PREDICTING APPLICATION PERFORMANCE FOR CHIP MULTIPROCESSORS

by

Ryan W. Moore

B.S. in Computer Science, Westminster College, 2007

M.S. in Computer Science, University of Pittsburgh, 2012

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Ryan W. Moore

It was defended on

November 21, 2013

and approved by

Dr. Bruce R. Childers

Dr. Sangyeun Cho

Dr. Mahmut Kandemir

Dr. Youtao Zhang

Dissertation Director: Dr. Bruce R. Childers

PREDICTING APPLICATION PERFORMANCE FOR CHIP MULTIPROCESSORS

Ryan W. Moore, PhD

University of Pittsburgh, 2013

Today's computers have processors with multiple cores that allow several applications to execute simultaneously. The way resources are allocated to an application affects whether performance objectives, such as quality of service (QoS), are satisfied. To ensure objectives are met, resources must be carefully but quickly allocated in response to changing runtime conditions.

Traditional approaches to resource allocation take place either purely online or offline. Online methods do not scale to large, multiple core systems because there are too many allocations to evaluate at runtime. Offline methods cannot handle unanticipated workloads or changes. A hybrid approach could combine the lower runtime overhead of offline approaches with the flexibility of online approaches.

This thesis introduces AUTO, a hybrid solution to perform resource allocation. AUTO dynamically adjusts thread count, core count, and core type. It does so in accordance with a user-provided policy to meet performance objectives. AUTO's capabilities come from four prediction techniques. The first technique builds and uses models that consider CPU contention and application scalability in order to select co-running applications' thread counts. The second technique predicts applications' preferred thread-to-core mappings. The predictions are thread count independent and are translated into concrete thread-to-core mappings based on resource availability. The third technique predicts application performance under thread-to-core mappings. The final technique selects thread count and core count for applications on a system with cores of different capabilities.

AUTO was tested in several scenarios. In each scenario, it was shown to be an effective, efficient solution to resource allocation. First, it was used to select the thread count of one or more co-running applications. Second, it was used to select application thread-to-core mappings. Third, it was used to make predictions about application performance under thread-to-core mappings. Finally, it was used to select both thread count and core type for applications on a computer with cores of different capabilities. AUTO's resource allocation and models allow for more effective and more efficient policies. By using hybrid online and offline techniques, AUTO solves the problem of allocating threads and cores to meet performance objectives.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

Whether your impact was big or small,
I dedicate this to you all

Thank you.

1.0 INTRODUCTION

In today's computers, multiple processor cores allow multiple instruction streams to execute simultaneously [105]. Several hardware contexts per core provide additional parallelism (e.g., up to 64 threads on a Sun UltraSPARC T2) [107]. Manufacturers are expected to build future processors with even more cores and contexts, supporting large parallel workloads on a single machine.

Applications make use of machine resources, such as CPU time, memory bandwidth, and cache space. An application's resource allocation directly affects application behavior and consequently, whether or not it meets performance objectives (e.g., QoS). Furthermore, applications may compete for the finite resources of a computer. Competition can occur across different applications and/or between the threads of an application. Resource contention (e.g., "stealing" CPU time, cache contention) negatively affects performance [26, 71, 120].

Resources must be carefully allocated to applications. Applications differ in how much benefit they obtain from additional resources (e.g., additional cache space). To achieve the most benefit, application resource assignments must be performed by leveraging application knowledge and user objectives.

Furthermore, the applications executing on a system change over time, necessitating changes in resource allocations. Without notice, new applications may arrive, requiring access to resources. Additionally, applications may finish, freeing resources. This thesis enables dynamic resource allocation by enabling policies to efficiently consider the effects of application thread count, core count, and core type on applications. This thesis also allows for the dynamic adjustment of each of those parameters. Dynamic resource allocation 1) minimizes the negative effects of resource contention, 2) more effectively controls application performance (according to user objectives), 3) and more efficiently utilizes the resources of modern chip multiprocessors.

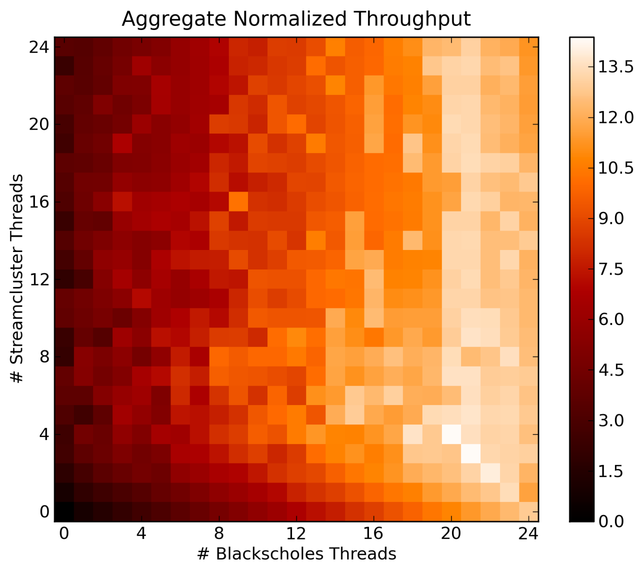

Figure 1: Aggregate normalized throughput for *BLACKSCHOLES* and *STREAMCLUSTER* as the number of threads in each is varied

1.1 RELEVANCE

Application threads use machine resources. If no threads exist, the system is idle. Controlling the number of threads directly affects the use of machine resources and application performance. To illustrate the importance of the proper selection of thread count, I conducted experiments with three applications from PARSEC [10]. The experiments executed a pair of applications under different thread configurations (i.e., number of threads in each application) on 24 cores in a multicore machine. The experiments measured aggregate normalized throughput, which is the sum of the speedups of each application relative to single-threaded performance on an idle system. Aggregate normalized throughput is a system-wide, higher-is-better performance metric.

Figures 1 and 2 show the results of these experiments. Warmer (brighter) regions indicate higher aggregate throughputs. Figure 1 shows the aggregate normalized throughput when *BLACKSCHOLES* is executed simultaneously with *STREAMCLUSTER*. The optimal configuration uses 21 threads for *BLACKSCHOLES* and 3 threads for *STREAMCLUSTER*, resulting in an aggregate normalized throughput of 14.34.

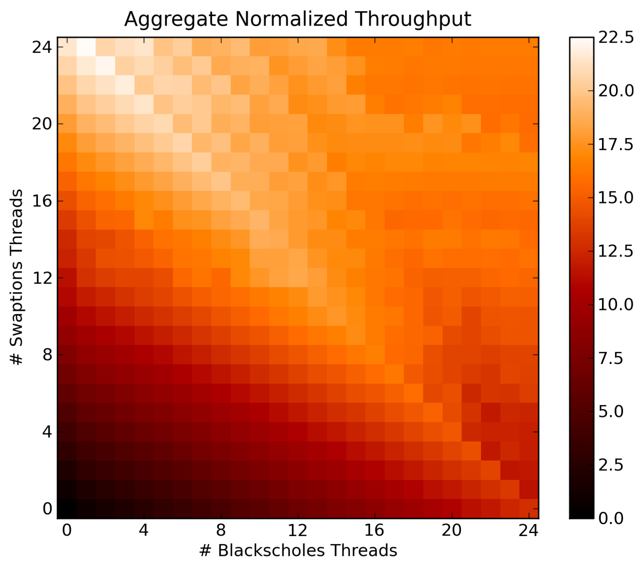

Figure 2: Aggregate normalized throughput for *BLACKSCHOLES* and *SWAPTIONS* as the number of threads in each is varied

Using a naïve thread configuration results in less performance. For example, 12 threads for each application (24 threads total) will result in a aggregate throughput of 9.27 (35% less than optimal). An alternative, but still naïve, thread configuration might oversubscribe the system's cores by using 24 threads in each application. In this case, aggregate throughput will be 11.27 (21% less than optimal).

However, if the system executes *SWAPTIONS* instead of *STREAMCLUSTER*, as shown in Figure 2, the best configuration uses 1 thread for *BLACKSCHOLES* and 24 threads for *SWAPTIONS*. This optimal configuration results in an aggregate throughput of 22.50. If each application has 12 threads, the system will achieve an aggregate throughput of 18.51 (18% less than the optimal throughput). With 24 threads in each application, the system achieves a throughput of 16.52 (27% less than optimal).

These results show that carefully selecting thread count based on corunners achieves significantly better performance than configurations which are chosen without consideration for the workload (i.e., application-agnostic thread configurations). The best thread configurations for one workload may differ greatly from the best configurations for another workload. Furthermore, the best thread count assignment is not intuitive; naïve policies (e.g., 12 threads to each, or 24 threads to each) significantly under perform. If the objective is to maximize aggregate normalized throughput, then *how can the system automatically choose the best configuration?*

The number of threads created by an application is just one parameter that may need to be adjusted in order to manage resource utilization and maximize performance. A second required parameter is the number of cores an application can use. A third parameter is the selection of cores that an application may use. The choice of cores given to an application may, for example, facilitate faster inter-thread communication by sharing one or more levels of cache. Cores may also differ in the amount of performance they can extract from an instruction stream (e.g., asymmetric cores). Application threads should be mapped to appropriate core types.

These three configuration parameters—number of threads, number of cores, and core choice—define a complex configuration space. To choose the best configuration, inter-thread and intra-thread behavior must be considered, along with physical hardware properties, application properties, and user objectives. As any of these considerations may change dynamically, configuration must also be done dynamically.

1.2 APPROACH

This thesis develops the AUTO framework. An overview of AUTO is shown in Figure 3. AUTO generates models for use in performance prediction. The models are used by an *allocation policy* to choose a resource allocation. The models enable allocation policies to take into account the effects of resource allocation. Models are built from profile data. The profile experiments and model-construction techniques depend on the *control knobs* used to change resource allocation.

To evaluate each technique, allocation policies are evaluated. These policies take advantage of AUTO's predictions to 1) determine the number of threads in each application, 2) determine the number of cores available to each application, and/or 3) determine the choice of cores that each application may use. *My hypothesis is that using performance prediction to dynamically allocate the number of threads, the number of cores, and the choice of cores can effectively maximize performance as compared to traditional allocation techniques without performance prediction.*

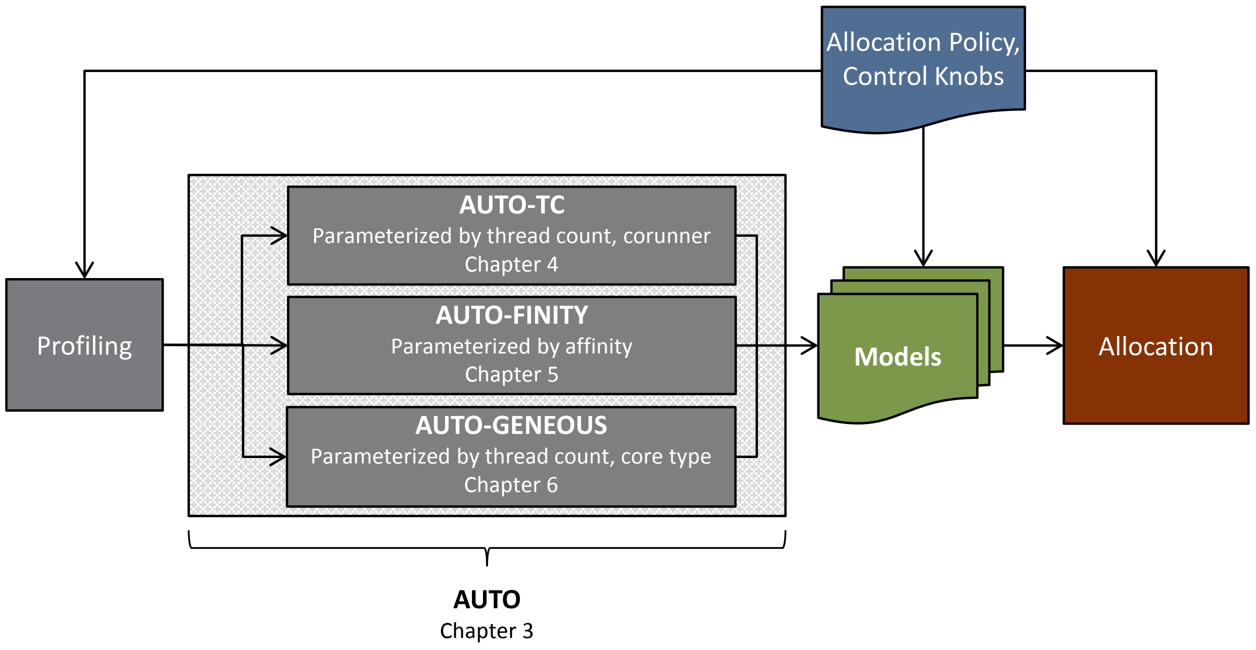
Allocation Policy, Control Knobs
Profiling
AUTO-TC
Parameterized by thread count, corunner
Chapter 4
AUTO-FINITY
Parameterized by affinity
Chapter 5
AUTO-GENEOUS
Parameterized by thread count, core type
Chapter 6
Models
Allocation
AUTO
Chapter 3

Figure 3: AUTO overview

AUTO is shown to enable policies that boost system performance according to a variety of metrics. This thesis's method of resource allocation requires capabilities not found in traditional systems. Application threads cannot in general be started or stopped without adversely affecting application correctness. Therefore, the framework works in collaboration with applications to adapt their thread counts.

1.3 ASSUMPTIONS

This thesis makes several assumptions about the applications and host system that it targets. These assumptions are necessary to focus the thesis.

This thesis considers CPU-bound and/or memory-bound applications. Applications which block (e.g., due to I/O) are not addressed. Blocking due to lock contention (i.e., poor scalability) is acceptable. It is permissible for an application to perform I/O in serial setup and shutdown phases. Applications must always be ready to do work (e.g., work queues are never empty). This thesis assumes that applications are trusted and do not disregard framework directives. For example, an application should eventually destroy a thread if instructed to do so. This assumption is reasonable in shared, trusted computing environments.

The host system must be able to execute multiple threads simultaneously via multiple cores and/or hardware contexts. This thesis therefore targets chip multiprocessors (CMPs). Systems that cannot simultaneously execute multiple threads do not benefit. Additionally, the host machine must have enough RAM for all applications. Otherwise, the operating system might perform page swapping due to low memory availability. An application whose pages are swapped to disk is I/O-bound, not CPU-bound. Additionally, a host's cores must be instruction-set compatible.

This thesis assumes that each thread executes on its own context (core). If the system is oversubscribed (i.e., more threads than cores) then the assumption is broken. CPU-bound threads cannot benefit from oversubscription when cores are idle.

1.4 CHALLENGES

Dynamic resource allocation involves multiple challenges. One challenge lies in making performance predictions. Application interactions and contention for shared resources (e.g., cores, caches) are inherently complex. Making useful, accurate predictions about the performance of the system is difficult. Therefore, machine learning and statistical techniques are used to build performance models. This challenge is the primary challenge addressed by this thesis.

The second challenge is how to dynamically change an application's thread count. Creating and destroying application threads must be done in an application dependent way. The developer is responsible for adding this functionality. For a particular class of applications, there may exist multiple ways to change the application's thread count. Modification guidance is provided for various classes of applications, including a discussion of the advantages and disadvantages of each (e.g., flexibility versus developer effort).

The third challenge lies in framework evaluation. Once performance predictions can be made by a model, the model must be shown to be useful. Each of AUTO's techniques is evaluated through the use of a policy, demonstrating the utility of AUTO. The evaluated policies consider application work or instruction throughput (e.g., to maximize performance, or ensure a minimum performance is met). These metrics are typical performance metrics on chip multiprocessors.

1.5 CONTRIBUTIONS

This thesis makes the following contributions:

1. A framework, AUTO, that predicts the effects of resource allocation (i.e., number of threads, core count, choice of cores) on applications according to various performance metrics;
2. A technique to build a model to predict application normalized throughput in the presence of one or more CPU-intensive corunners;
3. Techniques to models to select an application's preferred affinity;
4. A technique to build models to make application performance predictions across core types and with varying thread counts; and,
5. Evaluation of policies that use AUTO, in order to validate its techniques and accuracy;

1.6 ORGANIZATION

The rest of this thesis is organized as follows. Chapter 2 covers background and related work. Chapter 3 introduces the AUTO framework, necessary application support, and a discussion of how programmers might modify their applications for use with AUTO. In Chapter 4, the use of AUTO to dynamically select thread count in the presence of corunners is detailed. Chapter 5 shows the use of AUTO to choose application affinities in two ways. First, AUTO can predict the preferred affinities of an application. Second, the performance of the application executed with an affinity can be directly predicted. Chapter 6 shows how the framework can be used to simultaneously select application thread count and core type in a system with different capability cores. In each of chapters 4 through 6, one or more sample policies are evaluated to demonstrate AUTO's utility. A summary of this work, conclusions, and future work are in Chapter 7.

2.0 BACKGROUND AND RELATED WORK

This chapter groups related work into five inter-related categories. First, Section 2.1 explores general application adaptation techniques. Second, Section 2.2 shows works about general resource allocation. Third, Section 2.3 examines asymmetric computing. Fourth, knowledge of application behavior is discussed in Section 2.4. Last, Section 2.5 shows related work in the areas of performance modelling and runtime policies.

2.1 GENERAL ADAPTATION

This thesis is about building a device that manages itself. A self-managing system has several advantages as compared to a system which cannot control itself. By not requiring regular manual adjustments, employee costs are reduced or eliminated. More importantly, the system can react faster and more appropriately than a human ever could. Naturally, others have worked to build self-monitoring, self-modifying systems in the past. This thesis uses a set of reconfiguration parameters (i.e., thread count, core count, core choice) that others have not. Nevertheless, existing work in autonomic and adaptive computing may illuminate the limitations and/or promise of this thesis's work.

Some previous work has looked at how to adapt already existing systems and applications. Parekh et al. examine how to give autonomic capabilities to existing systems [76]. Their approach is generic and primarily reactive. This thesis targets a problem that requires a proactive solution. The configuration space (i.e., number of threads, core count, choice of cores) is too large to be explored and reacted to without serious lapses in performance (e.g., QoS). Diwan et al. present a programming environment within which reactive, high performance applications can be built [28].

Their work is on large, distributed systems. This thesis's focus is on a smaller granularity (i.e., a single CMP computer). This thesis could be used to coordinate applications *within* a computer, while their work coordinates applications *across* computers.

Cho et al. introduced MAESTRO, a framework for managing multicore systems [20]. They demonstrate the applicability of their approach. However, their work does not focus on how to build models for making predictions. This thesis's focuses is on the models. Without models, a predictive framework like theirs is limited.

Ramirez et al. provided valuable guidance (i.e., design patterns) for creating adaptive systems [86]. Dustdar et al. provide ways of "thinking" about the capabilities of autonomic systems [31]. Often, self-adapting work, like *AutoMate*, requires that applications be built on top of a specific framework or platform [1]. While this thesis does propose a framework, care has been made to ensure that application developers have nearly complete freedom in how they develop their applications. For example, developers can use their choice of programming language or parallel programming paradigm. Developers must be sure only to provide a way for the system to request a thread count increase or decrease. The implementation is left to the programmer, although this thesis provides some guidance.

Zhang et al. show how to modify applications to better share caches [118]. With their techniques, applications can have improved baseline performance and scalability. Chandra et al. presented a runtime to dynamically perform adaptive mesh refinement [14]. Similarly, Cleary et al. showed how scalability can be improved even in the presence of runtime lock contention [21]. Cleary's work uses message passing to alleviate lock and cache contention. Zhang's, Chandra's, and Cleary's techniques can be used to develop how high-performance, adaptive applications.

Programming languages might intrinsically give applications the ability to dynamically adapt their thread count. If so, they would free the developer from having to add this capability. For example, X10 [15] and Cilk++ [57] extend existing languages (Java and C++ respectively) to add fine-grained, explicit parallelism. X10 is intended for non-uniform cluster environments, while Cilk++ is for individual multicore computers. By explicitly adding information about parallelism, the programming language's runtime could automatically react to thread increase/decrease commands. Additional techniques to enable a changeable thread count include Lee et al.'s Thread Tailor [56]. Thread Tailor dynamically combines threads at runtime. Using offline analysis, Thread

Tailor takes into account thread communication patterns. If an application or its programming language does not allow for a changing thread count, then one solution would be to have the application create as many threads as there are cores. Next, Thread Tailor could be used to dynamically change application thread count.

Zhang et al. introduced a lightweight layer for streaming applications [117]. Their layer abstracts streaming tasks away from how the tasks are performed at runtime. Their layer can then make better resource allocations at runtime. Similarly, Phothilimthana et al. show how the same application can provide various algorithm implementations [80]. At runtime, the system can use the algorithm implementations that perform best given the current resource allocation. Applications can benefit from the ability to automatically adapt to runtime conditions.

Rahman et al. demonstrated how scientific applications can be automatically adapted to optimize for both performance and power [85]. Their techniques use the compiler to find the set of optimizations that are appropriate. The best performing set of compilation flags may dramatically increase power usage. Instead, they find a compiler-optimization that has great performance without poor power consumption. Their techniques could be used to automatically compile applications in accordance with runtime power or performance requirements. They require that each version be executed so that power consumption can be measured. This thesis's aims to be able to near-instantly predict and apply configuration changes. Future work could consider compiler-based techniques to perform further optimizations.

In general, the more information about a workload there is, the better the resource allocation can be. This thesis introduced the concept of application behavior "sketches" to inform resource allocation. Others have had similar ideas. Majo et al. added to OpenMP directives that hint at thread communication patterns [62]. These directives can be used to guide thread placement. Reduced communication latencies result in better performance and scalability.

2.2 RESOURCE ALLOCATION

Once the system is capable of reconfiguring and/or adapting applications, it must be able to enact changes. This thesis changes application thread count, core count, and choice of core. Existing work has looked at allocating resources to applications.

Nesbit showed how to grant machine resources to individual applications through the use of virtual private machines [74]. Nesbit's solution uses microarchitectural-level allocations and requires hardware support. This thesis uses commodity hardware. Nikolopoulos et al. allow for application-specific hypervisor policies to control, for example, how virtual machine cores are scheduled together [75]. Their work relies on manually generated scheduling hints, does not adapt thread count, and does not consider application affinity. Ahmad developed an adaptive non-contiguous processor allocation strategy [2]. His algorithm assigns cores to processes in large, shared distributed systems.

Other work has also examined how to do microarchitectural resource allocation. Duong et al. manage caches by sampling loads and stores to calculate reuse distances [30]. They allocate cache lines based on predicted reuse distances. McFarlin et al. and Hameed et al. examined theoretical application performance and inefficiencies in modern processors [40, 67]. Both works show that smarter resource allocation at the functional unit level and instruction set level can result in much lower power usage, smaller chip areas, and better application performance. Guo et al. show how to allocate resources to meet QoS constraints [38]. Their techniques require that applications know, in advance, what resources they require to meet QoS (e.g., a maximum cache miss rate). They present several policies to coschedule applications and distribute resources to applications. Unlike this thesis, they require microarchitectural support.

Wu et al. consider the prefetcher's effects on the LLC cache [113]. They derive policies to control the LLC prefetcher and reduce cache pollution if the prefetcher is mispredicting. As a result, application IPC is increased (5% on average).

This thesis is concerned with resource allocation above the microarchitectural level and yet below the level of a distributed system. The above techniques are therefore orthogonal.

Terboven et al., Blagodurov et al., and Dashti et al. examined NUMA contention management [12, 25, 103]. Terboven's OpenMP hints allow threads to control memory placement. Blagodurov introduced a NUMA aware policy to reduce memory contention. Dashti additionally manages contention through dynamic page migration. Dashti also allows page duplication across NUMA nodes to reduce bandwidth bottlenecks. Each authors' techniques result in decreased memory access latency. This thesis must take into account memory behavior across sockets, because it may affect performance, but does not consider page placement as a runtime control knob. Therefore, these works are complementary.

Like this thesis, Fengguang et al.'s work considers the effects of affinity on runtime performance [95]. Their techniques make use of memory traces to discover data sharing between threads. From this information, they can rank application affinities based on how well the affinity supports the application's sharing patterns. Their work does not consider private thread footprints. Memory traces can be very large and their solution requires approximation algorithms due to the complexity of the problem. This thesis uses faster methods of selecting application affinity.

Cui et al. address lack of application scalability by monitoring lock contention and migrating critical threads to a set of dedicated cores [24]. Ebrahimi et al. also monitor lock contention and change the memory scheduling policy to assist threads which most constrain the scalability of a application [33]. Resource allocation techniques like this are orthogonal to this thesis. This thesis assumes that, once an application's affinity has been selected, applications manage how their threads perform their work.

Perhaps most similar to this thesis are works by Kumar Pusukuri et al. [53], Xiang et al. [115], and Bhadauria [8]. Kumar Pusukuri et al. introduce a framework to coschedule applications. They schedule threads on a coarse socket basis (i.e., on a socket basis). They also do not change thread counts. Xiang et al. use traces gathered at runtime to drive scheduling decisions. Traces can introduce relatively large performance overheads (up to 31%). They do not consider multi-threaded applications. Bhadauria et al. use hardware performance counters to measure application behavior. Applications are profiled with a high thread count (one thread per core) at launch on an idle machine. Applications which scale well are, in turn, given exclusive access to the machine. Applications which do not scale well are forced to share CPU access with other poor-scaling applications. Their techniques do not adjust thread count and are designed only to maximize throughput per watt.

Juggle, by Hofmeyr et al., balances thread CPU access even if the number of threads and cores is imbalanced [41]. This thesis makes use of dynamic thread count changes to ensure that there is never more than one thread per core. Therefore, Juggle would not be necessary to use with AUTO. Ding et al. proposed Balanced Work Stealing (BWS) [27]. BWS alleviates CPU contention that occurs if work stealing applications have no work to do. This thesis removes the need for BWS by ensuring that resources are not overcommitted or application thread counts are too high.

Pusukuri et al.'s Thread Reinforcer determines the number of threads an application should use at runtime [82]. Thread Reinforcer does a good job selecting thread counts, but assumes the machine is not shared by CPU intensive processes. This thesis handles a system with more than 1 CPU intensive process.

2.3 ASYMMETRIC COMPUTING

One of the control knobs that this thesis considers is core type. Processor architects have always had to make design-time trade-offs due to limitations in, for example, manufacturing technology, power, heat dissipation, or area. A relatively recent trend is to incorporate ISA-compatible cores with different capabilities into the *same* chip or CMP system. Asymmetric systems are able to make power and performance tradeoffs better than traditional systems [51, 52]. Up to an order of magnitude improvement in energy consumption is possible [36]. These devices are available in commodity hardware (e.g., cell phones) [35, 37, 42, 68, 87]. "Dark silicon" will enable further variety in the performance capabilities and power consumption of cores [34, 36, 108, 109].

The "trick" to efficient computation on asymmetric systems is to map the workload to the most appropriate core. As the workload changes, remapping may be required to ensure that performance objectives are met. Measuring application performance on all core types is one approach [7, 51, 52]. As more core types are added, the cost of sampling becomes excessive. Analytic and experimental models avoid trying all core types [3, 17, 22, 49, 49, 77, 81, 91, 96, 106]. Nevertheless, they do not also consider application thread count: a critical factor in modern application performance.

The ability to change thread count allows for additional flexibility in meeting performance goals (e.g., QoS). For example, many threads running on small, low performance cores may be able to deliver the same performance as a few threads running on power-hungry, out-of-order cores. However, only one of those solutions will be optimal (e.g., the most power efficient solution). The additional flexibility creates a larger configuration space. Purely online techniques to select thread count and core type will waste time and energy exploring the large configuration space.

The disadvantages to online techniques are expected to grow, as architects create systems with increasingly more core types and counts.

2.4 APPLICATION KNOWLEDGE

To ensure good performance, a dynamically configurable system must react in an application-specific way. A single policy is unlikely to work well. Instead, the system must take into account application behavior and application resource needs. Only with application knowledge can the system find the best configuration. This section describes the diverse ways that others have gathered application knowledge.

Application knowledge may be embedded in the source code (potentially extracted later) [4, 15, 57, 70]. Knowledge may also come in the form of programmer inserted hints or static information, runtime-derived information, or profiling [16, 44, 66, 98]. Kandemir et al. use compiler-provided hints to manage shared caches [48]. Leung et al. allocate processes given their task graphs [58].

Wang et al. use machine learning to choose affinities [110]. Their approach requires a special compiler or dynamic binary instrumentation to extract features. Their offline approach requires a single-threaded profile of an application. Fengguang et al. also use dynamic binary instrumentation to choose affinities [95]. They use an analytical model and gathered information to predict application performance in various thread settings. Tian et al. model how an application's input affects runtime behavior [104].

Tam et al. use hardware performance counters to decide thread affinity settings [100]. Phadke et al. also use hardware performance counters, but they classify applications based on the type of memory that they prefer (e.g., low latency or high bandwidth) [78]. Xiang et al. give theory on various cache miss metrics and their relationships [116]. They show how to convert between different metrics using differentiation and integration.

Klug et al. propose *Autopin*, a tool to choose thread affinities [50]. Autopin uses a trial-and-error approach to eventually reach well-performing affinity settings. Radojković et al. have a similar approach (e.g., resource monitoring, trial and error) but focus on network applications [84]. Mars et al. and Jiacheng et al. profile an application's susceptibility to different types of memory interference [64, 119]. Profile information is useful online when predicting the effects of a co-scheduling on application performance.

The above works could be used to further inform AUTO's resource allocation process by supplementing profile-derived knowledge. The proper set of techniques may vary from system to system, or application to application. For example, a technique which requires a task graph should only be used on applications whose task graph is easily extractable. This thesis primarily uses profiling to extract application knowledge, because profiling is readily applicable across a variety of workloads.

2.5 PERFORMANCE MODELLING AND ALLOCATION POLICIES

Runtime policies use application knowledge and dynamic resource changes to meet performance objectives. The objectives may come from a system administrator or users. Objectives may be expressed in a variety of metrics (e.g., power, performance, response time). This thesis builds a general framework to meet diverse metrics.

Existing work considers the applications on a system and adjusts them to meet performance criteria (e.g., QoS, utilization without significant performance degradation) [63, 101, 111, 119]. This thesis primarily differs from past work in terms of setting (CMP systems) and configuration capabilities.

AbouGhazaleh et. al use machine learning to build a runtime policy that examines hardware performance counters. Their work is applied to embedded single-core systems and focuses on optimizing energy-delay product [84].

Application knowledge can be used to build utility models and to make configuration decisions. Machine learning approaches and statistical approaches are a natural fit for building predictive models that respond to application knowledge [65]. Barnes et al. [6], Ipek et al. [43], and Lee et al. [55] used a variety of machine learning techniques (e.g., regression, neural networks) to predict application performance in large dedicated clusters. Moseley et al. used linear regression and recursive partitioning to make performance predictions about co-scheduled threads on SMT contexts [72]. This thesis considers dozens of contexts on a single, shared CMP.

Scheduling techniques have been proposed for asymmetric CMPs to select the core type that an application should be executed on. Many scheduling approaches gather online information to predict application performance on various core types. Some directly measure runtime and/or power [51, 52], while others use hardware counters [88, 106].

Like this thesis, several researchers have used statistical models to predict performance of core types [17, 22, 49, 81]. Analytic models have also been developed that use online performance counters to guide migration decisions between core types [106]. A "signature" of an application's use of architectural resources has been used to select models that predict performance of different core types [91]. Recent work extended PIE for ARM big.LITTLE with compiler assistance to estimate cycles per instruction [81]. Those works do not address scalability for multithreaded applications.

Some works have considered modelling scalability. Wu et. Taylor model application scalability on large clusters [114]. Their models primarily consider available memory bandwidth and assume weak scaling. Ju et al. use queueing theory to analytically model application performance [47]. Their techniques require detailed programmer-provided information.

Some existing works detect and accelerate lagging threads in an asymmetric system by placing threads onto faster cores [9, 13, 54]. Other work detects and accelerate scalability bottlenecks caused by a thread's behavior [45, 99]. Joao et al. accelerate multithreaded applications, thus improving their scalability, by detecting lagging and/or bottleneck threads [46]. Their techniques accelerate threads by appropriately placing them on larger, more powerful cores. Those works help an application take advantage of a resource allocation. This thesis focuses on making the correct resource allocation, and lets applications use the resources however they may. Therefore, this thesis complements those techniques.

