the global Pareto optimality of its actions.

To illustrate this, we explore the behavior of the *barnes* application from the SPLASH2 benchmark suite (see Section 5.1 for details on methodology) on a multicore system with two knobs: the total number of cores assigned to it (from 1-64, by powers of 2), and the size of the last-level cache on each core (from 16-256KB, by powers of 2). We simulate each configuration on the Graphite simulator [14], measure application performance in instructions per second, total energy consumption in joules, and plot the normalized results in Figure 2.5. The solid diamond points represent all simulated configurations; the squares show configurations that appear optimal for a closed system which only considers cache adaptations; and the triangles show optimal configurations for a system that only considers core allocations. The Pareto-optimal frontier is depicted by those diamond points that are connected together. Notice there are triangles and squares that appear below the Pareto frontier: these points represent configurations that uncoordinated systems would believe to be optimal, but, in fact, are sub-optimal for the overall system. For the closed systems, these states are Pareto-optimal within its own space of possible configurations, but they are clearly sub-optimal in the global system.

To avoid sub-optimal configurations, adaptivity needs to be considered as a *first class object*, so that all knobs are considered globally instead of in piecewise, uncoordinated objects. In a coordinated self-aware system with a global view of all the knobs in the system, sub-optimal configurations can be filtered out and discarded, leaving only the

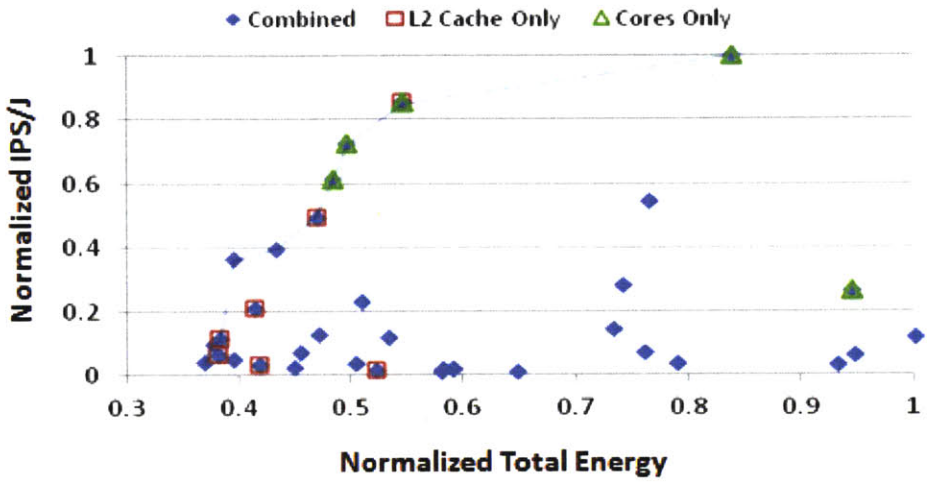


Figure 2.5: Normalized efficiency of barnes. The Pareto frontier is shown as the blue connected line.

Pareto-optimal configurations as actions for the ODA loop (the Pareto frontier).

2.3 SEEC Framework

In [15], Hoffmann et al. proposes the SEEC (SElf-awarE Computing) framework. In the SEEC model, there are three distinct roles with distinct responsibilities:

Role	Responsibilities
Application Developer	Specifies application goals.
Systems Developer	Specifies actions in the form of subroutines that tune the knobs.
SEEC Runtime System	Observes the system and determines the amount of speedup needed to achieve goal; decides on appropriate actions

Table 2.1: Roles and responsibilities in the SEEC model.

Note that the application developer is completely relieved from the optimization task, and needs only to set the performance goal that he/she desires. All adaptive knobs are registered into the SEEC framework by the systems programmer, who is exactly the

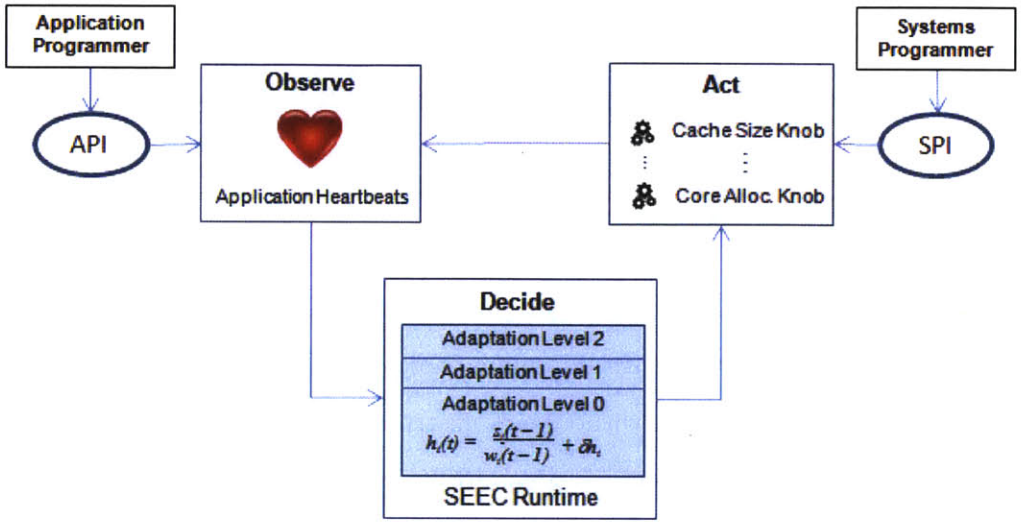


Figure 2.6: The SEEC model.

individual most familiar with a systems underlying adaptivity. Since all the knobs are registered in one place, SEEC filters out all sub-Pareto optimal configurations that would otherwise plague an uncoordinated adaptive system.

In this thesis, we will be using SEEC as the self-aware controller for our architecture, and use it to explore the effectiveness of the proposed knobs. A full description of SEEC can be found in [13], but is mostly beyond the scope of this thesis. Instead, we will describe only those features that are important for our interests. Figure 2.6 serves as a reference for the following subsections.

2.3.1 Observation

Application goals are specified through an API provided by the SEEC framework, which provides an abstraction of performance in the form of an Application Heartbeat [20]. The application developer instruments the application to emit heartbeats at important intervals, and sets the desired heart rate. The programmer can also specify the size of a rolling

window, which acts as a low-pass filter to smooth out the heartbeat data. This abstraction level allows the programmer the purchase for setting fine-grained goals, without having to understand the lower level features of the system itself.

It is interesting to point out that, unlike the self-aware systems we have already discussed, goals in SEEC are specified directly by the application developer. In the previous discussions, systems are optimized with extreme goals: run an application with maximum performance, run an application with minimum energy consumption, etc. While this does mean that these systems can run without any instrumentation in the application, it also means that there is no way for these systems to target application-specific goals. For instance, consider the target performance for a video encoding application: it could be 30 frames per second, 40 frames per second, or any number deemed necessary by the developer; there is no way for the application developer to specify such a goal. On the other hand, SEEC assumes that the application developer has the most knowledge of application demands, and provides the heartbeats interface to communicate these goals.

2.3.2 Decision

SEEC's decision engine is divided into several levels. At the simplest, or Adaptation Level 0, the decision engine is a basic model-based feedback controller. This controller continuously monitors the heart rate *h(t)*, and compares it with the target heart rate g to compute the required speedup *s(t)*. Using the set of actions made available by the system programmer and their associated models, SEEC decides on the appropriate action (or combination of actions) to move the system closer to its goals. This basic control system is the backbone of all the engines available.

The next level of sophistication, or Adaptation Level 1, addresses the issue of workload estimation. While the heart rate, *h(t)*, measured by SEEC is a general indicator of

the latency between heartbeats, it is not a direct measurement of application performance. This is because the actual application performance also depends on the actual work done. That is, the heart rate *h(t)* can increase either because the system is performing more efficiently, or the actual work being done for the application has decreased. In Adaptation Level 0, the work done is assumed to be time invariant, but this is patently untrue for some applications (*i.e.,* a video suddenly becomes more difficult to encode). In Adaptation Level 1, a 1-dimensional Kalman filter is used to estimate the workload of the application. This increases the speed of convergence substantially.

Adaptation Level 2 addresses the accuracy of the models provided to SEEC. While the Kalman filter can estimate application workload, it cannot differentiate between cases where workload actually changes, and cases where the action models provided by the systems programmer are inaccurate. The fact that the action models could be erroneous is not unlikely, because a single model for a knob is unlikely to be appropriate for all applications (although we discuss including multiple models per knob in Section 6.2). As an example, consider the knob that tunes the total number of cores available to an application. Obviously, depending on the application, the limits of parallelism will provide different speedup tradeoffs. Incorrect models imply that SEEC will select actions that are sub-Pareto optimal, because it will adjust to changes in estimated workload that might not exist. In Adaptation Level 2, another Kalman filter is used to estimate the actual cost and benefits of an action, and modifies the models if it finds them incorrect. In this way, SEEC gradually learns the correct action models, and will be able to recompute Pareto-optimal actions on-the-fly.

2.3.3 Action

The systems programmer specifies the action model for a knob by providing a set of actions A_0, A_1, .., A_n, and associating a relative speedup and cost over the baseline action A_0, which by definition has a speedup and cost of 1.

Action	Speed-up	Cost	Description
A_0	1.0	1.0	1 allocated core
A_1	2.0	2.0	2 allocated cores
A_2	3.0	3.0	3 allocated cores
...	...	...	...

Table 2.2: Example action model for core allocation knob.

For example, to specify actions for the knob that tunes the number of cores available to an application, the baseline action A_0 represents the allocation of a single core. Action A_1 might represent the allocation of two cores, and the associated speedup and cost could be set as 2.0, which represents double the performance and cost, and so on. This is shown in Table 2.2.

The systems programmer provides a set of actions for every knob, and SEEC interprets the combination of multiple knobs as the product of the associated speedup and cost for each constituent action. The flexibility of this framework allows several different systems programmers to provide action models for the knobs they are most familiar with. It also allows the removal and addition of knobs as is appropriate, which allows SEEC to use new knobs or a subset of existing knobs. If the models are accurate, SEEC will rapidly converge to the optimal configuration to achieve the specified goals. In the case that the models are inaccurate, either due to a single knob or a combination of them, the Adaptation Level 2 in SEEC's decision engine will resolve the inaccuracies.

Chapter 3

Angstrom: A Massively Multicore Self-Aware System

3.1 Architecture Overview

Angstrom is a large, cross-disciplinary project that aims to create a fundamentally new system to meet the challenges of extreme-scale computing. Solutions are posed for challenges at all levels, from circuits, to architectures, to operating systems, to applications. For this thesis, we will be using the Angstrom proposal as the backbone architecture for our study. The full specification of Angstrom is, of course, well beyond the scope of this thesis, but there are two important features of Angstrom that are relevant to our interests.

First, Angstrom is proposed as a massively multicore processor, with up to a thousand cores. This proposal comes from a projection of the multicore scaling trend, which has seen the number of cores double every three years since 2006 [4]. Cores in Angstrom are organized as a tile-based architecture, connected by a mesh network with endpoints at

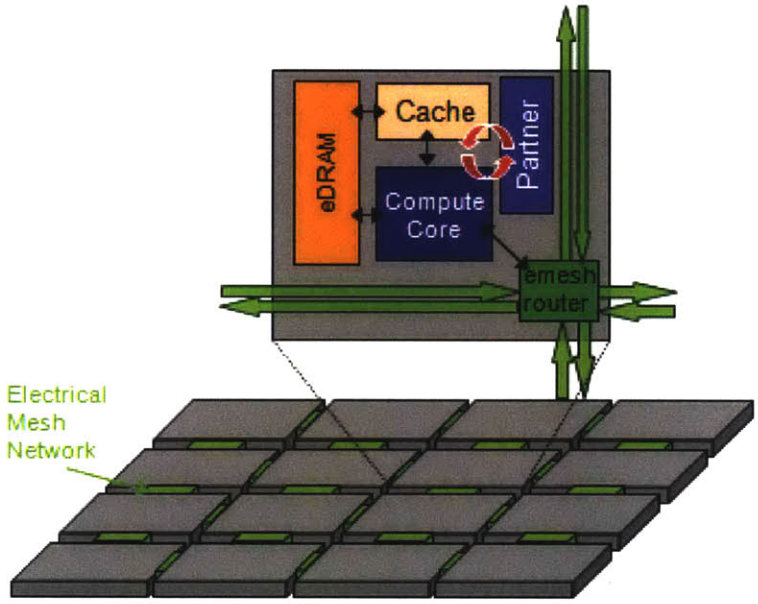


Figure 3.1: The Angstrom architecture.

each tile. Figure 3.1 illustrates a basic picture of the Angstrom architecture.

Cores in Angstrom are much simpler relative to modern processors, which otherwise have large re-order buffers, superscalar issue width, and a considerable number of pipeline stages. These architectural features are useful because they substantially improve performance, but come at the expense of high energy consumption. For future technologies, where transistor density will increase exponentially, the tradeoff of energy for performance is not as forgiving as in previous generations. In fact, the heat dissipated from the transistors pose a hard limit to the total number of active components, forcing large numbers of transistors to be powered off at any one point in time. Studies predict that at 8nm, up to 50% of a chip could be forced to be powered down, coining the term *dark silicon* to describe devices that cannot utilize all its constituent transistors simultaneously [5]. Therefore, it is important for a massively multicore system like Angstrom to be extremely energy-efficient. Notwithstanding, the actual hit in performance from removing the complex architectural features

is not fully realized anyway, because even if they were left in place, most of them could not be powered on for doing useful work. Adding to the fact that a massively multicore system targets parallel applications instead of sequential ones, then simple in-order cores are perfectly reasonable for Angstrom.

The second feature of Angstrom that is relevant to our interests is that it is designed to treat adaptivity as a first-class object. This means that all components of the architecture are designed to contribute to the specification of the ODA loop. As such goals are specified through a single interface, the self-aware controller will have purchase to adapt all levels of adaptivity to accomplish them. Angstrom uses the SEEC model, and exposes a wide array of both hardware and software observations and actions to the SEEC runtime. In the following sections, we will describe in detail the features in Angstrom that contribute to the ODA loop.

3.2 Observation

To provide SEEC appropriate feedback to model the tradeoffs associated with its knobs, Angstrom provides a means to monitor both performance and power. As we will describe, Angstrom goes beyond normal architectures to provide a level of introspection for SEEC that would be impossible on a traditional system.

3.2.1 Performance Monitoring

The main interface for setting goals and observing the performance of the system is provided by the Application Heartbeats API described in Section 2.3.1; this is an architecture independent feature, and is not reliant on Angstrom in any way. However, beyond the application-level performance monitoring enabled by the Application Heartbeats API,

Angstrom also provides several hardware features that provide valuable insight into the application behavior on the underlying architecture.

The first such feature is the implementation of performance counters. Of course, existing systems already possess a number of performance counters, but most of them either impose strict limitations to the number of counters that can be monitored simultaneously, or require heavy kernel interaction for each access [21]. As a result, either multiple profiling runs are required to comprehensively pre-analyze an application, or the programmer is left with limited and possibly out-of-date information about the system.