Tang et al. use a classifier and hardware performance counters to select application affinities at runtime [102]. Majo et al. consider NUMA systems and use hardware performance counters to consider both the costs of remote accesses and the costs of sharing a LLC [61]. Their policy can then migrate, as needed, a process to another NUMA node to improve performance. Pusukuri et al. developed Thread Tranquilizer to automatically reduce performance variation [83]. Dwyer et al. and Xiang et al. show how to predict performance degradation using hardware performance counters and dynamic binary translation respectively [32, 115]. Xiang does not consider multithreaded applications. Chen et al. model application characteristics using compiler-derived analyses (e.g., data flow) [18].

Sasaki et al. use hardware performance counters and statistical models to select DVFS settings at runtime [89]. Another work by Sasaki et al. monitors CPU usage to dynamically adjust application affinities and force poor scaling applications (i.e., those that block due to lock contention) to give cores to better scaling applications [90]. They model application scalability using Amdahl's law. Their work does not adjust thread count.

3.0 AUTO: A FRAMEWORK FOR CORE AND THREAD ALLOCATION

This chapter introduces the AUTO framework, its components, and the application support necessary to enable it. The AUTO framework predicts the effects of resource allocation. AUTO considers application thread count, core count, and choice of cores through the use of *predictive models*. An *allocation policy* will use AUTO's predictions to find the best resource allocation.

The AUTO framework is shown in Section 3.1. Section 3.2 discusses the framework's required application support. Section 3.3 discusses how programmers can modify their applications to fully enable dynamic resource allocation. Section 3.4 concludes.

3.1 OVERVIEW

Multicore CMP machines can be used for diverse purposes. For example, system administrators may want a machine to automatically maximize workload performance, meet quality of service goals, or minimize energy usage. An administrator may even want the machine to satisfy two or more objectives simultaneously (e.g., meet quality of service while minimizing energy usage).

To allow the machine to meet its performance goals (e.g., maximize workload performance), a system-wide coordination effort is needed. The AUTO framework is the primary component necessary to enable the coordination. AUTO considers dynamically changing thread count, core count, and/or core type.

In general, threads can only be created and destroyed in application-specific ways. AUTO therefore can make a wider range of predictions and policies can enact a wider range of allocations if applications can create or destroy threads in response to its decisions. The dynamic creation and destruction of application threads is called *inflation* and *deflation* respectively.

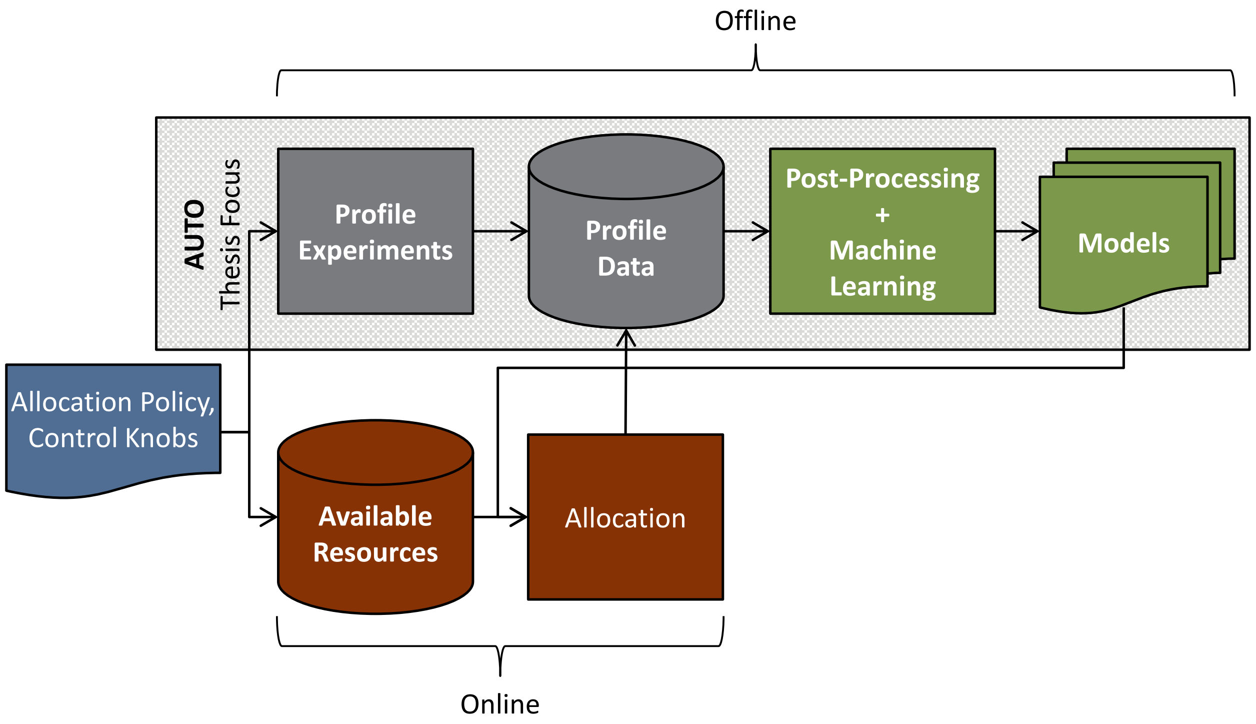

Figure 4: Framework usage and thesis focus

Figure 4 shows the use of AUTO. This thesis's focus is emphasized. AUTO uses an offline profiling step to gather application behavior. The profile data is processed. Machine learning techniques are used to build policy-informed models. The models are used by the allocation policy to make allocation decisions. Models can be updated online as runtime information is gathered.

The system architecture of my thesis is shown in Figure 5. A new layer, the *Runtime Configurator* , controls application resource allocation through communication with a new component, the *Application Resource Manager*. The Runtime Configurator is a dedicated layer above the operating system, to allow for a separation of concerns and easier runtime replacement or modification.

As in traditional systems, the operating system (OS) is the lowest software layer. However, in an AUTO-using system, the OS serves a narrow purpose: to control application access to hardware by following the Runtime Configurator's resource allocation. The operating system also provides usual services (e.g., file I/O). My thesis requires no operating system changes, as typical operating systems (e.g., Linux) already provide the services that the Runtime Configurator requires.

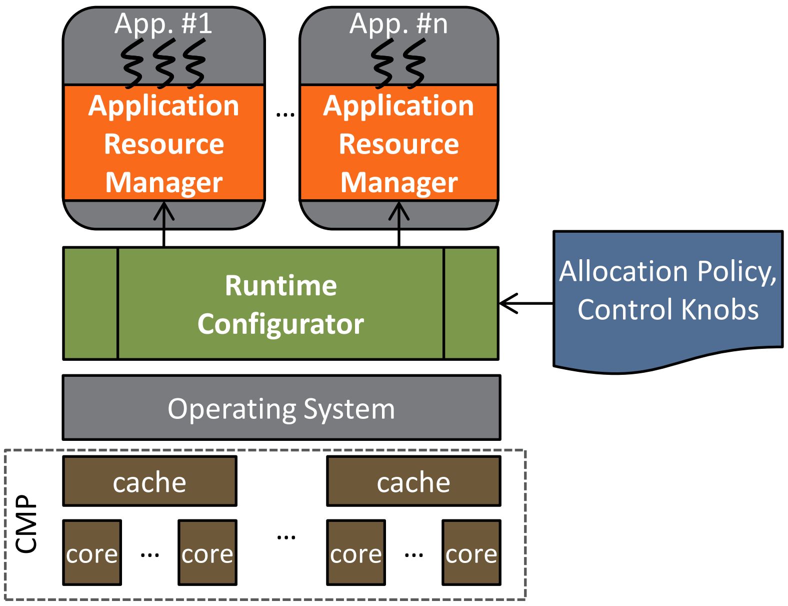
App. #1
Application Resource Manager
App. #n
Application Resource Manager
Runtime Configurator
Allocation Policy, Control Knobs
Operating System
CMP
cache
cache
core
core
core
core

Figure 5: System architecture

Applications are required to have an Application Resource Manager or they will not be admitted onto the system. This component is subservient to the Runtime Configurator. It changes the number of threads in a running application as directed by the Runtime Configurator. Because each application has unique behavior and resource needs, Application Resource Managers will be unique to each application.

3.2 INFLATE/DEFLATE PROGRAMMING MODEL

Inflation of an application is when a core is given to the application, and the application creates a thread to execute on that core. *Deflation* is when a thread is shutdown, so that the thread's core can be returned to AUTO. Inflation/deflation is a necessary mechanism to enable dynamic resource allocation. If applications on a system are not capable of dynamic inflation and/or deflation, AUTO's reconfiguration capabilities will be limited. The resource allocation and system performance will be suboptimal.

Table 1: Situations where inflate/deflate can be used for self-adapting applications

Situation	**Inflate Core?**	**Deflate Core?**
1. a core has failed	×	✓
2. power consumption rate is too high	×	✓
3. quality of service easily being met (over-provisioned)	×	✓
4. a core has come online	✓	×
5. an application has exited or freed cores	✓	×
6. cores have different capabilities	✓	✓
7. allocation policy	✓	✓

Table 1 lists some examples where inflate/deflate can be used in response to changes in cores. These examples are likely to necessitate a change in resource allocation. The figure also shows possible Runtime Configurator responses. The Runtime Configurator might inflate an application (give it a core) and/or deflate (take away a core).

The first three examples in Table 1 require deflation. The first two cases require deflation due to a core either failing or the system exceeding power constraints. In these cases, application threads must be moved off the cores to ensure that the system can continue to function. In the third case, the application is exceeding QoS constraints. The application could stop using one or more cores and still meet QoS. The newly-freed cores could be given to other applications.

The fourth and fifth examples benefit from inflation. In the fourth example, a new core has come online. Applications may now use it. In the fifth example, an application has exited, consequently freeing cores. The newly available cores may now be used by other applications.

The sixth and seventh situations may require both inflation and deflation. In the sixth situation, the system's cores are asymmetric. The threads in the system's applications may be more suited to some cores than others. The Runtime Configurator may take away a core from one application (deflation) to give the core to another application which can make better use of that core (inflation). In the seventh situation, the allocation policy may inform core assignments. For example, if the system is idle except for one application, the allocation policy may give all cores to that application. If another application starts, the allocation policy may take half of the cores away from the existing application in order to give them to the new application.

3.3 ADDING APPLICATION SUPPORT FOR INFLATE/DEFLATE

.

Traditional applications do not support inflate/deflate. Therefore, traditional applications are unable to respond to Runtime Configurator requests. Yet, this functionality is necessary to exercise the full range of resource allocations. To ensure proper allocation, the application developer must add this functionality.

To maximize inflation/deflation benefits, application-level knowledge must be utilized. If an application is granted a core, the application should take full advantage of it. Similarly, applications should properly respond to the loss of a core (e.g., by shutting down that core's thread).

Consider a multi-threaded, work-stealing application. At times, it will be granted a core or have a core taken away. The application should, upon receiving a core, create a thread to execute on that core. If assigned enough cores, the application might use a different algorithm to complete its work. For example, an algorithm which scales well may be used if many cores are assigned to the application. Applications might have limits to how much parallelism their algorithms can exploit (e.g., due to lock contention). If an application is unable to adequately utilize a core, the allocation policy will determine whether the Runtime Configurator will revoke the core.

The Runtime Configurator may also retrieve a core from an application (i.e., an application is forced to give up a core). It may be necessary for the application to prepare for the removal of the core. Additional time may be necessary to ensure that the application's threads remain in an acceptable state. For example, a thread may need time to commit its work before the thread can be shutdown and its core returned.

3.3.1 Programming Model

Applications must respond appropriately when granted or revoked resources. Figure 6 shows example pseudocode for what is expected for application developers to write. Application developers possess application-level knowledge, which can be leveraged for adaptation. Programming language constructs, such as those proposed, can allow developers to express application knowledge without undue burden.

```
def main(argv):
    resources = requestFromSystem(FAST + MULTIPLE + LENIENT + CORE)

    inParallel {
        for resource in resources:
            spawnThread(workerThreadFunc, resource)
        waitForWorkDone()
    } onInflate(CORE, newResources) {
        for resource in newResources:
            spawnThread(workerThreadFunc, resource)
    } onDeflate(CORE, removedResources) {
        #Signal the thread to quit (freeing the resource)
        #when convenient.
        for resource in removedResources:
            getThreadAssignedTo(resource).signal(SHUTDOWN)
    } onDeflate(CONTENTION, coresWithContention) {
        if (CONTENTION.type == CACHE && CONTENTION.value >= HIGH):
            #Signal threads to change how they compute.
            for resource in coresWithContention:
                thread = getThreadAssignedTo(resource)
                thread.signal(USE_LOW_FOOTPRINT_ALGORITHM)
    }
```

Figure 6: Example pseudocode for an inflate/deflate-capable application

Figure 6's pseudocode introduces two important concepts. First, the figure introduces the concept of a "sketch" of resources. A sketch is a specification of resources that a code section requires. Secondly, the pseudocode describes the application's reaction to various requests or commands.

Application developers, being familiar with the application, have high-level algorithmic knowledge. For example, the developer may know that an application computes large arrays of floating point values. This suggests that the application would benefit from fast cores with large caches. Another application, such as a static content web server, may not benefit from large caches.

The developer may also know other useful, low-level information. For example, a pipelined scientific process, in addition to using floating point operations, periodically communicates between threads in adjacent stages (i.e., one stage's output is another stage's input). Here, the programmer has some insight about thread communication but may not know the precise communication patterns and behaviors that manifest at runtime.

One purpose of the inflate-deflate model is to allow the developer to easily and succinctly express high-level application knowledge to the Runtime Configurator. By allowing the developer to easily express this knowledge, the system can more accurately predict the effects of resource allocation. Sketches, without being burdensome, allow the programmer to express their application knowledge. Application knowledge may also be obtained offline from the compiler, via profile information, or learned online.

On line 2 of Figure 6, the application requests multiple fast cores (`FAST + MULTIPLE + CORE`). It additionally indicates that its request is lenient (`LENIENT`). In this example, the application can make use of a wide range of cores, although it prefers multiple, fast ones. This is an example of a sketch. The sketch provides hints to the Runtime Configurator. Cores that match the sketch description are granted by the Runtime Configurator to the process (`requestFromSystem`). With its granted cores, the application performs computation (lines 4 - 21).

The `inParallel` construct is first executed. It creates threads to work on the cores. Once all cores have a thread assigned to them, the application waits until its worker threads have signaled that all work is done (`waitForWorkDone`, line 7). When the system grants the application a core, the application responds by spawning a new thread on the newly assigned core (`onInflate`, line 8). Similarly, when the application is told to return a core (`onDeflate`, line 11), it signals the thread assigned to that core to terminate early.

The application in Figure 6 can respond to changes in cache contention (`onDeflate`, line 15). The application checks to see if the core cache contention is high. The meaning of "high cache contention" is determined by the Runtime Configurator and depends on the system's cores and workload. If cache contention is high, the application threads will be signaled to reduce cache usage (line 20). Threads will receive the signal and utilize a different algorithm to reduce their cache footprint (not shown).

The difficulty of creating inflate/deflate-capable applications depends on how the original application was written. Some existing applications are already written in ways which allow for inflation/deflation. For example, queues between pipeline stages allow worker threads to be oblivious about which threads produced their stage's input. Work stealing approaches (e.g., Intel®

Thread Building Blocks[79]) abstract the production of work and the assignment of threads to the jobs that consume the work.

If all executing applications support inflate/deflate, the gamut of allocation changes is made possible. The Runtime Configurator can consequently make performance predictions and follow the allocation policy. Next, some remaining inflate/deflate challenges, and possible solutions, are discussed.

3.3.2 Inflate/Deflate Challenges

There are several challenges for the inflate/deflate programming paradigm. The challenges are:

1. how to handle underspecification of requested resources,
2. how to handle over-specification of requested resources,
3. time bounds on requests,
4. how to measure current state quality,
5. making accurate predictions, and
6. dealing with deceptive processes,

The first challenge is how to handle underspecification of requested resources. In Figure 6, the sketch dictating what type of resources an application desires is a high-level request (`FAST + MULTIPLE + LENIENT + CORE`, line 2). For example, what it means for a core to be fast is not clear. High-level sketches simplify the developer's task of describing which cores an application needs. High-level sketches may leave the Runtime Configurator unable to reason about how applications differ with regards to their resource requirements and performance under possible resource allocations. For example, on a multicore asymmetric system running two multi-threaded applications, the first application may benefit from fast floating point operations, while the other application may benefit from fast memory access. A lack of sketch detail may be due to programmer error or limitations on sketch expressiveness. Fortunately, offline profiling solves this challenge. Offline profiling can automatically determine the behavior of an application. Predictive models can fill in the "holes" left by unprofiled behaviors.

The second challenge is how to handle over-specification. The developer may request a resource that may not easily, if at all, be granted. Some over-specified requests may be impossible to be met on a specific system (e.g., a request for a core with 128 KiB of L1 cache space). Should the system ignore such a request? What if the request is necessary for the application to function correctly? One way to address this challenge is to have the Runtime Configurator lie about resource properties, while making a best effort to meet application requirements. As with the solution to underspecified sketches, automatically constructed models help inform the true needs of an application.

The third challenge is how to deal with time bounds on requests. Requests can be from the Runtime Configurator to an application (e.g., to retrieve a resource), or from an application to the Runtime Configurator (e.g., for an additional resource or change of resource). Quickly complying with requests allows both the Runtime Configurator and applications to react quickly and reason more accurately about system behavior. The Runtime Configurator and Application Resource Manager should quickly comply to commands and queries. However, this may not always be possible. For example, if a thread is signaled to shutdown, it must do so in a way that ensures application correctness. Terminating the thread immediately could put the application in an inconsistent state. If the application does not comply within a reasonable amount of time, what should the system do? Should the request or command be ignored? Should the application be punished for not complying in a timely manner? This thesis assumes that AUTO is used in a semi-cooperative environment. Therefore, a best effort approach is suitable.

The fourth challenge is how to accurately monitor performance. What metrics should be used? Given a current configuration and resource assignment, is the system in a good or bad state? Should the system switch to a different, predicted-to-be-better configuration? Hardware performance counters can be used to record low level performance metrics, such as IPC. Application-specific counters can also be used (e.g., work units per second). Because applications are trusted, both counters are acceptable. In this thesis, the users of the system set the allocation policy. The policy determines which allocation techniques and performance objectives will be used. The policy also determines if and when AUTO should change configurations.

The fifth challenge is how to ensure that the Runtime Configurator's performance predictions are accurate. The Runtime Configurator and applications possess an incomplete understanding of system behavior, due to the inherent complexity of modern CMPs, application behavior, and application interactions. Figure 6's application can reduce cache contention. Ideally, an application would be able to predict how much it can reduce cache contention. Often however, such a capability is practically impossible. Fortunately, online learning, hardware performance counters, and offline profiling can be used to measure and predict different aspects of system behavior.

The sixth and last challenge is that an application may be deceptive. For example, a multi-threaded application may request a minimum of four cores at all times. This may be a legitimate request (e.g., if the application is hard-coded to create four threads). However, the application may lie in order to gain more resources than it would normally be given, to let itself execute faster. A deceptive process may be a result of a malicious programmer or programmer error. The applications running on a system are assumed to be semi-cooperative. Thus, deceptive applications are only possible due to programmer error, not maliciousness.

3.3.3 Demonstration Implementations

Two use cases are explored to show whether or not adaptation is beneficial: energy conservation and pipeline balancing. Each use case evaluated a multi-threaded application from the PARSEC benchmark suite[10] (*BLACKSCHOLES* and *DEDUP*). The evaluation also serves to demonstrate how to create inflate/deflate-capable applications.

3.3.3.1 Energy Conservation Use Case (Data Parallel Workload) Consider a scenario where a server continuously performs batch work and runs two applications: a multi-threaded application (*BLACKSCHOLES*) and a single-threaded application, referred to as the side process (described later). *BLACKSCHOLES* calculates European options pricing using Black-Scholes partial differential equation. The goal of this exploration is to demonstrate that inflate/deflate enables efficient sharing of cores.

Suppose the system administrator has instituted a policy to dynamically restrict the number of usable cores in a chip multiprocessor, to correspond with changing electricity prices. The Runtime Configurator is used to implement this policy.

Figure 7 shows the sample policy for the use case's test machine. Electricity costs vary throughout the day. To reduce power costs, the number of cores used by the machine will also be varied. If the only goal was to limit how many cores are active at any given time, no application

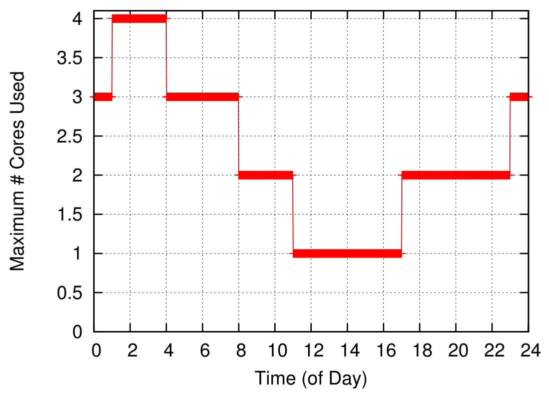

Figure 7: Sample core usage policy

modification would be necessary. A mechanism like Linux's `sched_setaffinity` could be used to restrict which cores the threads are allowed to execute on.

However, restricting the scheduling of threads has several disadvantages. To make full use of all the system's cores (e.g., as desired late at night) there must be as many threads as there are cores. Yet, when fewer cores are available (e.g., in the afternoon when electricity prices are highest), fewer threads are necessary. The *BLACKSCHOLES* application cannot change thread count at runtime. This limitation can lead to a mismatch between the number of CPU-bound threads and the number of usable CPU cores.

On a system with more cores than threads, cores will be idle and performance will suffer. A system with more threads than cores may cause threads to be starved for CPU time. More threads than cores can also increase the cost of context switching; thread working sets are likely to be evicted caches due to context switching.

A better system would feature adaptable processes. As the number of usable cores is decreased, multi-threaded applications should use fewer threads until the system is no longer oversubscribed. Each application has at least one active thread. As the number of usable cores is increased, applications should create worker threads until the system is no longer underutilized. This policy avoids over-subscription and unnecessarily high context switching overhead, while also ensuring good performance.

BLACKSCHOLES cannot dynamically change its thread-count. It accepts a thread-count pa-

Table 2: Inflate/deflate *BLACKSCHOLES* modifications

Modification:	**Lines of Source Code Changed:**
Work unit data structure	5
Work units initialization	9
Worker thread code changes	14
Main thread inflate response ability	7
Main thread deflate response ability	7
Support code to listen to inflate/deflate requests	13
Total changes	**55**

rameter on the command line. The input array is read from file and stored in an array of length $workUnits$. The desired number of threads is created. Each thread is given an identification number which is used to evenly divide the work among threads. Threads compute a result for each entry in the workload array. The main thread waits at a barrier for all worker threads to finish. The main thread then writes the results to a file.

To support the dynamic addition and removal of worker threads, *BLACKSCHOLES* was modified. A shared, thread-safe linked list stores tuples of the form $\langle i, j \rangle$ where $0 \leq i < j < workUnits$. List initialization is done by the main thread. Worker threads execute a loop, popping a tuple from the work list on each iteration. Threads compute results for the elements in the array between the tuple's i and j indices. The modification of the work thread code was straightforward and required little modification, as shown in Table 2.

How quickly an application reacts to a message can be adjusted by changing the size (range) of work units. The size of work units is controlled by an environment variable. Larger work units cause messages to be checked less often, but may cause less lock contention in the work unit queue. Smaller work units cause quicker responses to thread messages but may incur locking overhead. Before obtaining a new work unit, a thread checks to see whether it should exit prematurely. If no work units are available, the worker thread exits. Other modifications to *BLACKSCHOLES* include adding the code to connect to the Runtime Configurator, a function to create new threads, and a function to signal the shutdown of worker threads.

Table 3: Test system properties

System Property	**Value**
Processors	Intel® Xeon® E5335 @ 2.00GHz
number of cores	4
RAM	8 GiB
OS	Linux (kernel 2.6.29.6)

3.3.3.2 Energy Conservation Experimental Results The energy conservation policy was applied, and the throughput of both versions of *BLACKSCHOLES* and the throughput of the side process were observed. Experiments with the original sample policy (Figure 7) would be 24 hours in length. Instead, a miniaturized policy is used. Experiments are divided into seven five-minute periods. To ensure an execution time of at least 35 minutes, the *BLACKSCHOLES* input is processed multiple times. The first period allows only one core to be active. Each period thereafter allows one more core than the previous period. In period four, four cores are allowed, which is the number of cores in the test system. Subsequent periods use one less core than the previous one. In period seven only one core is active. The test system runs *BLACKSCHOLES* and the side process. Table 3 shows the properties of the test system.

At the end of each of the periods, the number of work units computed by each version of *BLACKSCHOLES* and the side process are measured. A *BLACKSCHOLES* work unit is one array entry calculation. The side process performs memory and arithmetic operations to calculate a value (single-threaded, CPU bound). A side process work unit is one completed calculation. Work units for each application are normalized to the maximum number of work units completed by that application, in any period.

With the inflate/deflate-version of *BLACKSCHOLES*, the side process is always given exclusive access to a core, except when both applications are constrained to execute entirely on one core (as scheduled by the OS). The unmodified *BLACKSCHOLES* uses four threads, regardless of how many cores are available in a period.

Figure 8 shows the results of this experiment. The height of the first two bars in each period is the number of work units completed by the unmodified *BLACKSCHOLES* (BS) and inflate/deflate-

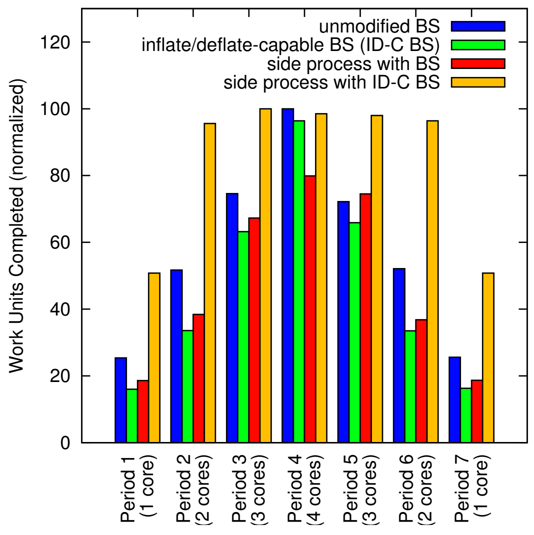

Figure 8: Work units completed by *BLACKSCHOLES* and the side process

capable *BLACKSCHOLES* (ID-C BS), respectively. Each BS has a similar pattern: as the number of cores increases, so does the rate of work unit completion. Consistently, the unmodified BS completes slightly more work units than ID-C BS.

The unmodified BS performs better than the ID-C BS because it uses more threads and hence can obtain more CPU time (i.e., by stealing CPU time from the side process). Additionally, threads in BS do not make use of a shared work unit queue. This design avoids linked list contention.

Figure 8 also shows the relative number of work units completed by the side process when executed alongside the unmodified and inflate/deflate-capable versions of *BLACKSCHOLES* (third and fourth bars respectively). In general, the number of work units completed by the side process increases as the policy allows more cores to be used in each period. However, its corunner has a dramatic effect on the number of work units completed. When the number of cores is low (periods one, seven), the side process executed with ID-C BS completes more than twice as many work units than if executed with BS. When the allocation policy allows all cores to be used (period four) the side process executing with ID-C BS completes 23% more more work units than if executed with an unmodified corunner. When more than one core is available (periods two through six) the side process running with ID-C BS always performs nearly at its maximum.

Table 4: Efficiency of the energy conservation experiment

Period	**Efficiency (unmodified)**	**Efficiency (ID-C)**	**% Increase in Efficiency**
1	44.0	66.8	51.8%
2	45.1	64.6	43.2%
3	47.3	54.4	15.0%
4	45.0	48.7	8.2%
5	48.9	54.6	11.7%
6	44.5	65.0	46.1%
7	44.3	67.1	51.5%

Table 4 shows an efficiency comparison between the unmodified system and the inflate/deflate-capable system. Efficiency is the sum of normalized work units completed by both processes within a period, divided by the number of cores used in that period. The inflate/deflate-capable system (third column) is always more efficient than the unmodified system (second column), especially when core resources are few (e.g., the first and last period). The inflate/deflate-capable system shows efficiency increases from 8.2% to 51.8%. Increased efficiency is a direct consequence of the fairer, more efficient use of system resources.

In conclusion, the inflate/deflate-capable *BLACKSCHOLES*, combined with a power conservation policy, causes cores to be utilized more fairly and efficiently across processes. Fairer utilization allows the system to execute with fewer active processors while still maintaining good performance, enabling power savings.

3.3.3.3 Balanced Pipeline Use Case This use case evaluates a multicore system in which the number of available cores changes dynamically, as cores come online and go offline without warning[1]. The maximum number of cores is also unknown (e.g., cores may be hot pluggable).

A process on this system should respond to core availability to maximize performance regardless of the number of cores available. In this use case, we evaluate *DEDUP*. *DEDUP* was modified to take advantage of such a system. *DEDUP* performs data stream compression using both global and local compression.

[1]Processes are given adequate time to migrate their work from soon-to-be-offline cores.

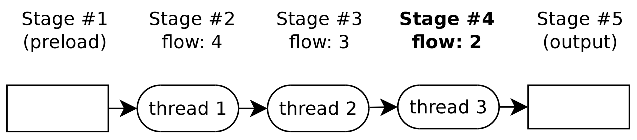

Figure 9: *DEDUP*'s initial pipeline with three cores

DEDUP uses a multi-stage pipeline. The modified *DEDUP* supports dynamic insertion and removal of threads in each parallel pipeline stage. Stages are instrumented to calculate the flow of work units through them. The instrumentation allows the identification of bottleneck and over-provisioned stages. The changes to enable dynamic thread creation and destruction are discussed. Second, the modified and unmodified versions of *DEDUP* are evaluated with regards to their performance and ability to handle changing resource availability.

The base version of *DEDUP* is a five stage pipeline, as shown in Figure 9. In the first stage, the application loads an input file before spawning threads. This step reduces I/O bottlenecks during the parallel phase. Next, three thread pools are created: one for each of the middle stages of the pipeline. The main thread waits for the middle stages to finish their work. After the middle stages are done, the last stage's work is performed (outputting the results to a file). The time when the middle three stages do their work is the region of interest (ROI). The third stage may bypass the fourth stage and send work units directly to the fifth stage.

A command line argument (tc) specifies the thread count for each pool. With tc threads per pool and three pools, $3 \cdot tc$ threads are created. To maximize performance, all n cores should have schedulable threads. To guarantee this constraint, the base version of *DEDUP* must spawn $3 \cdot n$ threads[2].

DEDUP's static thread creation oversubscribes the system. When a stage has no more work to perform, that stage's threads will exit. Oversubscription is lessened only as the application finishes stages of work. Oversubscription may starve other processes for CPU time, or cause overhead due to unnecessary context switching or lock contention. Furthermore, once *DEDUP*'s threads are created, no more threads may be created. If new cores become available, *DEDUP* cannot create

[2]In the worst case, only the fourth stage is unfinished.

Table 5: Inflate/deflate *DEDUP* modifications

Modification:	**Lines of Source Code Changed:**
Tracking pipeline flow and bottlenecks	81
Worker thread code changes	111
Main thread inflate response ability	49
Main thread deflate response ability	32
Support code to listen to inflate/deflate requests	14
Miscellaneous changes	32
Total changes	**319**

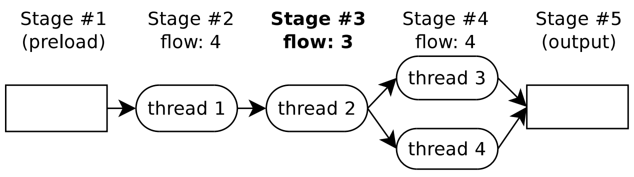

Figure 10: *DEDUP*'s pipeline after inflate (four cores)

more threads to use the additional resources.

To enable dynamic pipeline adaptation, *DEDUP* was modified to measure work unit flow. Flow indicates the fastest and slowest stages. These changes required adding profiling code and counters, as shown in Table 5. When a core becomes unavailable, a thread corresponding to the stage with the highest throughput will be shutdown. Bottleneck stages are assigned threads as cores come online. These changes took 81 lines of changes. An example of dynamic thread creation is shown in Figure 9. In the figure, the fourth stage has the lowest flow. *DEDUP* detects this and creates another thread to do the fourth stage's work. The result is the inflated version in Figure 10. In the inflated version, the new bottleneck is the third stage. If another core was offered to *DEDUP*, the third stage would be granted another thread/core.

3.3.3.4 Balanced Pipeline Experimental Results

The unmodified and modified versions of *DEDUP* are evaluated to determine: 1) Whether the modified *DEDUP* performs work more efficiently than the unmodified version given the same resources, and 2) how well the modified

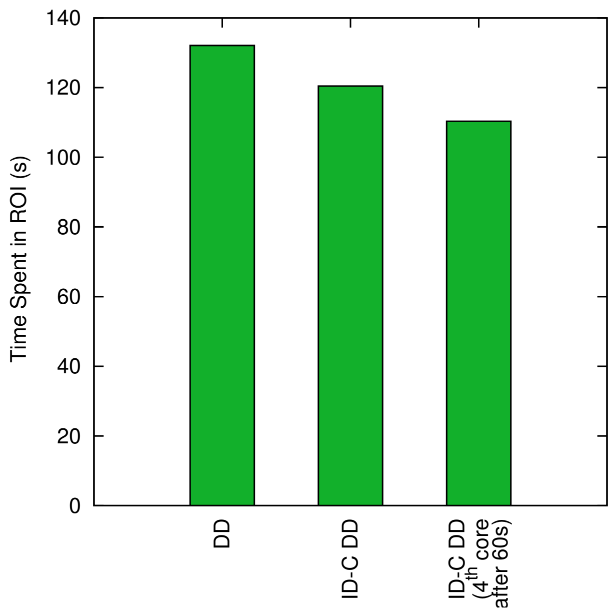

Figure 11: *DEDUP* experimental results

DEDUP responds to resources that dynamically come online.

DEDUP's native input is processed in approximately 30 seconds. To conduct longer experiments, a 3 GiB file is used. All experiments were performed on the machine described in Table 3.

The first experiment is to determine how much more quickly the modified *DEDUP* executes as compared to the unmodified *DEDUP*. Each version is executed on a separate set of cores. Both versions are forced via Linux's `taskset` command to execute on three distinct sets of cores. The time required to execute *DEDUP*'s region of interest (ROI) is measured. The unmodified *DEDUP* to spawn three threads per pipeline stage (nine threads total), as recommended for a system with three cores available.

The second experiment starts like the first. However, after 60 seconds in the ROI, an unused core becomes available (four cores total). The modified *DEDUP* creates a thread on it. The unmodified *DEDUP* is unable to respond to the availability of a new core; performance is the same as in the first experiment.

Figure 11 shows the results of these experiments. The y-axis is the time spent in the ROI. Lower is better. The first bar shows the unmodified *DEDUP*'s results (DD). The unmodified *DEDUP*

Table 6: Suggested modification strategies for to create inflate/deflate-capable applications

Application Type:	**Application Change:**
Data Parallel	Work stealing computation
Fork/Join	Work stealing computation
MapReduce	Work stealing computation
N-Dimensional Simulation	Redistribute N-dimensional space on thread count change
Pipeline Parallel	Monitor work units/second of each stage, add threads to stage with slowest throughput, remove from stages with fastest throughput
Stream processing	Work stealing computation, or, see pipeline parallel application type
Unmentioned	Work stealing computation
Last resort	Time multiplexing by OS

processes the input file in 132 seconds. The second bar shows the inflate-deflate capable *DEDUP*'s ability to process the workload (ID-C DD). It processes the input file in 120 seconds. The third bar shows the results of giving the modified *DEDUP* process an additional core after 60 seconds; the workload is processed in 110 seconds (17% speedup versus the unmodified version).

With the first two bars of Figure 11, both *DEDUP* versions are only given 3 cores yet the inflate-deflate capable *DEDUP* (ID-C DD) is 12 seconds faster. With ID-C DD, each thread is always schedulable. No thread has to wait for a core to become available. Threads will not be context switched out. This causes their working sets to remain across scheduling quanta. Additionally, with fewer threads, lock contention is reduced.

In conclusion, the modified version of *DEDUP* responds as available resources change. The source code modifications made were significantly more involved than the modifications made to *BLACKSCHOLES*. However, *DEDUP* is also significantly more complex than *BLACKSCHOLES*. These methods and results are expected to be applicable to other pipeline-parallel applications.

3.3.4 Guide for Application Developers

Most applications are not inflate/deflate-capable. Table 6 provides guidance to developers about how to make their applications inflate/deflate-capable. This reference table tells developers how they should rewrite their application to become inflate/deflate-capable. The guidelines take into account current application design. For example, the table shows that data parallel applications should be rewritten to become work stealing. A work stealing application naturally supports inflate/deflate as worker threads can be created or destroyed on demand. At least one compute thread per application is guaranteed by AUTO.

Work stealing is often the recommended solution. Work stealing is the most flexible model and is well suited for inflate/deflate. Other solutions, though less flexible, may have better cache locality and therefore better performance. For example, N-dimensional simulations (e.g., N-body molecular simulation) can have excellent cache locality if each thread works on a subspace of the N-dimensional space. Work stealing could cause cache thrashing for such applications.

As a last resort, AUTO can resort to time multiplexing an application's threads instead of creating or destroying threads. This approach can result in poor performance (e.g., if multiple threads are scheduled onto one core). If this approach is used, the application should be launched with as many threads as there are online or offline cores. With as many threads as cores, resources can never be underutilized.

3.4 CONCLUSION

This chapter introduced the AUTO framework. The system administrator provides an allocation policy and specifies control knobs to AUTO. AUTO will then make predictions about resource allocation, thus enabling the policy. To dynamically adjust thread count, the inflation/deflation programming model was introduced.

Inflation/deflation allows for dynamic application adaptation and resource allocation. To explore the difficulty and potential of creating applications which are inflate/deflate-capable, two PARSEC applications were modified: *BLACKSCHOLES* and *DEDUP*. Each application was tested in a use case.

BLACKSCHOLES was tested in a scenario where a power usage policy was in effect. *BLACKSCHOLES* shared the system with another workload. The inflate/deflate-capable *BLACKSCHOLES* achieved good throughput without sacrificing performance of the other application, resulting in increased efficiency (8.2% increase in the worst case, 51.8% increase in the best case). An inflate/deflate-capable version of *DEDUP* was tested in a scenario where compute cores dynamically arrive and leave. *DEDUP* completed work faster than the unmodified version (9% faster) and dynamically adapted itself to core availability at runtime (17% faster). Modification of *BLACK-SCHOLES* and *DEDUP* to support inflate/deflate was nontrivial but worthwhile. Based on the modification difficulty (or lack thereof) and experiments, inflate/deflate is a viable way to create autonomic applications and systems, like AUTO. Pulling from the experiences learned while modifying these applications, guidance was given to application developers to help create inflate/deflate-capable applications.

With inflate/deflate-capable applications, the full range of AUTO resource allocations can be made (i.e., thread count, core choice, core type). In the following chapters, AUTO's techniques for performance prediction are shown and evaluated.

4.0 AUTO-TC: MODELS PARAMETERIZED BY THREAD COUNT

As applications launch and terminate, demand for cores changes, perhaps radically. Too many threads in an application can cause harmful contention with other application threads (e.g., thread migrations, poor cache locality, stealing CPU time). Too few threads may underutilize the system.

The choice of how many threads to use for an application, i.e. *thread count*, should be influenced by the number of cores, number of threads in other applications, and the allocation policy. Coordination of thread count across applications ensures that applications meet performance goals, while not unnecessarily competing or underutilizing the system. This chapter introduces AUTO-TC, a technique to dynamically control thread count. AUTO-TC uses profiling to build predictive models and therefore enable policies that adjust thread count.

A solution like AUTO-TC is necessary because manual coordination of thread count among applications is burdensome and slow to react to quickly changing workloads. Even automatic methods to coordinate the creation of threads when an application is launched have limitations: As other applications come and go, the ideal number of compute threads, according to the allocation policy, for each application may vary.

The *thread configuration* of an application is the number of threads used by an application. Choosing a configuration requires considering each application's scalability, in addition to the effects of contention and interaction.

Application models can be used dynamically to choose thread configurations. A model predicts the performance of an application under a particular resource allocation. AUTO-TC automatically builds application models. The models consider inter-process contention, without a priori knowledge of the workload. The models are constructed offline and used at runtime. The effectiveness of this approach is demonstrated by a policy, DYCA (Dynamic Cooperative Allocation). DYCA uses AUTO-TC to maximize performance while ensuring that applications meet QoS.

Table 7: AUTO-TC characteristics

Property:	**Value:**
Model Function	$M_{program}(tc_{program}, tc_{corunners})$
AUTO Control Knob	thread count
Profile Space	$O(n^2)$, where n is the number of cores

Table 8: Overview of steps to use AUTO-TC

Step:	**Description:**	**Offline?**	**Online?**
1	Select sampling budget (p)	✓	
2	Select sampling method	✓	
3	Use sampling method to determine profile points	✓	
4	Profile	✓	
5	Use linear regression techniques to build model	✓	
6	Use allocation policy to determine thread counts		✓

This chapter makes the following contributions:

1. AUTO-TC: a technique to automatically build models for multithreaded CPU-bound or memory-bound applications;
2. Sampling techniques that reduce the time to build a model;
3. Experimental evaluation of sampling and model accuracy; and,
4. Experimental evaluation of the techniques in a policy, DYCA, and comparison to conventional resource allocations.

4.1 OVERVIEW OF APPROACH

AUTO-TC's characteristics are shown in Table 7. Models are application-specific and consider the effects of changing application thread count. Applications are not admitted into the system unless they have a model. In the worst case, the profiling is quadratic in the number of cores.

An overview of using AUTO-TC is shown in Figure 8. AUTO-TC builds models to make predictions about application performance. To prevent exhaustive profiling, the user selects a budget and sampling method. The sampling method will determine which profile experiments must be performed. Profiling records multithreaded application performance across thread counts of the application and a CPU-intensive, microbenchmark corunner. The performance metric is selected by the allocation policy (e.g., IPC). The profile data is used by linear regression to generate an application model.

The model predicts application performance given the thread count of the profiled application and the thread count of one or more corunners. The allocation policy will use AUTO-TC's models and, examining the model predictions, the policy will manipulate application thread counts.

The allocation policy is consulted if the workload changes. When a new application enters the system, it will create threads to obtain CPU time. Existing applications may be forced to respond to the arrival of the new application (e.g., by decreasing their thread count). Similarly, when an application exits, CPU time may become available. As a result, the number of threads in the remaining applications might be increased.

4.2 OFFLINE PROFILING AND MODEL GENERATION

Profiling is used to observe application performance across thread counts. We also consider a second parameter, the number of other CPU-bound threads in the system[1], in order to observe and model the effects of CPU contention on application performance. A multithreaded synthetic benchmark (described in Section 4.4) varies CPU contention. The profiling space is three dimensional:$\langle tc_{program}, tc_{corunners}, z\rangle$. $tc_{program}$ is the number of CPU-bound threads in the application. $tc_{corunners}$ is the number of other CPU-bound threads. z is the ratio of the application's performance with $tc_{program}$ threads, sharing the system with $tc_{corunners}$ CPU-bound threads, over the application's performance with one thread on an idle system. Profiling determines z.

Ideally, the profiling step should examine an application's performance at every value of $tc_{program}$ and $tc_{corunners}$. However, exhaustive profiling is prohibitively expensive. If an appli-

[1]For brevity, the term "CPU-bound threads" refers to both memory-bound threads and CPU-bound threads.

```
RandomSample($numcpus, numpoints$)
1  // Return a set of two-dimensional points, each a different profiling experiment.
2  $testPoints = \{\langle 1, 0\rangle\}$
3  while $|testPoints| < numpoints$
4      $otherThreads = $ UniformRandom$(0, numcpus)$
5      $appThreads = $ UniformRandom$(1, numcpus)$
6      $testPoints = testPoints \cup \{\langle appThreads, otherThreads\rangle\}$
7  return $testPoints$
```

Figure 12: Algorithm for performing random sampling of the profile space

cation is allowed to use between 1 and n threads on an n CPU system, a complete profiling run would require $n^2 + n$ data points (n application thread counts $\cdot$ $(n + 1)$ other CPU-bound thread counts). As shown in Figure 2 of this thesis's introduction, it can be beneficial to use more than n application threads on an n-CPU system.

Two sampling methods are used to reduce the amount of profiling required: 1) *random sampling* and 2) *stepwise sampling*. These methods determine the set of points that are profiled. The number of points to profile, known as the *budget*, depends on how much time the user can for the model to be produced. The trade-off between accuracy and profile budget is evaluated in Section 4.4.

4.2.1 Random Sampling

Random sampling, shown in Figure 12, uses a discrete uniform distribution to choose profile points. The random sampling algorithm accepts two parameters: 1) the number of cores ($numcpus$) and 2) the number of profile points ($numpoints$).

The algorithm generates tuples in the form $\langle tc_{program}, tc_{corunners}\rangle$, ensuring that one of the

```
STEPWISESAMPLE($numcpus$, $appStep$, $otherStep$)
1  // Return a set of two-dimensional points, each a different profiling experiment.
2  $testPoints = \emptyset$
3  for $appThreads = 1$ to $numcpus$ step $appStep$
4      for $otherThreads = 0$ to $numcpus$ step $otherStep$
5          $appThreadsInt = $ ROUND($appThreads$)
6          $otherThreadsInt = $ ROUND($otherThreads$)
7          $testPoints = testPoints \cup \{\langle appThreadsInt, otherThreadsInt \rangle\}$
8  return $testPoints$
```

Figure 13: Algorithm for performing stepwise sampling of the profile space

profile points is $\langle 1, 0 \rangle$ (i.e., the point in which the application is executed with a single thread on an idle system). This point determines baseline application performance.

Random sampling is flexible, and can readily determine a set of profile points that meets the specified budget. Because the profile points are chosen randomly, the quality of a model built from those points may vary. In Section 4.4, the quality of models built via random sampling is examined.

4.2.2 Stepwise Sampling

Stepwise sampling selects profile points at regular intervals (i.e., *steps*) from one another. The step distances of each dimension can be different.

Figure 13 shows the stepwise sampling algorithm. The algorithm accepts three parameters: 1) the number of cores ($numcpus$), 2) the step distance of the application thread count ($appStep$), and 3) the step distance of the other CPU-intensive thread count ($otherStep$). Step distances may be any real number greater than or equal to 1.

$$\mathit{shift}(\mathit{perf}, \mathit{appThreads}, \mathit{otherThreads}, n) = \frac{\mathit{perf} \cdot \max(\mathit{appThreads} + \mathit{otherThreads}, n)}{n}$$
$$\mathit{shift}^{-1}(\mathit{perf}, \mathit{appThreads}, \mathit{otherThreads}, n) = \frac{\mathit{perf} \cdot n}{\max(\mathit{appThreads} + \mathit{otherThreads}, n)}$$

Figure 14: The shift transformation compensates for CPU contention

Starting at the point $\langle 1, 0 \rangle$, the algorithm varies the number of application threads between 1 and $\mathit{numcpus}$, incrementing by $\mathit{appStep}$. For each setting of the thread count, the algorithm varies the number of other CPU-intensive threads between 0 and $\mathit{numcpus}$, incrementing by $\mathit{otherStep}$. Depending on the values of $\mathit{appStep}$ and $\mathit{otherStep}$, some points will not lie on integer coordinates. As thread counts must be integers, the stepwise algorithm rounds non-integer thread counts to the nearest integer.

The number of selected profiling points is approximately $\lfloor \frac{\mathit{numcpus}}{\mathit{appStep}} \rfloor \cdot \lfloor \frac{n+1}{\mathit{otherStep}} \rfloor$. Increasing a step value results in fewer profile points, but diminishes the amount of information about an application's performance. For example, increasing $\mathit{appStep}$ gives fewer application-thread-count settings, which may make it more difficult for a model to capture scalability trends as application thread count is changed.

For a given budget, there may be several settings of $\mathit{appStep}$ and $\mathit{otherStep}$ that meet that budget. In Section 4.4, $\mathit{appStep}$ and $\mathit{otherStep}$ are varied to observe how the best settings of each depend on the application and budget.

4.2.3 Building an Application Model

After sampling chooses the profile points, AUTO-TC profiles applications to observe their performance in the selected configurations. An application model is then built from the profile data, capturing the trends in the performance information.

Application performance often grows non-linearly due to a lack of access to cores (e.g., if the system is overcommitted) and inherent scalability limitations. Modeling a non-linear phenomenon (e.g., by using polynomials of degree 2 or higher) is possible, but subject to overfitting and may require burdensome amounts of training data. Predicting a plane (i.e., with multivariate linear regression) requires less training data and is less inclined to overfitting. Therefore, linear regression is preferable *if* the phenomenon is linear.

A post-processing transformation is used to account for the non-linear growth in performance and to enable linear regression: the *shift* transformation. The shift transformation is shown in Figure 14. This transformation accounts for the fact that if more threads than cores are created, the threads will compete for CPU time. The transformation compensates for contention by scaling performance proportional to how much the system was overcommitted. For example, if three applications each create 12 threads (36 threads total) on 24 cores, then each application effectively has access to only 8 cores—not 12 cores. After the transformation, the profile data is better suited to linear regression. At runtime, if the model is consulted to make a prediction about performance, the inverse is done (also in Figure 14).

Given the set of three-dimensional profiling points after the shift, multivariate linear regression finds the three-dimensional plane that best fits the observed performance of the application. An application often has linear growth as thread count increases (at least for a while), and thus, a plane captures performance as thread count and corunner(s) thread count are varied.

An accurate model must incorporate scalability limitations of the application. Applications may not scale linearly beyond a specific thread count. The range of thread counts in which an application's performance does not scale is referred to as the *performance plateau*.

FINDPLATEAUMODEL in Figure 15 shows how to determine the model's performance plateau. The plateau is found by building many candidate models, each with a different performance plateau. The candidate model that best fits the profile data is chosen.

Figure 16 shows BUILDMODEL, used by FINDPLATEAUMODEL, to construct a model with a specified performance plateau. Because performance stops increasing once the application's thread count is beyond the plateau point, the procedure ignores profile data where the application's thread count is greater than the plateau point. Linear regression finds the plane that fits the remaining data. Because the data is three-dimensional, the plane has 3 components: an offset and two slopes. One slope, the *scalability factor*, captures how performance is affected by changing thread count. The other slope, the *interference factor*, captures how performance is affected by the presence of

```
FINDPLATEAUMODEL($trainingData, plateauPoints$)
1  // Evaluate many models by varying the plateau setting. Return the best one.
2  $bestModel = \emptyset$, $bestModelError = \infty$
3  for $p \in plateauPoints$
4      $model$ = BUILDMODEL($trainingData, p$)
5      $error$ = TESTMODEL($model, trainingData$)
6      if $error < bestModelError$ // A better fitting performance plateau.
7          $bestModel = model$
8          $bestModelError = modelError$
9  return $bestModel$
```

Figure 15: Algorithm for determining an application's performance plateau

```
BUILDMODEL($trainingData, plateauPt$)
1  $confPts = \emptyset$, $perfPts = \emptyset$
2  for $point \in trainingData$
3      if $point.threadConfiguration.appThreads \leq plateauPt$
4          $confPts.append(point.threadConfiguration)$
5          $perfPts.append(point.result)$
6  $model.plateauPt = plateauPt$
7  $model.linearModel$ = REGRESSION($confPts, perfPts$)
8  return $model$
```

Figure 16: Algorithm for building an application model

other CPU-intensive threads.

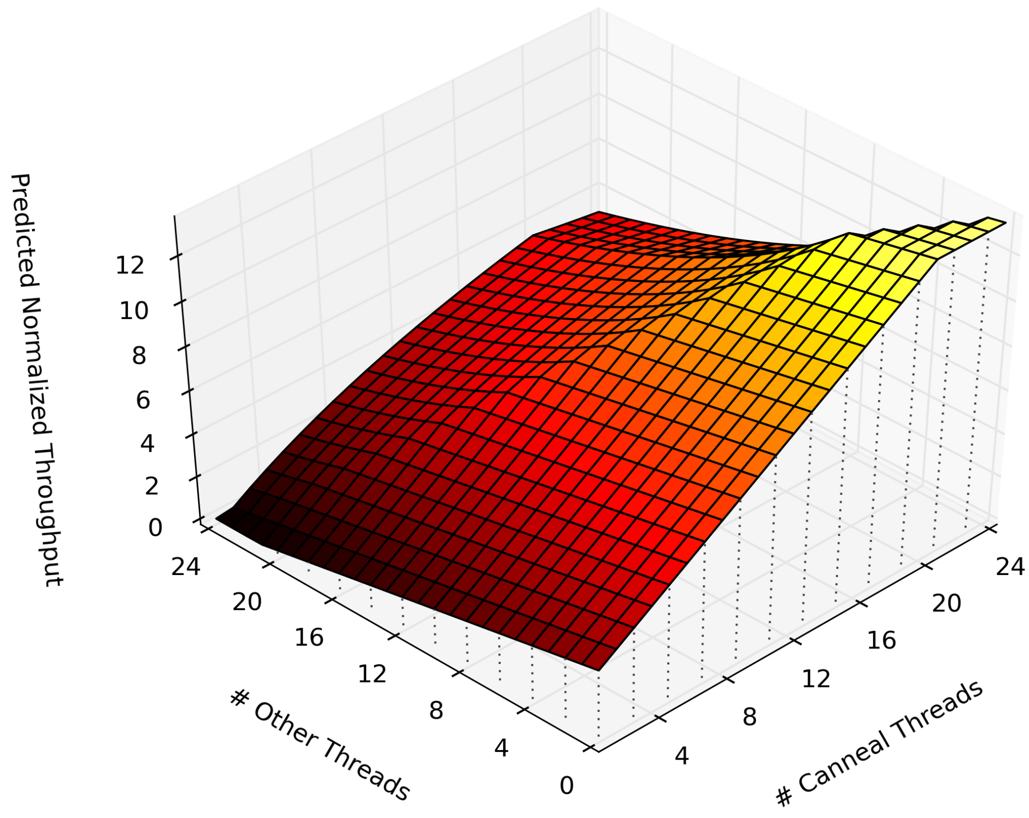

Figure 17: Visualization of an example *CANNEAL* model

Figure 17 shows a sample model for *CANNEAL* [10]. The model shows that if *CANNEAL*'s thread count is greater than 20, its performance stops increasing. This flat region is due to *CANNEAL*'s inability to scale past 20 threads (i.e., *CANNEAL*'s performance plateau). The reduction in performance as the number of other threads increases, shows that *CANNEAL* is susceptible to contention (i.e., its interference factor is negative). This characteristic can be seen by observing that *CANNEAL*'s performance decreases as additional corunner threads are used. *CANNEAL* scales well (i.e., it has a good scalability factor): if executed with 20 or more threads on an idle system, *CANNEAL* achieves a speedup of about 12.

Figure 18 shows the algorithm to query a model (CONSULTMODEL). Intuitively, an application using zero threads has no throughput. If the model is queried about the application's perfor-

```
CONSULTMODEL($model$, $conf$)
 1  if $conf.appThreads == 0$
 2      return 0
 3  $originalAppThreadCount = conf.appThreads$
 4  if $conf.appThreads > model.plateauPt$
 5      // Application model says performance does not scale past $model.plateauPt$.
 6      $conf.appThreads = model.plateauPt$
 7  $prediction = model.offset + model.scalabilityFactor \cdot conf.appThreads +$
 8                  $model.interferenceFactor \cdot conf.otherThreads$
 9  $prediction = \text{SHIFT}^{-1}(prediction, originalAppThreadCount, conf.otherThreads)$
10  if $prediction < 0$
11      // Prevent nonsensical predictions.
12      $prediction = 0$
13  return $prediction$
```

Figure 18: Algorithm for consulting a model to make a performance prediction

mance at the performance plateau, the procedure makes a prediction about the performance at the start of the plateau. To make a prediction, the procedure uses the multivariate linear regression model. The inverse of the SHIFT function (SHIFT^{-1}), is then used (Figure 14).

A model captures application behavior as revealed through profiling. AUTO-TC assumes that application behavior (e.g., scalability) is nearly consistent across inputs. For many modern multi-threaded scientific applications (e.g., the majority of PARSEC) this assumption holds [11]. Once the model is constructed, it can be used across many invocations of the application. If an application's input does significantly affect behavior, methods exist to ascertain the input's effect(s) on application scalability and susceptibility to interference [104]. The effects could then be merged with the application's model.

```
ChooseConfiguration($models, systemPolicy, possibleConfigurations$)
// Consider each possible configuration in $possibleConfigurations$,
// and return the best configuration according to the $systemPolicy$.
$bestConfiguration.policyMetric = -\infty$
for $conf \in possibleConfigurations$
    $conf.policyMetric = systemPolicy(conf, models)$
    if $conf.policyMetric > bestConfiguration.policyMetric$
        $bestConfiguration = conf$
return $bestConfiguration$
```

Figure 19: Algorithm to find the best configuration (thread counts) for applications given their models

4.3 ONLINE CONFIGURATION

The model for an application can be used to select a system configuration (number of threads in each application). Figure 19 shows the algorithm to choose a system configuration. The algorithm accepts three parameters: 1) a model for each application, 2) an evaluation function to judge the quality of a configuration (i.e., an allocation policy), and 3) the configurations to evaluate. In Section 4.4.2, AUTO-TC is used in a policy to maximize performance while meeting QoS goals.

With n cores, applications can use between 1 and n threads. With m applications there are n^m possible configurations. This algorithm uses brute force examination of each possible configuration. Because consulting a model is fast, a brute force approach is reasonable. For example, it takes less than 0.05 seconds to consider 24^5 configurations on the evaluation machine. That is, for 5 applications and 24 cores, a configuration can be chosen in 0.05 seconds. A brute force approach is also amenable to parallelization.

AUTO-TC selects a configuration when an application starts or stops execution. Application behavior may change while the application executes, necessitating that reconfiguration take place more often. Handling phase changes is expected to be a natural extension: each phase of an application can correspond to a different model, as opposed to the current approach that has one model for the whole execution of the application.

4.4 EXPERIMENTAL EVALUATION

This section examines how the models capture the behavior of the full profiling space, make accurate predictions of normalized application throughput, and enable an allocation policy. The evaluation uses a subset of the PARSEC suite: *BLACKSCHOLES* (BS), *BODYTRACK* (BT), *CANNEAL* (CN), *STREAMCLUSTER* (SC), and *SWAPTIONS* (SW) [10]. In addition, *LUXRENDER* (LX), a physically accurate multithreaded raytracer, is also used [60]. The applications were modified to track their throughput (work units per second). If multiple implementations of an application are available, the `pthread` implementation is used. Some applications from PARSEC are not used due to limitations in available inputs and/or thread configuration flexibility (e.g., requiring thread counts to be a power of 2).

For each workload, applications are suspended to RAM once fully initialized. Once all applications are ready to start their parallel work, they are simultaneously released to create worker threads. This methodology ensures maximal CPU contention. Throughput measurements are taken while all applications perform their parallel work.

The corunner is a multithreaded benchmark developed for use in profiling experiments. Each corunner thread performs integer arithmetic on a private array that fits within an L1 cache. The threads do not share data. In Section 4.5, corunners which generate various amounts of memory traffic are considered.

All experiments were done on the machine described in Table 9. Due to the number of experimental configurations, it is infeasible to perform full experiments on all 48 cores (4 sockets) of the machine. Experiments are instead performed on 24 cores (2 sockets). The reduction in available cores is enforced with Linux's `taskset` command. To provide more model flexibility, and therefore, better capture application behavior, performance plateau points can be any multiple of 0.025

Table 9: AUTO-TC experimental machine

Machine Component	**Component Details**
Processors	4 AMD Opteron 6164 HE sockets (48 cores total)
Socket	2 NUMA nodes
NUMA Node	1 L3 cache (LLC)
L3 Cache	5 MiB cache, 6x cores
Core	1.7 GHz, private L1 $ (64 KiB), private L2 $ (512 KiB)
Operating System	Linux 2.6.39.1

between 1 and 24.

4.4.1 Model Accuracy

In this section, the advantages and disadvantages of each sampling method (random and stepwise) are evaluated. Each sampling method is evaluated with four time budgets: 15, 30, 60, and 120 minutes. These budgets capture a range of delay that a developer would be willing to experience before he/she can make use of the application model. Each profiling experiment takes 2.5 minutes. Therefore, the time budgets correspond to 6, 12, 24, and 48 profile points.

Each profiling experiment consistently takes 2.5 minutes. 1.5 minutes is allowed for application initialization. Many of the applications take less time to initialize (e.g., *LUXRENDER* and *STREAMCLUSTER* start their parallel work within 10 seconds of application launch). For experimental simplicity, all applications are treated as requiring 1.5 minutes of startup time.

At the end of the initialization period, the application parallel region-of-interest is started. At the end of the minute, throughput is measured and the application is stopped. The relative increase of application throughput compared to single-threaded throughput is the speedup.

For each combination of budget, application, and sampling method, the models are built and compared. First, models are compared with regards to their ability to capture the behavior of the full, unsampled profiling space. Second, models are compared based on their ability to predict individual application and system performance.

Table 10: Comparison of the sampling techniques with a 15 minute budget (6 points)

Application	Average Random Sampling Test MSE	Best Stepwise Sampling Parameters	Best Stepwise Sampling Test MSE
BLACKSCHOLES	0.43	(12.0, 9.5)	0.18
BODYTRACK	3.84	(11.75, 11.0)	1.72
CANNEAL	3.48	(12.0, 10.25)	1.42
LUXRENDER	0.99	(12.0, 8.75)	1.43
STREAMCLUSTER	1.06	(12.0, 8.25)	1.11
SWAPTIONS	2.99	(12.0, 9.5)	0.82

Table 11: Comparison of the sampling techniques with a 30 minute budget (12 points)

Application	Average Random Sampling Test MSE	Best Stepwise Sampling Parameters	Best Stepwise Sampling Test MSE
BLACKSCHOLES	0.22	(10.25, 7.0)	0.15
BODYTRACK	2.13	(10.75, 7.5)	1.46
CANNEAL	1.91	(6.5, 10.25)	1.16
LUXRENDER	0.48	(7.25, 11.75)	0.31
STREAMCLUSTER	0.67	(6.5, 9.5)	0.52
SWAPTIONS	1.52	(7.5, 11.75)	0.77

Table 12: Comparison of the sampling techniques with a 60 minute budget (24 points)

Application	Average Random Sampling Test MSE	Best Stepwise Sampling Parameters	Best Stepwise Sampling Test MSE
BLACKSCHOLES	0.19	(7.25, 4.25)	0.14
BODYTRACK	1.55	(4.25, 6.5)	1.34
CANNEAL	1.35	(4.0, 6.5)	1.08
LUXRENDER	0.37	(3.25, 8.75)	0.29
STREAMCLUSTER	0.53	(4.0, 6.25)	0.41
SWAPTIONS	0.91	(4.5, 7.5)	0.69

4.4.1.1 Sampling Behavior For evaluation purposes, all profiling data is gathered. A model is built for each application, sampling method, and budget. Each model is used to predict the profile points not used for training. The mean squared error (MSE) of each model at predicting the unseen profile points is compared. This MSE is the *test MSE*. Tables 10–13 show these results. Table 14 shows the overall average.

Table 13: Comparison of the sampling techniques with a 120 minute budget (48 points)

Application	**Average Random Sampling Test MSE**	**Best Stepwise Sampling Parameters**	**Best Stepwise Sampling Test MSE**
BLACKSCHOLES	0.17	(3.25, 4.25)	0.14
BODYTRACK	1.40	(3.25, 4.25)	1.28
CANNEAL	1.12	(2.0, 6.5)	1.01
LUXRENDER	0.33	(3.25, 4.25)	0.29
STREAMCLUSTER	0.47	(2.0, 6.75)	0.40
SWAPTIONS	0.73	(2.0, 7.5)	0.67

Table 14: Overall comparison of the sampling techniques across profiling budgets

Budget	**Random Sampling Test MSE, σ**	**Stepwise Sampling Parameters**	**Stepwise Sampling Test MSE, σ**
6 points	2.13, 1.34	(12.0, 10.25)	1.21, 0.54
12 points	1.16, 0.73	(7.25, 9.5)	0.77, 0.47
24 points	0.82, 0.50	(4.5, 6.75)	0.69, 0.43
48 points	0.70, 0.44	(3.25, 4.25)	0.65, 0.40

Because stepwise sampling can meet a single budget in multiple ways[2], the best setting, in terms of error, is shown. The parameters to the stepwise sampling are shown in the "Best Stepwise Sampling Parameters" column in the form of ($appStep$,$otherStep$).