In Angstrom, all performance counters are exposed directly to software by mapping a portion of memory dedicated to observation hardware. As such, all levels of the software stack have access to all of the counters. This not only allows SEEC to gather information from all the counters at one time, but provides low enough overhead per access to allow SEEC to dynamically analyze the information during runtime. Counters enumerate any events that may provide insight to application behavior. Some examples include: the number of cache hits and misses, pipeline stall cycles, network flit traffic, etc. Figure 3.2 illustrates the architectural design of the performance counters.

Since performance counters are only enumerators, it is far too expensive for a runtime system to continuously poll them for information, especially if it is trying to observe the occurrence of a rare event. For such cases, Angstrom provides *event probes* that can be associated to a performance counter, or some other architectural state. These event probes contain a programmable comparator that continuously watches for a specific state, and triggers an interrupt when the event occurs. Event probes can be set to different operations - such as equal to, greater than, or less than etc. - and may be masked to only watch selected bits. In any case, programmable event probes effectively move the busy-polling loop from software to hardware.

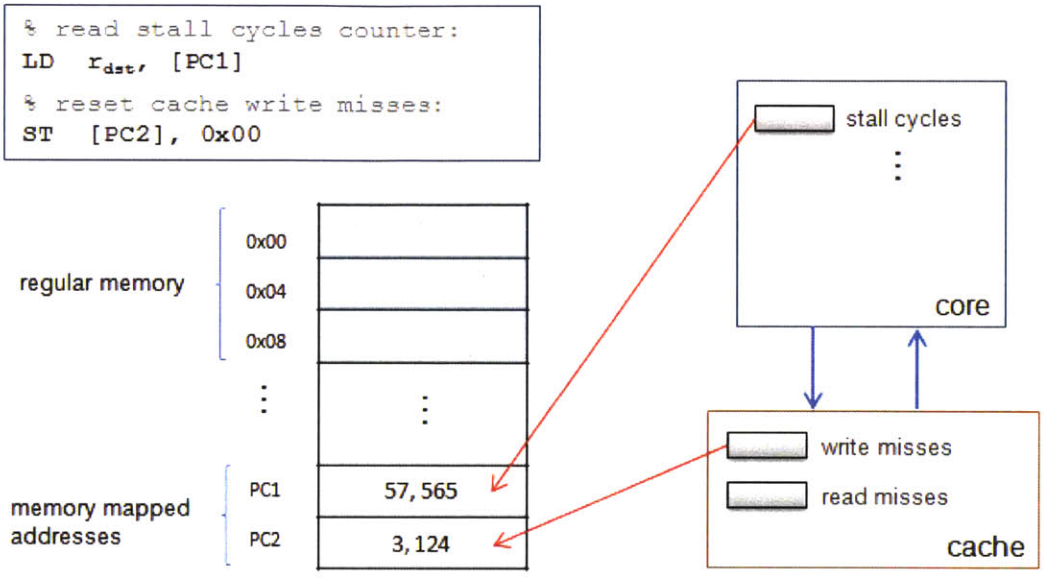


Figure 3.2: Performance counters are memory mapped, access is made through regular load/store instructions.

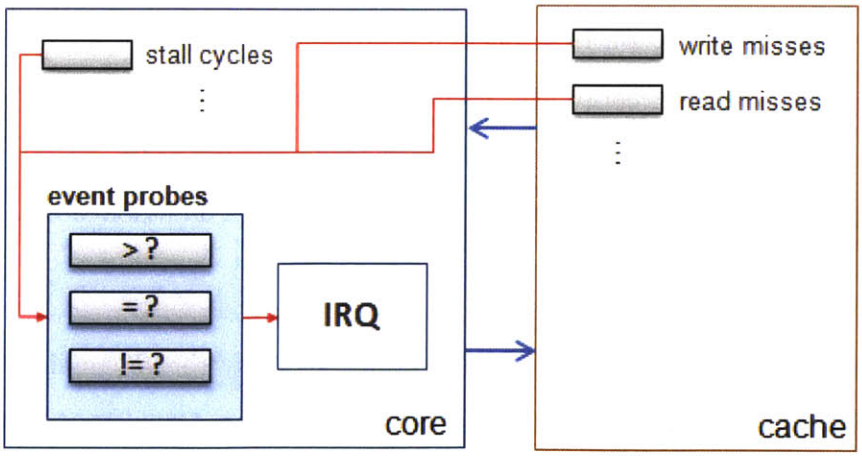


Figure 3.3: Event probes allow observation of rare events without polling from software.

3.2.2 Energy Monitoring

To effectively monitor the performance versus power tradeoff, the self-aware runtime requires some sort of energy consumption feedback. Accurate measurements of energy allow the controller to gauge the actual effect of its actions during exploration. In modern systems, power can be measured either by placing current meters at major power delivery inputs of the system, or through an ad-hoc power model developed to estimate the energy consumption of the specific architecture. The former approach fails to provide fine-grained information on energy efficiency, and the latter is only as accurate as the model, which can be especially vulnerable to rare corner cases, or unforeseen events such as thermal emergencies.

Accurate, fine-grained power monitoring is key to architectural adaptivity because it shows exactly which components of the system are the least efficient, allowing SEEC to target knobs for improving efficiency on exactly those components. Otherwise, SEEC runs the risk of unknowingly sacrificing the performance of components that might otherwise be efficient. For example, consider an application that is compute-bound and uses very few cache accesses, then the focus of optimization should be localized around the computational part of the system. Application Heartbeats has no means of conveying this information a priori; without fine-grained energy measurement, it is up to SEEC to discover the energy tradeoff through exploration of the configuration space in both components. To be sure, SEEC will eventually converge to the optimal configuration, but power observation can improve the performance of the SEEC runtime.

To motivate fine-grained energy observation, we study the *fft* application first described in Section 2.1.1. This Fast Fourier Transform algorithm alternates between a transpose and computation phase, which alternately stresses different parts of the core architecture. Figure 3.4 shows the dynamic energy breakdown of *fft* across the two phases. During

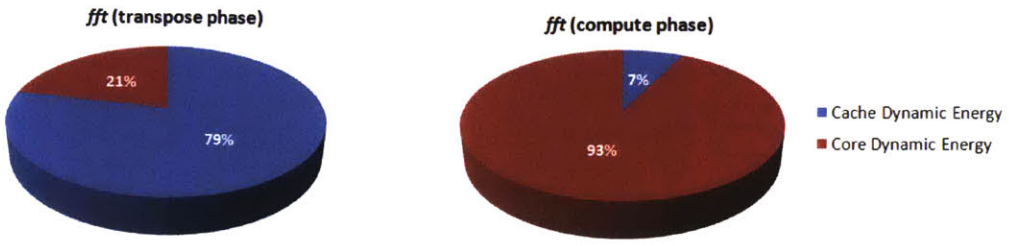


Figure 3.4: Breakdown of cache and core component energy between phases in the *fft* benchmark.

the transpose phase, when *fft* is focused on memory operations, the core pipeline spends most of its time stalled. The results show that during this time the core consumes only 21% of dynamic energy, but core dynamic energy consumption jumps to 93% during the compute phase. This information is extremely insightful, as it can direct the optimizations in a very clear direction, allowing any self-aware controller to converge to an optimal configuration much faster.

This leaves the question of implementation. One way to provide this feature is to implement circuits to monitor energy directly on-chip. One approach is to track the current through the component under measurement, and the number of times it charges a pre-defined capacitance. This number can then be converted to a power measurement by comparing dividing the count with the system cycle counter:

$$P = \frac{\#\ cap\ charges \times V_{cap}}{\#\ cycles\ elapsed \times f}$$

The counters that track the number of capacitor charges and the system clock cycles are exposed to software as memory mapped locations, much like performance counters. Therefore, accessing power information has the same overhead as any regular load or store for memory mapped registers. Note, to allow for power measurements during specific win-

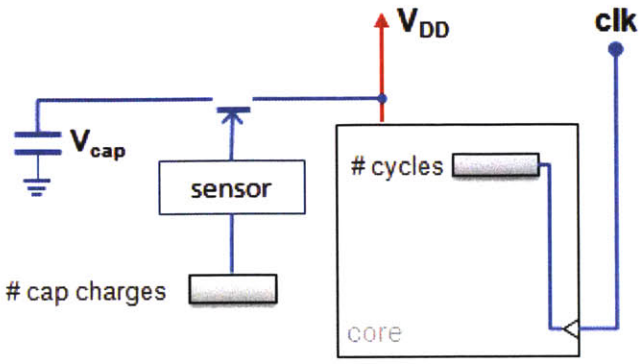


Figure 3.5: Block diagram of energy monitor circuit.

dows of time, the energy monitoring register can be reset. Figure 3.5 is a high-level diagram that shows how such a circuit is implemented.

By integrating multiple energy monitoring circuits within a single tile, Angstrom provides a granularity of observation to SEEC never before available. Any such number of these energy monitoring circuits can be implemented, limited only by the area available. One concern is that current prototypes of this circuit use an off-chip capacitor to serve as V_{cap}. This may stress the I/O constraints of the physical chip, and add considerable routing complexity to connect the I/Os to every single energy monitoring circuit. However, work in integrated switched capacitor technology has been promising, and can potentially be used as a replacement for off-chip capacitors.

3.3 Decision

As we described in Chapter 2.3.2, the SEEC runtime has a decision engine that dynamically converts observations from the system into actions for its knobs. Unfortunately, this type of self-awareness does not come for free: the decision engine is real code, and must be given some execution context to run upon. Angstrom provides a number of places to run the

SEEC code:

(i)	Instrumented Source	The decision engine code is instrumented into the same context as the application.
(ii)	Same Core	The decision engine code is run in a separate thread on the same core, and receive heartbeats through the built-in API.
(iii)	Separate Core	The decision engine code is run in a thread on a separate core.
(iv)	Partner Core	The decision engine code is run in a separate thread on a specialized, low-power core.

The efficiency of each approach can be summarized by two key metrics: communication latency and application slowdown. Communication latency is related to the speed in which SEEC receives observable data, and the speed that it can effect change to the tile containing the application; this is important because it directly affects the speed in which SEEC can make adjustments. Application slowdown, for obvious reasons, is undesirable in terms of application performance. Table 3.1 summarizes each approach.

	Communication Latency	Application Slowdown
Instrumented Source	Low	High
Same Core	Low	Low to High
Separate Core	Med to High	None
Partner Core	Low	None

Table 3.1: Characteristics of SEEC decision engine placement.

The *instrumented source* approach epitomizes low communication latency because it is co-located with the application thread, but suffers from slowdown because the application must stop to allow the SEEC code to run. In the *same core* approach, or similarly an SMT approach, communication is still cheap because hardware is still shared between the threads. The application slowdown, however, is application specific. Slowdown could be kept low if the scheduler is able to execute the SEEC runtime during periods where the application thread is otherwise stalled. In cases where this is not possible, the application thread must stall and wait for the SEEC thread to compute its decisions, resulting in substantial application slowdown. On the *separate core* approach, the SEEC thread is completely

decoupled from the application, allowing it to make its decisions asynchronously. The communication latency can be variable, as it is a function of the distance between the cores executing the SEEC runtime and the application. At best, the SEEC runtime thread is scheduled on a tile adjacent to the application, which is inherently slower than the previous two approaches but still reasonable. However, in the case where the SEEC runtime is scheduled on a tile several network hops away, the communication latency grows quickly.

To address all these issues, Angstrom provides a novel architectural feature, called *Partner Cores*, to allow for low-latency communication and no application slowdown. At a high-level, a partner core is simply a small, low-power core on the same tile as the core running the application. The uniqueness of the partner core is its emphasis on low power rather than performance, and the way in which it communicates to the application core. To minimize the power consumption and area of the partner core, each partner core includes a low-power pipeline with very few architectural frills. Such a pipeline will be much slower than the application core, but this is acceptable because SEEC makes its decisions asynchronously and can avoid application slowdown. To observe and communicate with hardware, the partner core makes use of direct hardwired connections to the main cores hardware, resulting in bare minimum communication latency (Figure 3.6). Details of the partner core architecture are described in [22], so we will not include them here.

3.4 Action

The Angstrom processor provides several adaptive mechanisms, or knobs, that can be tuned on-the-fly by SEEC to reflect the speedup changes deemed necessary by its decision engine. To register knobs with SEEC, the programmer must associate each knob configuration with a speedup and cost, and then provide a valid subroutine that modifies the values in the appropriate memory-mapped locations. Depending on where in the compute stack a knob

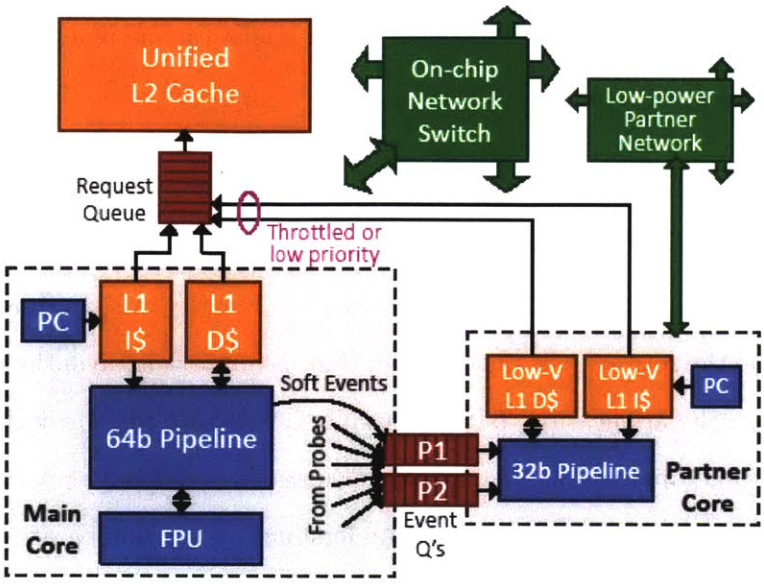


Figure 3.6: Architecture of a single tile in the Angstrom architecture, showing the main and partner cores.

is located, knobs may be registered by different developers.

If a knob is built in hardware, such as a knob that tunes the cache size or instruction issue width, Angstrom will map the relevant control hardware into memory. Thus, a normal load and store to the appropriate location can directly adjust the underlying hardware. The process of creating these hardware interfaces is the responsibility of the architect, and is completely invisible to the application developer. A detailed example of how such an interface is built will be given in Chapter 4.

If a knob is built in software, such as a knob that schedules threads on cores, the system programmer must again provide the appropriate subroutine and associated speedup and cost. Unlike hardware-based knobs though, which are fixed with the specific processor architecture, there is no limitation on the number of software-based knobs. To be sure, designing a software-based knob requires deep knowledge of the system, and likely also

requires kernel-level permissions, but there is no physical limit constraining the number of software-based knobs. In this way, software-based knobs can be much more flexible than hardware-based knobs.

Staying faithful to the definition of an open adaptive system, all knobs - whether based in software or hardware - are dealt wholly by the SEEC runtime. This adds another level of flexibility to Angstrom: through SEEC we can identify exactly those knobs we wish to tune, and leave the rest at their initial state. This is important when the programmer has a priori insight on the application, but is normally invisible to SEECs decision engine. For example, a real-time application that possesses unstable performance patterns might pose a challenge for SEEC. If the programmer understands which knobs are most important to the application, then he/she can identify those knobs through SEEC to improve the convergence behavior. Since real-time applications have hard deadlines, the improvement in convergence time can be pivotal to completing on-time.

This leaves the question of what type of knobs are actually available. In the design of Angstrom, this is still an open question, but it is this question that this thesis attempts to answer by demonstrating knobs that are useful, and providing a framework for measuring their effectiveness. In the next chapter, we will explore the implementation of knobs that were found to be useful in a massively multicore system.

Chapter 4

Knobs for Self-Aware Multicore Systems

4.1 Caches Associativity and Size

4.1.1 Motivation

As technology scaling continues to increase transistor density in silicon, processor design has tended towards using those transistors for larger on-chip caches [23, 24]. However, the performance improvements from large caches come at the expense of increased energy consumption. While this tradeoff is fine for a well-utilized cache, caches that are not well-utilized will waste substantial energy. Due to the fact that caches are usually implemented as DRAMs, periodic refreshes are required for all cache lines, even if there are lines that are unused. This issue is exacerbated by smaller technologies, because the narrowing of the transistor channel creates an even higher rate of leakage, requiring even higher refresh rates.

In a perfect world, a cache would be sized to match the needs of the application

programmer. Unfortunately, there is no single cache size that would be optimal in general, because cache utilization depends on the application-specific *working set size*. The working set of an application represents its temporal locality, which is a function of the amount of data it requires during execution and the way data is distributed in main memory. To design an appropriately sized cache, it is important to understand the working set sizes of the applications it is designed to serve.

Beyond working set sizes, the specific access patterns of an application will determine its miss behavior in the cache. In cases where an application reuses large blocks of memory that share the same cache line, a highly associative cache is ideal. On the other hand, if the required cache lines do not collide in the cache, the extra overhead used to maintain the associativity of the class is wasted.

In the following sections, we will describe in detail the implementation of two mechanisms for cache adaptivity. The first knob adjusts associativity by powering down ways, and the second adjusts the total number of sets per way by powering down blocks. In both cases, the total size of the cache is changed, allowing SEEC to adapt the cache for both the working sets sizes and access patterns of an application.

4.1.2 Tradeoffs

Working Set Size

Working set size can vary widely between applications. In Figure 4.1, we reproduce a study of working set sizes in the SPLASH2 benchmark suite [2], which depict the cache miss rate of each application as a function of cache size. The points of inflection (or *knees*) in the figures correspond to the cache sizes where the working set of an application fits entirely in the cache. Knowing this, it is clear that the working set of an application can vary by

orders of magnitude. Consider *radiosity*, where the first knee is found at 16KB, compared with *radix*, which has its first knee at 256KB - a 16x difference!

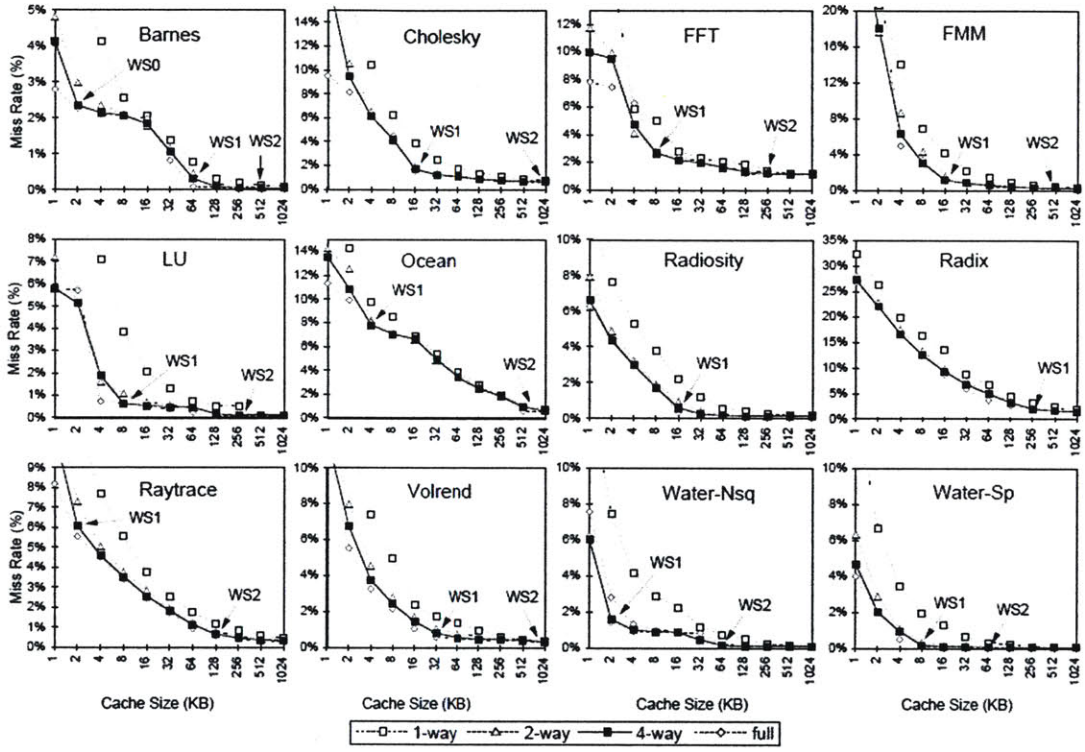


Figure 4.1: Miss rate vs. cache size and associativity for the SPLASH2 benchmarks, taken from [2].

Since it is impossible to provide a single cache for all varieties of working sets, we implement as large of a cache as possible, and allow its size to be dynamically tunable. This makes a larger cache available for applications that possess a large amount of data locality, but the flexibility of reverting to a smaller cache to save power for applications that do not.

This knob also provides SEEC the license to adjust caches for time-variant working sets. Figure 4.1 shows that multiple benchmarks actually contain multiple points of inflection, representing the fact that an application may have multiple working sets. In fact,

it is not uncommon for an application to have a hierarchy of working sets. One simple scenario where this is possible is an application with nested loops: if at each level the loops require specific but partially shared data, then there is a tiered hierarchy of working sets. Another example: if an application experiences several sequential phases containing different amounts of data locality, then it will have multiple working sets spread out in time. In these cases, a statically configured cache cannot remain optimal, whereas the cache knobs allow SEEC to maintain cache optimality for the entire lifetime of an application.

Lastly, Figure 4.1, also shows us how the miss rate can change with associativity. In most cases, increasing the associativity from 1-way to 2-way provides a stark improvement in miss rate, whereas increasing from 2-way to 4-way the change is less pronounced. The exact tradeoff for increasing associativity lies in the extra energy, area, and time required for a cache line access. However, suffice it to say that the relationships between miss rate, associativity, and these inherent cache properties are extremely complex, and difficult to predict by an application programmer. In fact, what may be an expected trend, where increasing associativity should improve miss rate, is not even true for all applications. In *fmm*, the fully associative cache actually performs worse than the 4-way associative cache. The impact of associativity on working set size is not predictable in general, and hence it can be an attractive knob for a self-aware controller like SEEC, but not for an application programmer.

Miss Behavior

To optimize a cache, we must implement the smallest cache possible which retains as low of a miss rate as possible. Therefore, the miss behavior of an application is incredibly important to the ultimate configuration of the cache. A miss in the cache can be categorized into three categories:

Miss Type	Cause
Cold Misses	Memory locations that have never been accessed will always miss in the cache.
Conflict Misses	Memory locations that have been evicted due to the replacement policy.
Capacity Misses	Memory locations that are evicted because of the finite size of the cache.

Table 4.1: Miss behavior and causes.

Cold misses can be addressed by the cache line size, but dynamically adapting the line size has unfortunate implications to the cache mapping, and requires the entire cache to be flushed. Memory prefetching is also a technique that can address cold misses, which we explore in another paper [22]. However, we will not investigate this type of cache miss any further in this thesis.

Conflict misses are those that are closely related to the associativity of a cache. These misses are caused be evictions from the *mapping function*, and the corresponding *replacement policy*. Providing associativity allows locations that map to the same cache line to be stored in separate *ways*, so that multiple lines are simultaneously valid for the same cache address. Increasing the associativity of a cache comes at the expense of higher area, energy consumption, and latency per access, because a line from every way must be checked per access. Further, if the same total cache size is to be maintained while increasing associativity, the number of sets per way must decrease, making the cache more susceptible to capacity misses.

Lastly, capacity misses are closely related to the total number of sets in the cache. If an application accesses a large amount of memory, then there will be misses resulting from the lack of capacity. Having higher associativity will improve miss rate, but add additional dynamic energy overhead, whereas simply adding to the number of sets would accomplish the same without adding energy consumption to each access. Therefore, for an application that demonstrates this sort of behavior, the best configuration is a cache that is big enough to fit the working set with minimal associativity.

We will exploit the latter two types of misses in the form of two different knobs. The first is *associativity adaptivity*, which dynamically adjusts the associativity of the cache by powering down entire cache ways when necessary; the second is *set adaptivity*, which dynamically adjusts the size of the cache by powering down the sets when necessary. As we will show, the associativity increases the dynamic energy consumption per access, so the energy tradeoffs between the two knobs also differ. It is up to SEECs decision engine to decide the appropriate amount of associativity that performs most efficiently for a specific application.

All novel architectural features described in this section are feasible, as we have designed this cache in RTL and have shown them working in hardware simulation.

4.1.3 Implementation

Associativity Adaptivity

The associativity adaptivity knob allows SEEC to adjust the associativity of the cache, giving it the ability to improve the conflict miss behavior for an application. In addition, it also allows SEEC to control the energy tradeoff, because powering down cache ways decreases both static power and dynamic energy per access.

In general, caches are organized in several banks surrounded by control logic. There are several types of bank organizations, but for our purposes Figure 4.2 details the architecture of the cache we will analyze.

In this design, each cache bank matches the width of a single word (32- or 64-bit), and each way is associated with a separate bank. To access a word in a bank, the appropriate word line is charged across all ways, and the corresponding bit lines are driven by the contents of SRAM cell. Since banks usually contain a considerable number of lines, it is

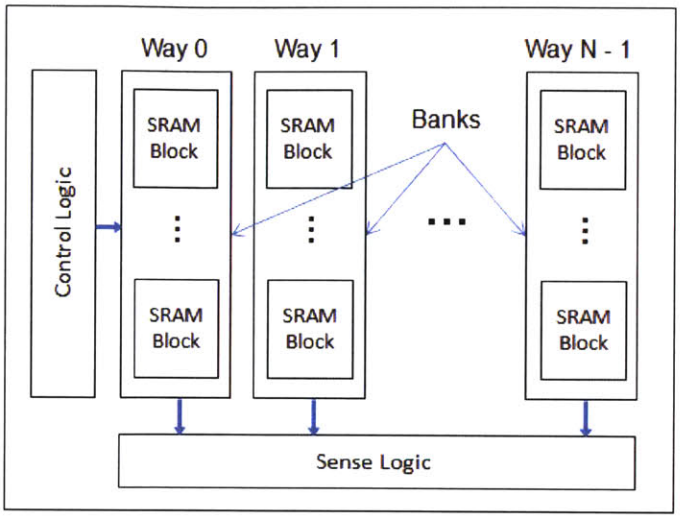


Figure 4.2: Cache bank organization with surrounding control and sense logic.

difficult to fabricate a single consistent SRAM block for an entire bank. Instead, SRAMs are provided as smaller blocks, and together they form the banks of the cache.

In an associative cache, writes occur by accessing a single way, so the access energy per write does not change with greater associativity. For reads, however, an associative cache accesses every way in parallel, and the data is compared with the tag of the read address. Therefore, an N-way associative cache in general consumes $O(N)$ times dynamic energy than a direct-map cache.

To power down a way, the corresponding SRAM blocks are powered down. This requires an extra input to each SRAM block that will gate the voltage and clock inputs of the block, and is implemented at the circuit-level. What is left then is to allow software the ability to control these SRAM blocks, so that SEEC can dynamically tune the associativity of the cache.

As we mentioned before, the control of hardware knobs is mapped in memory, so SEEC controls the associativity adaptivity knob through regular memory load/stores to

a special status register. Once a cache way powers down, there are two tasks the status register is used for:

(i) Determining the number of ways available for a write.

(ii) Decoding the power signals to the appropriate banks.

In *(i)*, when the cache controller determines a way to write its target data, that way must be guaranteed to be powered on - exceptions to this rule would cause the data to be lost. To make this guarantee, a decoder is used to mask the signal generated by the cache controller that indicates the way to be written. Figure 4.3 shows a simple method to convert a 4-way associative cache to a 2-way associative cache. In this figure, the decoder is simply a modulus function where the base is controlled by the associativity stored in the status register. As the associativity was initially 4-way, the replacement policy in the cache controller decides that the next write should target Way 3. However, knowing from the status register that the associativity has changed to 2-way, the new target way is 3 (mod 2) = 1. To be sure, the resulting evictions may violate the replacement policy temporarily, but the cache is functionally sound, and the policy is eventually upheld with sufficient time. Alternatively, the decoder can be as complex as desired by the architect, and made to always honor the replacement policy. However, keeping with the theme of energy efficient architectures, a simple decoder like this one is likely more desirable.

Likewise, in *(ii)*, a simple encoder controlled by the status register provides the power signals to the appropriate banks. Figure 4.4 shows how the associativity value in the status register is encoded as a 4-bit vector to the power inputs of the banks when the associativity changes from 4-way to 2-way. Again, more complex encoders can be implemented to select specific ways for power down, but a simple scheme is to turn off the higher order ways.

Lastly, to maintain consistency in the cache, ways must be guaranteed to be completely empty when they are powered on. This is important because cache lines that were valid in powered down ways could have be replaced during the time they are off. Other-

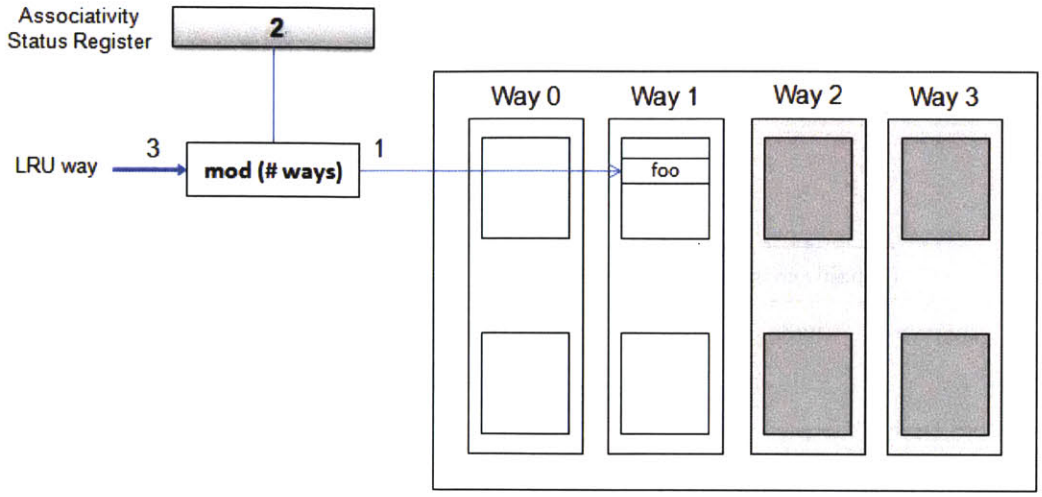


Figure 4.3: A 4-way associative cache converted to a 2-way associative cache. A write access intended for Way 3 is decoded to Way 1 due to the new associativity.

wise, if a way powered up with valid data, then there would be cache lines living in separate places. This task does not require much implementation work, as caches need to be flushed during their initial power up anyway; the only extra requirement here is that flushes can be made at a per-way granularity.