Random sampling is, by definition, unpredictable. It may accidentally select very useful or useless profile points. Across the evaluated budgets, it takes no more than 22 models built with random sampling before average prediction converges to within 5% of the previous prediction (i.e., the models "settle" on a prediction once there is, at most, 22 models). For the models predicting *BLACKSCHOLES*'s, *LUXRENDER*'s, and *STREAMCLUSTER*'s throughputs, the models settle once there are 12 or fewer models. To provide additional confidence that the randomly built models agree, 50 models are built and their average prediction is used. The average prediction is an *expected prediction*. In practice, a model built with random sampling (i.e., not the average-case prediction) may make better or worse predictions.

Table 10 shows the results for the smallest budget (15 minutes, 6 points). Stepwise sam-

[2]I.e., multiple settings of $appStep$, $otherStep$ can result in the same number of profile points being selected.

pling outperforms random sampling, except for *STREAMCLUSTER* and *LUXRENDER*. The poor models generated by stepwise sampling in these two cases are due to the low budget. Stepwise sampling uniformly chooses profile points, and thus a low budget may fail to reveal crucial performance trends. With a larger budget, stepwise sampling consistently outperforms random sampling.

With a 15 minute budget, some applications are more easily predicted than others. Regardless of sampling method, *BLACKSCHOLES* has a much lower test MSE than *BODYTRACK*. There is little variation among the best parameters to the stepwise sampling method: due to limited flexibility with such a small budget, all applications prefer an $appStep$ of about 12.

The second smallest budget (30 minutes, 12 points) results are shown in Table 11. Both sampling methods have improvement with 12 profile points versus the previous budget of 6 points.

However, models built with stepwise sampling have less improvement than random sampling, which is due to the already good stepwise sampling results with a 6 point budget. The best stepwise parameters for each application are now different. For example, *BLACKSCHOLES* prefers an $appStep$ of 10.25, resulting in *BLACKSCHOLES* being trained on thread counts of 1, 11, and 22. *STREAMCLUSTER* prefers an $appStep$ of 6.5 (thread counts of 1, 8, 14, and 21). Examination of the full profile data reveals that *BLACKSCHOLES*'s performance scales linearly. Therefore, experiments that vary its thread count reveal little additional information. The profile budget is better used to vary the number of other CPU-intensive threads (a smaller value of $otherStep$).

Using a 60 minute budget (24 points) results in better models, although the relative improvement in MSE diminishes. These results are in Table 12.

The biggest budget results (120 minutes, 48 profile points) are shown in Table 13. Like the 60 minute results, the 120 minute budget continues to improve a small amount. It is interesting to note that the best application model generally prefers $appStep$ to be half the value of $otherStep$. The relationship between $appStep$ and $otherStep$ suggests that for each corunner thread setting to be profiled, two different application thread settings should be profiled.

Table 14 shows overall MSE averages for each sampling policy and budget. Also shown is the standard deviation and the best average stepwise sampling parameters. This figure shows which sampling policy, on average, produces a model that best matches application behavior under full training data. In this regard, stepwise sampling is clearly more accurate than random sampling. On average, it produces *better*, more *consistent* models. For all sampling methods, larger budgets produce better models but with dwindling returns. The model design inhibits overfitting, so MSE will never be zero.

4.4.1.2 Sampling Accuracy Analysis Although Section 4.4.1.1 showed that stepwise sampling was better than random sampling, those experiments only examined the models in reflecting application behavior with a known corunner. It is also necessary to consider the behavior with one or more unknown corunners.

Experiments were conducted with all sets of 1 to 4 applications. Application thread counts were taken from $\{1, 8, 16, 24\}$. System configurations with more than 48 CPU intensive threads are disregarded, as the system is overloaded by more than a factor of 2. The evaluation of an overloaded system serves as a stress test of AUTO-TC. There were over 3,000 experiments conducted.

In each experiment, actual application throughput is measured and compared to the model predictions (one model per application per sampling method per budget, in addition to models built without sampling).

For random sampling at a given budget, the average prediction from 50 models to produce an expected prediction. The previous section determined the single set of parameters that gave the overall best performance across applications. Therefore, those parameters are used (Table 14).

Results are averaged based on how many applications were executed simultaneously (Tables 15–18). In each figure, the first column is the sampling method and budget. The next six columns show application average MSEs. The overall MSEs is in the eighth column and the ninth column shows the overall MSE of experiments not involving *CANNEAL* or *SWAPTIONS* due to their unusual behavior (discussed later).

Table 15 shows the model accuracy summaries for experiments with 1 application. Generally, using more sampling points leads to better accuracy due to more information. Stepwise sampling on average (whether the overall average or average without *CANNEAL* and *SWAPTIONS*) slightly outperforms random sampling. Some applications exhibit unusual behavior as sampling budget is increased. The extremely low MSE of *BLACKSCHOLES*'s (0.02 MSE) when using stepwise sampling (6 points) is an uncharacteristic result. The expected error is 0.14 (MSE of 0.02 is a mean error of 0.14). *LUXRENDER* performs poorly (9.22 MSE) under stepwise sampling (6 points) because its performance is overestimated at 24 threads (a normalized throughput of 19.78

Table 15: Average model accuracy using 1 application

Sampling Method	**BS MSE**	**BT MSE**	**CN MSE**	**LX MSE**	**SC MSE**	**SW MSE**	**Overall MSE**	**No CN, SW Overall MSE**
None (All Points)	0.71	2.98	10.39	0.66	0.22	1.67	2.77	1.14
Random (6 Points)	1.23	0.20	10.27	0.45	0.92	2.49	2.59	0.70
Random (12 Points)	0.77	0.58	10.48	0.33	0.40	1.86	2.40	0.52
Random (24 Points)	0.68	1.19	9.57	0.48	0.28	1.62	2.30	0.66
Random (48 Points)	0.56	1.80	9.94	0.55	0.24	1.52	2.44	0.79
Stepwise (6 Points)	0.02	1.76	5.06	9.22	1.83	0.22	3.02	3.21
Stepwise (12 Points)	0.31	0.60	8.77	0.45	0.34	1.09	1.93	0.42
Stepwise (24 Points)	0.13	1.46	6.68	0.26	0.11	1.07	1.62	0.49
Stepwise (48 Points)	0.33	1.40	9.03	0.54	0.06	1.03	2.06	0.58

versus 13.75). Other experiments show that models for *LUXRENDER* generally perform well.

BODYTRACK, *CANNEAL*, and *SWAPTIONS* have counterintuitive results. *CANNEAL* and *SWAPTIONS* are sensitive to the choice of profile points. For *CANNEAL* and *SWAPTIONS* models built with stepwise sampling, more profile points results in worse models due to the selected profiling points.

BODYTRACK's error comes from how the model-building process builds trends from the profile data. *BODYTRACK* scales well (slope of 1) for 1 to 4 threads, then its scalability drops, curving but never flattening. As more profile points are observed, the models "notice" the area of sharp performance increases and try to account for it. Without more information and flexibility[3], they overestimate performance for low thread counts and overestimate performance for higher thread counts. Nevertheless, *BODYTRACK*'s MSE is reasonable (under 3.00). This behavior is also exhibited in experiments with corunners.

Table 16 shows model accuracy in experiments with two applications. Applications may contend for CPU time and/or processor resources (e.g., cache space). The total number of CPU-intensive threads may exceed the number of cores. In general, the models have more prediction error due to application interactions on a shared, potentially overloaded system (about 40% higher MSE). Overall averages show that stepwise sampling consistently outperforms random sampling, but not drastically so. The advantage of stepwise sampling is a result of relatively simple system

[3]Models are limited to a single area of linear growth, followed by an optional area of no-growth.

Table 16: Average model accuracy using 2 applications

Sampling Method	BS MSE	BT MSE	CN MSE	LX MSE	SC MSE	SW MSE	Overall MSE	No CN, SW Overall MSE
None (All Points)	2.27	3.74	4.60	3.18	0.49	8.27	3.76	2.12
Random (6 Points)	2.46	2.57	4.21	2.92	0.59	9.49	3.71	1.88
Random (12 Points)	2.22	2.56	3.80	2.78	0.42	8.90	3.45	1.73
Random (24 Points)	2.24	2.77	3.79	2.95	0.39	8.53	3.44	1.81
Random (48 Points)	2.16	3.11	4.14	3.05	0.39	8.38	3.54	1.89
Stepwise (6 Points)	1.82	2.43	3.02	4.20	1.12	5.89	3.08	2.36
Stepwise (12 Points)	2.30	2.56	3.73	3.27	0.33	7.95	3.36	1.81
Stepwise (24 Points)	2.07	2.96	3.67	3.10	0.34	8.31	3.41	1.81
Stepwise (48 Points)	2.06	2.94	3.97	3.02	0.49	7.86	3.39	1.82

Table 17: Average model accuracy using 3 applications

Sampling Method	BS MSE	BT MSE	CN MSE	LX MSE	SC MSE	SW MSE	Overall MSE	No CN, SW Overall MSE
None (All Points)	2.86	4.01	3.29	3.84	0.42	9.08	3.92	2.34
Random (6 Points)	2.92	2.61	2.56	3.63	0.60	8.49	3.47	2.06
Random (12 Points)	2.77	2.59	2.36	3.35	0.41	8.59	3.34	1.92
Random (24 Points)	2.83	2.89	2.53	3.58	0.35	8.61	3.47	2.03
Random (48 Points)	2.74	3.33	2.86	3.70	0.35	8.87	3.64	2.12
Stepwise (6 Points)	2.43	3.12	2.26	3.76	0.68	6.57	3.14	2.33
Stepwise (12 Points)	3.15	2.78	2.38	4.22	0.47	8.09	3.52	2.25
Stepwise (24 Points)	2.80	3.42	2.92	3.70	0.37	9.45	3.77	2.18
Stepwise (48 Points)	2.67	3.19	2.86	3.70	0.46	8.70	3.60	2.10

contention as compared to evaluations with three or four applications.

For two applications at once, models for *CANNEAL* show a significant reduction in MSE as compared to predictions for performance when executed alone. The improvement in accuracy is due to the models underestimating *CANNEAL*'s performance when not sharing the system. Models for *SWAPTIONS* have the opposite tendency. They predict *SWAPTIONS*'s performance relatively accurately if *SWAPTIONS* is executed alone, but consistently underestimate performance on a shared system.

Tables 17 and 18 show results with three and four applications executing at once. Overall MSEs are not significantly worse than two applications at once. Disregarding two outliers (*CANNEAL*,

Table 18: Average model accuracy using 4 applications

Sampling Method	**BS MSE**	**BT MSE**	**CN MSE**	**LX MSE**	**SC MSE**	**SW MSE**	**Overall MSE**	**No CN, SW Overall MSE**
None (All Points)	2.32	3.55	2.68	3.40	0.34	7.59	3.31	1.97
Random (6 Points)	2.30	2.19	1.86	3.20	0.64	6.67	2.81	1.72
Random (12 Points)	2.23	2.23	1.82	2.91	0.37	6.99	2.76	1.59
Random (24 Points)	2.30	2.54	2.04	3.15	0.28	7.05	2.90	1.69
Random (48 Points)	2.22	2.95	2.32	3.26	0.27	7.35	3.06	1.78
Stepwise (6 Points)	2.05	2.93	1.98	3.18	0.48	5.50	2.69	1.95
Stepwise (12 Points)	2.73	2.47	1.91	3.90	0.48	6.63	3.02	2.01
Stepwise (24 Points)	2.37	3.08	2.57	3.26	0.30	8.03	3.27	1.85
Stepwise (48 Points)	2.18	2.84	2.38	3.30	0.39	7.30	3.06	1.77

and *SWAPTIONS*), stepwise sampling generally leads to models with lower error as the budget is increased. This effect is due to additional interference information.

For three or more applications, stepwise sampling is no longer the best sampling method: random sampling almost always has slightly better MSE. Random sampling is now better because it is better at predicting interference across multiple running applications. Random sampling picks profile points uniformly at random. In these experiments, stepwise sampling's parameters are hard-coded to vary the number of application threads about twice as much as the number of corunner threads (Table 14).

Random sampling results in models that *on average* work about as well as stepwise sampling. Due to the random selection of sampling points, a single model *may* behave better or worse than the average model. To get an expectation, the average prediction of 50 randomly produced models was computed.

Across trials and sampling methods, AUTO-TC's models often predict application performance quite well (e.g., MSEs of less than 3). Even if an application does not perform as well as desired (e.g., *CANNEAL*, *SWAPTIONS*), the models still capture general trends.

Further testing was performed with all pairs of applications across all thread counts. Models built with stepwise sampling and a 6 point budget select configurations that achieve on average 90% of the available system performance. This indicates that the models do a good job maximizing system performance across diverse workloads.

Based on the results in this section, stepwise sampling with a budget of 6 points is the preferred sampling technique and budget. Models built in this way achieve a good balance between accuracy and the time necessary to build them. Using a larger sampling budget can result in slightly better predictions but the additional profiling cost may double (or more) the time required to build a model.

4.4.2 DYCA: A Policy to Meet Quality of Service Constraints

To demonstrate the utility of the models, the *Dynamic Cooperative Allocation policy (DYCA)* policy is implemented and evaluated. DYCA makes use of the AUTO-TC models to meet QoS. Its secondary objective is to maximize system throughput. DYCA is useful in, for example, shared scientific-computing environments (e.g., a shared university machine). It allows applications to utilize idle resources, while still respecting individual user needs. The model building techniques are expected to enable additional policies that work with other performance metrics.

In Section 4.4.1.2, stepwise sampling with a budget of 6 points was selected as the best way to build models. Table 19 contains the parameters of the models used by DYCA[4].

Figure 20 gives the algorithm for DYCA. The policy uses the models to quantify the quality of a configuration (i.e., a prediction of how many applications will meet QoS). Configuration quality is equal to the number of applications that are expected to meet QoS. This number is multiplied by a large constant, as meeting QoS is the primary goal. To break ties, the expected system throughput is added[5]. The configuration that best meets QoS while maximizing aggregate system performance will be applied by DYCA.

The effectiveness of the DYCA is compared to three static policies: 1) Free-for-All (FREFORA), 2) "Uniform Partition" (UNIPAR), and 3) "Application Maximum" (APPMAX). FREFORA grants each application as many threads as cores. It ensures that the system is fully utilized, at the risk of overcommitting cores. UNIPAR considers that multiple applications will be executed. It grants each application an equal share of cores (i.e., if there are m applications and n cores then each is allowed to create $\frac{n}{m}$ threads). This policy avoids overcommitting the system.

APPMAX takes into account limits in scalability. Each application is granted as many threads

[4]A performance plateau at ∞ indicates expected continued scaling.

[5]The constant large number should be greater than any possible expected system throughput.

Table 19: QoS experiment application models

Application	Offset	Interference Factor	Scalability Factor	Performance Plateau
BLACKSCHOLES	0.66	-0.02	0.50	∞
BODYTRACK	1.62	-0.12	0.66	∞
CANNEAL	1.85	-0.13	0.57	∞
LUXRENDER	0.55	-0.03	0.80	∞
STREAMCLUSTER	0.59	0.02	0.21	∞
SWAPTIONS	1.04	-0.09	0.85	∞

```
DYNAMICCOOPERATIVEALLOCATIONEVALUATOR($configuration$, $models$)
1  // Return a metric signifying the quality of $configuration$. Prioritize meeting quality of
2  // service goals. As a tie breaker the configuration that maximizes system performance
3  // will be selected.
4  $prediction = $ FINDPREDICTEDBEHAVIOR($configuration, models$)
5  $metric = prediction.QoSGoalsMet * LARGE\_CONSTANT + prediction.SystemThroughput$
6  return $metric$
```

Figure 20: Algorithm for Dynamic Cooperative Allocation

as necessary to ensure that it reaches its performance plateau on an idle system. For this policy, performance plateaus were determined with full profiling data, without a training corunner, and the FINDPLATEAUMODEL algorithm. Therefore, this policy requires a profiling step. With multiple applications, it may overcommit the system. This policy has *BLACKSCHOLES* and *SWAPTIONS* use 24 threads. *BODYTRACK* and *CANNEAL* use 19 and 20 threads. *LUXRENDER* uses 16 threads and *STREAMCLUSTER* uses 5.

To compare the policies, the system executes 1 to 4 applications simultaneously. Each application is assigned a QoS, Q. A QoS of Q means that the application should execute at least Q times faster than if executed with a single-thread. Q should be chosen by the user after considering minimum performance requirements and application scalability. We select Q from the set $\{2, 5, 8\}$. These Q values represent a range of priorities from low to high. All valid combinations of applica-

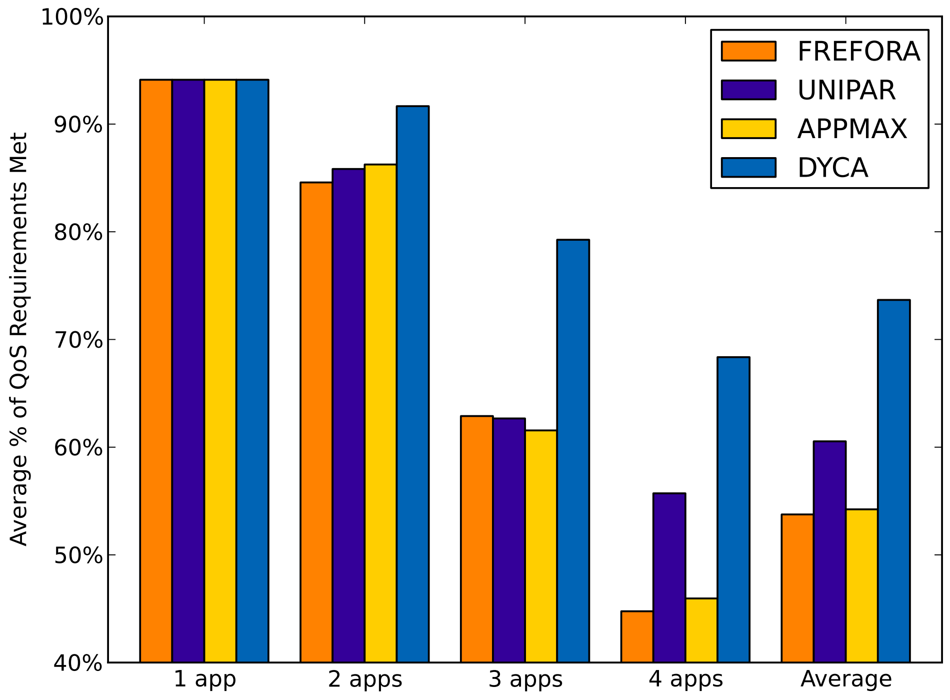

Figure 21: Percentage of QoS met, across number of concurrent applications

Table 20: Percentage of QoS met, across workloads using a specific application

Application	FREFORA QoS Met	UNIPAR QoS Met	APPMAX QoS Met	DYCA QoS Met
BLACKSCHOLES	48.5%	56.1%	48.0%	72.0%
BODYTRACK	52.6%	59.3%	53.8%	72.2%
CANNEAL	52.1%	60.4%	53.7%	71.9%
LUXRENDER	54.0%	61.3%	53.6%	74.4%
STREAMCLUSTER	46.3%	53.9%	49.4%	68.0%
SWAPTIONS	53.0%	61.5%	51.8%	74.0%
across all applications	53.8%	60.5%	54.2%	73.7%

tions and QoS settings are evaluated (over 1,700 experiments). *STREAMCLUSTER* scales poorly and cannot achieve a Q of 8 on the evaluated machine, so it is evaluated for $Q \in \{2, 5\}$.

4.4.2.1 Achieved QoS Figure 21 shows the average number of times QoS is met, divided into categories based on how many applications executed at once. Executing a single application, regardless of policy, results in QoS being met 94% of the time. QoS is not 100% due to *STREAM-CLUSTER*. As shown in Figure 1, *STREAMCLUSTER*'s behavior has several local maxima and minima. Although *STREAMCLUSTER* can, in the right circumstances, meet a QoS goal of 5, none of the evaluated policies select a thread count which does so.

For 2 applications executing at once, the static policies perform nearly the same, meeting QoS about 85% of the time. However, DYCA meets QoS 92% of the time (about 8% more often than the next best policy) because it considers application scalability and susceptibility to interference.

Executing 3 or 4 applications at a time, all policies have a more difficult time meeting QoS. The increased difficulty is due to more applications contending for the same cores. Once again, the static policies perform worse than DYCA. With 3 applications, the best static policy (FREFORA) meets QoS 63% of the time. DYCA meets QoS 79% of the time (25% more often). With 4 applications, it meets QoS 68% of the time, which is 23% better than UNIPAR, the next best static policy. As the average results columns show, regardless of the number of applications, DYCA meets QoS 22% more often than the next best static policy, UNIPAR.

Because applications exhibit diverse behavior, experiments are summarized to reveal how the application itself affects the ability of the policies to meet QoS. Table 20 shows the results grouped in this way. For example, a system executing *LUXRENDER*, among other applications, allows DYCA to more easily meet QoS (74%). *LUXRENDER* scales well and can meet the toughest QoS constraint ($Q = 8$) with a low thread count (9), leaving cores available for other applications. However, if executing *STREAMCLUSTER*, DYCA meets QoS 68% of the time. *STREAMCLUSTER* scales poorly, causing other applications to be unable to meet QoS due to limited CPU availability.

4.4.2.2 System Performance In addition to examining how well each policy meets QoS, it is also important to consider overall system throughput. The policies may conservatively meet QoS rather than maximize throughput while also meeting QoS. Figure 22 shows the average system aggregate throughput as a function of how many applications execute together.

For a system with a single application, each policy, except for APPMAX, achieves a throughput of 15. APPMAX achieves a throughput of about 14. It may underestimate how many threads an application requires to reach maximum performance.

If multiple applications are executed, performance differences become apparent. With two ap-

Figure 22: System performance for each policy, across number of concurrent applications

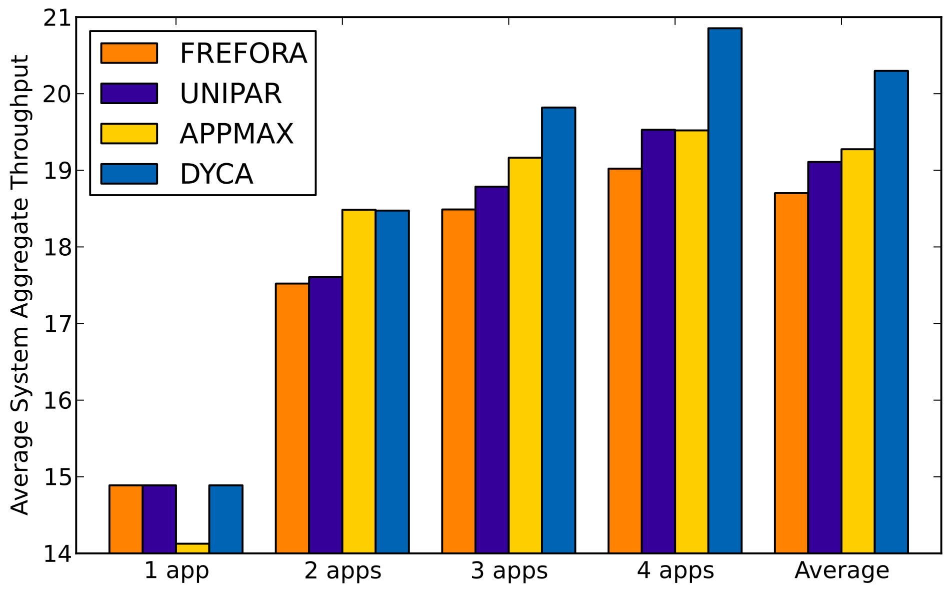

Table 21: System performance for each policy, across workloads using a specific application

Application	**FREFORA Throughput**	**UNIPAR Throughput**	**APPMAX Throughput**	**DYCA Throughput**
BLACKSCHOLES	17.7	18.6	17.9	20.0
BODYTRACK	19.4	19.5	19.9	20.5
CANNEAL	19.0	19.8	19.6	20.5
LUXRENDER	19.2	19.6	19.7	20.6
STREAMCLUSTER	17.6	17.5	18.9	19.8
SWAPTIONS	19.8	20.1	20.0	21.3
across all applications	18.7	19.1	19.3	20.3

plications, DYCA and APPMAX essentially tie, outperforming FREFORA and UNIPAR by about 5%. If three applications are executed, DYCA has slightly better system performance, achieving 19.8 versus the next best policy at 19.2 (APPMAX). For four applications, DYCA continues to outperform the other policies. It achieves an average system throughput of 20.9 versus 19.5 for APPMAX. Policies that consider application performance at various thread counts, like DYCA, have a distinct advantage over purely static policies. AUTO-TC clearly enables such policies.

Table 21 shows the average aggregate system throughput separated into categories based on

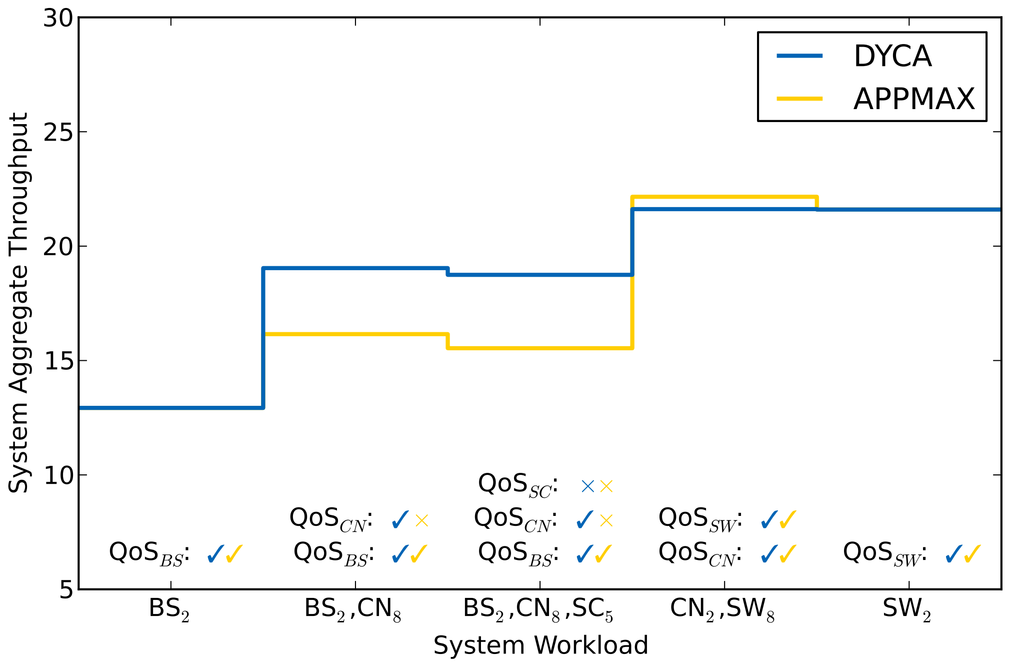

Figure 23: Example dynamic workload scenario showing system throughputs and whether QoS goals of applications are met. Numerical subscripts indicate QoS goals.

whether an experiment used a particular application. There is some variation to be observed, suggesting that certain applications help or hinder a policy to maximize system performance. For example, experiments with *SWAPTIONS* have the highest average system throughput across all policies. This fact suggests that *SWAPTIONS* behaves "well" and allows policies to maximize throughput. Experimental data shows that *SWAPTIONS* scales very well and can achieve a throughput of 21.6 if using 24 threads. Perfect linear scaling (slope of 1) would achieve a throughput of 24. Experiments involving *STREAMCLUSTER* have the worst average system throughput due to *STREAMCLUSTER*'s poor scalability.

A representative scenario is examined, wherein the workload is changed throughout multiple intervals (Figure 23). The system throughput and QoS achieved by DYCA and APPMAX are shown. The workload in each interval is shown. Numeric subscripts indicate target QoS. The height of the lines indicates normalized system aggregate throughput across each policy (higher is better). Whether an application met QoS during an interval is shown near the x-axis. A ✓ indicates

QoS was met. A $\times$ indicates the application did not meet QoS constraints.

In the first interval, both policies achieve identical system throughput and QoS by giving *BLACKSCHOLES* 24 threads. The second interval again shows that both policies meet QoS, but DYCA chooses a configuration that improves system performance. APPMAX assigns 24 threads to *BLACKSCHOLES* and 20 threads to *CANNEAL*. DYCA assigns 4 and 20 threads respectively. In the third interval, with 3 applications executing, both policies fail to achieve all QoS constraints. However, DYCA achieves better system throughput and meets 2 out of 3 QoS goals. APPMAX only meets 1 QoS goal. It assigns 24, 20, and 5 threads to *BLACKSCHOLES*, *CANNEAL*, and *STREAMCLUSTER* respectively. This assignment overloads the system (i.e., more CPU intensive threads than cores). DYCA assigns 4, 19, and 1 threads. *BLACKSCHOLES*'s desired QoS is low ($Q = 2$) and therefore needs only four threads to meet QoS. The remaining cores are not overcommitted by DYCA. *CANNEAL*, which scales the best, is given the majority of cores. *STREAMCLUSTER* scales poorly and DYCA "gives up" on it, as it is expected to be unable to meet QoS.

In the following interval, both applications satisfy QoS, although APPMAX results in marginally better system throughput. APPMAX statically gives 20 threads to *CANNEAL* and 24 threads to *STREAMCLUSTER*. DYCA gives 5 threads to *CANNEAL* and 19 threads to *STREAMCLUSTER*. In this interval, DYCA behaves conservatively, missing an opportunity for slightly increased system performance.

In the final interval, both policies tie (24 *SWAPTIONS* threads). Overall, DYCA was better able to meet QoS and maximize system aggregate throughput than APPMAX.

Across all experiments, on average, DYCA achieves an average system throughput of 20.30. The next best static policy is APPMAX, which achieves an average system throughput of 19.3. However, system performance is secondary to QoS. Even though DYCA meets QoS 22% more often, it also improves system performance by about 6% (DYCA's 20.3 average system throughput versus UNIPAR's 19.1 average system throughput).

4.5 SENSITIVITY TO MEMORY SUBSYSTEM INTERFERENCE

Multithreaded applications contend for memory resources (e.g., shared caches and memory buses) in addition to CPU access. Memory contention could significantly degrade application performance. This section evaluates the use of DYCA models built with a memory intensive corunner. Recommendations are provided about the choice of corunner to use when building application models.

Three memory intensive corunners are considered. Each corunner thread operates on a private 10 MiB array of integers. The corunners use different array access patterns. The first corunner reads and writes each integer (stride of 1), the second corunner operates on every 8th integer (stride of 8), and the last corunner does the same to every 16th integer (stride of 16). On the evaluation machine, every 16th integer lies on a new L3 cache line.

The access patterns capture a range of memory behavior, from good locality (stride of 1) to poor locality (stride of 16: each read and write occurs on a new cache line). Application models are built using the new memory intensive corunners and the best sampling procedure (stepwise sampling with 6 points). The use of new corunners creates three model variants, each of which is evaluated under DYCA: DYCA+M$_{stride_1}$, DYCA+M$_{stride_8}$, and DYCA+M$_{stride_{16}}$.

Table 22 shows the percent of experiments that met QoS constraints, across each model variant. The more locality there is in the training corunner, the better the policy is able to meet QoS. DYCA+M$_{stride_1}$ meets QoS more often than DYCA+M$_{stride_8}$ (16% more often). DYCA+M$_{stride_8}$ is slightly better than DYCA+M$_{stride_{16}}$ (meeting QoS about 3% more often). The original corunner's working set fits within each core's L1 cache. Therefore, it has excellent locality. The original corunner achieves QoS more frequently than any other policy (71.2% of the time versus the second best policy's 70%). These results show that the original corunner results in more accurate models and that its interference is more similar to that of the evaluated applications.

Table 23 shows the average system throughput achieved by the policies. The best policy is still the base one with the CPU-intensive corunner (DYCA). DYCA+M$_{stride_1}$ and DYCA+M$_{stride_8}$ on average perform the same. For experiments involving *BLACKSCHOLES*, DYCA+M$_{stride_1}$ performs slightly better than DYCA+M$_{stride_8}$. However, in experiments involving *BODYTRACK*, *STREAMCLUSTER*, and *SWAPTIONS*, DYCA+M$_{stride_8}$ performs better than DYCA+M$_{stride_1}$. The

Table 22: Percentage of QoS met under the DYCA policy across workloads for a specific application, as the corunner is varied

Application	**DYCA+M$_{stride_1}$ QoS Met**	**DYCA+M$_{stride_8}$ QoS Met**	**DYCA+M$_{stride_{16}}$ QoS Met**	**DYCA QoS Met**
BLACKSCHOLES	68.7%	65.1%	63.2%	72.0%
BODYTRACK	69.2%	65.0%	63.3%	72.2%
CANNEAL	67.9%	63.0%	60.4%	71.9%
LUXRENDER	69.6%	64.9%	62.4%	74.4%
STREAMCLUSTER	63.9%	60.2%	58.1%	68.0%
SWAPTIONS	70.2%	66.3%	64.5%	74.0%
across all applications	70.0%	66.0%	63.8%	73.7%

Table 23: Average system aggregate throughput under the DYCA policy across workloads for a specific application, as the corunner is varied

Application	**DYCA+M$_{stride_1}$ Throughput**	**DYCA+M$_{stride_8}$ Throughput**	**DYCA+M$_{stride_{16}}$ Throughput**	**DYCA Throughput**
BLACKSCHOLES	19.8	19.7	19.4	20.0
BODYTRACK	20.2	20.3	20.1	20.5
CANNEAL	20.3	20.3	20.0	20.5
LUXRENDER	20.3	20.3	20.1	20.6
STREAMCLUSTER	19.1	19.4	19.1	19.8
SWAPTIONS	21.1	21.2	21.0	21.3
across all applications	20.0	20.0	19.8	20.3

policy that uses the models trained on the most memory intensive corunner, DYCA+M$_{stride_{16}}$, always performs worst.

DYCA+M$_{stride_8}$ is occasionally better than DYCA+M$_{stride_1}$ at maximizing throughput because of the policy and the effects of the training corunners on the application models. Across experiments, it predicts that it will meet 8% fewer QoS goals compared to DYCA+M$_{stride_1}$. Consequently, it has more freedom to maximize system throughput. However, the quality of models built with the $stride_8$ corunner varies across applications, resulting in DYCA+M$_{stride_8}$ only occasionally surpassing DYCA+M$_{stride_1}$ in throughput.

The model accuracy directly affects whether DYCA can choose a configuration that meets QoS and maximizes throughput. The original DYCA performed best: It met QoS more often while also achieving better performance. Based on this evidence, the use of a non-memory-intensive corunner for these applications is better than using a memory intensive one.

Because the non-memory intensive corunner resulted in better models, we conclude that the evaluated applications are not memory intensive. Although the PARSEC applications and *LUXRENDER* contend for memory subsystem resources, they do not contend enough to need a memory intensive training corunner. In fact, training with a memory intensive corunner leads to less accurate models. Consequently, the decisions made by a policy using the models may harm performance. Furthermore, some modern processors, like those in the evaluation system, use hardware techniques to avoid unnecessary cache interference [23].

For applications which do cause significant memory interference with each other, it may be beneficial to build the models with a memory intensive corunner (i.e., the models should be trained with corunners that are similar to the future expected corunners). *Regardless of the corunner used,* all policies achieved better QoS and performance than the best static policy: Even training with a “wrong” corunner (e.g., a corunner that is significantly more or less memory intensive than the runtime applications) will benefit the users of the system by enabling dynamic allocation policies.

4.6 CONCLUSION

This chapter introduced AUTO-TC, an automated technique to construct performance models, make performance predictions, and dynamically adjust thread count to meet performance goals. Its models take into account the thread count of the application, the number of other CPU-bound threads, and number of cores. Sampling methods were introduced to reduce the number of training points to build a model. The models were evaluated in two ways. First, their prediction accuracy was examined. Second, their ability to predict application performance with untrained-for corunners was tested.

After performing those evaluations, a preferred sampling technique (stepwise sampling) and budget (6 profile experiments) were chosen. Models built using this preferred technique and budget were used by a policy, DYCA, to dynamically select application thread counts, maximize system performance, and meet QoS. Compared to static policies which do not consider the dynamic workload, our models and dynamic configuration policy increased system throughput (6% higher) and better met application QoS (22% more often). Therefore, using AUTO is a viable solution to dynamically configure thread counts on shared CMP machines.

5.0 AUTO-FINITY: MODELS PARAMETERIZED BY AFFINITY

This chapter shows the importance of selecting affinity (i.e., thread-to-core mappings), the challenge involved in selecting affinty, and how AUTO enables choosing affinity to improve application performance. Two techniques to choose application affinity are introduced and evaluated. These techniques together constitute AUTO-FINITY.

The first technique determines an application's preferred affinity via a thread-count independent model. It is discussed in Section 5.2. The second technique predicts application performance given an affinity via models built for a particular thread count. It is shown in Section 5.3. Both techniques can be updated online.

This chapter:

1. Shows the importance of choosing application affinities;
2. Presents AUTO-FINITY, a set of techniques to select application affinity;
3. Evaluates a policy, MAGNET, which uses AUTO-FINITY's first technique to minimize execution times across applications and thread counts; and,
4. Evaluates a policy, COMPASS, which uses AUTO-FINITY's second technique to predict the performance of affinities and select affinities that minimize execution time.

5.1 MOTIVATION

Today's computers for scientific and server workloads are multisocket, multicore machines, capable of massive amounts of parallelism. Cores share multiple resources, such as caches, memory controllers, and interconnects. For a multithreaded application, core allocation decisions can be

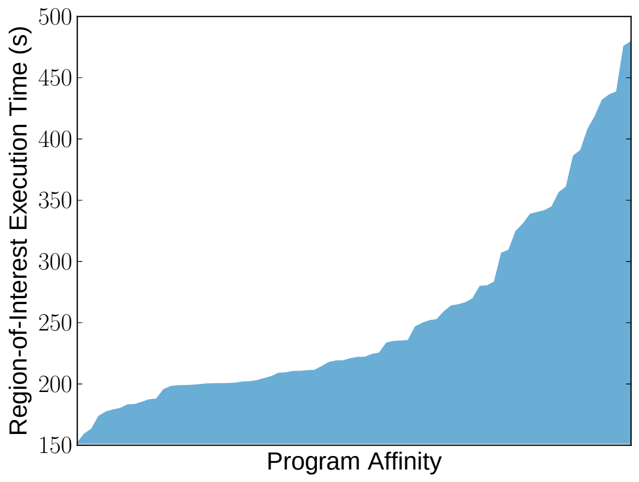

Figure 24: Execution times of *STREAMCLUSTER*'s region-of-interest for different affinities on a 48-core machine (8 threads)

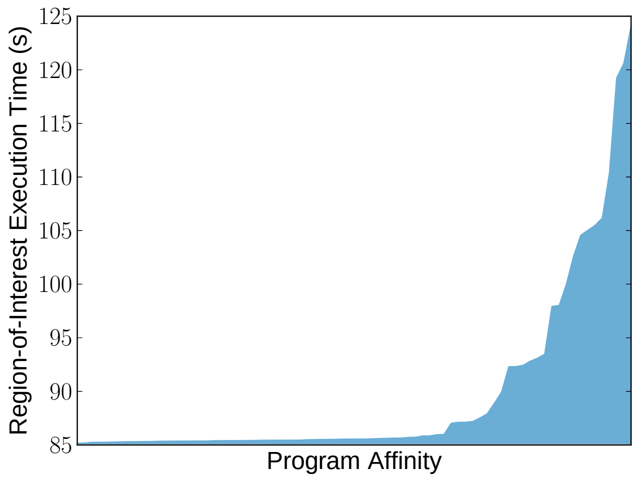

Figure 25: Execution times of *SWAPTIONS*'s region-of-interest for different affinities on a 48-core machine (8 threads)

made for whether threads share data caches. Alternatively, threads can be given exclusive access to resources. Because resource allocation impacts performance, it must be performed carefully and in response to runtime conditions, such as core availability and workload demand.

It is well known that there is often single "best" application affinity. For example, Figure 24 shows *STREAMCLUSTER*'s region-of-interest (ROI) execution time when executed with different affinities with 8 threads on a 48-core machine (machine described in Table 25) [10]. The ROI is the application's parallel section where the "work" is done.

For *STREAMCLUSTER* (Figure 24) the first 5% of affinities achieve an execution time of around 150 seconds. Beyond this point, quality diminishes, resulting in an execution time of about 200 seconds (33% slower). The remaining execution times steeply increase: The worst affinity has an execution time of over 450 seconds! Because so few affinities are good, it is unlikely that a randomly chosen affinity will result in good performance.

In contrast, *SWAPTIONS* (Figure 25) is relatively insensitive to affinity [10]. The majority of possible affinities have nearly identical execution times. If an affinity was randomly selected, the resulting execution time would likely be good because *most* affinities performed well. However, some affinities cause *SWAPTIONS* to perform poorly. The worst affinity has an execution time of 124 seconds (45% slower than the best one). Even in this affinity-insensitive application, it can be beneficial to select affinity because there are some mappings that should be avoided.

It is important to understand why *STREAMCLUSTER* and *SWAPTIONS* behave so differently (Figure 24 and 25). Certain affinities allow applications to communicate more quickly by sharing cache space. However, sharing caches among threads can reduce effective cache capacity, causing the working set to overflow the cache. Other affinities give a thread more cache space by spreading threads across sockets and caches at the cost of communication speed. Cache and communication demands of an application directly influence which affinities are best.

STREAMCLUSTER makes frequent use of barriers [10]. Because affinity affects communication, and thus, the speed at which threads reach and pass through barriers, *STREAMCLUSTER* is affected dramatically by affinity. *SWAPTIONS* does not use many barriers or locks [10]. Furthermore, *SWAPTIONS*'s working set is relatively small: Each thread's working set can fit in the experimental machine's private L2 cache. Thus, *SWAPTIONS* is more resilient to affinity choice.

If one affinity worked well across applications, then selecting application affinity would be trivial. Unfortunately, this is not the case. Figures 26 and 27 show *CANNEAL* and *STREAMCLUSTER*'s performance across affinities [10]. In these graphs, performance is normalized to the slowest performance for that application (higher is better). Applications are not executed simultaneously. In Figure 26, affinities are sorted by *CANNEAL*'s corresponding performance. In

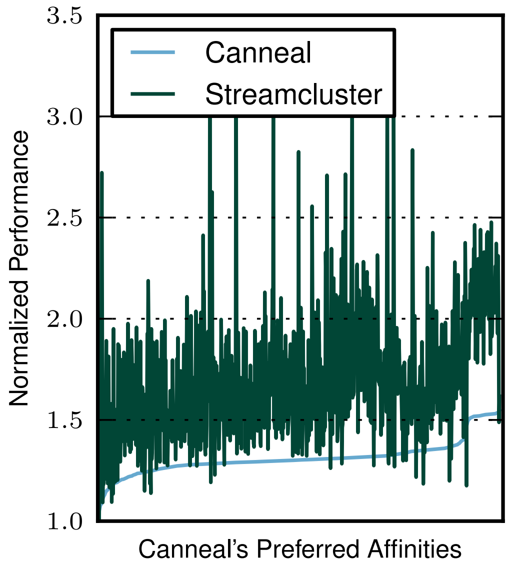

Figure 26: Performance across affinities, sorted by *CANNEAL*'s preference (16 threads)

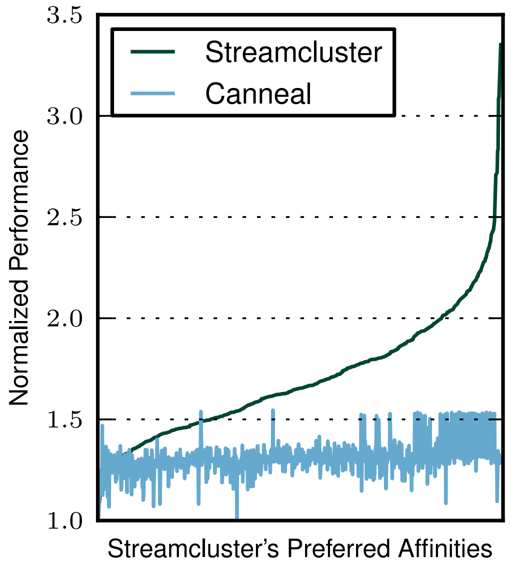

Figure 27: Performance across affinities, sorted by *STREAMCLUSTER*'s preference (16 threads)

Figure 27, affinities are sorted by *STREAMCLUSTER*'s performance. As better affinities are used for one application, the other application may or may not have better performance under that affinity. There is no sorting of affinities that works across applications. Therefore, affinities must be

Table 24: Non-isomorphic affinities for one application across thread counts, available sockets

Number of Affinities				
Thread Count	**1 Sockets**	**2 Sockets**	**3 Sockets**	**4 Sockets**
2	2	3	3	3
4	3	8	10	11
6	4	16	27	32
8	3	26	57	80
10	2	34	105	174
12	1	38	168	339
14	n/a	34	231	585
16	n/a	26	280	912
18	n/a	16	300	1,282
20	n/a	8	280	1,632
22	n/a	3	231	1,884
24	n/a	1	168	1,979

Table 25: AUTO-FINITY experimental machine

Machine Component	**Component Details**
Processors	4 AMD Opteron 6164 HE sockets (48 cores total)
Socket	2 NUMA nodes
NUMA Node	1 L3 cache (LLC)
L3 Cache	5 MiB cache, 6x cores
Core	1.7 GHz, private L1 \$ (64 KiB), private L2 \$ (512 KiB)
Operating System	Linux 2.6.39.1

selected on a per application basis.

Affinity selection is made even more difficult due to the large number of affinity choices. Table 24 shows the number of affinities for various thread counts and available sockets for the machine described in Table 25. Only non-isomorphic affinities are considered. For certain thread counts, there are nearly 2,000 affinities to select from! Furthermore, the number of affinities can greatly increase if affinity is also considered for a corunner. An exhaustive online evaluation of all affinities to find the best one is too costly.

As these figures show, application behavior varies across affinities. Automated techniques are

Table 26: Characteristics of AUTO-FINITY's affinity preference classification

Property:	**Value:**
Model Function	M(HPC Values)
AUTO Control Knobs	affinity, thread count
Profile Space	$O(t \cdot a)$, where t is a number of training trials for a applications

needed to select affinity. AUTO-FINITY provides models to guide an allocation policy.

5.2 AFFINITY PREFERENCE CLASSIFICATION

The characteristics of AUTO-FINITY's first technique to selecting application affinity are shown in Table 26. The primary goal of this technique is to support unknown application and select application affinity with only one sample of hardware performance counters (HPCs). A secondary goal is to handle runtime resource constraints (e.g., an unavailable socket). This technique assumes that machine resources (e.g., sockets) are statically partitioned to provide strong isolation between processes.

AUTO-FINITY's first technique automatically builds a model that selects a *affinity hint* that will maximize a user-defined performance metric. An affinity hint is a set of possible affinities that will result in good application performance. A hint is independent from thread count. The model is consulted at runtime by an allocation policy. Training data for this technique comes from running applications under different affinities. The training applications are not necessarily those used at runtime.

Figure 28 shows the steps to build a model. First, AUTO-FINITY profiles training applications for a range of affinities (step 1). This step gathers data that indicates how well particular affinities behave. The behavior is recorded by hardware performance counters (HPC) during an application's ROI. The HPC sampled. Each sample is timestamped to facilitate the examination of cross-affinity application behavior (e.g., two samples each from the beginning of an application's execution). A sample observes application behavior over a fixed amount of time. As such, multiple consecutive

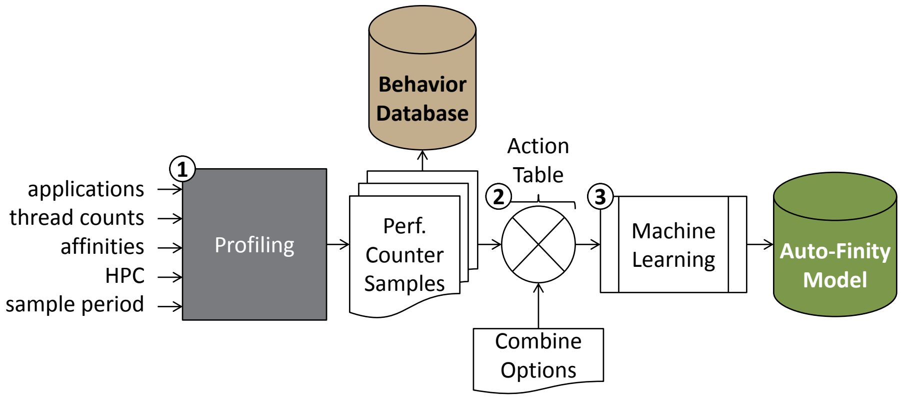

Figure 28: Affinity preference model generation steps

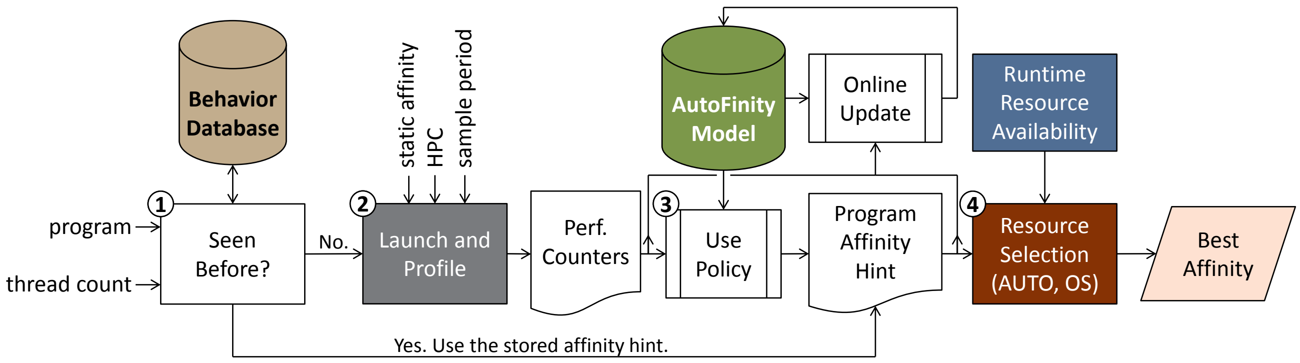

Figure 29: Model usage steps

sample periods capture the full ROI behavior.

The samples are analyzed and transformed into an *action table* (step 2). The action table is built to maximize a policy-defined performance metric (e.g., IPC). A row in the action table states what affinity hint should be used if an application exhibits a particular behavior at runtime. The samples are also archived in a behavior database to build future policies or update existing ones.

Machine learning is used to condense and summarize the rows of the action table, turning it into a model. Machine learning will also handle contradictory or missing table information (step 3). Machine learning resolves the conflicts and "holes" through statistical analysis.

The generated model is used at runtime to select an affinity hint. Figure 29 shows this process. To use the policy, the application's thread count must be known. The thread count can be discovered on the command line, in an environment variable, or by monitoring syscalls.

With the thread count, the behavior database is consulted. If the application and thread count have been previously observed then the behavior database gives the application's affinity hint (step 1). Otherwise, the model is consulted to obtain the application's affinity hint. To consult the model, application behavior must be observed. A starting affinity will be used and the application's HPC behavior will be observed. The application's behavior is examined once, during a single sample period (step 2). Continuous sampling and adjustment are naturally supported by this technique, but require a separate model to predict the costs of thread migration.

The sampled HPC values are used by the allocation policy to select an affinity hint (step 3). The HPC values and selected hint are saved into the behavior database. The affinity hint can be used to choose a new application affinity. To use an affinity hint, the Runtime Configurator considers the hint and allocates cores, ultimately choosing an application affinity for the application (step 4).

Because AUTO-FINITY continuously records application behavior, it can generate a new policy as application information is accumulated (e.g., when the behavior database has grown by some threshold). If a stored affinity hint corresponds to an out-of-date policy, the hint will be removed.

5.2.1 Affinity Hints

Application affinities explicitly define which cores an application may use (e.g., cores 0–3 and 5–7). For a particular thread count, there may be thousands of possible affinity choices. It is difficult to directly pick an appropriate affinity, and selecting an affinity is made harder by supporting different thread counts. To reduce the number of affinity options that must be considered, application affinities are generalized into classes, called *affinity hints*.

An affinity hint suggests a relationship between cores (e.g., cores should be distributed across different caches). The generalization enables techniques that work *across* thread counts. Using a hint, the allocation policy will select an affinity based on available resources (e.g., sockets) and number of active threads.

There are two requirements for affinity hints. First, affinity hints should not be overly specific: A hint should be realized at runtime, even under runtime resource constraints (e.g., an unavailable

Table 27: Affinity hint parameters

Affinity Hint Parameter	**Parameter Description**	**Values**	**Symbol**
spreadAcrossSockets	Prefer use of many sockets	{True, False}	S
spreadAcrossLLCs	Prefer to not share LLCs	{True, False}	L
socketOptionGetsPriority	Socket preference is priority	{True, False}	P

processor socket). Second, affinity hints must capture the effects of core assignments on thread: a) cache space, b) communication, and c) access to main memory (DRAM).

Threads may benefit from a large effective cache space (e.g., to hold their private working set). The space available may be negatively impacted by the presence of one or more threads in the same last-level cache (LLC). However, sharing LLCs also allows threads to more quickly communicate. Regardless of whether cores share one or more levels of cache, a thread will occasionally suffer cache misses. It is important to consider the impact of affinity on NUMA access. For example, applications may better exploit memory bandwidth if spread across NUMA domains.

To meet these requirements, affinity hints have three boolean parameters, leading to eight possible hints (Table 27). The first parameter dictates whether threads should be distributed across sockets ($spreadAcrossSockets$, symbol: S). The use of multiple sockets may allow greater memory bandwidth, by simultaneously employing memory banks. On the contrary, assigning threads to the same sockets allows better communication in the same memory domain, potentially avoiding remote accesses.

The second parameter, $spreadAcrossLLCs$ (symbol: L), controls whether threads should be assigned to cores that share a LLC. Threads that share a cache may communicate with each other more quickly or may prefetch data for sharers. However, sharers may contend for cache space. The third parameter ($socketOptionGetsPriority$, symbol: P) captures whether the sharing sockets or sharing LLCs parameter is more important.

The three parameters define eight hints. Boolean notation is used to represent each hint. For example, $\mathrm{S}\overline{\mathrm{LP}}$ specifies $spreadAcrossSockets$ = True, $spreadAcrossLLCs$ = False, and $socketOptionGetsPriority$ = False. This hint dictates that application threads share as few

Figure 30: Algorithm for selecting an application affinity

```
GETAFFINITY($resources, threadCount, spreadAcrossSockets, spreadAcrossLLCs,$
$socketOptionGetsPriority$)
1  $affs$ = ALLAFFINITIES($resources, threadCount$)
2  if $spreadAcrossSockets$
3      $socketSortFunc$ = PRIORITIZEBYLARGERUSEDSOCKETCOUNT
4  else
5      $socketSortFunc$ = PRIORITIZEBYSMALLERUSEDSOCKETCOUNT
6  if $spreadAcrossLLCs$
7      $LLCSortFunc$ = PRIORITIZEBYSMALLERLLCOCCUPANCY
8  else
9      $LLCSortFunc$ = PRIORITIZEBYHIGHERLLCOCCUPANCY
10 if $socketOptionGetsPriority$
11     $prioritizedAffs$ = SORT($affs, socketSortFunc$ THEN BY $LLCSortFunc$)
12 else
13     $prioritizedAffs$ = SORT($affs, LLCSortFunc$ THEN BY $socketSortFunc$)
14 return HIGHESTPRIORITY($prioritizedAffs$)
```

LLCs as possible ($\overline{\text{L}}$)[1]. The hint also prefers LLCs that do not share sockets (S). The priority in this example hint is to pack threads onto as few LLCs as possible ($\overline{\text{P}}$).

Once the allocation policy is given an affinity hint, it can select an affinity while also taking into account resource availability. This process is shown in Figure 30. GETAFFINITY takes a list of available cores and the amount needed ($threadCount$). A list of affinities for the available cores is obtained (line 1). Only affinities that use as many cores as threads are considered. The value of $spreadAcrossSockets$ prioritizes potential affinities (lines 2–5). If threads should be spread across sockets then affinities which utilize more sockets are preferred; otherwise, affinities that use fewer sockets are selected. In lines 6–9 $spreadAcrossLLCs$ is similarly processed. A hint that states that threads should spread across LLCs will prioritize affinities accordingly. Next, the available affinities are sorted based on $socketOptionGetsPriority$'s value (lines 10–13). Ties are broken by the secondary priority. The highest priority affinity is the one that best matches the affinity hint. This affinity will be returned (line 14)[2].

[1]There is at most one thread per core.

[2]Multiple affinities may tie for the same priority. In this case, the affinities will be isomorphic and one will be

```
BUILDACTIONTABLE($observations$, $condense$, $leeway$)
 1  $possibilities = \emptyset$, $actionTable = \emptyset$
 2  for $application \in observations$
 3      for $samplePeriod \in application$
 4          for $aff \in samplePeriod$
 5              $behavior = aff.observedPerformanceCounters$
 6              for $destAff \in samplePeriod$
 7                  $perf = destAff.perf$
 8                  $possibilities[behavior][aff][destAff].append(perf)$
 9  for $behavior \in possibilities$
10      for $aff \in behavior$
11          for $destAff \in aff$
12              $affinityHints = $ AFFINITYHINTFROMCONF($destAff$, $leeway$)
13              if $affinityHints = $ NONE
14                  continue // Configuration does not map to an affinity hint.
15              $consequence = condense(possibilities[behavior][aff][destAff])$
16              // An affinity may map to $>1$ affinity hints (e.g., due to $leeway$).
17              for $hint \in affinityHints$
18                  $actionTable[behavior][aff][hint] = consequence$
19  for $behavior \in actionTable$
20      for $aff \in behavior$
21          $bestAffinityHint = $ MAX($actionTable[behavior][aff]$)
22          WRITETRAINACTION($behavior$, $bestAffinityHint$)
```

Figure 31: Algorithm for building the action table

5.2.2 Action Table Generation

chosen arbitrarily.

AUTO-FINITY analyzes application behavior (i.e., hardware performance counters) and performance across affinities. This information is used to generate a model that predicts at runtime which affinity hint is best for an application. This process consists of two steps: 1) building the action table and 2) condensing the table into a policy.

The action table contains observed application behavior and the affinity hint(s), that will improve the user-defined performance metric in those situations. It is built with initial training data: applications in various affinities have their behavior (HPC values) recorded. Additional behaviors can be gathered online.

Figure 31 shows BUILDACTIONTABLE. This function condenses performance information ($observations$) into an action table. BUILDACTIONTABLE has two more parameters: $condense$ (a function) and $leeway$ (an integer). $Condense$ resolves different performance values coming from similar HPC samples across multiple applications. This phenomenon is expected to occur occasionally, as applications may have similar behavior (e.g., cache misses) but different performance.

For a fixed thread count, GETAFFINITY selects one configuration per affinity hint. Each of the eight affinity hints correspond to a best match affinity (eight affinities total). $Leeway$ is introduced to allow training on affinities which *almost* would be selected by GETAFFINITY. Leeway allows an affinity hint to correspond to affinities that are not the affinity hint's best match. A leeway of 0 means that only the best match affinities are considered.

Leeway is the maximum edit distance between two affinities. A leeway of l allows an application affinity to be considered as belonging to an affinity hint if, by rearranging up to l threads across LLC domains, the affinity becomes identical to that affinity hint's best match affinity. The rearrangement of threads across LLC domains may also transfer threads across sockets. Therefore leeway also places a limit on affinity similarity in terms of socket usage.

For example, an affinity that has six threads sharing a LLC might be selected as best corresponding to the hint $\overline{\mathrm{SLP}}$ (for 6 threads). With a leeway of 1, the affinity which causes 5 threads to share a LLC while putting another thread on a separate socket, would still be considered as corresponding to $\overline{\mathrm{SLP}}$. These two affinities are considered equivalent because only 1 thread was transferred to make the affinity equivalent to the best match affinity.

In Figure 31 the first loop nest (lines 2–8), build a table, $possibilities$, which stores the affinity (aff) and associated $behavior$ (HPC values). For each sample, alternate affinities with the same thread count and corresponding to the same time stamp are considered. These are affinities ($destAff$) that the application could have been executed under. For each alternative affinity, the performance metric of the application in that sample period affinity is recorded ($perf$).

Applications may exhibit similar behavior, and therefore, the first loop nest accumulates a list of multiple performance metrics (line 8). List items are reduced into one entry using the $condense$ function (lines 9–18). BUILDACTIONTABLE also converts destination affinities to their corresponding affinity hint(s), with AFFINITYHINTFROMCONF (the inverse of GETAFFINITY). After this step, the action table contains a list of affinities and associated behaviors, affinity hints that could have applied, and the consequent performance for the affinity hints.

Finally, the action table is written. In lines 19–22 the action table's behaviors are examined. For each type of behavior seen under each affinity, BUILDACTIONTABLE finds the best affinity hint. These results are written as a series of rules (WRITETRAINACTION). Each rule states an affinity that should be used if a particular behavior is observed. Contradictions between rules are expected, and may be due to noise and/or different applications exhibiting similar behavior, yet operating best in different affinity hints. The rules are transformed into an affinity hint model, described next.

5.2.3 Building the Affinity Hint Model

WEKA's JRIP is used to convert the action table to a model that can be consulted at runtime [39]. JRIP is an implementation of Repeated Incremental Pruning to Produce Error Reduction (RIPPER), a propositional rule learner that produces a series of rules for classification. Each rule is a set of conditions joined by `and` operators. RIPPER fills in "holes" due to incomplete data (i.e., combinations of HPC values that were not observed). It also prunes rules that are statistically insignificant. RIPPER is used because it creates rules for missing data, produces simple, human-readable policies. Furthermore, the rules are easily converted into a code implementation to use at runtime. WEKA's default JRIP settings are used.

An example policy produced by JRIP is shown in Figure 32. This policy is used by *SWAPTIONS* in Section 5.2.4. For readability, numeric values have been replaced with constants. The policy's parameters are the values of four HPCs. The selection of these counters is discussed later

```
$\text{AFFINITYHINTPOLICYSWAPTIONS}(dCacheAccesses, invalidDCacheLinesEvicted,$
        $exclusiveReadRequestsToL3CacheFromAnyCore, retiredUops)$
if $retiredUops \in (\text{BOUNDLOWER0}, \text{BOUNDUPPER0}]$
    if $dCacheAccesses < \text{BOUND1}$ and $invalidDCacheLinesEvicted < \text{BOUND2}$
        return $\overline{\text{S}}\text{LP}$
    if $invalidDCacheLinesEvicted < \text{BOUND3}$ and
            $dCacheAccesses \in (\text{BOUNDLOWER4}, \text{BOUNDUPPER4}]$ and
            $exclusiveReadRequestsToL3CacheFromAnyCore < \text{BOUND5}$
        return $\overline{\text{S}}\text{L}\overline{\text{P}}$
return $\overline{\text{SL}}\text{P}$
```

Figure 32: Example affinity hint policy

(Section 5.2.4.1). Depending on the values of these counters, one of three affinity hints will be selected ($\overline{\text{S}}\text{LP}$, $\overline{\text{S}}\text{L}\overline{\text{P}}$, or $\overline{\text{SL}}\text{P}$). In this policy, there is no fourth affinity hint. As more performance data is obtained and the model is rebuilt, additional hints may become available.

5.2.4 MAGNET: A Policy to Select Application Affinity

This section examines AUTO-FINITY's ability to determine an application's preferred affinity. Results are gathered on a set of applications and thread counts. AUTO-FINITY's usefulness in a sample policy, MAGNET, is examined. MAGNET aims to minimize application execution time across a range of thread counts through the proper selection of application affinity.

All evaluations were performed on the machine described in Table 25. The machine is a 48-core AMD Opteron 6164 NUMA system. Each of the four sockets has 2 NUMA domains (8 NUMA domains total). Each NUMA domain has 6 cores that share an L3. The cores have private L1 and L2 caches. Applications are executed on an otherwise idle system.