Set Adaptivity

The set adaptivity knob allows SEEC to adjust the size of the cache by adjusting the total number of sets in the cache. This is useful for data that does not generate a lot of conflict misses, but would fit into a smaller cache size. This way, the total size of the cache can be changed without changing the underlying associativity. Unlike the associativity knob, dynamic energy per access does not change when adjusting the number of sets, but shrinking the total number of sets will decrease static power consumption.

To dynamically power down specific sets is non-trivial because most SRAM arrays

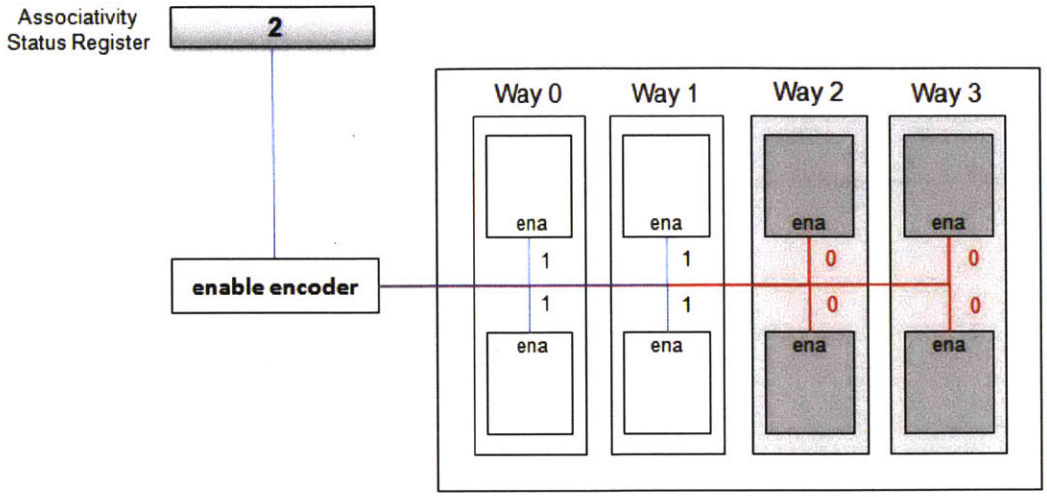


Figure 4.4: A 4-way associative cache converted to 2-way associativity. The power signals for each bank are delivered from an encoder that converts the value stored in the associativity status register.

are densely packed, and do not provide gating at such a granularity. However, as shown in Figure 4.2, ways are composed of several separate blocks, and as previously discussed these blocks can be powered down independently. Thus, the set adaptivity knob is easily implementable at the granularity of SRAM blocks. This is reasonable, as powering down sets at a cache line granularity is unlikely to change the cache behavior by very much anyway.

Much like the associativity adaptivity knob, a status register can be read and written directly by SEEC from software. Figure 4.5 shows an example where each way in a 4-way associative cache is composed of two separate blocks. Each block will have its power enable signal controlled by a decoder connected to a status register, which holds the set configuration of the cache. In this simple example, as there are only two blocks per way, the status register may hold either 0 or 1.

Since the number of valid sets changes with this knob, the cache mapping function

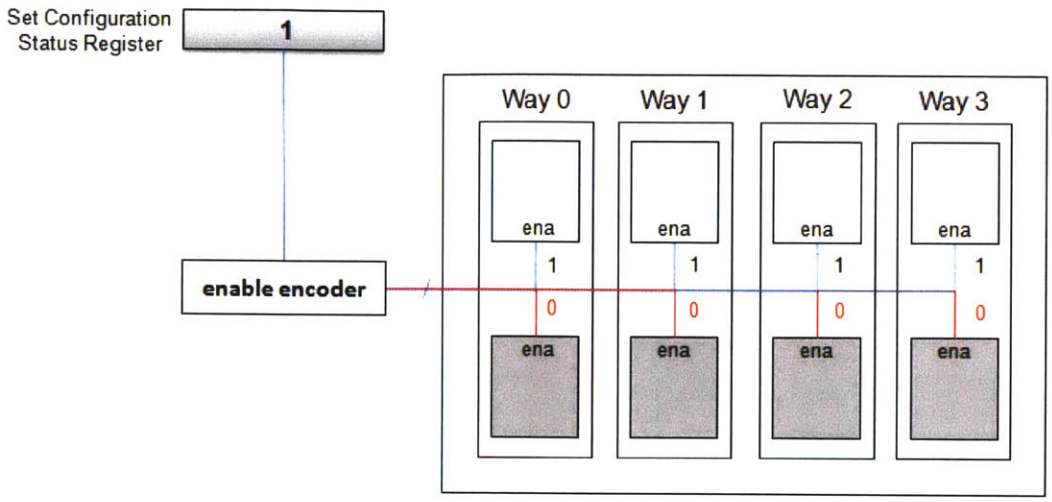


Figure 4.5: A 4-way associative cache where half the number of sets is powered down. Set adaptivity is achieved by turning off constituent blocks in a bank.

needs to be adjusted as well. In most modern caches, addresses are mapped into the cache by a simple mask that divides the address into parts. Starting from the least significant bit 0, the appropriate number of bits is used for the line offset, the set index, and the tag, respectively. The bits for the cache line index are exactly enough to uniquely identify every set in the cache, but if the total number of sets changes, so too must the set index. Figure 4.6 illustrates how the set index bits grow with the total number of sets.

The number of tag bits necessary to identify a cache line decreases as the size of the cache grows. However, the physical SRAM blocks that compose most tag caches have a fixed width, and implementing a tag cache with adjustable bit width is a very expensive circuit-level feature. Therefore, in our investigation, the length of the tags is fixed to the length of the largest tag required (*i.e.*, the tag required for the smallest possible cache, N - k - *1* bits in Figure 4.6). For SRAM blocks that can efficiently power down bitlines, it may be possible to make efficiency gains in an adaptive tag cache, but this is outside the scope of this thesis.

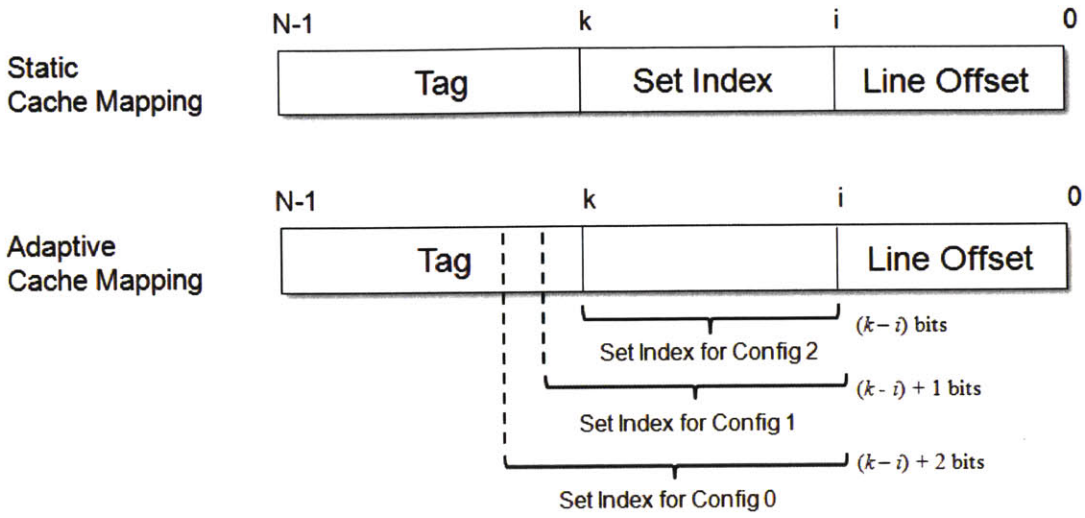


Figure 4.6: Address mapping for static and adaptive caches. The number set index bits changes with the set adaptivity knob, but the number of tag bits stays the same.

Adjusting the mapping function actually requires very little extra hardware, as it can be done simply by using the set configuration register to directly control the address mask. Once the mask is updated, address locations that map to powered-down locations are remapped to still functioning sets. Figure 4.7 shows the procedure where a cache begins with all its sets available, and then shrunk by half. An address that would have been in the bottom half of the cache is then remapped to the top half. One drawback of this method of adjusting the mapping function is that the number of blocks powered on must be powers of two. As such, the number of available configurations for the set adaptivity knob is *log* M, where M is the total number of SRAM blocks per bank.

Finally, maintaining the consistency in the cache as the number of sets changes is non-trivial. Cache lines are susceptible to aliasing when the number of sets is dynamically adjusted. Consider the case where the cache grows in the number of sets, as shown in Figure 4.8. In this example, address 0x1100 is originally mapped to the top half of the cache, but when the cache expands it ss mapped to the bottom half instead. As the cache

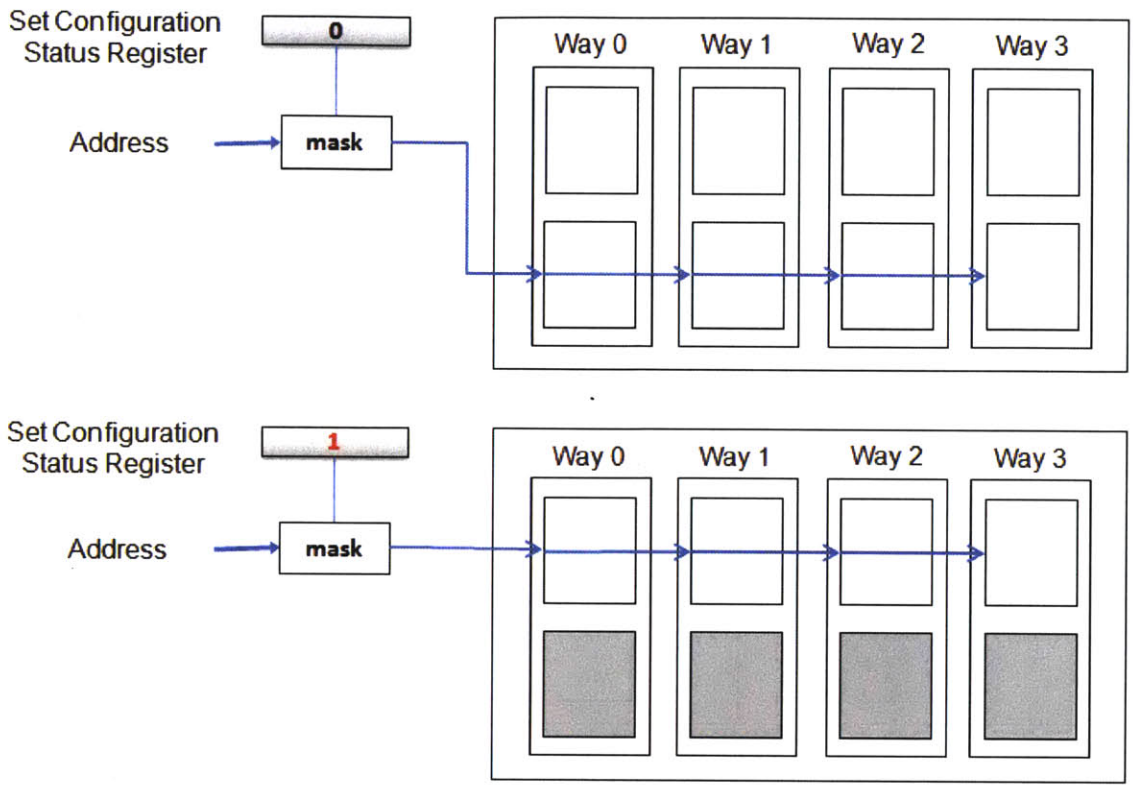


Figure 4.7: An address is that is mapped to the bottom half of the cache is re-mapped to the top half when the set configuration is adjusted.

remains in this state, data will be written to its location in the bottom half of the cache. When the cache shrinks back to the half cache state, and the original cache line in 0x0100 was not evicted, pursuant reads to address 0x1100 will hit on an outdated version of the data!

To solve this issue, not only do sets coming out of power down have to be clear, but the entire cache must be flushed when the cache grows. Flushing the entire cache may be inefficient, as this aliasing issue is rare, and is usually localized over just a few addresses even when it does occur. However, as the memory access patterns of most applications are relatively stable over time, SEEC should not be growing the cache size often over short periods of time. So for our purposes, this model for maintaining consistency within the

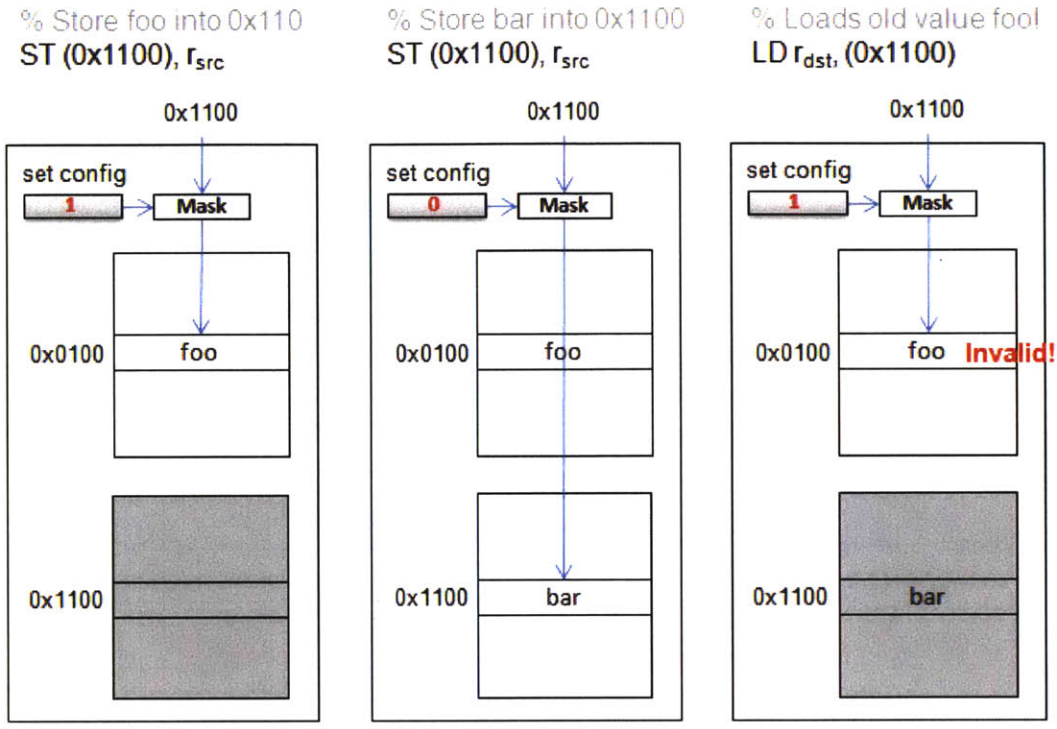


Figure 4.8: Illustration of the aliasing problem in the set associativity knob.

cache is acceptable. Other methods of tracking aliasing are possible, but keeping with the theme of low-power architectures, we will not pursue this path further.

Memory Hierarchy

As cache lines are invalidated on-the-fly, the system must be careful to maintain the cache coherency of the memory hierarchy. In this thesis, we will consider two levels of private cache per core, and implement a directory-based, write-back MSI protocol. Directories are distributed across the tiles with memory controllers that access DRAM off-chip, and L2 caches are inclusive. Figure 4.9 shows a representation of this memory hierarchy for four cores.

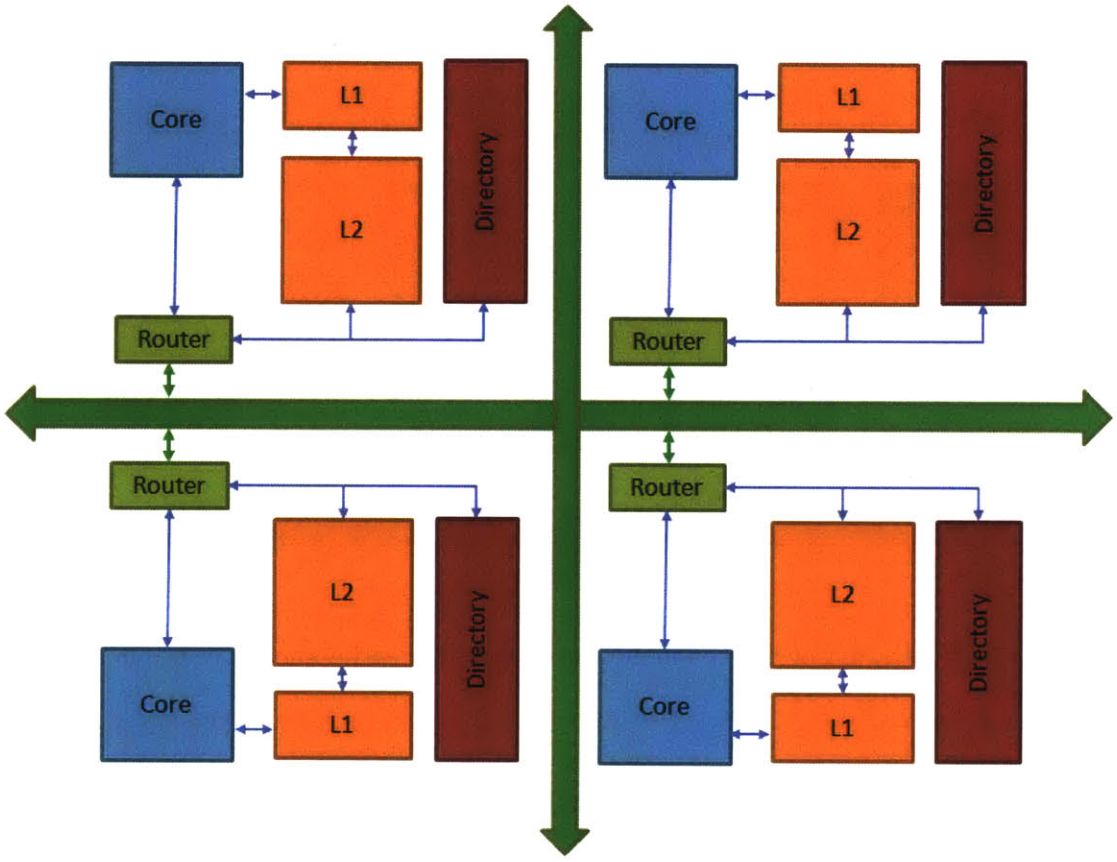


Figure 4.9: Memory hierarchy showing private L1/L2 caches, and a distributed directory.

We only consider systems where the L1 cache is adaptive. There are two main reasons for this decision:

(i) The L1 cache is the fastest and is the most frequently used cache in the hierarchy.

(ii) The overhead of maintaining coherency for L2 cache adaptivity is prohibitively high.

The first point is clear: sizing the L1 cache to fit the working set of the application can provide tremendous performance gains to the alternative. The second point is more subtle. Consider the case where SEEC makes an adaptation to shrink the L1 cache to half its size, therefore invalidating all the lines that are powered down. Since the L2 cache is

inclusive, there is no further need of any updates. On the other hand, Figure 4.10 shows that changing the size of the L2 cache is much more involved.

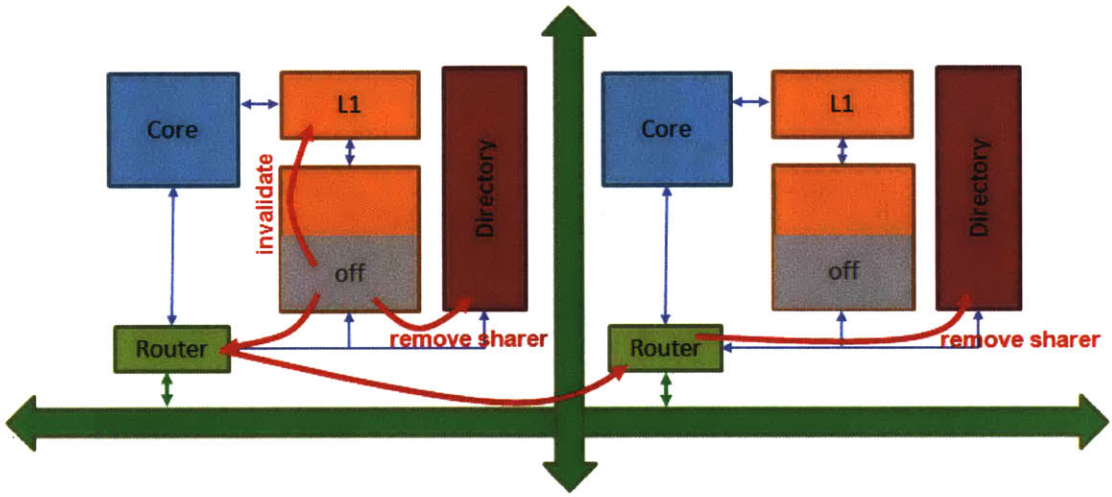


Figure 4.10: Illustration of the coherency problem if the cache knobs are implemented for the L2-cache.

First, because the L2 cache is inclusive, it must invalidate the corresponding lines in the L1 cache, which requires an invalidation request for every invalidated L2 cache line. Second, and more important, the appropriate entry in the directories must be updated to remove the appropriate sharers. Since the directory is distributed, not only does the on-tile directory need to be updated, but the directories on other tiles need to be updated as well. To this end, a network message for every invalidated cache line must be sent to the appropriate tile with the directory entry. This not only consumes considerable amounts of energy, but also generates an extreme amount of network traffic and latency. To make matters worse, since multiple lines from multiple tiles could be tracked in a single directory entry, the network router on the tile of heavily used directories would be severely congested.

Relatively speaking, the overhead of the first requirement is trivial; the overhead required to maintain directory coherency is extremely expensive. For this reason, we provide the cache knob only for the first level of caches.

4.2 Core Allocation

4.2.1 Motivation

The core allocation knob controls the total number of cores available for an application. Traditionally, a parallel application divides its workload into parts, and distributes the work among several threads of execution. When the number of threads is smaller than the number of total cores in the system, then the point of this knob is simple: allocate at most the corresponding number of cores to the applications threads, and idle the rest. However, the exact number of cores to allocate is unclear, because allocating one core per thread is not necessarily the most efficient configuration. There are several tradeoffs associated with running a core, and the optimal allocation might have multiple threads running on the same core, leaving the other cores to idle and save power. In the case that there are more threads than cores, then the problem becomes even more difficult. The complexity of this knob makes it ideal for a learning self-aware controller such as SEEC.

4.2.2 Tradeoffs

The main decision SEEC must make for this knob is whether the performance provided by running an extra core can offset the extra energy required to run it. The extra energy required to run a core is the difference between the power consumption of a running core and an idle core. This difference can be considerable, as most state-of-the-art cores provide deep-sleep states that can completely power down the core at the expense of a longer start-up time the next time it is used. As a result, allocating extra cores for performance comes with signficant cost in energy.

What makes this knob hard to tune is the non-monotonic relationship between performance and the number of cores. The speed of multithreaded applications depends on the

amount of parallelism available, the division of workload, and the data locality of threads. Moreover, these dependencies may change over the lifetime of an application, so a single optimal configuration may not even exist.

We first consider how the type of parallelization and synchronization can affect the performance of an application. A common approach for parallelizing an application is to split the total workload into parts, and distribute the parts among a specific number of threads. Another approach is to divide the workload into smaller tasks and place them into a queue, where a specified pool of threads then request for work when they are able. In any approach, applications require synchronization at the end of each iteration so they can consolidate the results before continuing.

Unfortunately, dividing the workload into exactly equal parts is non-trivial, and in fact a lot of time and effort is made by the application developer to manage this behavior. As an example, consider the matrix multiplication shown in Figure 4.11. The common block multiplication divides the matrices into equal-sized blocks, distributing the block multiplications across parallel threads, and then consolidating the solution at the end. Depending on the sparsity of each block, the time it takes each thread to compute a solution can vary widely. If multiple short-running threads can finish their work before a single long-running thread, then allocating a separate core for each short-running thread is likely to be wasteful, as they would have to wait to synchronize with the long running thread (Figure 4.11(a)). Alternatively, scheduling multiple short-running threads on a single core, and leaving the others idle, would be much more energy efficient (Figure 4.11(b)). Of course, it is possible for an expert application developer to identify the ideal distribution of work for the specific input type, but SEEC can automatically accomplish the same thing with this knob. Further, SEEC is also able to adapt the core allocation as unforeseen workload changes occur, which is a luxury not available to the application developer.

Secondly, depending on the applications access patterns, data locality plays a large

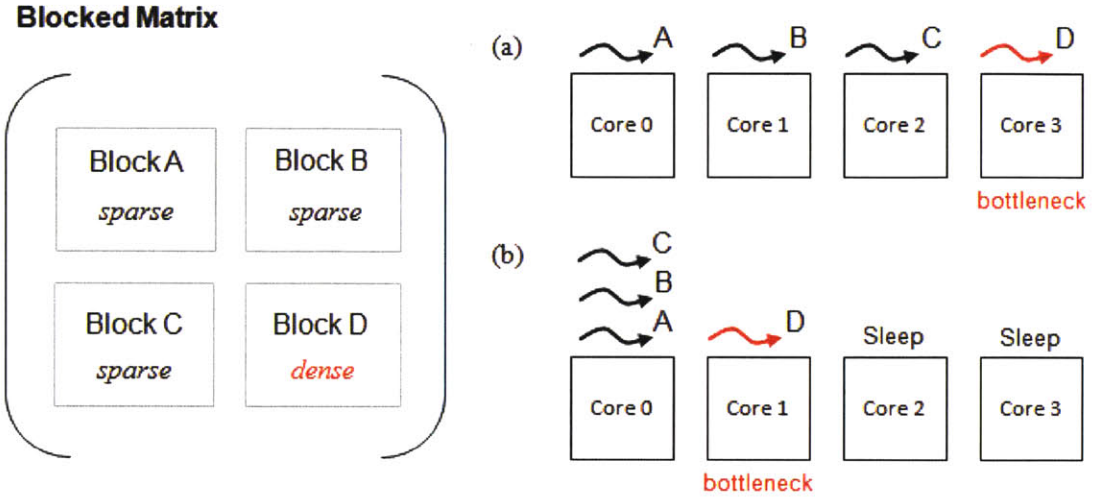


Figure 4.11: Unbalanced workloads in Matrix Multiplication: both distributions (a) and (b) execute with the same latency because block D is a bottleneck, so distribution (b) is much more energy efficient.

part in the optimal number of cores. Consider the case where threads actually share parts of their working set. Executing threads on the same core could avoid what would otherwise be cold misses that require long-latency DRAM accesses. Having warm caches can make a substantial difference in performance as well as energy efficiency, but the issue of data locality is application specific, and can also change during the lifetime of an application.

The data locality issue brings another advantage to coordinated self-aware systems such as SEEC, because it is largely related to the size of the caches. Since SEEC has the ability to tune both the size of the cache as well as the number of cores through a singular interface, it can converge on the optimal configurations for both knobs simultaneously. On the other hand, a closed adaptive system might configure its core allocation optimally at one moment for a specific cache size, only to find that the cache size changes shortly afterwards, and forcing it to readjust the number of cores.

4.2.3 Implementation

The core allocation knob is strictly an OS-level feature, and can be implemented by the thread scheduling features built-in to most kernels. In this thesis, we consider a system with a pre-emptive thread scheduler that allows the user to specify a core affinity for each thread. The feature for setting core affinity, or CPU pinning, is not unique, and can be found in most operating systems.

Core affinity can be interpreted as a set of bits that specifies a list of cores that a thread may prefer over another. When a thread possesses non-trivial affinity, the scheduler prioritizes those cores when scheduling threads. For instance, if a thread is running on a core that it has no affinity for, the scheduler will migrate the thread to a core that it does have affinity with. Therefore, to implement the core allocation feature, SEEC needs only set an affinity containing the appropriate total number of cores for all the threads in the application. In Figure 4.12, SEEC is running on a 4-core system, and each thread has an affinity mask that indicates that all cores are available. When SEEC decides to adjust the total number of cores available to two, it tells the OS to change the affinity masks for each thread. Eventually, the scheduler will honor the affinities, and the cores will be scheduled to two cores.

Note that once the threads settle onto the cores associated with their affinity masks, the scheduler is free to optimize the performance of these threads among those cores. Therefore, the effects of load balancing and data locality are as good as the scheduler. Furthermore, there is no need for SEEC to deal with shutting down idle cores, as the OS should move the cores into a sleep state after sufficient amount of time.

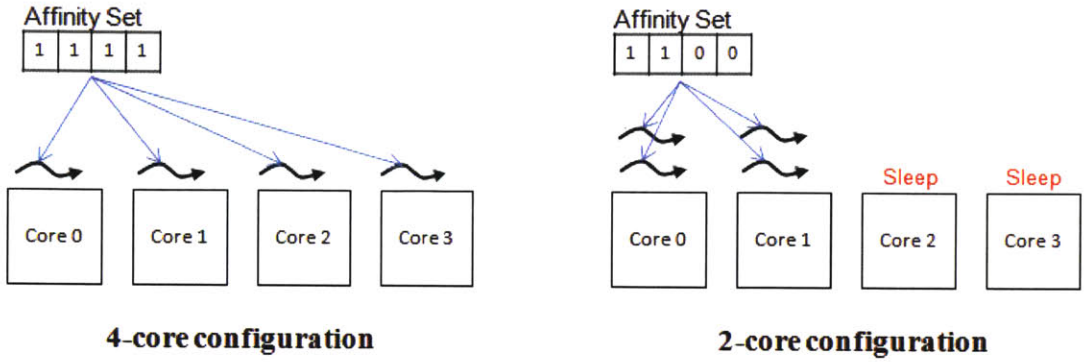


Figure 4.12: To adapt the core allocation knob, SEEC sets the affinity mask of each thread to correspond to the desired configuration.