The 18 benchmarks are from PARSEC 2.1 [10], OmpSCR 2.0 [29], and the NAS Parallel

Benchmark Suite (NPB) 3.3.1 [73]. The PARSEC benchmarks use the largest input size ("native") and the NPB benchmark uses the "A" input size. Because OmpSCR does not provide official inputs, applications in that suite were configured to execute for a similar amount of time as PARSEC (one to two minutes with eight threads).

The following PARSEC applications are used: *BLACKSCHOLES* (BS), *BODYTRACK* (BT), *CANNEAL* (CN), *DEDUP* (DD), *FACESIM* (FS), *FLUIDANIMATE* (FA), *FREQMINE* (FM), *RAYTRACE* (RA), *STREAMCLUSTER* (SC), *SWAPTIONS* (SW), *VIPS* (VP), and *X264* (X). From OMPSCR, *C_FFT* (FT), *C_FFT6* (FT6), *C_LU* (LU), *C_MANDEL* (MN), and *C_MD* (MD) are used. Lastly, *DC* (DC) from NAS is evaluated. Some applications (e.g., PARSEC's *FERRET*) were not used due to compilation errors, framework incompatibilities, or short execution times.

MAGNET maximizes instructions per second (IPC), consequently minimizing application execution time. This metric is used because it captures the rate of application progress and is easy to obtain via hardware performance monitoring. As later shown, choosing IPC works well to reduce execution time.

Before using MAGNET, that application's data is removed from the training data. This causes the application to be treated as a never-before-executed application. The AUTO-FINITY model is built on the remaining applications' data (leave-one-out cross-validation).

To evaluate the quality of the generated policy, applications are launched in a fixed affinity. The HPC values are sampled once (as shown in Figure 29). The AUTO-FINITY model uses the HPC values to predict a best-performance affinity hint. An affinity is selected from the hint. The application with that affinity is then started and ROI runtime is observed. Average performance is calculated by computing the geometric mean of each application's execution time.

By default, previously unseen applications and thread counts are executed with a *distributed* affinity. A distributed affinity allows the AUTO-FINITY model to make better affinity hint predictions. A distributed affinity spreads an application's threads across sockets and LLCs. It is equivalent to the affinity hint SLP.

5.2.4.1 Choosing and Gathering Hardware Performance Counters Before an AUTO-FINITY model can be used, it must be built. This requires profiling applications using HPCs. Feature selection automatically chooses appropriate counters. The counters are measured during the initial training period as well as online. To determine the counters, HPC values of 60 manually selected counters were periodically collected across a subset of evaluated applications.

The comprehensive HPC measurements required 15 runs of each application, as the experimental machine can only record 4 counters at a time. Additional counters bring dwindling returns in terms of utility. The machine learning algorithm (i.e., RIPPER) will ignore counters and/or counter values that are statistically insignificant.

WEKA's `CfsSubsetEval` feature selection algorithm chooses four performance counters that best correlate with the objective metric, IPC. This method evaluates features by considering the individual predictive ability of each feature along with the degree of redundancy between them. Features were selected with greedy search.

Out of the 60 counters considered, the selected counters were the following: 1) data cache accesses, 2) invalid data cache lines evicted, 3) exclusive read requests to a LLC from any core, and 4) retired micro-ops.

The first counter indicates the memory demands of an application. The second counter tracks how frequently cache lines become invalid due to modification by another core. Larger values indicate thread communication or data sharing. The third counter correlates with thread communication and memory demand. A cache line in the exclusive state may become shared if another thread accesses that line. Exclusive LLC read requests may also allow one thread to prefetch another's data. The last counter tracks the number of operations completed. The experimental machine, an x86-64 processor, executes instructions that decode into one or more RISC-like operations. Therefore, this metric correlates with IPC.

To ensure that the techniques work on many thread counts and sampling period lengths, performance counter values are normalized relative to thread count and number of seconds over which the counter was gathered.

The selected counters and IPCs for each application were gathered. The applications are executed with 8 threads. The counters are recorded for 78 different affinities (8 threads). This step gathers initial training data.

5.2.4.2 Selecting a Discretization Method To build the action table, the training data must be put into discrete bins. Two methods are used to discretize the data:

Table 28: Impact of the discretization method on average application execution time

Method	**Geo. Mean**
equal-width	79.4
equal-frequency	83.1

Table 29: Impact of the condense function on average application execution time

Function	**Geo. Mean**
AVERAGE	80.6
GMEAN	80.6
HMEAN	80.6
MAX	81.5
MEDIAN	79.4
MIN	83.1
SUM	80.6

1. Equal-width binning: Each bin corresponds to a constant range over the possible values of a HPC. Equal-width binning uniformly covers the range of observed counter values.
2. Equal-frequency binning: Bin widths are uneven to allow higher resolution where HPC values are most concentrated. Bins have the same item counts.

Five bins for each discretization method worked well, without making individual bins too wide or small. WEKA's equal-width discretization supports automatic adjustment of bin count to be data appropriate. The adjustment is influenced by the number of requested bins. This feature was enabled. Equal-frequency binning does not support this option.

Cross-validation is used to evaluate the discretization method. The geometric mean of the application execution times is computed. Because discretization method is not the only adjustable parameter, the other parameters (e.g., the condense function) are fixed to their best values, described later.

Table 28 shows the results of varying the discretization method. The results show that equal-width binning improves the geometric mean by approximately 5% (a mean of 79.4s versus 83.1s). Equal-width binning does better because equal-frequency binning creates very wide bins to allow for occasional narrow bins. However, a model built with equal-frequency binning may be unable to later differentiate between HPC values which, though the difference between them is large, are grouped into the same wide bins.

5.2.4.3 Selecting a Condense Function Multiple condense functions are evaluated. The condense function combines performance measurements from similar samples gathered for the same affinity. Other parameters are set to their best values. Evaluated condense function are: a) arithmetic mean (AVERAGE), b) geometric mean (GMEAN), c) harmonic mean (HMEAN), d) MAX, e) MEDIAN, f) MIN, and g) SUM.

For example, suppose that application a is similar to b (i.e., both have similar counter values). After analyzing performance in different affinities, it is determined that both applications execute fastest with affinity hint h. However, due to differences in behavior, a will obtain an IPC for h of 2, whereas b will obtain an IPC of 1.5.

During action table construction, AUTO-FINITY must consider expected IPC of an unknown application, c, that is similar to a and b. In hint h, the action table could predict c to have a runtime of 2 (MAX), 1.5 (MIN), or 1.75 (AVERAGE). The condense function determines the expected value.

Table 29 shows the results of varying the condense function. The best function, MEDIAN, has a geometric mean of 79.4s. The worst condense function is MIN, with a geometric mean of 83.1s, about 5% worse than MEDIAN's result. MIN results in the worst performance because it creates a pessimistic policy. It penalizes types of behavior which are exhibited by multiple applications: If three applications exhibit the same behavior, the worst performing one is the only one whose performance will be considered.

Although MIN makes a pessimistic policy, and MAX makes an overly optimistic one, MEDIAN is balanced and works the best. The other means also had good behavior.

5.2.4.4 Selecting Other Parameters There are three remaining AUTO-FINITY parameters.

The first one is whether to normalize IPCs. IPC can be normalized to the smallest IPC recorded for an application. This allows for easier comparison of IPC across applications. For example, if using a particular affinity hint can boost an application's IPC by 0.1, then this performance increase may be minor (e.g., 3% if the original IPC is 3.0) or relatively major (e.g., 10% if the original

IPC is 1.0). Normalization allows these trends to be captured. Using the best parameter settings for the other model parameters (e.g., the best condense function, the best discretization method) causes normalization to have no significant benefit. Nevertheless, IPCs are normalized because, for suboptimal parameter settings, normalization resulted in better models.

The second parameter is whether affinity-insensitive applications are included in the training data. Affinity-insensitive applications are applications whose standard deviation of performance is less than 1% of average performance across affinities. Intuitively, an application that is insensitive to affinity contains no useful information to guide affinity selection. Potentially, the data from insensitive applications may dilute important information. As with IPC normalization, using the best possible parameter settings causes the removal or inclusion of affinity-insensitive applications to make little difference (less than 1%). However, the removal of affinity-insensitive applications was occasionally beneficial for less optimal settings. Therefore, in the performance evaluation, applications that are affinity insensitive are removed from training data.

The last remaining parameter is leeway. Leeway equal to 2 was experimentally determined to have good flexibility while still only considering affinities similar to their best match affinity. For 8 threads, this value resulted in 31 affinities to train on.

5.2.4.5 Application Behavior Coverage Exposure to diverse application behavior allows the action table and policy to appropriately respond to unseen applications. To gain insight into this diversity, application behavior across all possible affinities for 8 threads is analyzed.

An application exhibits a behavior, b, if during the training process it produces a discretized HPC sample that is equal to b. Each application, a, produces a set of unique behaviors, B_a, where $B_a = \{b_0, b_1, ...\}$. Examining the discretized behaviors, the diversity of an application ($|B_a|$) and whether other applications exhibit the same number of behaviors can be examined. An application behavior examination can also reveal whether some applications have more diverse behavior than others (i.e., whether $|B_{a_i}| < |B_{a_j}|$), and the effect of an application's behavior on the cumulative training information.

Figure 33 shows how coverage is influenced by the applications. The x-axis is sorted by the number of unique behaviors in each application. The figure has two lines. The "+" line shows the number of unique behaviors in each application, while the "·" line shows the number of cumulative

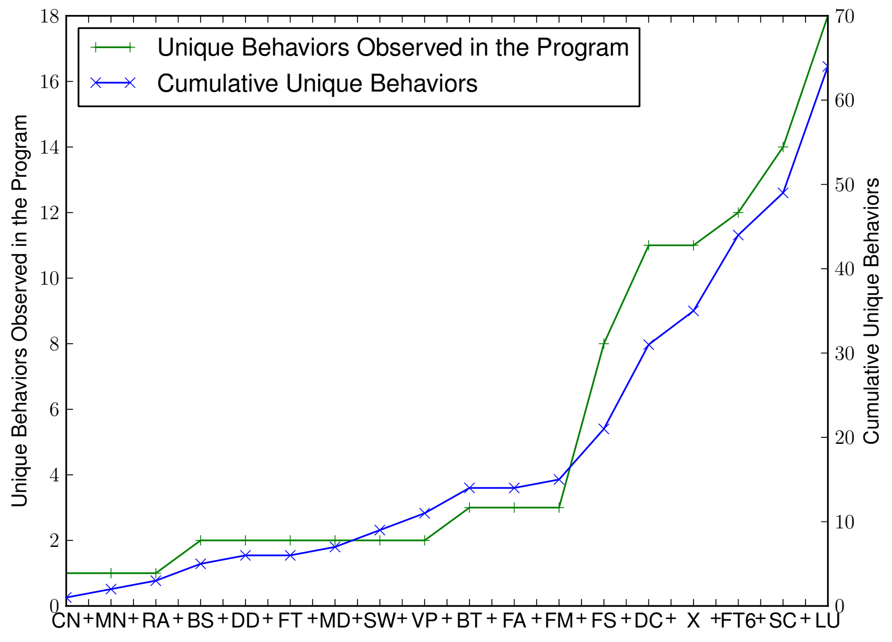

Figure 33: AUTO-FINITY cumulative and per-application discretized behavior coverage

behaviors as applications are added (left to right).

Each application is placed into one of two categories: 1) applications that have few unique behaviors and 2) applications that exhibit diverse behavior. Approximately half of the applications, those before and including FM (*FREQMINE*), have few behaviors, and therefore, exhibit consistent behavior. This is shown by the low height of the "unique behaviors" line. Even though 12 applications fall into this category, each with an average of 2 unique behaviors per application (24 total), there are only 15 unique behaviors between them (i.e., at the point on the x-axis for FM, the cumulative unique behaviors line has a y-value of 15). Each application, therefore, contributes on average only 1.25 unique behaviors to the cumulative training data (15 behaviors over 12 applications = 1.25 behaviors per application).

The other applications (after FM) have great diversity. This diversity may be due to phases and/or sensitivity to affinity. These 6 applications contribute 49 unique behaviors to the training data, for an average of more than 8 unique behaviors per application. *C_LU* has the highest count of unique behaviors. It exhibits 18 unique behaviors and contributes 15 to the cumulative list. *STREAMCLUSTER* has the second highest number of unique behaviors (14). It contributes 5 unique behaviors to the behavior coverage plot.

From Figure 33's data, it may be possible to train on only a few, specially selected, applications (i.e., those that exhibit a range of behavior) to produce a good model.

5.2.5 Performance Evaluation of MAGNET

This section evalutes a policy built from AUTO-FINITY, MAGNET. MAGNET is used to choose affinity hints for unknown applications in order to minimize execution time. The resulting execution times are compared against the execution times of applications under affinities chosen by two static policies:

1. *Packed*: This policy places threads together, causing them to share LLCs, and preferring to use few sockets (corresponding affinity hint: $\overline{\mathrm{SLP}}$).
2. *Distributed*: This policy distributes threads uniformly to sockets and LLCs (corresponding affinity hint: SLP).

For MAGNET, AUTO-FINITY uses a leeway of 2. Affinity insensitive applications are removed from the training data. IPCs from an application are normalized to the slowest IPC from that application. HPC monitoring uses a sample size of 10 seconds for the four counters described in Section 5.2.4.1. The condense function is MEDIAN.

5.2.5.1 Fixed Thread Count Evaluation The runtime of each application with 8 threads under the evaluated policies is compared. Training data consisted of applications with 8 threads. Figure 34 shows these results.

The y-axis is ROI execution time (lower is better). The x-axis shows each application, with the geometric mean at the far right. Subscripts indicate thread count. For each application, three bars are shown. The first bar is the runtime under a packed affinity. The second bar is runtime under a distributed affinity. Finally, the third bar is runtime for the affinity chosen by MAGNET.

Some applications are insensitive to the policy. *BLACKSCHOLES*, *FREQMINE*, *C_MANDEL*, *C_MD*, *RAYTRACE*, *SWAPTIONS*, *VIPS*, and *X264* are not significantly affected by affinity. To a lesser extent, *FLUIDANIMATE* is insensitive too. However, for most applications, the affinity is important. Five of the applications prefer a packed affinity (*BODYTRACK*, *CANNEAL*, *DEDUP*,

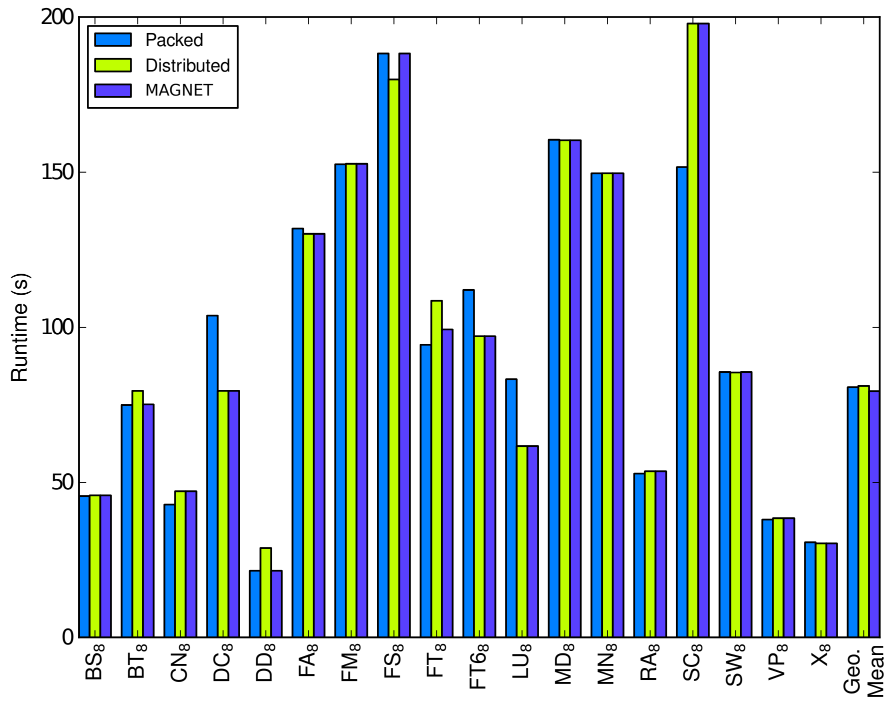

Figure 34: Packed, distributed, and MAGNET ROI execution times for 8 threads (lower is better)

C_FFT, *STREAMCLUSTER*). The threads in these applications communicate and/or share data. For *BODYTRACK*, *DEDUP*, and *C_FFT*, MAGNET selects an affinity that performs nearly as good or just as good as the best static policy.

MAGNET selects a poor affinity hint for *STREAMCLUSTER* ($\mathrm{S}\overline{\mathrm{LP}}$). The result is due to *STREAMCLUSTER*'s unique behavior. As shown in Figure 33 (discussed in Section 5.2.4.5), *STREAMCLUSTER* has the second highest number of unique behaviors. Due to this fact, AUTO-FINITY did not build an appropriate rule. Additional training data (e.g., from *STREAMCLUSTER* or applications which exhibit behavior like it) would allow MAGNET to make better affinity hint selections.

DC, *FACESIM*, *C_FFT6*, and *C_LU* prefer a distributed affinity. These applications have large memory footprints, as evidenced by the distributed preference. For *DC*, *C_FFT6*, and *C_LU*, MAGNET chooses an affinity hint and application affinity which performs just as well as the distributed policy.

In 7 out of 9 cases involving affinity sensitive applications, MAGNET chose an application affinity which performed nearly as well or just as well as their preferred static policy. The geometric mean (last set of bars in Figure 34) shows that it has a slight runtime advantage over either static policy. The advantage of MAGNET is that the user does not have to manually select the proper affinity: It can automatically select the best affinity to use.

MAGNET is also compared against an oracle policy that was built through exhaustive experimentation. On average, MAGNET achieved 96% of the oracle's performance. For three applications, it tied with the oracle (*DC*, *C_FFT6*, *C_LU*).

5.2.5.2 Varying Thread Count The utility of MAGNET across a range of affinities is now examined (4, 8, 16, and 24 threads)[3]. Training data was obtained *only* for a thread count of 8. To test MAGNET on an application, that application is removed from the training data before building the model.

As thread count is changed, many applications do show a wide variety of behavior. Affinity insensitivity is even sometimes thread count dependent. Applications under a thread count that have at least a 20% ROI runtime difference between their packed and distributed affinities, are referred to as *interesting cases*. Interesting cases have a potential for bad or good affinity choice. To focus the discussion, only interesting cases across thread counts are shown and discussed. There are 15 interesting cases.

Figure 35 shows the interesting cases. The x-axis shows the application's name with subscripts indicating thread count. The y-axis is the runtime of the application's ROI (lower is better). The first bar is the runtime for the packed affinity, the second bar is the runtime for the distributed affinity, and the third bar is the runtime for the affinity chosen by MAGNET.

The figure is annotated to indicate whether MAGNET chooses an affinity that is within 1% of the best static policy. Wins (i.e., within 1%) are marked by "✓." If performance is not within 1% of the best static policy, the graph is marked with "!."

DEDUP (DD) appears 3 times (4, 8, and 16 threads). Each time, *DEDUP* does best under a packed affinity. Because it is a pipeline parallel application, there is communication between threads as data is passed from stage to stage [10]. Therefore, it is natural that *DEDUP* would benefit

[3] *FACESIM* and *FLUIDANIMATE* do not support executing with 24 threads.

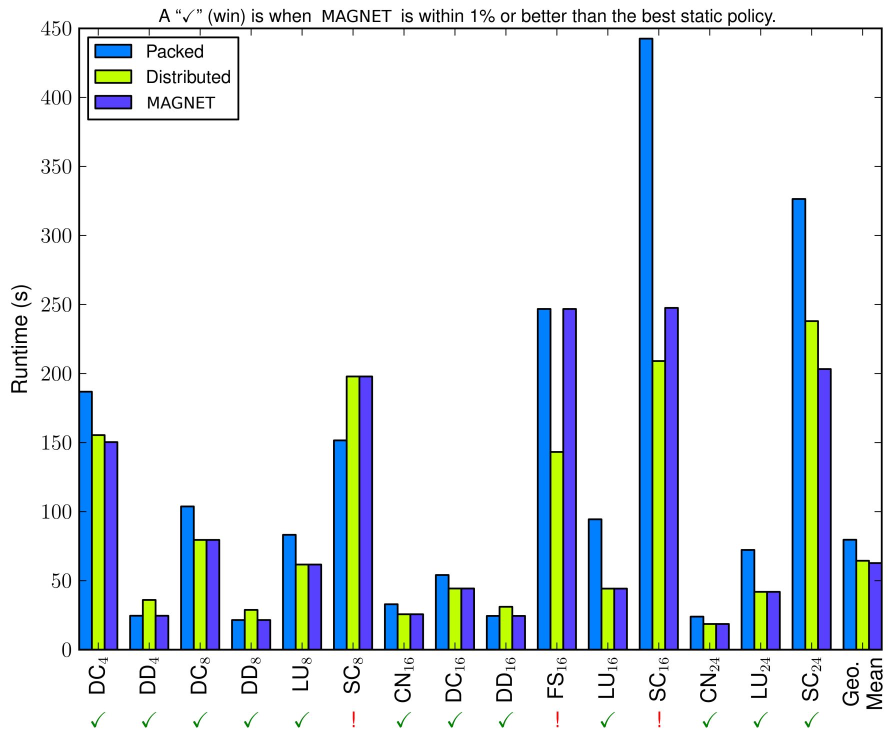

Figure 35: Packed, distributed, and MAGNET ROI execution times for interesting cases across applications and thread counts

from packed affinities, which cause threads to share cache space. In each case, MAGNET chooses an appropriate application affinity, and thus allows *DEDUP* to execute quickly even though the policy was not built on data obtained from observing *DEDUP*.

DC also appears 3 times in Figure 35 (4, 8, 16 threads). Each time, it prefers a packed affinity. If *DC* is executed with 4 threads, MAGNET chooses an affinity hint that performs better than the best static affinity ($\mathrm{S}\overline{\mathrm{LP}}$).

STREAMCLUSTER (SC) is another interesting case. If executed with 8 threads (SC_8), it benefits from a packed affinity. Unfortunately, MAGNET chooses an affinity whose runtime is similar to the distributed affinity runtime. If using 16 threads (SC_{16}), *STREAMCLUSTER* prefers a distributed affinity. MAGNET chooses an affinity which performs almost like (in terms of execution time) the preferred distributed affinity, but it is not within 1%. Therefore, it is not classified as a win. If 24 threads are used (SC_{24}), *STREAMCLUSTER* still prefers a distributed affinity. MAGNET actually beats both the packed and distributed policies and chooses an affinity hint that performs better than either ($\overline{\mathrm{S}}$LP).

Finally, in *FACESIM* (FS_{16}) the packed affinity is almost 75% slower than the distributed affinity. Unfortunately, MAGNET chooses an affinity hint that behaves like the packed affinity. Additional behavior data would potentially improve hint selection.

Overall, MAGNET, a policy enabled by AUTO-FINITY, chooses high performance affinities in most cases (12 out of 15 cases). It even beats the distributed and packed static policies in two cases. These results across thread counts show that affinity hints and models which use AUTO-FINITY's first technique for affinity selection work well.

5.3 AFFINITY PERFORMANCE PREDICTION

This section discusses AUTO-FINITY's technique to build models that make performance predictions given affinity. This technique quantitatively predicts the performance of an affinity. In comparison, AUTO-FINITY's first technique predicted which class of affinities would be best to use. The disadvantage to this technique is that each affinity must be considered, so that the best can be selected. This technique's characteristics are shown in Table 30.

The models in this section are application- and thread-count specific. Models are trained offline using statically derived aspects about affinities, called *features* (e.g., average number of threads sharing each L3), to make online predictions.

Models are built in two steps. First, feature selection determines which features are most useful to predict performance across applications. Second, applications are profiled and a performance model is constructed. These steps are described next, followed by an example use of the models.

5.3.1 Feature Selection

The choice of features directly affects the quality of the models. Features need to capture aspects about affinities that contribute to, or, detract from application performance. This offline process needs to be done once per machine. The selected features will be used in all subsequent models on

Table 30: Characteristics of AUTO-FINITY's affinity performance prediction

Property:	**Value:**
Model Function	$M_{program,tc}(\mathit{aff})$
AUTO Control Knob	affinity
Profile Space	$O(p)$, where p is the user-specified budget

$$M_{program,tc}(\mathit{aff}) = \beta_0 F_0(\mathit{aff}) + \beta_1 F_1(\mathit{aff}) + \beta_2 F_0(\mathit{aff}) F_1(\mathit{aff}) + \beta_3 {F_1}^2(\mathit{aff})$$

Figure 36: An example model for AUTO-FINITY's second affinity selection technique

that machine.

Features are functions which take an affinity as their parameter. They return the feature value for that affinity. To determine the best features, for a subset of applications their performance across a range of affinities was gathered. Regression models were built with many different combinations of features. Generated models are then compared based on how well they predict performance. The best performing features are selected.

Feature selection is an automated process. Features are statically derived from affinities. Therefore, new features and combinations of features can be considered without increasing profiling time or redoing profile experiments. Features were generated by me. The feature generation process will select the features that work best (discussed in Section 5.3.4).

5.3.2 Model Training

Models are trained on a subset of affinities. To ensure that the models make accurate predictions in such a large space, affinities are selected such that they uniformly cover the range of features. The optimal set of these training affinities is computationally infeasible to find[4].

[4]There would be $\binom{a}{t}$ combinations to evaluate.

Instead, multiple sets of affinities are randomly selected. The set of affinities that have the highest average distance from one another are used as the training affinities. The chosen affinities approximate a uniform distribution of affinities across the feature space.

Application performance is normalized to the worst observed performance. Models predict application speedup relative to the worst observed performance. Least squares multivariate linear regression is used to find the importance of each feature and the feature's effect (i.e., feature coefficients) on application performance. An example model is shown in Figure 36. The figure shows that the application's performance is affected by two features: F_0 and F_1. The value of a coefficient reflects how important a feature is to the application's performance.

5.3.3 Model Usage

After training, the model is ready for performance prediction. The time to consult a model is nearly instantaneous, as models are simply a linear combination of products between feature values and coefficients. Feature values do have to be computed for each affinity that must be evaluated. However, this process is fast due to the static features (e.g., number of L3s used by an affinity). Feature values could also be precomputed and looked up on demand.

The allocation policy can consult each application model and determine the affinity that is best. For example, a policy might be designed to improve average performance relative to their worst performance. Another policy might give higher priority to a specific application, preferring affinities that improve that application's performance to the detriment of corunners.

5.3.4 Evaluation

The models and feature selection were implemented on the machine described in Table 25. All applications were compiled with GCC 4.6.3. From PARSEC, *BLACKSCHOLES*, *BODYTRACK*, *CANNEAL*, *FACESIM*, *STREAMCLUSTER*, and *SWAPTIONS* were evaluated [10]. Additionally, experiments were performed with *DETBFS*, *KNN*, and *RAYCAST* from PBBS [93]. When applicable, the `pthreads` version of each application was used.

Table 31: Evaluated features chosen by various objective metrics

Maximized Obj. Metric	**Feature 1**	**Feature 2**	**Feature 3**	**Feature 4**
average R^2	mean thread distance	bandwidth facilitation^2	L3s	socket imbalance^2
average ρ^2	L3s	footprint and communication	socket load	sockets
max R^2	mean thread distance	L3s	normalized average L3 MiB/thread	socket imbalance^2
max ρ^2	bandwidth facilitation^2	L3s	socket imbalance	socket σ
min R^2	mean thread distance	density^2	L3s	socket σ
min ρ^2	density^2	median L3 MiB/thread	median thread distance	max non-empty socket imbalance

For feature selection, application performance on a variety of affinities was gathered (8 threads). After building a model for each application, predictions are compared with the observed performance according to the R^2 and ρ^2 values. R^2 measures linear correlation. Spearman's rank correlation coefficient (ρ) is used to measure the dependence between two variables (i.e., to what extent one is a function of the other).

To aggregate model quality during feature selection, the highest, smallest, and average correlation measure across models are computed. Feature selection chooses the set of features that maximizes the aggregate measures. Features selected via each method are shown in Figure 31.

Some common features include *bandwidth facilitation*^{2}, *socket imbalance*^{2}, and *L3s*. *Bandwidth facilitation*^{2} measures the average number of threads on each socket. Socket loads are squared to more heavily account for sockets with more active threads. *Socket imbalance*^{2} measures the average difference between the number of threads on each L3 of a socket. Differences are squared to place more emphasis on larger imbalances. Finally, *L3s* is the number of L3 caches whose cores have at least 1 thread assigned.

5.3.4.1 Evaluation of Model Construction Models can be trained on as many points as the user is willing to profile. In this evaluation, the time in minutes, T, to train p training point is given by $T(p) = (1 + \mathit{SerialSetupTime}) \cdot p$ where, for the evaluated applications, $\mathit{SerialSetupTime}$ ranges from 0 seconds (i.e., the application instantly starts performing parallel work) to 2 minutes. The application executes its parallel section for 1 minute, its performance is observed, and then the application is terminated. This process is automated. Models are built with 10 and 20 training points. In many contexts (e.g., scientific computing), the cost of training is small compared to the benefits that the models provide over the application's lifetime.

Once the training data is gathered, the time required to build a model is minimal. Regression is fast and the number of training points is relatively small and has few dimensions (tens of points, 5 dimensions).

Figure 37 shows an example model for *STREAMCLUSTER* (16 threads). The model is shown with two features due to the complexity of visualizing models with additional features (dimensions). The figure shows *STREAMCLUSTER*'s performance as the number of L3s is varied along with the average thread distance. Average thread distance is how "far" threads are away from each

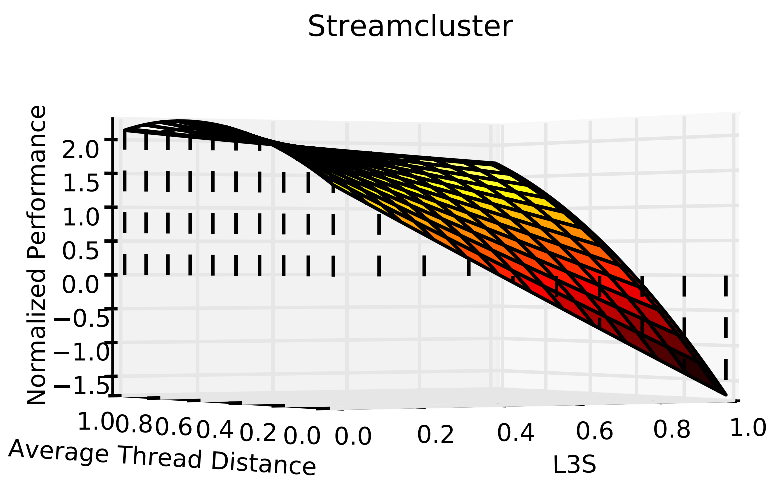

Figure 37: Visualization of a model which predicts *STREAMCLUSTER* performance given an affinity. The model's has two inputs: Number of L3s and average thread distance

other. Threads sharing an L3 are closer than threads on a different L3 on the same socket, which are further away than threads on separate sockets. The model predicts that decreasing the number of L3s in use while increasing the average thread distance increases *STREAMCLUSTER*'s performance[5]. Intuitively, the models predict that *STREAMCLUSTER*'s threads should be packed onto as few L3s as possible and that the L3s chosen should span sockets. Such configurations help decrease the communication cost while increasing available memory bandwidth. Some points on the surface do not correspond to valid affinities (e.g., those points that correspond to a large negative performance).

5.3.4.2 COMPASS: A Policy to Maximize Application Performance The use of the models is evaluated. Models are used to improve system-level performance objectives. In these experiments, a AUTO-FINITY policy, COMPASS, is built to maximize average application speedup. Equal weight is given to each application. Pairs of applications are executed on two of the available four sockets of the evaluation machine, in order to force 100% utilization. For each pair of

[5]These objectives can conflict for certain affinities.

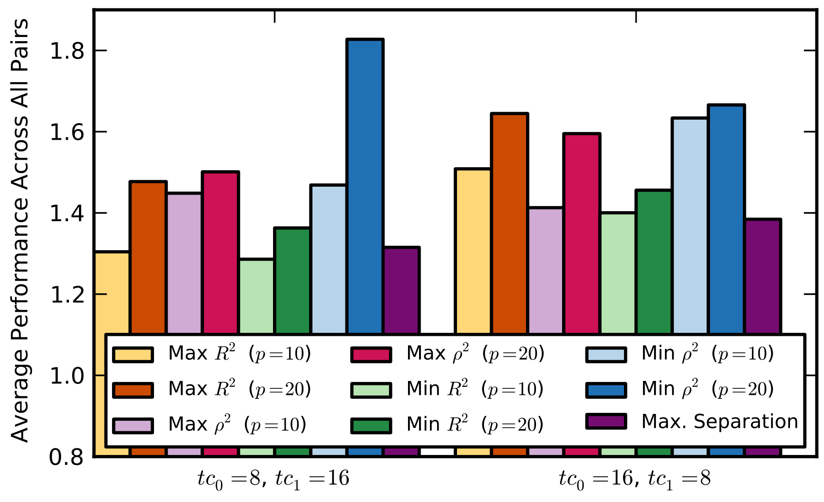

Figure 38: COMPASS's performance versus the best application-agnostic policy's performance (Maximum Separation)

applications, two cases are considered. In the first case, the alphabetically first application has 8 threads ($tc_0 = 8$) and the second application has 16 ($tc_1 = 16$). In the second case, the alphabetically first application has 16 threads ($tc_0 = 16$) and the second application has 8 ($tc_1 = 8$).

The models are compared against an application-agnostic scheduling policy: *maximum separation*. *Maximum separation* was the best performing application-agnostic allocation policy. It isolates applications from one another.

These results are shown in Figure 38. Models are evaluated with a different number of training points ($p = 10$ and $p = 20$) and four sets of features (max R^2, max ρ^2, min R^2, min ρ^2), for eight comparisons in total. These four features sets proved to be the best performing features.

For the "max" features, the feature selection maximizes the single largest R^2 or ρ^2 value across application models. In effect, this metric chooses the features that resulted in the best single model. For the "min" feature sets, the feature selection tried to maximize the single smallest R^2 or ρ^2 value across application models. This feature selection goal chose the features that most helped the worst performing model.

The models regularly beat the best static policy. Naturally, the choice of features greatly impacts the ability of the models to make accurate predictions. Additional training (i.e., a higher value of p) results in better performance.

The best set of features to use depends on application thread count. This is a result of changing cache and memory subsystem behavior. For some thread counts, communication speed may be a bottleneck. In others cases, cache space becomes the bottleneck. Consequently, the features used to predict and select application performance might also change to reflect the causes of application slowdown.

The policy increases performance by taking advantage of per-application nuances in best-performing affinities. Some pairs of applications prefer to be spread out across L3s and sockets, whereas others perform best when packed tightly together on the same sockets. An application-agnostic policy, like maximum separation, cannot adjust to individual application preferences. The results in Figure 38 show that the models can make the correct thread-to-core mapping decisions.

5.4 CONCLUSION

Application affinity can have a large impact on performance. It is important to select the “right” affinity to maximize performance. This chapter showed AUTO-FINITY’s two techniques to automatically select application affinities.

The first technique predicts performance for an untrained-for application using a potentially untrained-for thread count. From training data, AUTO-FINITY uses machine learning to derive thread-count-independent models. Using AUTO-FINITY’s models, a policy, MAGNET, was built. MAGNET minimizes application runtime via affinity selection.

MAGNET was used on untrained-on applications across a range of thread counts. In 12 of 15 cases where affinity has a significant impact on performance, MAGNET’s selection performs within 1% of or better than fixed assignments that do not consider application behavior. In two of these cases, MAGNET outperforms the fixed assignments.

The second technique quantitatively models application performance given affinity. The generated models rely on statically derived information and offline profiling. The models allow for the fast examination of performance across the large number affinities possible on modern multicore, multiprocessor machines. The second technique was used to build a policy, COMPASS. COM-

PASS can be used to predict and improve application performance (up to 39% improvement) as compared to the best static policy.

6.0 AUTO-GENEOUS: MODELS PARAMETERIZED BY CORE TYPE AND THREAD COUNT

Today's mobile and server computers execute a range of workloads under varying quality-of-service (QoS) and power (energy) requirements. To satisfy these requirements, *asymmetric chip multiprocessors (ACMPs)* are emerging that employ multiple *core types*. The cores are instruction-set compatible but optimized for different objectives. This architecture style allows a careful balance between performance and power by mapping the workload to the most appropriate number of threads and core type, given workload properties, QoS, and power/energy requirements.

In an ACMP, a power-efficient core type typically has a relatively low clock frequency and simple microarchitecture, while a performance core type has a high clock frequency and sophisticated out-of-order microarchitecture to extract instruction- and memory-level parallelism (ILP/MLP). Multiple cores of each type are usually available. For example, ARM's big.LITTLE architecture has two core types: a "big" Cortex-A15 for performance and a "little" Cortex-A7 for power efficiency [19, 37]. Initial products using ARM big.LITTLE have 2 to 4 cores of each type. NVIDIA, Samsung, TI, MediaTek, Renesas and others are all pursuing asymmetric CMPs [35, 42, 68, 69, 87]. The trend toward asymmetry, with possibly dozens of cores of different types in a single ACMP, will continue and likely accelerate as "dark silicon" takes hold [34, 36, 108, 109].

This chapter describes AUTO's modeling technique, AUTO-GENEOUS, to predict application performance under different mappings in ACMPs. An overview of the technique is shown in Table 32. AUTO-GENEOUS automatically creates estimation models to help the allocation policy choose both thread count and core type. Models predict performance scalability curves that are parameterized by core type and thread count. AUTO-GENEOUS is the first approach to handle both parameters for multithreaded applications executed on ACMPs.

Table 32: AUTO-GENEOUS characteristics

Property:	**Value:**
Model Function	$M_{program}(tc, ty)$
AUTO Control Knobs	core type, thread count
Profile Space	$O(tc + \|inductees\| \cdot inflection\_points)$

Profile data is used to build the models. To minimize this data, a *projection function* transforms a scalability curve for one core type to another target core type without full training for the target core. AUTO-GENEOUS updates the models with information gathered during actual application execution to increase prediction accuracy. This reduces *a priori* profiling and ensures models are portable across ACMP architectures.

This chapter makes the contributions:

- Estimation models to capture a multithreaded application's performance scalability when executed on asymmetric core types;
- A technique, AUTO-GENEOUS, to create and use estimation models;
- Algorithms to select the best modeling methods;
- Algorithms to transform a model for one core type to another core type with minimal additional profile data; and,
- An extensive evaluation of a sample policy, QoST, to show AUTO-GENEOUS's effectiveness.

The rest of this chapter is organized in the following way. Motivation and an overview of AUTO-GENEOUS are presented in Section 6.1. The overall modeling technique is described in Section 6.2. Detailed approaches are in Section 6.3. An evaluation of AUTO-GENEOUS and a sample policy, QoST, is described in Section 6.4. The conclusion is in Section 6.5.

6.1 MOTIVATION AND OVERVIEW

Thread count and core type affect the performance of a multithreaded application in an asymmetric CMP, influencing whether quality-of-service is met (e.g., performance is in some range under a power cap). The relationship of these parameters is complex, depending on both software and hardware properties.

The best thread count choice is affected by communication and synchronization (software scalability) and the hardware resources (core type) allocated to the application. Core type can change the relationship between application behaviors, including the ratio of time spent in critical sections versus non-critical sections, the time spent on computation versus communication, wait time at synchronization points (e.g., at a barrier), etc. For example, using a fast core type may cause an application to spend less time in critical sections, scaling better due to less lock contention. Thread count also has interactions with hardware factors, such as cache and memory subsystem pressure. With more threads and better scaling, there can be more resource demand on the hardware. This increased demand may be satisfied by using big cores with large caches. Alternatively, the demand may even limit inherent application scaling on little cores due to small caches. Similarly, an application that does not scale may be unable to take advantage of hardware resources of big cores.

In addition to thread count, the way individual threads use core resources affects performance. Specifically, big cores are designed to extract instruction- and memory-level parallelism: For a thread to "run fast" on a big core, it needs to exhibit both ILP and MLP—in fact, the thread may execute as fast on a small core as a big core, if it is memory-bound and lacks sufficient ILP to mask memory operations [106]. ACMPs may cluster resources, such as big cores that share a large L2 cache [37]. For an application with a large working set (i.e., cache footprint), this cluster might achieve better performance than a cluster with little cores and a small shared cache.

There can even be multiple configurations of thread count and core type that have equivalent performance, further compounding the challenge of the mapping problem. Figure 39 demonstrates this behavior. The figure gives a histogram of performance bins for several multithreaded applications (see Section 6.4 for experimental details) on three core types. The x-axis shows uniformly-spaced bins of *instructions per nanosecond (IPN)*. This metric is aggregate instructions per cycle (across all threads) scaled by core clock frequency. IPN measures both microarchitecture and

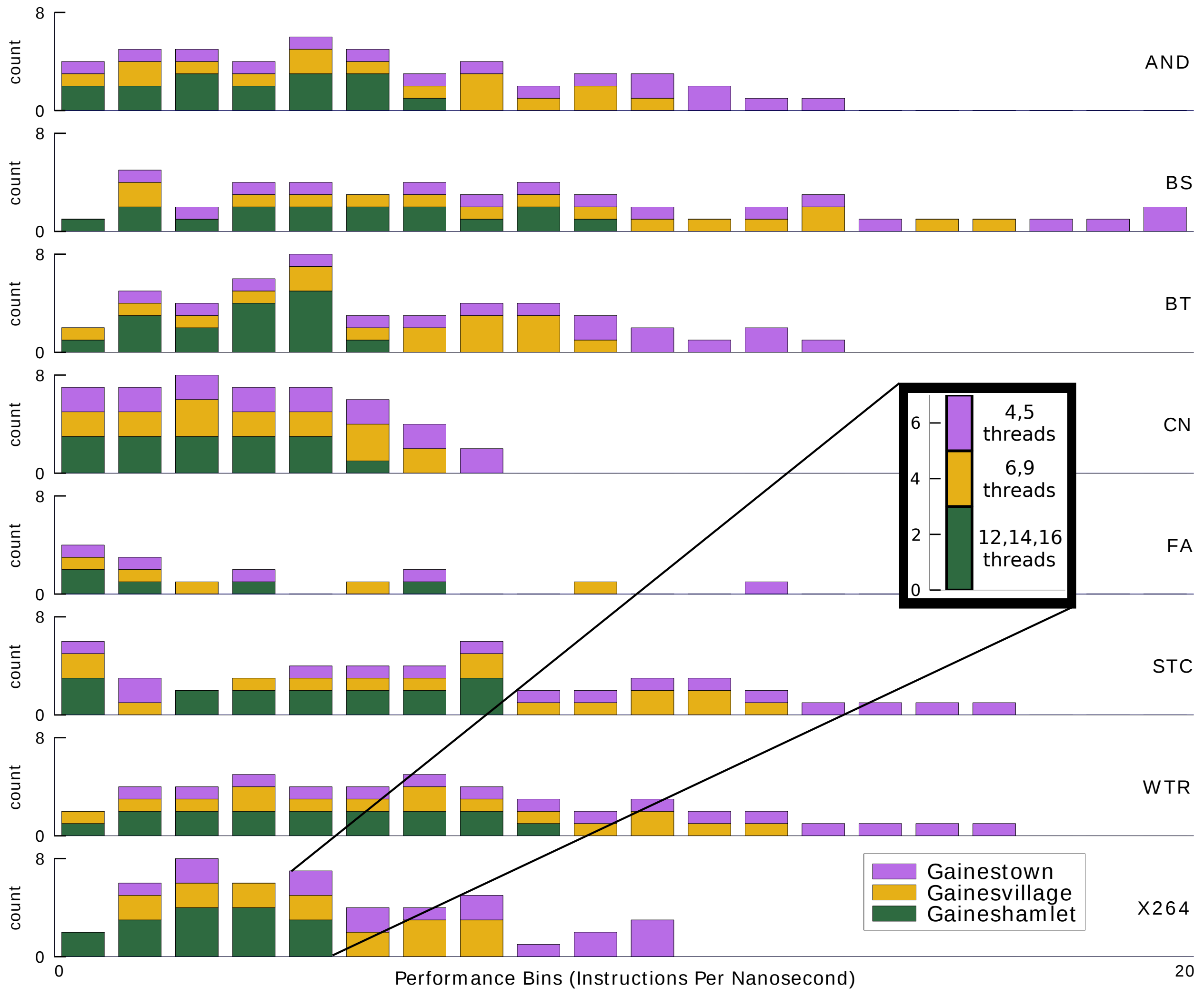

Figure 39: Effect of thread count and core type on performance

clock frequency differences among core types. Work unit throughput could also be used to measure application performance. For the evaluated applications, IPN is expected to correlate strongly with throughput. In the figure, there are three core types: Gaineshamlet (a slow, narrow pipeline; shown in green), Gainesvillage (a moderately fast, medium width pipeline; shown in purple) and Gainestown (a fast, ILP/MLP-oriented pipeline; shown in orange).

The results in the figure reflect several factors. First, some applications do not scale well regardless of core type. For example, in *CANNEAL*'s (CN) histogram, all thread count and core type configurations are found in the lowest performance bins (left side). Other applications have better scalability, such as *BLACKSCHOLES* (BS), where all bins have at least one configuration. Second, the choice of core type influences scalability. In *X264*, performance is limited on Gaineshamlet: scaling stops after the fifth bin. The application scales better on the larger core types (Gainesvillage and Gainestown), as the configurations are more widely distributed in the bins. on scalability. Finally, different thread count and core type configurations often have similar performance. The call-out for *X264* demonstrates this behavior, which shows the fifth bin's detail. In this bin, a configuration with 12, 14 or 16 threads mapped to Gaineshamlet (little core) is performance equivalent to 4 or 5 threads mapped to Gainestown (biggest core). Although certain thread count–core type configurations may be performance equivalent, their power/energy may differ. The availability and number of core types may also vary; e.g., in a clustered ACMP architecture, only clusters of cores may be powered on/off, influencing the relative cost of one core type versus another. The selection of a specific thread count and core type, within a performance bin, is driven by these issues external to performance requirements.

All of these factors should be balanced in mapping the application to an ACMP to achieve QoS. Thus, the allocation policy should consider the relationship between thread count and core type during mapping. AUTO-GENEOUS assists in determining the mapping. AUTO-GENEOUS's *estimation models* predict an application's performance scalability. The models are parameterized by thread count and core type, enabling the Runtime Configurator and allocation policy to simultaneously consider both parameters to pick a mapping. The models can be used to identify performance equivalent configurations during mapping; external requirements on other important factors, including power, energy and thermal behavior, can then drive the selection of a specific configuration within the set of performance equivalent ones.

For the models to be most useful, they must be accurate and automatically portable across ACMP architectures. For this purpose, AUTO-GENEOUS's approach empirically constructs models from profiling data to expose the interaction between an application's properties (scalability, ILP and MLP) and ACMP architecture (core types and caches). From the profile data, models are automatically built, with regression analysis, to estimate performance for different mappings of thread count and core type. Automatic portability is desirable due to the expected range of ACMP architectures enabled by dark silicon. It is also necessary to ensure the approach is feasible since

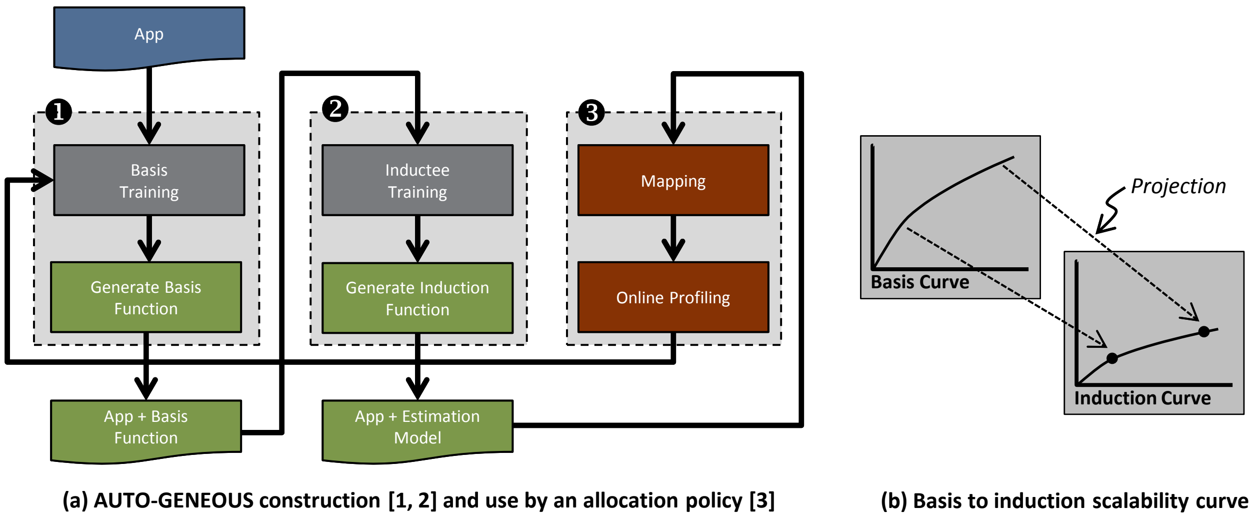

Figure 40: Overview of AUTO-GENEOUS

an application's model is bound to a specific processor.

Similar to other experimental approaches, the time to profile (train) and build a model is a concern. To gather full profile data would require multiple application runs using different thread counts on each core type available in the target ACMP. For example, the eight applications in Figure 39 would take approximately 20 hours (total) to train and build the models for an ACMP with three core types. Instead, as explained in the next section, AUTO-GENEOUS builds a model for the basis core type from available profile data and projects it to the other core types in ACMP. The basis core is the fastest core in the ACMP to improve profiling speed. Available basis profile data is augmented with selected data for each additional core type in the target ACMP. These additional types are called *induction cores* (or, simply "inductee"). A model is built by projecting performance on the basis to the inductee types. This allows constructing models in-situ on the target ACMP to best capture thread count–core type relationship with low profiling cost.

6.2 MODELING PERFORMANCE ON ACMPS

The approach, AUTO-GENEOUS, to build and use a model is depicted in Figure 40(a). It has three steps: 1) basis, 2) induction, and 3) model update and use. Each step is described in the following sections.

6.2.1 Basis Model

In the first step, a *basis function* is created for the fastest, largest core type. This function estimates an application's performance on the biggest core as thread count is changed. The basis function has the form $B(tc) = perf$, where $perf$ is performance according to the allocation policy. tc is the thread count of the application on the basis core type. In essence, this function defines an application's performance scalability curve across thread count on the biggest core.

B must accurately estimate performance because it is used to make mapping decisions for big cores *and* to approximate scalability for inductee (little) cores. Profile data about application execution on the basis type is used to create B. Profiling can be done during application development or installation. The profile can be partial: Missing data can be gathered online during actual application execution to improve basis function accuracy. Typical usage of AUTO-GENEOUS will gather hardware counters (e.g., instructions retired, cycles) for multiple thread counts. Application-specific metrics can also be profiled. From this data, AUTO-GENEOUS generates B using *adaptive regression analysis* to select among different function forms and parameters to represent basis performance.

6.2.2 Induction Model

In the second step, functions are created to estimate performance scalability on the induction (little) core types from the basis core. These functions are *projections* that transform B to fit expected application scalability for the inductee cores. A projection function for a core type, $P_{inductee}$, is derived from basis training with a small amount of additional profiling on the inductee. Figure 40(b) illustrates the concept. In this figure, the basis function is transformed by a projection to match scalability for the induction core. There is one projection per induction core type.

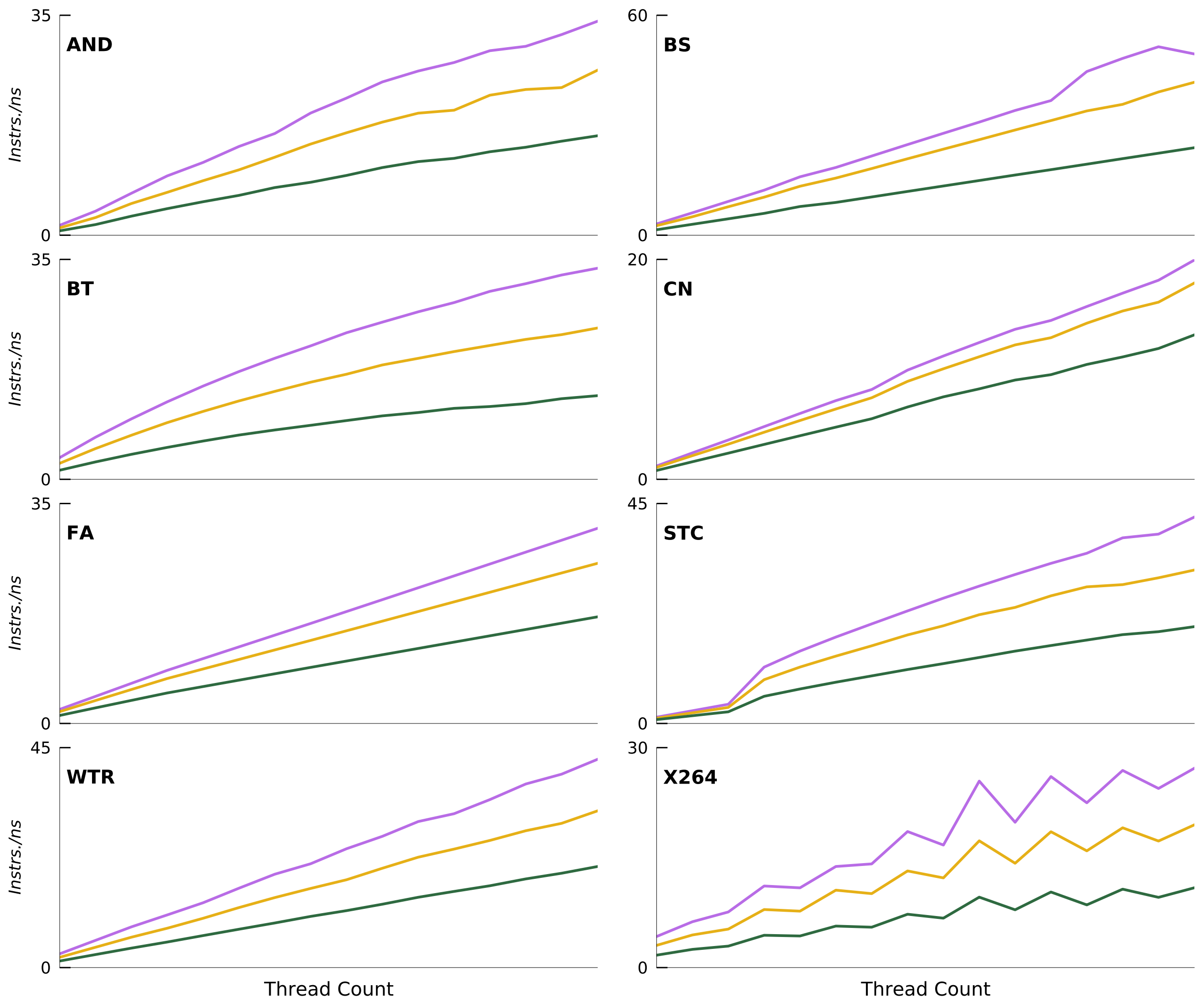

Figure 41: Performance for 1-16 threads on three core types

To get P for each inductee, AUTO-GENEOUS uses regression analysis between basis and inductee types. Thus, performance is estimated for each induction core using B in P. The analysis requires training on some profiled thread counts for each induction core type. These thread counts are called *inflection points*. For example, there are two counts used in Figure 40(b). This process minimizes overall profiling because only partial data is used for induction types. The check for inflection can use fewer or more thread counts to trade accuracy versus training cost.

This approach works because the actual scalability curves of the induction core types typically follow the basis core type. Figure 41 shows performance as thread count is changed for eight multithreaded applications on the Gainestown, Gainesvillage, and Gaineshamlet core types. The y-axis is observed IPN at the thread count on the x-axis; the curves were generated by running all applications on each core type for 1 to 16 threads with full evaluation data. These curve shapes capture the expected range of behavior for most applications.

As the figure shows, in general, the curve shapes for an application are similar across core types. For example, in *BODYTRACK*, the shape of the curves match one another, except IPN is shifted down for smaller core types relative to bigger cores. Likewise, in *X264*, the curves have similar shapes, despite many "ups and downs." The figure also illustrates how many inflection points are needed. For *CANNEAL*, two points are necessary to project Gainestown to Gainesvillage and Gaineshamlet cores. However, in some cases, more inflection points can be beneficial, such as *BODYTRACK* where three points are useful to describe the curve's two segments. *X264* has the most erratic behavior, but even in this case, only four training points are needed. While using more training points improves projection accuracy, fewer points can be used, if some error is tolerable.

The similarity of performance scaling on basis and induction core types depends on how much the core types differ. That is, two core types that are quite different may not have similar curves. In particular, big differences in the cache and memory subsystem may cause this behavior. Despite this issue, no situation has yet been observed where the curves are so different that the performance trends for the induction core types are mispredicted enough to harm mapping. This observation has held even for core types with different size last-level caches, as shown in the experiments.

Together, the basis and projection functions define an estimation model. Figure 42 shows an example model, M_{X264}, for *X264*. This model was automatically generated by AUTO-GENEOUS. The model is parameterized by core type, ty, to select the function to use for a core type. The model is also parameterized by thread count, tc, which is used by the individual functions to make a prediction. The functions predict instructions per cycle (IPC) for the basis and inductees. The IPC is scaled by core clock frequency in the model to account for different core speeds. The basis function, $B_{Gainestown}$, in this model is piecewise, with a degree 3 polynomial, to fit the "ups and downs" in *X264*'s scalability curve. Because scaling on the inductees is similar to the basis, a linear function is effective for $P_{Gainesvillage}$ and $P_{Gainestown}$. Section 6.3 describes the regression process to generate a model.

Estimation Model

$$M_{X264}(tc, ty) = \begin{cases} C_{Gainestown} \cdot B_{Gainestown}(tc) & \text{if } ty = Gainestown, \\ C_{Gainesvillage} \cdot P_{Gainesvillage}(tc) & \text{if } ty = Gainesvillage, \\ C_{Gaineshalmet} \cdot P_{Gaineshamlet}(tc) & \text{if } ty = Gaineshamlet \end{cases}$$

Basis Function

$$B_{Gainestown}(tc) = \begin{cases} 0.707 \cdot tc + 0.886 & \text{if } tc \in [1, 9), \\ 1.641 \cdot tc^3 - 51.937 \cdot tc^2 + 545.269 \cdot tc - 1890.809 & \text{if } tc \in [9, 13), \\ -2316.501 \cdot tc^3 + 481.250 \cdot tc^2 - 33.111 \cdot tc + 0.758 & \text{if } tc \in [13, 16] \end{cases}$$

Projection Functions

$$P_{Gainesvillage}(tc) = 0.794 \cdot B_{Gainestown}(tc) - \epsilon_0$$

$$P_{Gaineshamlet}(tc) = 0.484 \cdot B_{Gainestown}(tc) - \epsilon_1$$

Figure 42: Example model for *X264* (tc is thread count; ty is core type; C_{ty} is clock frequency of core type)

6.2.3 Model Update and Use

In the third step, the estimation models may be updated and used for mapping. The profile data for the basis and induction core types is updated as more information becomes available from more application invocations. Initially, the scalability curves may be sparsely populated, i.e., there are only a few profiled thread counts used to derive the basis and projection functions. As the application is executed under more varied situations, including different QoS, corunners, and input data sets, more profiled thread counts with a range of behaviors are seen. This profile information, i.e., hardware performance counter values, is passed back to the regression process to improve model accuracy by taking into account more execution scenarios. With up-to-date information, an application's model is periodically reconstructed. The next time the application is executed, the updated model is used.

An estimation model allows for performance prediction and identification of configurations (thread count and core type) that are performance equivalent. A configuration can then be chosen to meet constraints (e.g., power, energy, thermal, architectural). In many situations, a static (fixed) mapping of the multithreaded application by itself to the target ACMP is sufficient. For example, in mobile OSes, like iOS and Android, the foreground application has highest priority and "owns" the system. Background processes sleep, or have lightweight periodic services that check for critical events. In this situation, the estimation models are useful to map an application in isolation since the application effectively runs by itself. In other situations, like data center consolidation, it may be desirable to consider interference among co-running applications [63, 101]. Although the models reflect interference through online profiling (i.e., of the application with corunners), it may be possible to directly include interference in the estimation model to map multiple applications. Models may also be used for dynamic remapping (i.e., dynamically changing thread count); whenever remapping is triggered, an application's model can guide allocation. When to remap is separate from estimating the application's performance under a given mapping (which AUTO-GENEOUS does address).

6.3 GENERATING AN ESTIMATION MODEL

An estimation model has three parts (see the example in Figure 42): the model function (e.g., M_{X264}), the basis function, and the projection functions. AUTO-GENEOUS constructs each of these functions. The model function is straightforward because it simply scales the basis or projection functions by a core type's clock speed. This accounts for speed differences of the core types.

The basis and projection functions are more involved to generate. For this purpose, AUTO-GENEOUS uses *adaptive regression analysis*. This analysis identifies an accurate way to estimate performance with low profiling cost.

The analysis generates a piece-wise basis function, B, with s pieces. Each piece i in B is estimated by a polynomial of some degree, x_i, where $1 \leq i \leq s$, $x_i \leq maxdegree$, and $maxdegree$ is a fixed limit on the degree. A piece is defined by two inflection points, i.e., a range of thread counts, $[tc_{in,i}, tc_{in,i+1})$. $tc_{in,i}$ is the start inflection point for pieces i. As an example, consider Figure 42. The basis has three pieces at thread counts $[1, 9)$, $[9, 13)$, and $[13, 16]$. The first piece is estimated by a degree-1 polynomial and the second and third pieces are estimated by degree-3 polynomials.

There is a trade-off between the choice for s and each x_i. A larger s implies B will better match sharp changes in application scalability. However, with more pieces, there is potentially less profile data available to fit a higher degree polynomial. There may not even be "enough" profile points (thread counts) in a piece to build a particular degree polynomial, or the function may over fit. The number of pieces also affects how much profiling is needed for each induction core type. With more pieces, more profiling is necessary to project from basis to inductees. Thus, the goal of the analysis is to minimize s and select each x_i to most accurately fit basis training data without overfitting.

To find the basis function, adaptive regression analysis varies s, $tc_{in,i}$, and x_i to generate a set of possible candidate functions from which one is selected as B. The analysis proceeds in several steps:

1. Collect initial profile data for the basis core type by executing the application with different thread counts. This data may be incomplete but should reflect the expected range of thread counts for the application.
2. Select a portion of profile data uniformly distributed among available profiled thread counts to train the basis function.
3. Select s, $tc_{in,i}$, and x_i to maximize basis accuracy without overfitting, while reducing induction training. This step outputs the best B.
4. For each piece i in B, profile the application on each induction core type at the start inflection point, $tc_{in,i}$. For the last piece, profile the application with the first and last inflection point for the piece. If an inflection point is already available in the profile data, then do not profile it again.
5. Using B and the induction profiles, generate projections to transform B to fit the profile data. This step outputs $P_{inductee}$ for each inductee.

6. Collect online profiles during application invocation and periodically reconstruct the estimation model (repeat from step 2).

Figure 43 shows SELECTBASIS, the algorithm to build the basis function. This function is used in step 3. The algorithm inputs available profile data for an application. The profile is a database of tuples $\langle tc, ty, perf \rangle$ that record performance ($perf$) for different core types (ty) and thread count (tc) combinations. The profile can be incomplete, containing a mix of thread counts and core types.

Regression extrapolates performance scalability of an application for the basis core type from the available profile data. On Line 4 in the figure, profile data for the basis core is extracted to serve as *train data* for regression. This portion of the basis profile (TRAINAMT) is selected to be uniformly distributed (UNIFORM) across available thread counts. To evaluate fitness of a basis candidate function, the full basis profile data is extracted as *test data* on line 5.

The algorithm traverses the iteration space of number of pieces (line 7), assignment of thread counts to inflection points (line 10), and polynomial degree (line 11). The assignment of thread counts as inflection points defines the start and end points for the s pieces. The assignments, called "splits", are determined by SPLITPIECES on line 8. Similarly, all combinations of polynomial degrees for the splits are computed on line 9. The maximum number of pieces and polynomial degrees have fixed limits (constants MAXSPLITS and MAXDEGREE) to constrain the size of the iteration space. On line 12, GENBASIS generates a candidate basis function from the number of pieces, splits, and polynomial degrees using least squares regression. GENBASIS is shown in Figure 44.