4.3 Domain-Specific Dynamic Voltage and Frequency Scaling (DVFS)

4.3.1 Motivation

DVFS (Dynamic Voltage and Frequency Scaling) is a technique used to tune down the operating frequency of a component if that amount of performance is not required. When frequency is decreased, the supply voltage can be correspondingly decreased, thus saving on total energy consumption.

DVFS is a well-studied knob, because the scenarios where it is useful are plentiful, and the level of energy savings is very attractive. Consider a core that is stalled frequently, and is bottlenecked by the speed of the memory system; then the core does not need to be at maximum frequency at all times. For this reason, many modern systems dynamically vary the voltage and frequency of its cores, such as Intel's Turboboost system described in Section 2.1.2.

For this study, we take this knob one step further, and provide DVFS knobs for

each of the core and cache components for each tile in the system. This is useful because applications frequently have vastly different demands on the components of the system, and providing separate DVFS knobs to SEEC allows it to tune the performance of each component to the exact demands of the application. This requires the clock domains of the core and cache components to be separate, and also some complexity overhead for communication between clock boundaries, but is a necessary expense of implementation.

4.3.2 Tradeoffs

The DVFS knob allows SEEC to adjust the performance of both the core and cache components by tuning the clock frequency. In exchange, the power consumption of the component moves in the same direction. Fortunately, this tradeoff is what makes dynamic frequency scaling such a popular technique, because the frequency versus power relationship is super-linear, given by:

$$P_{total} = \alpha C V^2 f + P_{static}$$

Where C is the gate capacitance of the CMOS technology, f is the operating frequency, V is the supply voltage, and α is the activity factor, which indicates the relative factor of transistors switching at a time. From this relationship, it is clear that f is directly proportional to the total power consumption. Moreover, scaling down frequency allows for a corresponding scaling of supply voltage, so the actual benefit of frequency scaling is P $\propto fV^2$!

DVFS is useful because different applications put different demands on each system component. For example, an application that is memory-bound does not require a fast processor core, because it spends a large portion of cycles stalled and waiting for the memory

subsystem. In such a case, tuning down the frequency of the core component while maintaining the frequency of the cache component can save a considerable amount of energy while sacrificing very little overall performance. On the other end of the spectrum, there are applications that are compute-bound, and require a fast processor, but have very little use of the memory system. In this scenario, the cache component can save a considerable amount of energy by slowing down, while the compute component maintains its speed. By providing separate DVFS knobs for the cache and core components, we allow the self-aware controller to capitalize on both these types of applications.

With that said, the exact clock frequencies for the cache and core components can depend on a number of factors: how many memory accesses miss on the cache, what proportion of instructions are memory operations, what the relative delay in memory accesses over the core clock period is etc. Again, like the other knobs, the optimal DVFS configuration can be difficult to determine for an application programmer, but SEEC relieves this burden.

4.3.3 Implementation

To provide a DVFS knob that allows separate control of the core and cache frequencies, the clock domain of each component must be separate. The support this feature, the core pipeline must stall and wait for any cache accesses to complete before progressing. This way, even if the core and the caches run at different speeds, no data is lost because no more than one operation completes at a time. In addition, signals that travel between the core and cache domains must pass through hardware synchronizers to avoid metastability.

Like the core allocation knob, frequency scaling for the DVFS knob can be implemented as an OS-level feature. Many modern operating systems provide hooks for specifying the desired frequencies, and then voltage and frequency are scaled accordingly in the hardware. At the architectural level, exposing this knob to software requires only the

proper input to the clock controller, and those values can be set through memory mapped addresses, similar to the previous knobs.

Lastly, recent studies have pointed towards the diminished returns of DVFS [23]. One main concern is that as the technology nodes become smaller, supply voltages also scale down. As a result, as nominal supply voltages fall to the range of 0.9-1.0V, there is little margin for voltage scaling between the threshold and supply voltages. To address this concern, one approach is to develop future systems with subthreshold circuits [25]. In this work, logic and memory are designed with a novel methodology, such that circuits maintain functionality even in the subthreshold domain of the transistors. To be sure, the tradeoff for running logic at a subthreshold voltage is an exponentially large sacrifice in performance, but this is fine as long as the action models in SEEC are aware of them. A full description of subthreshold logic and memory is well beyond the scope of this thesis, but this technique can provide a much larger swing in supply voltage, vastly improving the degree of effectiveness of the DVFS knob.

Chapter 5

Experimental Evaluation

5.1 Methodology

5.1.1 Graphite Simulator

The SEEC framework has been tested on real x86 machines, and reports its effectiveness for those systems have been shown in [13]. However, since our interests lie in new adaptive mechanisms for future architectures, we cannot use traditional machines because the systems we are interested in simply do not exist. Therefore, for our purposes, all of the measurements for experimental evaluation will be made using the Graphite simulator [14].

For these experiments, the application under evaluation is linked to the Application Heartbeats and SEEC libraries, and analyzed by the Graphite simulator. The SEEC runtime is instrumented into the application source such that no extra threads of execution are required for the self-aware portion of the application. To make observations, applications will also be instrumented with the Application Heartbeats API so that heartbeats are

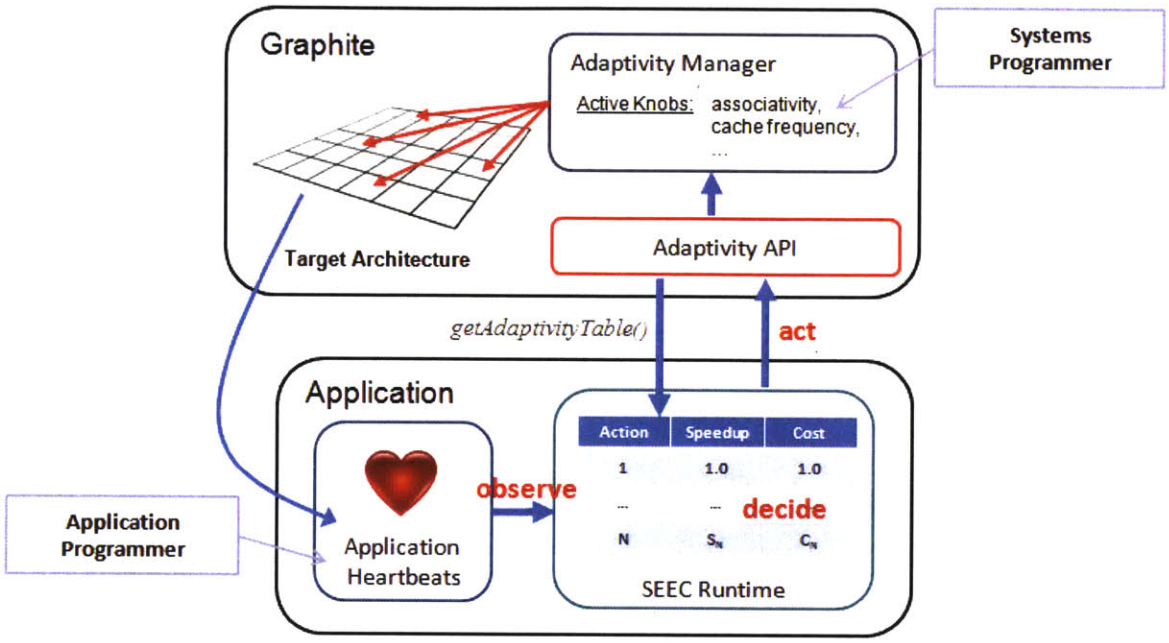


Figure 5.1: An application is simulated on a self-aware multicore system using Application Heartbeats, SEEC, and Graphite.

emitted at the beginning of significant inner loops. For making actions, the SEEC runtime can use a Graphite API specifically designed for adaptivity.

Once Graphite is configured with the desired knobs, SEEC can retrieve a list of all possible actions and their associated speedups and cost through a single API call during initialization. Knobs are configured by the Graphite programmer (*i.e.*, systems programmer in a real system), and the actual details are abstracted away so that SEEC needs only provide a configuration number to specify an action. Figure 5.1 illustrates the high level diagram of this setup.

5.1.2 Benchmarks

To evaluate the knobs, we will study the SEEC's behavior of the SPLASH2 benchmark suite [2]. SPLASH2 is a collection of shared memory parallel benchmarks representing a wide range of computation behavior in the scientific, engineering, and graphics domains. These benchmarks provide a good measure of effectiveness because they contain different amounts of parallelism, working set sizes, and communication to computation ratios. Each one of these characteristics directly affects the behavior of a knob:

- The amount of *parallelism* in an application characterizes how many cores can be effectively utilized. By Amdahl's Law, the more inherent parallelism in an application, the more effective extra cores are in speeding up the application. This is controlled by the Core Allocation Knob.
- The *working set* of an application indicates the locality inherent in an application. As described in Section 4.1.2, whether or not the working set fits in a cache can have tremendous impact on miss rate, and therefore the local memory bandwidth and inter-core communication. This is controlled by the Cache Associativity/Set Adaptivity Knobs.
- The *communication to computation ratio* indicates the potential impact of communication latency on performance. Communication is largely proportional to memory operations, which can result in local cache accesses, or off-chip memory accesses. Depending on the breakdown of time spent in computation, memory accesses, or otherwise, the performance of each of these components can be dynamically adjusted. This is controlled by the Cache and Core DVFS Knobs.

While the algorithms of the benchmarks are pre-defined, the aforementioned characteristics are extremely sensitive to the application parameters. Table 5.1 shows the exact applications that were used, and the associated application parameters.

Benchmark	Application Parameters
volrend	input head, 8x8 block, 8 steps, 1 rotation
raytrace	a = 1, m = 32
barnes	nbody = 16,384, dtime = 0.025, tol = 1.0, fcells = 2.0, fleaves = 5.0, tstop = 0.075
ocean	N = 258, e = 1e-7, R = 20,000, T = 28,800
water	tstep = 1.5e-16, nmol = 512, nstep = 3, norder = 6, cutoff = 3.21
cholesky	input tk15.O
lu	512x512 matrix, 16x16 blocks
radix	256K integers, 1024 radix

Table 5.1: Application parameters for benchmarks.

5.2 Results

5.2.1 Setting Goals

To evaluate a goal-oriented system such as SEEC, we must set an appropriate goal. Unfortunately, goals are not only application specific, but also usage specific: different application users may set different goals for the same application. The sheer number of possible goals makes it impossible to evaluate SEEC for every goal and application. Instead, we evaluate SEEC by targeting it towards challenging goals, and assume that it will perform well for easier goals. In this way, we can at least provide a worst-case bound on SEEC performance.

To make the job of SEEC difficult, we create a performance goal where a large number of *sufficient* configurations with similar effciency exist. In this case, SEEC must make intelligent decisions to select the optimal configuration from a pool of similar configurations. The maximum performance goal would not accomplish this, because there is only one sufficient system, and SEEC will uninterestingly selecting the fully-loaded configuration. On the other end of the spectrum, if the goal is minimum performance, SEEC will notice that all configurations are sufficient, and select the lowest performing configuration. Instead, we set the goal to one-quarter of the maximum performance attained by the follow loaded

$$Adjusted\ IPS/J = \frac{min(IPS_{target}, IPS_{config})}{E_{total}}$$

Where IPS_{config} is the instructions per second over the lifetime of the application, IPS_{target} is the target average instructions per second, and E_{total} is the total energy across all components in the system. By using the minimum of the target performance and the actual configuration performance, we penalize configurations that provide more performance than necessary.

The system we will be using for our experiments is the one described in Table 5.2, added to it the knobs we described in Chapter 4. The knob configurations are given in Table 5.3.

Knob	Available Configurations
Cache Associativity	direct-map, 2-way, 4-way
Cache Sets	4KB, 8KB, 16KB per bank
Cache DVFS	200MHz(0.5V), 400MHz(0.6V), 600MHz(0.7V), 800MHz(0.8V), 1000MHz(0.9V)
Core DVFS	200MHz(0.5V), 400MHz(0.6V), 600MHz(0.7V), 800MHz(0.8V), 1000MHz(0.9V)
Core Allocation	1, 2, 4, 8, 16, 32, 64, 128, 256

Table 5.3: Possible knob configurations.

We first identify the configuration that has the highest *cumulative* Adjusted IPS/J. This is the optimal configuration for a system that cannot change during runtime, and is statically configured once for all the applications. We call this configuration the *non-adaptive* configuration:

	Cores	Core Frequency	L1 Size	L1 Associativity	Cache Frequency
Optimal Non-Adaptive Config	128	800MHz	64KB	4-way	800MHz

Table 5.4: Optimal non-adaptive configuration over all benchmarks.

To identify the static oracle configurations, each application is simulated on Graphite with every combination of the knob settings shown in Table 5.3. The configurations with

system given in Table 5.2.

Parameter	Setting
Cores	256
Technology	32nm
Core Frequency	1 GHz
L1 Cache	64 KB 4-way
L2 Cache	256KB 4-way
Cache Frequency	1 GHz

Table 5.2: Simulation settings for fully-loaded system.

To confirm that this assumption is correct, consider the Pareto plot in Figure 2.5. There are very few configurations at the top of the performance spectrum, but many near the bottom. This is due to the fact that the knobs configurations grow exponentially, and only very few configurations are capable of high performance. That is to say, for instance, that core size is tuned by doubling or halving, and so performance scales exponentially. One-quarter maximum performance will have a large number of sufficient and similar configurations, making the decision of picking the optimal one difficult for SEEC. The effect is similar to guessing the values of two dice when given only the sum: seven is most difficult because it possesses the most possible combinations.

5.2.2 Static Oracle

The static oracle configuration attains the target goal with the best energy efficiency possible, on a system that cannot change during runtime. That is, it is the best configuration that an application developer could select given an infinite amount of time.

We define the best configuration as the one that attains *at least* the target performance goal, and consumes the least energy. This is equivalent to the configuration with the maximum *Adjusted Instructions per Second per Joule*, given by the following:

Application	Cores	Core Frequency	L1 Size	L1 Associativity	Cache Frequency
barnes	256	400MHz	64KB	4-way	600MHz
cholesky	128	800MHz	32KB	4-way	800MHz
lu-contig	128	800MHz	32KB	2-way	800MHz
ocean_contig	128	200MHz	16KB	direct-map	600MHz
ocean_non_contig	128	200MHz	16KB	direct-map	600MHz
radix	32	800MHz	32KB	2-way	800MHz
volrend	32	400MHz	64KB	4-way	400MHz
water-nsquared	256	800MHz	32KB	4-way	800MHz
water-spatial	64	600MHz	16KB	direct-map	600MHz

Table 5.5: Static oracle configurations for all benchmarks.

the highest Adjusted IPS/J are shown in Table 5.5.

In Figure 5.2, the efficiency of each static oracle is normalized against the non-adaptive system, showing remarkable improvements of up to 4.2x. Clearly, the static oracle can make good use of the knobs, and leverage them effectively to cater to each applications needs. This makes a strong case for providing any adaptivity possible: even if a system cannot be dynamically self-aware, the ability to at least provision on a per application basis is significantly better than a system with no configurability at all.

5.2.3 SEEC Results

To evaluate the behavior of SEEC, we again utilize the five knobs and the configurations presented in Table 5.3. The only difference is that for the following results, the cache knobs were split among the instruction and data caches, effectively doubling the amount of adaptivity in the memory hierarchy.

There are a total of 32 possible combinations of these knobs, and our goal is to determine how each knob performs with the others. To this end, each combination of knobs

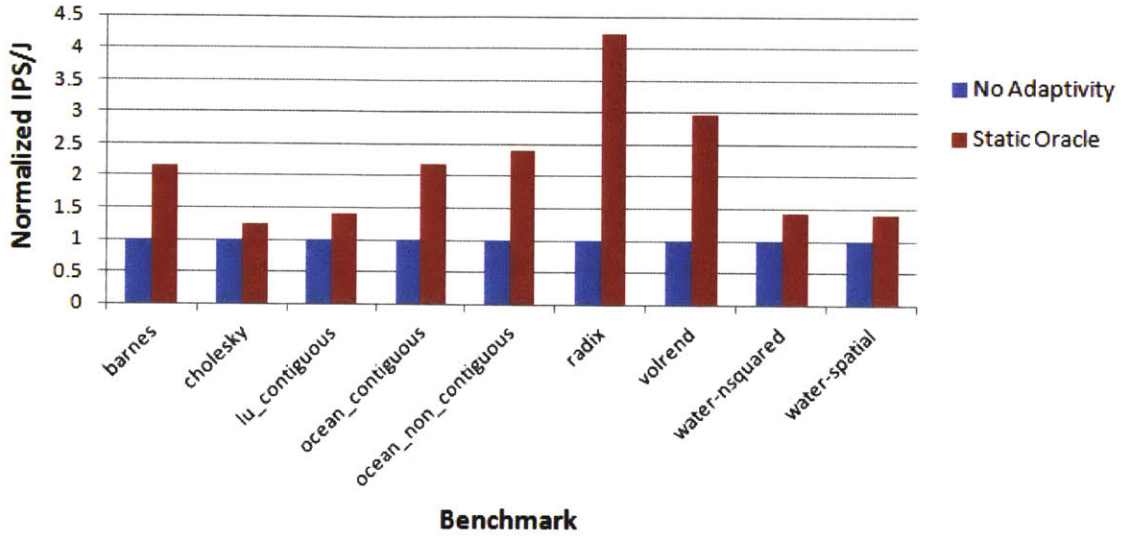


Figure 5.2: Results comparing static oracle configurations against an optimal non-adaptive system.

is simulated by enabling specific knobs, and leaving the other knobs disabled. When disabled during simulation, a knob is held at the configuration with maximum available performance. For example, if the Cache Associativity Knob is disabled, then the associativity of the caches will always remain 4-way associative, and if the Cache Allocation Knob is disabled, then the application will always have 256 active cores. This emulates the case where real systems are usually over-provisioned because there is no a priori information on the applications.

For these results, we use *heart rate* as the metric for best performance. The heart rate is the total number of heartbeats emitted through the Application Heartbeats API during a predefined window size. Again, we use a goal of one-quarter the maximum heart rate on the fully-loaded configuration. Since the SEEC model is designed to optimize performance while minimizing power, the appropriate metric to compare knob combinations is therefore *Adjusted Heart Rate/ Watt.* Like with our previous study, we cap the heart rate of each knob combinations to the target heart rate, so unnecessary performance is penalized:

$$Adjusted\ Heart\ Rate/W = \frac{min(HR_{target}, HR_{comb})}{P_{ave}}$$

Where HR_{config} is the average target heart rate over the lifetime of the application, HR_{comb} is the average heart rate of the knob combination, and P_{ave} is the total average power over the lifetime of the application.

This metric is useful for measuring the quality of the knob combinations, because it not only measures energy efficiency, but also includes information about SEECs convergence behavior. If a set of knobs produces bad convergence behavior, then the total energy efficiency would suffer, but on the other hand, if SEEC converges quickly, then the application will correspondingly execute with higher energy efficiency. In a system with more knobs enabled, SEEC is given more efficient, low-power states. However, having more states to explore may cause delays in convergence. Therefore, we must determine if the addition of more knobs actually provides enough improvement to performance and energy consumption to offset the detrimental effect on convergence behavior. The Adjusted Heart Rate/Watt metric directly reflects all these behavioral characteristics.

Over all 32 knob combinations, Table 5.6 shows the top five knob combinations as ranked by Average Adjusted Heart Rate/W over *volrend*, *raytrace*, *barnes*, and *water-spatial*.

Knob Combination	Knob Class	Total States	Normalized Heart Rate/W
Core Alloc, Core Freq, Cache Freq	3-knob	1000	2.57
Core Alloc, Core Freq, Cache Sets	3-knob	360	2.43
Core Alloc	1-knob	9	2.29
Core Freq, Cache Assoc, Cache Sets	3-knob	405	2.29
Core Freq	1-knob	5	2.20

Table 5.6: Top five knob combinations as ranked by average Heart Rate/W. Values are normalized to the non-adaptive configuration.

To study the effect of different numbers of knobs, and the tradeoff of extra states

and overall efficiency, we define the term *knob class* to describe combinations with a specific number of knobs. That is, every knob combination with 3 knobs enabled is part of the 3-knob class, and every combination with 4 knobs is in the 4-knob class. In Figure 5.3, we identify the best knob combination in each knob class, and show their relative energy efficiency normalized against the non-adaptive system. Table 5.7 contains the knob combinations of each knob combination in the figure.

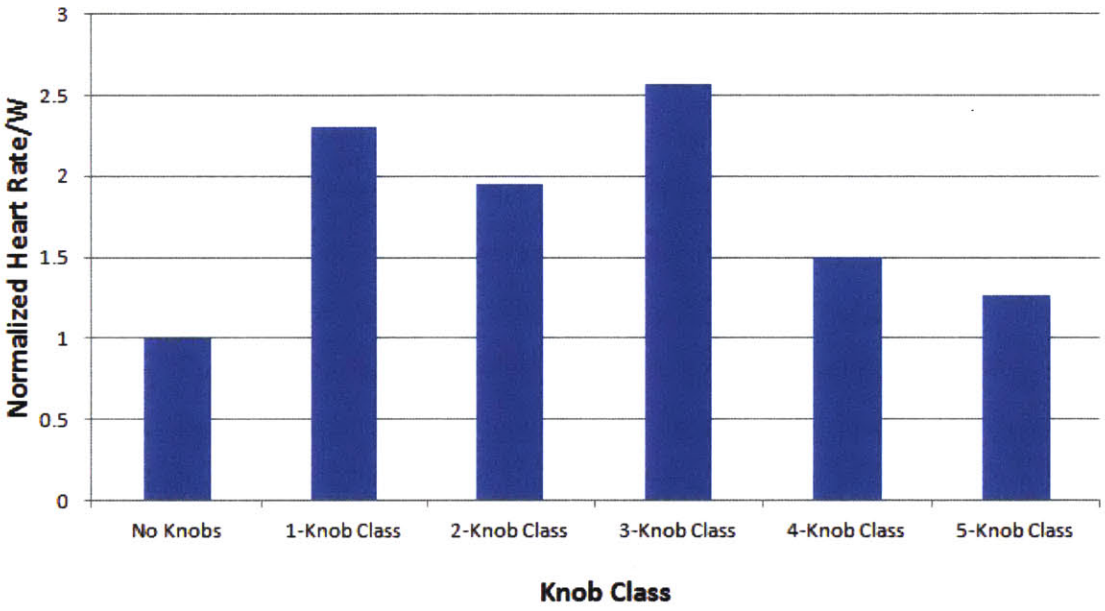


Figure 5.3: Energy efficiency of the best knob combination in each knob class when targeted to one-quarter maximum performance. Results are normalized against the non-adaptive, fully-loaded system; actual knob combinations are shown in Table 5.7.

5.3 Discussion

It is clear from Figure 5.2 that even a statically adaptable system is far superior to a non-adaptive system. The reason for such a large efficiency gain is the drastic disparity between application behaviors: each application's unique idiosyncrasies cause it to put pressure on

Knob Class	Combination with Highest Adjusted Heart Rate/W
1-Knob Class	Core Alloc
2-Knob Class	Core Alloc, Core Freq
3-Knob Class	Core Alloc, Core Freq, Cache Freq
4-Knob Class	Core Alloc, Cache Freq, Cache Sets, Cache Assoc
5-Knob Class	Core Alloc, Core Freq, Cache Freq, Cache Sets, Cache Alloc

Table 5.7: Best knob combinations for each knob class. The efficiency of each combination is shown in Figure 5.3

specific components. While the non-adaptive optimal configuration in Table 5.4 provides reasonable performance on every benchmark, it underperforms for some benchmarks, and outperforms for some others. Providing adaptivity can cater to these idiosyncrasies, and statically configuring them for each application results in large gains in efficiency.

Consider *barnes*, which is a highly parallelizable, and benefits from using all 256 cores in the system. Even though adding extra cores is very expensive in terms of power consumption, the performance gain from adding cores for *barnes* outweighs the costs. To wit, compared with the optimal non-adaptive configuration which contains 128 cores, the static oracle burns 2.01x more power throughout the lifetime of the application. However, by adjusting the number of cores to the system to 256, the static oracle completes *barnes* 3.26x faster, offsetting the cost of extra power.

Conversely, *radix* does not scale with the extra cores, and completion time actually stops improving after 64 cores. The reason for this is a consequence of the algorithm, where prefix computation in each phase cannot be completely parallelized [2]. Therefore, unlike *barnes*, the optimal configuration is the one that attains the target performance using as few cores as possible. Whereas *barnes* sacrificed higher power consumption and offset it with faster time to completion, *radix* maintains power consumption as low as possible. The static oracle for *radix* burns 0.24x power, and completes in 1.01x the time over the non-adaptive configuration.

We show the contrasting behaviors of completion time and total energy of *barnes* and *radix* in Figure 5.4. For *radix*, the completion time flattens out completely after 32 allocated cores, whereas *barnes* continues to reap the benefits of parallelism. As a result, the total energy consumption of *barnes* actually decreases up to 256 cores, whereas total energy for *radix* rises exponentially after 32 cores.

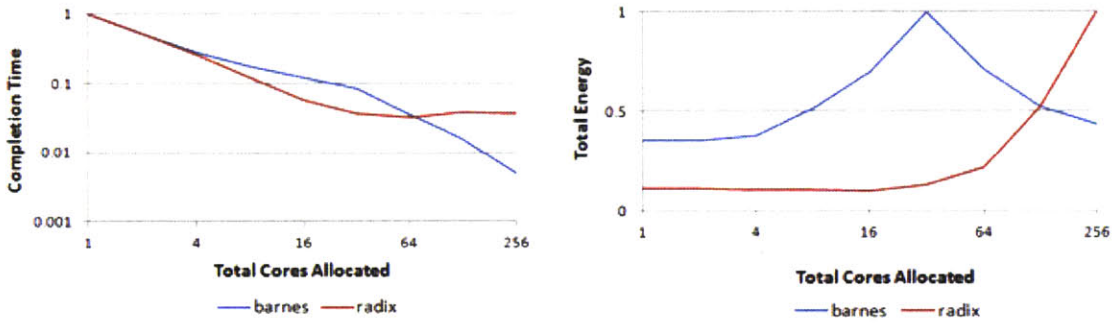


Figure 5.4: Contrasting performance and energy behavior of *barnes* and *radix*. *barness* completion time scales with the number of cores, but *radix* does not.

The point here is that, depending on the application and its input, adaptivity can provide huge gains in efficiency. However, the space of configurations can grow quickly: even with the five knobs we described in Table 5.3, there are a total of 1,800 configuration states that an application developer must test per application! As it is likely that there would be more knobs with even more states in a massively multicore system, it is hard to expect that an application developer can be as good as the static oracle. In fact, just by splitting the cache-base knobs over the instruction and data caches, the total number of possible configuration states for 5 knobs is 81,000!

In Table 5.6 we showed the best knob combinations as ranked by Heart Rate/Watt. As expected, SEEC makes significant gains in energy efficiency over the non-adaptive system. The Core Allocation, Core Frequency Knob, and Cache Sets Knob are present in a majority of the top combinations. This suggests that these knobs offer the most useful

tradeoffs to SEEC. This is not entirely surprising, because as we discussed in Section 4.3, the frequency knobs also allow for corresponding voltage scaling, which greatly improves the total power efficiency. As well, there are 256 cores in the simulated system, so the effect is compounded by the large number of cores that are tuned. For that same reason, it is also expected that the Core Allocation Knob has a large impact on efficiency. Finally, the Cache Sets Knob can effectively tune a system to the appropriate working set size.

While the cache-based knobs are useful to an extent, the core-based knobs are clearly stronger for this application set, as the top combinations all include at least one core-based knob. This is likely due to the nature of the benchmarks, which are dominated by compute-type operations. To wit, only an average of 31% of the instructions in the applications simulated were memory accesses. In addition, in the architecture of the tiles we are analyzing, the total power consumption is dominated by the application core, and so core-based knobs offer a larger impact on the total efficiency of the system. The cache-based knobs would likely be more effective for multicore systems with even simpler core architectures and larger caches, perhaps as in an embedded multicore system. However, in our analysis of an Angstrom-like system, we show that cache size based knobs are not as effective as the others.

According to Table 5.6, the 3-knob class is the most effective, and not the 5-knob class. In an ideal world, adding more knobs and adaptivity would only improve SEECs behavior, because adding more knobs merely adds granularity, never eliminating existing good states. However, in reality, adding more knobs to the system usually adds to the configuration space, creating a larger space for SEEC to explore. The fact that the 2-knob class is uncharacteristically lower than the 1-knob class, while the 3-knob class is still higher than both shows how difficult it is to make these predictions a priori. Depending on the synergies available between the knobs, SEEC can perform better even if the configuration space is larger.

Finally, we will discuss the convergence behavior of the knob classes. Figure 5.5 shows the heart rate behavior of the knob combinations with the most number of states in each knob class for *volrend*. The 1-knob and 2-knob classes converge almost immediately, as SEEC only needs to search through at most 25 and 225 states, respectively. However, because of the lack of knobs, SEEC does not have the ability to put the system in lower power configurations even if it hits the target heart rate, hence the lower overall efficiency. As it turns out, SEEC actually has trouble converging for the 3-knob combinations, as it only makes the goal right at the end of the simulation. That being the case, the 3-knob combination possess more states that are efficient, so it can avoid being in wasteful states even as it continues to converge to the goal. The 4-knob and 5-knob combinations show much more erratic behavior, but by the end of the simulation they too begin to converge.

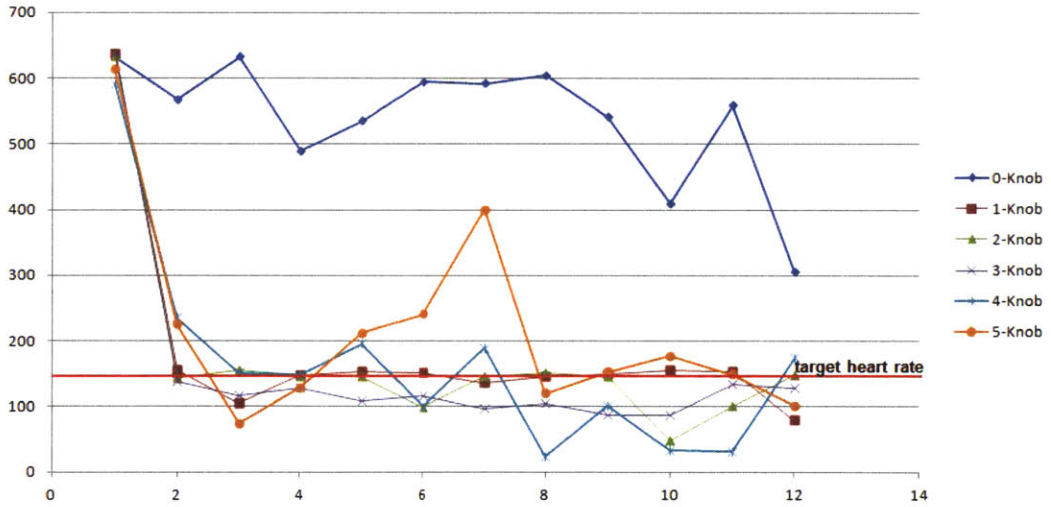


Figure 5.5: The convergence behavior of the knob combination in each knob class with the largest configuration space for *volrend*.

The source of SEEC's difficulty in the higher knob classes is not solely the number of states; the quality of the action models is likely a big contributor to the erratic convergence behavior as well. Recall in Section 2.3.3 that the action models for different knobs are combined together by multiplication, which does not take into account the actual relationship

between knobs. To be sure, SEEC can cope with incorrect values in the action models as long as the values are relatively correct with each other (*i.e.*, State 1 is faster than State 2, even if the values are incorrect). However, even to correctly maintain the relative correctness of the models is not guaranteed by multiplication, as the unique synergies between knobs is non-trivial and not modeled by simple multiplication. For this reason, multiplication of the action models is less accurate as the total number of knobs increases. The result is that the action models for the higher knob classes are less accurate, making it a challenge for SEEC to converge quickly.

Chapter 6

Conclusions and Future Work

6.1 Contributions

With the multicore revolution in full force, the corresponding growth in transistor density is threatening progress because of extreme constraints on power dissipation. *Dark silicon* describes a scenario where the transistors in future technologies cannot all be active simultaneously, leaving large parts of the silicon unpowered. This effectively stunts the drive for more cores on a chip, because even if the cores were available they could not be fully utilized. Hence, designing energy efficient systems has come to the forefront of computer architecture.

In this thesis, we share the vision that adaptive, self-aware systems may help solve this challenge. An adaptive system provides mechanisms, called *knobs*, that can be tuned to provide exactly enough performance for a specific application goal. This avoids providing systems with too much performance, which would consume more power than would otherwise be necessary. However, to take full advantage of adaptivity, it is necessary to

have enough configurability to cover all possible application behaviors. Unfortunately, the configuration space scales quickly with the number of knobs and cores, and the task of optimizing in such a space is unrealistic for an application programmer.

In Chapter 2, we present the background on *self-aware systems*, which can manage adaptivity without pressing any burden on the application programmer. These systems are introspective, allowing them to observe their own behavior and adjust the knobs as the application is executed. Most existing self-aware systems can only deal with a single piece of adaptivity, and cannot explore the configuration space in a global, consolidated manner. We introduce a coordinated adaptive system, SEEC, as a possible solution to this problem, and use this model for our evaluation.

In Chapter 3 we presented Angstrom, our proposed architecture of a massively multi-core system. We focused on those components of Angstrom that are designed for facilitating self-awareness and adaptivity. These include specific features specifically designed for the parts of the ODA loop, which are ubiquitous in self-aware systems. For observation, we presented memory-mapped performance counters and programmable event-probes for performance monitoring, and integrated energy monitoring circuits that provide fine-grained energy observations. For the decision component of the loop, we proposed the Partner Core as a viable architecture for running the SEEC engine with low communication latency and zero application slowdown. Finally, we described the SEEC and Angstrom interface that allows actions to be made efficiently.

The main goal of this thesis explores the usefulness of different types of knobs, especially for systems with high core counts, because studies of knobs in such systems have never before been done. We presented five knobs in Chapter 4, and described the tradeoffs that they are capable of controlling. Knobs were implemented and simulated in functional RTL, and the details of each knob for a system such as Angstrom were included.

In Chapter 5 we showed experimental results gathered from the Graphite simulator. First, we showed that adaptivity is critical for massively multicore systems, as even allowing per application configurability is far superior to a system that is non-adaptive. Results showed that for the SPLASH2 benchmarks, and a target of one-quarter the performance of a fully-loaded system, the static oracle executed with 2.2x IPS/J on average. We then went on to evaluate SEEC on selected benchmarks that were known to have contrasting application behaviors from the SPLASH2 suite. We found that the 3-knob combination of Core Allocation, Core Frequency, and Cache Frequency provided the highest Heart Rate/Watt on average, when targeting a performance goal of one-quarter the heart rate of a fully-loaded system. Even though the 3-knob combinations performed hit their goals most efficiently, we showed that the convergence behavior of SEEC was affected with the addition of knobs. As the number of knobs was increased, the convergence behavior continued to get worse. We attributed this to growth of the configuration space, and also the potential for action model inaccuracies due to the combination of several knob models.

This thesis is the first to study the behavior of adaptive mechanisms in a massively multicore self-aware system. Adaptivity and self-awareness provide an attractive way to build energy efficient systems without unrealistic effort by the application programmer. With the imminent danger of power dissipation threatening the utilization of a chip, this is a necessary step in the further development of multicore systems.

6.2 Future Work

6.2.1 Partner Cores

Preliminary design of the Partner Core and the high-level architecture of an Angstrom tile has been presented in [22], but there has been no work investigating its effectiveness

for running the SEEC decision engine. In Section 3.3, we claimed that partner cores can execute SEEC code with minimal communication latency and zero application slowdown, and that this would give the freedom for the partner core to run slower than the main application core. While this is true, the speed of the partner core directly affects the speed at which SEEC makes its decisions, which ultimately impacts the speed of convergence. If it is *too* slow, then SEEC could act on decisions that are well out-of-date, and never converge to the target heart rate. Therefore, it is critical for us to empirically determine that the frequencies we intend to use for the partner cores are sufficient for SEEC.

Furthermore, design work will need to be done for the interface between the partner core and the tile hardware. Care must be taken, because direct changes to the hardware by an asynchronous thread could break the functionality of the application thread. For instance, the partner core cannot change the size of the cache without making sure that all outstanding memory operations are first completed, lest a memory location gets invalidated even before it is complete. The design of the action interface must be bug-free so SEEC does not unintentionally violate functionality.

6.2.2 Application Classifier

As we have shown, SEECs convergence behavior tends to diminish as the number of knobs increase. This is expected, because as we discussed in Section 2.1.1, the configuration space scales by $O(N^M)$, where N is the number of knobs, and M is the number of configurations per knob.

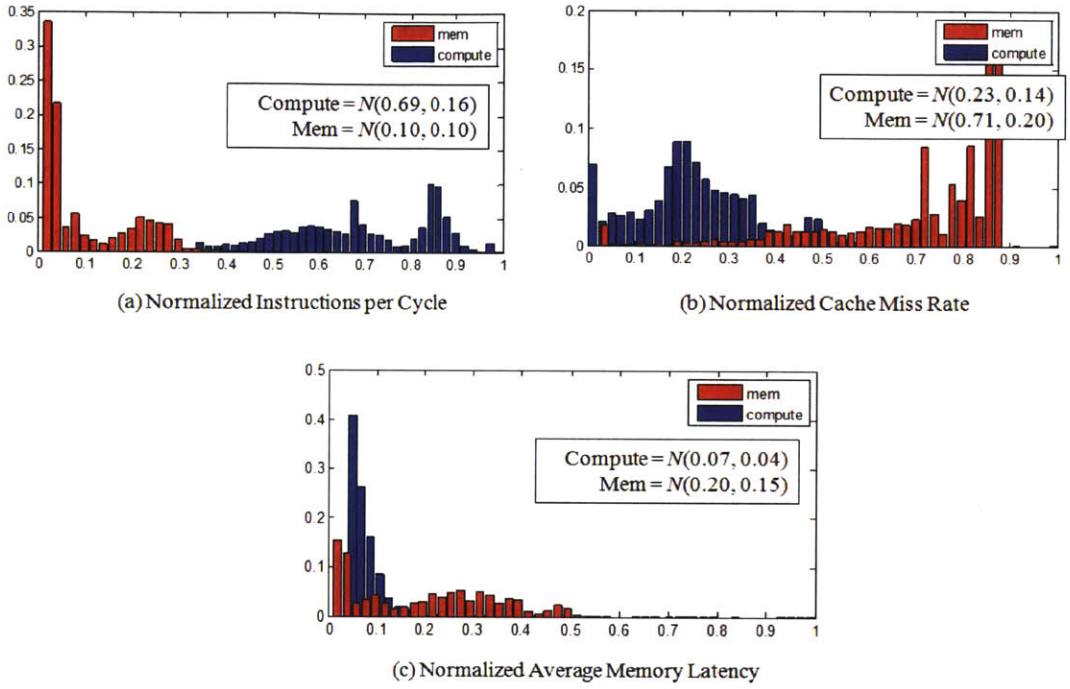
To help SEEC prune the configuration space, we propose a machine learning approach to classify the type of behavior of an application a priori. By predicting the behavior of an application, a classifier can improve the convergence time of SEEC by guiding its exploration of the configuration space. For instance, if a classifier identifies that an application

is compute-bound, then SEEC can focus on only the space where the compute component is optimized for performance. This feature can make good use of the performance counters described in Section 3.2.1.

A preliminary study was done to determine if a classifier could be effective using only information that can be tracked by performance counters. To this end, statistics for instructions per cycles, cache miss rate, and average memory access latency were gathered for several known compute- and memory-bound applications running under numerous system configurations. We used this data to form a training set for a number of different types of classifiers, and measured the prediction error on a test set formed from the SPLASH2 benchmarks. Figure 6.1 shows well-defined peaks between compute- and memory-bound applications, which show that these hardware metrics can be very useful for a classifier. Using these metrics, we showed that a Bayesian classifier correctly predicted 86% of the test set.

Effectively classifying an application provides benefits beyond just pruning the configuration space. One challenge in the SEEC model is providing accurate action models for each knob, because the behavior of knobs is application specific. For instance, an application containing large amounts of parallelism can benefit from a lot of cores, so the action model of the Core Allocation knob would describe a directly proportional relationship between number of cores and performance. On the contrary, an application with no parallelism will not benefit from extra cores, and in fact would suffer from the extra overhead of communication, resulting in an inverse relationship between the number of cores and performance.

One approach to address this issue is to allow the systems programmer to provide *multiple* models for each knob, and associate each model to a type of application. Then, with an effective application classifier, SEEC can determine which models to use. Making sure that SEEC uses accurate action models at all times is extremely important for maintaining



(a) Normalized Instructions per Cycle

(b) Normalized Cache Miss Rate

(c) Normalized Average Memory Latency

Figure 6.1: Histograms comparing behavior of compute- and memory-bound applications. Well-defined peaks allow classifiers to make accurate predictions.

healthy convergence behavior, and a classifier could choose among a collection of models to suit every application type.

6.2.3 Extra Knobs

While we have shown the effectiveness of five knobs in this thesis, the ultimate question of what knobs to implement in Angstrom is still an open question. There are still a host of unstudied sources of adaptivity, and any combination of them could benefit SEEC in an impactful way.

A potential source of adaptivity is the number of memory controllers available in

a system. The existence of multiple controllers increases the total bandwidth of memory operations because pages can be spread across several controllers, and allow for more simultaneous accesses than a single memory controller. The tradeoff is that each extra memory controller adds to total energy consumption. For applications that require high memory bandwidth, this tradeoff is acceptable, but for applications that do not require many off-chip memory accesses, the ability to adjust the number of memory controllers can greatly improve efficiency.

Another source of adaptivity is in the network routers. Depending on the scheme used by the routers, the latency per hop can be made to vary, thus affecting the total communication latency of the application. Simple routing schemes, such as XY-routing, require very little overhead, and produce very little latency per hop. However, simple schemes risk the danger of blocking live packets if a router is congested, thus increasing the total communication latency across the network. On the other hand, more complicated schemes, such as ones that manage virtual channels, can manage congestion much more easily. However, the extra overhead in virtual channel management will increase the latency per hop. Therefore, depending on the type of network traffic in the application, the optimal routing algorithm will differ. A knob that adjusts the flow-control algorithm can be used to select the ideal network router depending on the type of traffic in the system.

Bibliography

[1] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, December 2003.

[2] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, feb. 2008.

[4] A.Agarwal. Realizing a Power Efficient, Easy to Program Many Core: The Tile Processor. Presented at the Stanford Computer Systems EE380 Colloquium. Available online: http://www.stanford.edu/class/ee380/Abstracts/100203-slides.pdf.

[5] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

[6] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[7] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[8] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[9] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02), ICDCS '02, pages 301–, Washington, DC, USA, 2002. IEEE Computer Society.

[10] Seungryul Choi and Donald Yeung. Learning-based smt processor resource distribution via hill-climbing. In *In ISCA 06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 239–251. IEEE Computer Society, 2006.

[11] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 485–496, Washington, DC, USA, 2010. IEEE Computer Society.

[12] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 international symposium on Low power electronics and design*, ISLPED '00, pages 90–95, New York, NY, USA, 2000. ACM.

[13] Henry Hoffmann, Martina Maggio, Marco Santambrogio, Alberto Leva, and Anant Agarwal. Seec: A framework for self-aware management of multicore resources. 2011.

[14] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12. IEEE Computer Society, 2010.

[15] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 259–264, New York, NY, USA, 2012. ACM.

[16] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *In Proc. of the 9th International Symposium on High-Performance Distributed Computing*, pages 11–20, 1999.

[17] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2(3):195–205, July 1999.

[18] Baochun Li and Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, 17:1632–1650, 1999.

[19] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. Technical report, Intel Corporation White Papers, 11 2008.

[20] 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.

[21] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 353–364, New York, NY, USA, 2011. ACM.

[22] Eric Lau, Jason E. Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[23] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: the laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[24] Reid J. Riedlinger, Rohit Bhatia, Larry Biro, William J. Bowhill, Eric S. Fetzer, Paul E. Gronowski, and Tom Grutkowski. A 32nm 3.1 billion transistor 12-wide-issue itanium® processor for mission-critical servers. In *ISSCC*, pages 84–86, 2011.

[25] Alice Wang and Anantha Chandrakasan. A 180-mv subthreshold fft processor using a minimum energy design methodology. *IEEE J. Solid-State Circuits*, 40:310–319, 2005.