Some parameter combinations can lead to invalid candidate basis functions. This situation happens when there are not enough profile points in the train data to allow regression on the function defined by the parameters. For instance, a large number of pieces (s) and high-degree polynomials for each piece ($degrees$) may lead to an underspecified function. GENBASIS returns NULL on line 12 in Figure 43 in this case.

The fitness of a candidate function is evaluated on lines 15 to 20 by determining a penalty value of relative benefit to cost. The penalty is computed on line 17. Benefit is measured indirectly as mean squared error (MSE) on the test data; a larger MSE implies worse accuracy, and therefore, less benefit. Cost is measured as how many inflection points in the candidate basis function have

```
SELECTBASIS($profile$)
B = NULL // at end, the best basis function
$minPenalty = +\infty$
// extract train and test data sets for basis
$train$ = SELECT($profile$, BASIS, TRAINAMT, UNIFORM)
$test$ = SELECT($profile$, BASIS, ALL)
// iterate over pieces, splits of pieces and degrees
for $n = 1$ to MAXSPLITS
    $splits$ = SPLITPIECES($train, n$) // inflection points
    $D = \{1, \ldots,$ MAXDEGREE$\}^n$ // Cartesian exp.
    for $split$ in $splits$
        for $degrees$ in $D$
            $candidate$ = GENBASIS($train$, $split$, $degrees$)
            if $candidate \neq$ NULL
                // evaluate appropriateness of function
                $E$ = COMPUTEMSE($candidate$, $test$)
                $C$ = NUMMISSINGPOINTS($split$, $profile$)
                $penalty = E^2 \cdot (C + 1)$ // penalty function
                if $penalty < minPenalty$
                    $minPenalty = penalty$
                    $B = candidate$
return $B$
```

Figure 43: Algorithm to adapt and select basis function

not been previously profiled. The thread count for each unseen inflection point on each inductee core is executed to get data for projection, and thus, the unseen points impose cost. In the penalty calculation, the error (E) is weighed more heavily than the cost (C). The penalty is minimized

```
GenBasis($train$, $split$, $degrees$)
candidate = [ ]
for $i = 1$ to NumSplits($split$) // number of pieces
    // extract piece $i$'s profile data
    $regres$ = Select($train$, ThreadCount, $split[i]$)
    // regression on piece $i$'s profile data
    $pieceFunc$ = LSRegression($regres$, $degrees[i]$)
    if $pieceFunc$ == NULL
        // Underspecified polynomial (not enough data)
        return NULL
    // add the piece's function to basis
    $candidate$ = Append($candidate$, $pieceFunc$)
return $candidate$
```

Figure 44: Algorithm to generate candidate basis function

(lines 18 to 20) to find B.

Once the basis function is selected, projection functions for the inductees can be generated (step 5 of adaptive regression analysis). The algorithm to generate a projection function is shown in Figure 45. The algorithm is invoked for each induction core type in the ACMP. The projection function "reshapes" the basis function by adjusting it to fit the application's performance on an inductee.

Similar to basis generation, GenProjection adapts the projection function based on fit. In this case, only polynomial degree is changed and a single piece is used. A higher degree polynomial is useful to describe a more complex relationship between basis and induction scalability than a simple linear shift. As more profile data becomes available about the induction core type, the accuracy is improved and a higher degree polynomial can be used.

In Figure 45, GenProjection extracts profile data for an induction core. This data are tuples

```
GENPROJECTION($profile, B, split, inductee$)
// generate profile data for unseen inflection points
for $tc$ in $split$
    if not EXIST($profile, inductee,$ THREADCOUNT, $tc$)
        INSERT($profile$, $tc$, RUNPROFILE($inductee$, $tc$))
// extract all profile data for inductee
$train = $ SELECT($profile, inductee,$ ALL)
$regres = [\,]$ // regression data
// create regression data of basis and inductee IPCs
for $tuple$ in $train$
    INSERT($regres$, $B(tuple.tc), tuple.ipc$)
// find highest degree polynomial to fit regression data
$P =$ NULL
$degree =$ MAXDEGREE
while $P$ == NULL
    $P =$ LSREGRESSION($regres, degree$)
    $degree = degree - 1$
return $P$
```

Figure 45: Algorithm to generate projection for an inductee

of thread count and performance (e.g., hardware performance counters). B is used to predict the IPC of the basis core type, which is mapped by regression to the profile data for the induction core type. The mapping gives the necessary (x, y) data required by regression. For a given thread count, B is used to compute the basis aggregate performance as x and the profile data is used to get performance for the inductee as y. Using the data, the algorithm first tries a high degree polynomial. If this degree cannot be used (i.e., due to missing data), then the next lowest degree is tried until a function is generated.

Table 33: PERF architecture parameters

Core architecture	
Gainestown	Freq=2.66GHz,Dispatch=4,Window=128,LSQ=10
Gainesvillage	Freq=2.4GHz,Dispatch=2,Window=64,LSQ=5
Gaineshamlet	Freq=2.2GHz,Dispatch=1,Window=32,LSQ=2
Cache architecture (L1, L2 per core and L3 per processor)	
L1 cache	private I&D, 32 KB, 4-way, 64 B block
L2 cache	private 256 KB, 8-way, 64 B block, 8-cycle hit
L3 cache	shared 8 MB, 16-way, 64 B block, 30-cycle hit
Main memory (DRAM)	
Config.	45 ns access, 1 controller/proc. w/2 channels
Multiprocessor organization (multiple sockets/clusters)	
Processor	3 clusters of each core type, 4 cores per cluster
System	4 processors

6.4 EVALUATION

AUTO-GENEOUS was implemented and evaluated to determine its effectiveness. The evaluation examined model accuracy as key parameters are varied. Additionally, AUTO-GENEOUS is used in a policy, QoST, to predict IPC and meet QoS while minimizing the cost (e.g., power) of doing so.

6.4.1 Methodology

The Sniper CMP simulator, version 5.1 [94], was used to evaluate two ACMPs, PERF and MOBI. These designs were chosen to study AUTO-GENEOUS on diverse processors. PERF has three core types, shown in Table 33. This configuration models a performance-oriented multiprocessor ACMP desktop or server machine; its three core types cover a range of microarchitecture/speed ratios. PERF has four processors, where each processor has three clusters of each core type, with four cores per cluster. Within each processor, only one cluster can be powered on at a time, and all clusters share the last-level cache (LLC) and memory subsystem. The microarchitecture for each core type differs in ILP and MLP aggressiveness. MOBI is a many-core design for future mobile systems; the parameters are shown in Table 34. It has 16 cores of two types, Grande and Petite.

Table 34: MOBI architecture parameters

Core architecture	
Grande	Freq=1.86GHz,Dispatch=3,Window=64,LSQ=10
Petite	Freq=1.36GHz,Dispatch=2,Window=32,LSQ=2
Cache architecture (L1, L2)	
L1 cache	private I&D, 32 KB, 4-way, 64 B block
Grande L2 cache	shared 2 MB, 8-way, 64 B block, 8-cycle hit
Petite L2 cache	shared 512 KB, 8-way, 64 B block, 8-cycle hit
Main memory (DRAM)	
Latency	45 ns DRAM access, 1 controller/proc. w/1 channel
Processor organization	
Processor	2 clusters of each core type, 16 cores per cluster

The cores for each type are clustered. Unlike PERF, the clusters do not share the LLC. The Grande cluster has a large LLC and the Petite cluster has a small LLC. Most experiments evaluate only PERF; the trends are similar for MOBI. MOBI is evaluated in the QoS study.

Multithreaded benchmarks were selected to cover a spectrum of MLP, ILP, and scaling. The benchmarks are executed to completion with the simulator. The applications are: *BLACKSCHOLES*, *BODYTRACK*, *CANNEAL*, *FLUIDANIMATE*, *STREAMCLUSTER*, and *X264* from PARSEC [10]; Andersen's parallel points-to analysis (*ANDERSEN*) [97]; and, *WATER_NSQUARED* from Splash-2 [112]. *ANDERSEN* is a proxy for graph-intensive applications. The benchmarks were compiled with clang 3.2 (LLVM) and -O3 optimization flag. The same applications are used for PERF and MOBI to ease comparison between processors. One data set (*simmedium*) was used to generate the models, and another one was used for evaluation (*simlarge*). Unless stated otherwise, the models were generated from 50% of possible *simmedium* training data (even thread counts).

Several evaluation metrics are used, including mean relative error, performance, and runtime. *Mean relative error* is the absolute value of the difference between a model prediction and actual performance reported from the simulator for *simlarge*. *Performance* is reported as IPN. *Runtime* for profiling and analysis is collected by executing AUTO-GENEOUS on a real machine. QoS and cost (e.g., power) are also evaluated, as described later.

Table 35: Parameters for models generated by AUTO-GENEOUS for QoST on PERF

Program	Pieces	Degrees	Inflection Points
AND	2	1, 1	1, 9, 16
BS	2	2, 2	1, 11, 16
BT	2	2, 2	1, 7, 16
CN	1	3	1, 16
FA	2	1, 1	1, 5, 16
WTR	3	1, 2, 2	1, 5, 11, 16
STC	2	1, 1	1, 13, 16
X264	1	2	1, 16

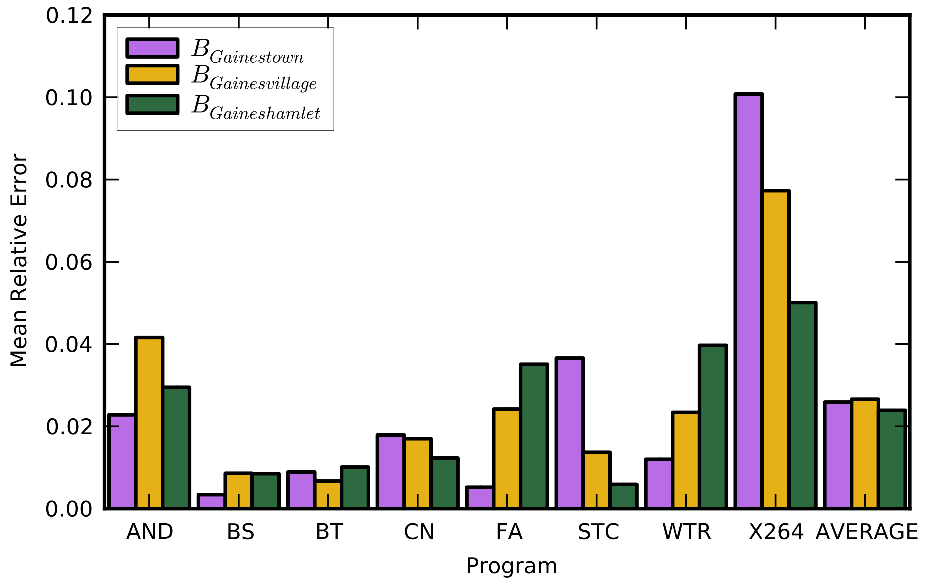

Figure 46: Basis model accuracy

6.4.2 Model Evaluation

Table 35 summarizes the models generated by AUTO-GENEOUS. Typically, the models have one or two pieces (splits) with low degree. For instance, *FLUIDANIMATE* scales mostly linear with a slight inflection; consequently, AUTO-GENEOUS generated a basis function that has 2 pieces, each with degree 1. Interestingly, *FLUIDANIMATE*'s curve in Figure 41, gathered for *simlarge*, has linear scaling, yet a 2 piece function was generated for *simmedium*. This happened as a consequence of using a different and partial data set to generate the model. *X264* illustrates this phenomena as well: The model generated by AUTO-GENEOUS has 1 piece with degree 2, which follows the "curving" trend in the profile. Because the profile is partial (50% of thread counts), the "ups and downs" in Figure 41 are not fully seen by adaptive regression analysis. A model was constructed from *X264*'s full evaluation data; this "optimal" model is shown in Figure 42 as an earlier example. This model is more complex than the one generated from partial training due to missing points. As this case illustrates, models can have error depending on data set and completeness of their profiling. Model error is examined next.

6.4.2.1 Model Accuracy Figure 46 shows mean relative error depending on the core type chosen for generating the basis function. On a per-application basis, the choice of basis core type has an impact. For example, on *FLUIDANIMATE*, the basis generated for Gainestown has minimal error (0.005), whereas for Gaineshamlet, the error is 0.04. *FLUIDANIMATE*'s behavior varies from one core type to another. On Gainestown, it scales linearly, while on Gaineshamlet it scales with a slight curve. Curves are harder to predict than lines, resulting in slightly increased error. In *STREAMCLUSTER*'s case, the basis for Gainestown has higher error than the basis for Gaineshamlet (0.04 versus 0.005). *STREAMCLUSTER*'s scalability, beyond 12 threads, shifts and slightly flattens. The flattening is more dramatic for Gainestown due to increased scaling (higher slope) between 1 and 12 threads. For Gaineshamlet, the drop is less significant, leading to a more accurate function.

Despite differences for some applications, the choice of basis for PERF has a negligible overall affect, as the average mean error shows. MOBI has similar behavior. Because the fastest processor, i.e., Gainestown and Grande, reduces profiling time, it should be used as the basis type. In the rest of the paper, models use Gainestown for PERF and Grande for MOBI as the basis types.

Next, the accuracy of projections is examined. Figure 47 shows mean relative error, with Gainestown as the basis. Inflection points were used to build projections for Gainesvillage and Gaineshamlet. The error of the Gainesvillage projection is consistently smaller than Gaineshamlet's projection (except *ANDERSEN*). Because Gainestown is more similar to Gainesvillage than Gaineshamlet, the error is lower for this projection. Gainesvillage's characteristics are between those of Gainestown (the large, aggressive PERF core) and Gaineshamlet (the smallest PERF

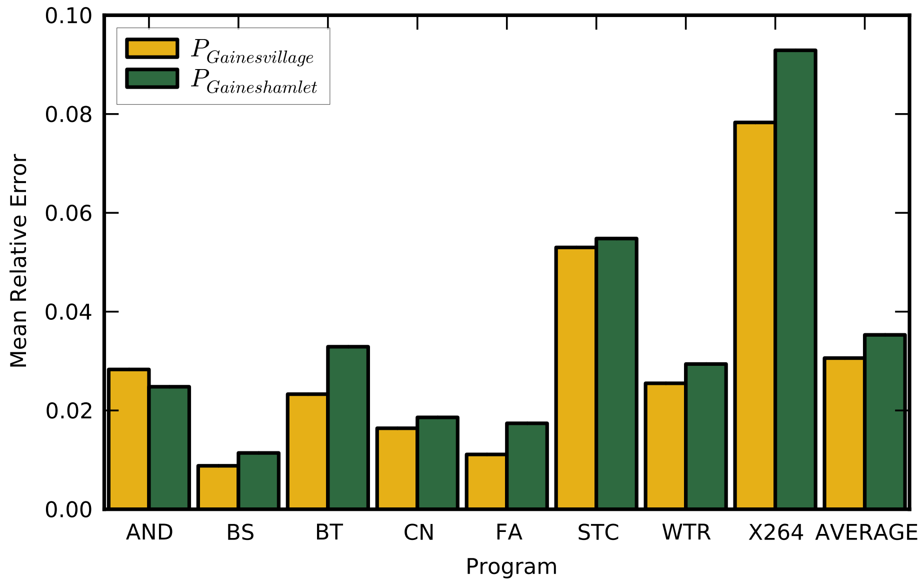

Figure 47: Projection model accuracy

core). Gainesvillage predictions are more accurate because performance of the two core types are more similar and more easily described by projection given limited training data. Nevertheless, the Gaineshamlet error is small: On average Gaineshamlet has less than 0.04 relative model error.

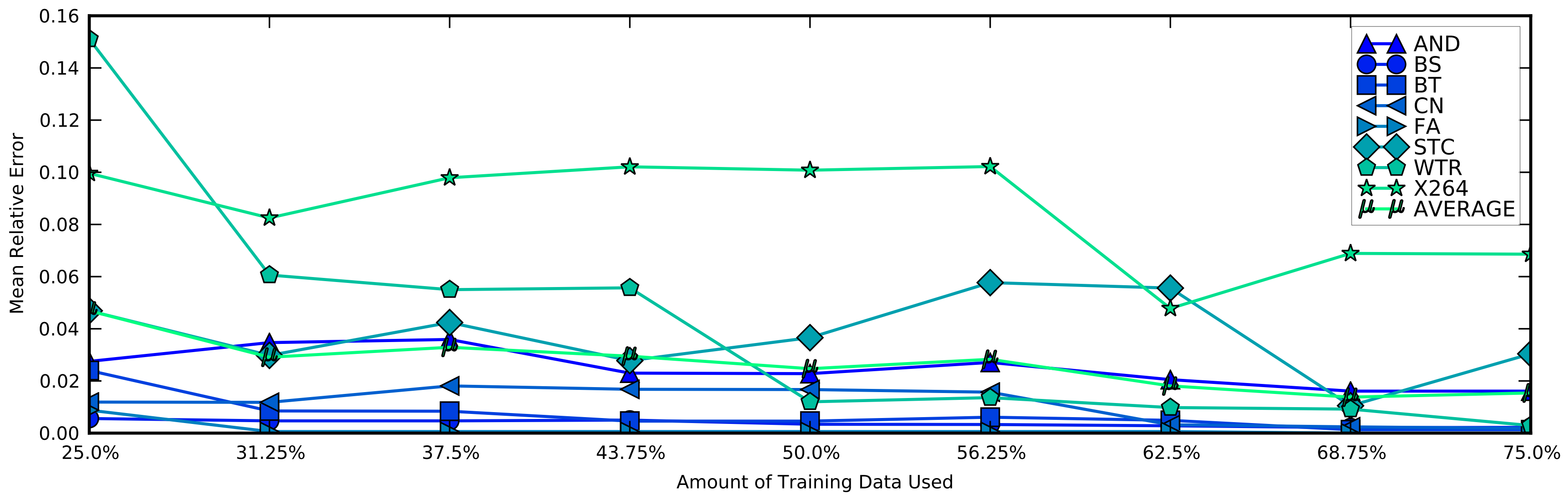

Figure 48: Basis model (Gainestown) as profiling increased

6.4.2.2 Online Updating of Models The amount of profile data to construct the basis function can have an impact on accuracy. Figure 48 shows change in error with increasing quantity of profile (training) data. This figure indicates how AUTO-GENEOUS's online profiling can improve accuracy as more data becomes available. The far left shows 25% profile data (4 thread counts: 4, 8, 12, 16). Additional points are selected uniformly until 75% of the full profile space is used. The results demonstrate that, as expected, additional training reduces basis function error. *ANDERSEN* and *BLACKSCHOLES*'s basis functions slowly improve with more profile points. *BODYTRACK*'s and *FLUIDANIMATE*'s initial error (0.025 and 0.01, respectively) at 25% profile data is halved once an additional point is used (31.25%). After this point, the error continues to decrease consistently, although slowly. The error for *CANNEAL* is steady until 62.5%. At this point, the error drops, which indicates a training sensitivity to thread count.

The error for *STREAMCLUSTER*'s basis function alternates up and down as points are added. The functions generated for each set of profile data do not quite capture *STREAMCLUSTER*'s scaling trend. Indeed, the accuracy of the basis function in this application strongly depends on the choice of thread counts for training. A non-uniform training approach, which focuses on areas where *STREAMCLUSTER*'s performance changes, could address this issue to improve error. Nevertheless, once *STREAMCLUSTER* is trained on 68.75% of profile points, the error significantly drops.

WATER_NSQUARED's error has "plateaus." At 25% training, the error is high (0.15). Given one more point (31.25%), the error drops to 0.06 and remains steady. Once 50% training is used, the error again drops and plateaus. A final drop happens at 75%. Finally, the error is constant until 62.5%. At this point, the error is below 0.08. *X264* has many local minima and maxima, and requires a relatively large quantity of data to accurately capture and predict performance.

Projection function accuracy, as more profile data is used, was also examined. Because the scalability curves are similar between core types, the inflection points alone were usually sufficient to accurately determine a projection. Thus, most applications had minimal improvement (less than 0.01) in projection accuracy as more thread counts beyond the inflection points were used. However, *X264* was, once again, sensitive: The mean relative error varied from 0.07 to 0.15 (average 0.08). A mean relative error of 0.15 was an outlier. This relatively large error occurred if 2 more points than the inflection points were used for training. With just the inflection points, the error

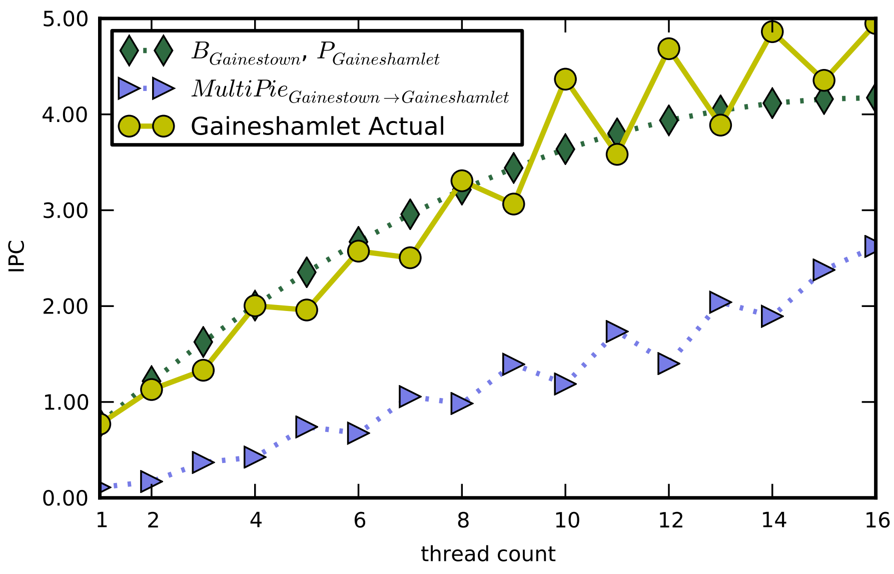

Figure 49: QoST versus MultiPie for *X264*

was 0.08, which is close to the minimum 0.07.

These results show that AUTO-GENEOUS's methods to gather online profile data is useful to increase basis function accuracy. It is less important to gather more profile data for the projection function, but it can help for applications with local peaks and valleys in scaling (e.g., *X264*).

6.4.2.3 Comparison to PIE To put the mean relative error for AUTO-GENEOUS in context to other approaches, AUTO-GENEOUS was compared with Performance Impact Estimation [106]. PIE is extended for multithreaded applications and to make predictions between out-of-order cores. The original PIE supported only in-order to out-of-order and vice versa. This extension is called MultiPie. Figure 49 shows how AUTO-GENEOUS and MultiPie compare for a representative application, *X264*. The model from AUTO-GENEOUS, trained on *simmedium*, is used to predict performance for *X264* on *simlarge* for Gaineshamlet. MultiPie is designed to be used online, therefore it is evaluated on *simlarge* input data. As the figure shows, the model from AUTO-GENEOUS smoothly matches *X264*'s behavior. Sometimes AUTO-GENEOUS over predicts (low thread counts) and sometimes it under predicts (high thread counts), but it captures the overall

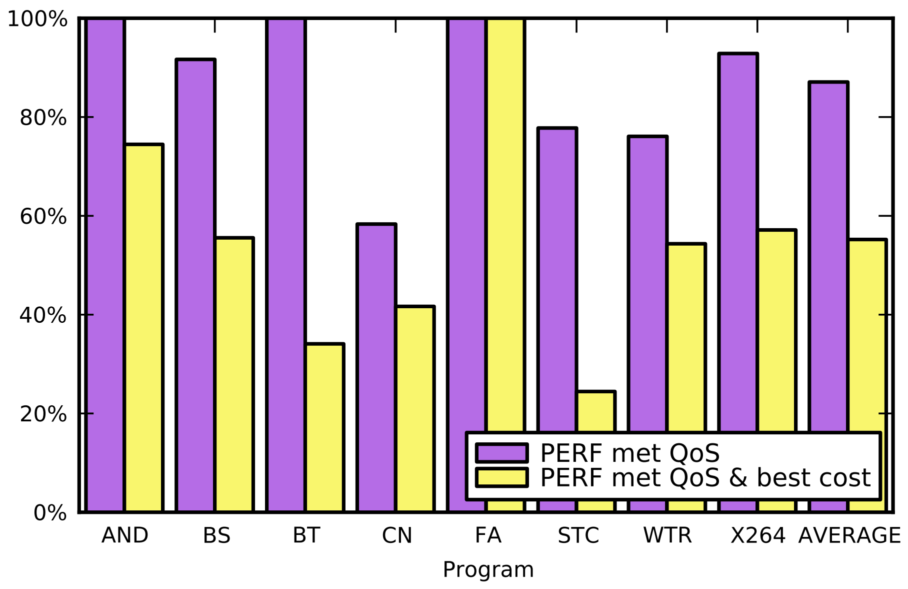

Figure 50: QoS met by QoST on PERF for cost ratios [1.84, 1.32, 1.0]

trend. On the contrary, MultiPie consistently underestimates performance. Although MultiPie identifies local minima and maxima for *X264*, the predictions from the model are inaccurate. The inaccuracy is due to PIE's assumption that processors are "balanced", and in the absence of LLC misses, the IPC is equal to dispatch width. Other researchers have noted a similar observation for PIE and pointed out that no commercial processor is balanced [81]. CPI stacks from Sniper reveal *X264* has multiple sources (e.g., floating point, branch misprediction) of slowdown on Gainesham-let. Similar inaccuracies were found for other applications and core types as well, even for simple cases, like *CANNEAL*.

6.4.2.4 Profiling and Adaptive Regression Analysis Runtime The time to profile and construct AUTO-GENEOUS models was examined. Profile runtime depends on number of profile points; the full space for PERF has 48 points (16 thread counts on 3 core types) per application and data set. By profiling 50% of the thread counts for the basis core type and only the inflection points for each induction core type, the number of profiling points is reduced from 48 to 12-16, with an average of 13.25. To understand how these simulated inputs translate into actual runtime, profile times on a real machine with native data sets were measured. With AUTO-GENEOUS, the profile runtime was improved by 6.83· (*FLUIDANIMATE*) to 9.4· (*X264*), average 8.7·, versus profiling the full space. Per benchmark, the average profiling runtime was reduced from approximately 2.5 hours to 17 minutes.

The time to generate a model is dominated by building the basis function. Projection models are built nearly instantaneously since only the number of degrees is varied. Generating the basis, however, depends on maximum degrees and maximum splits. In the most complex case (16 basis training points, MAXDEGREE $= 3$, MAXSPLITS=3), basis generation took approximately 20 seconds. Relative to profile runtime, model generation contributes only a small amount to AUTO-GENEOUS's runtime. In conclusion, AUTO-GENEOUS's runtime is fast enough to build models as part of normal application development. If small, representative data sets can be used, then the time could even be fast enough to build models during application installation (e.g., on a mobile device).

6.4.3 QoST: A Policy to Select Thread Count, Core Type

To evaluate AUTO-GENEOUS's utility in mapping applications to ACMPs, the models were used in several execution scenarios. The scenarios had different QoS targets and cost ratios of one core type to another (e.g., ratio of power between big and little core types). The evaluated policy, QoST[1], tries to satisfy a QoS target and minimize relative cost. QoST uses an exhaustive search to find the mapping that is predicted to best meet QoS and minimize cost, within a threshold. This mapping is "scored" against the optimal mapping determined through simulation with the evaluation data set for all mappings.

Four cost ratios were used. The ratios are derived from a variety of sources to show that AUTO-GENEOUS is useful in a plethora of situations, regardless of the actual cost model used at runtime. The ratios were: [1.84, 1.32, 1.0], [3.96, 1.98, 1.0], [2.568, 1.565, 1.0], and [6.0, 3.0, 1.0]. The first number in a list is the cost of Gainestown (PERF) or Grande (MOBI) types. The last number is Gaineshamlet or Petite's cost. The middle value is Gainesvillage's cost.

These ratios are derived from McPAT power predictions from core microarchitecture and sup-

[1]$QoST = QoS + cost$

ply voltage (the first and second set of ratios), published ARM big.LITTLE relative power consumption (the third set of ratios), and expected dynamic power ratios in asymmetric quad-core processors (the fourth set of ratios) [5, 59, 92] The ratios give a range of expected relative cost for ACMPs. The cost for a specific mapping is computed as: $\textsc{Cost}(tc, ty) = tc \cdot \textsc{Cost}(ty)$.

QoS is specified as a target IPN (instructions per nanoseconds), Q_{target}, and a threshold, α. QoS is satisfied when actual performance meets or exceeds $(1-\alpha) \cdot Q_{target}$. To test many Q_{target} values for an application, *all* IPN values for that application are evaluated. Cost is deemed minimized when it falls within a factor of α of the best mapping. The choice of α is taken from the set $\{0.2, 0.1, 0.05\}$ to adjust evaluate different QoS and cost strictnesses.

Figure 50 shows how well the models meet QoS ($\alpha = 0.1$). The graph shows two bars for each application; the first bar is the percentage of mappings selected with an application's model (for all Q_{target} values) that satisfied QoS, and the second bar is the percentage that met QoS and achieved minimal cost. In most cases, mappings were selected that met QoS (87% on average). Furthermore, a mapping with the best cost was also typically chosen (55% of cases on average). *ANDERSEN*, *BODYTRACK*, and *FLUIDANIMATE* achieved 100% of the QoS targets due to very accurate models. When cost is considered, these models do well. *FLUIDANIMATE* always lead to the selection of the best mapping. Using the AUTO-GENEOUS models, QoST always meets QoS and minimizes costs. Because the application requires power-of-two thread counts, there are large gaps between mappings, which minimizes the impact of misprediction error on the policy's selections. For *BODYTRACK*, many mappings are performance equivalent (see Figure 39), which causes sensitivity in selection with cost. Another interesting case is *CANNEAL*: 60% of the QoS targets were satisfied, with only 40% having minimal cost. This error is due to a small, consistent bias toward optimism in predicting performance, which lead to the selection of underperforming mappings. This same situation happens for *STREAMCLUSTER* and *WATER_NSQUARED*. Using profiles gathered online can mitigate this problem.

Figure 51 shows how well QoS and cost are met for different cost ratios, α, and processors. Also shown are the results for MultiPie. The figure shows the average percentage of mappings across all applications that achieved QoS and minimized cost. Overall, the models generated with AUTO-GENEOUS do consistently well and accommodate the cost ratios, QoS targets and core types. At $\alpha = 0.2$, for PERF, 97% (average) of QoS targets were satisfied and 73% (average) of the QoS and best cost targets were satisfied. For MOBI, the averages were 95% (QoS) and 68% (QoS and best cost). Even at a very strict target, i.e., $\alpha = 0.05$, the mappings for PERF satisfy QoS 78% (PERF) and 77% (MOBI) on average, with over 37% (PERF) and 38% (MOBI) of mappings matching the optimal. Across cost ratios, for all settings of α, the models do better as the cost ratios increase (increasing from left to right in the figure) due to more "separation" between cores types, leading to fewer cost-equivalent mappings. This affect is most pronounced for $\alpha = 0.05$. In comparison between processors (PERF vs. MOBI), the models for the mobile processor satisfy the constraints slightly less often. This result is attributed to the different cache hierarchies for the two processors.

MultiPie meets QoS in 85% of experiments but, as Figure 51 shows, it rarely meets QoS while also minimizing best cost. MultiPie consistently underpredicts performance, because it does not take into account non-MLP sources of slowdown (e.g., branch misprediction, lack of ILP). Therefore, it prefers the more powerful processor on each platform at the detriment of cost.

These results show that AUTO-GENEOUS generates models that are accurate enough to successfully guide application mapping, i.e., the selection of thread count and core type, across a broad range of QoS targets, cost ratios, and ACMP core types.

6.5 CONCLUSION

Asymmetric chip multiprocessors have multiple instruction-set compatible core types to trade performance and power/energy. This chapter describes, AUTO-GENEOUS, a technique that allows AUTO to automatically construct estimation models that predict a multithreaded application's performance for different thread count and core type mappings. The approach empirically derives an estimation model with low training cost. AUTO-GENEOUS is the first technique to consider both thread count and core type in estimating application scalability for ACMPs. The use of a policy that uses AUTO-GENEOUS, QoST, shows that model predictions are accurate (0.02 error) and beneficial for mapping multithreaded applications. The policy met QoS constraints 88% of the time.

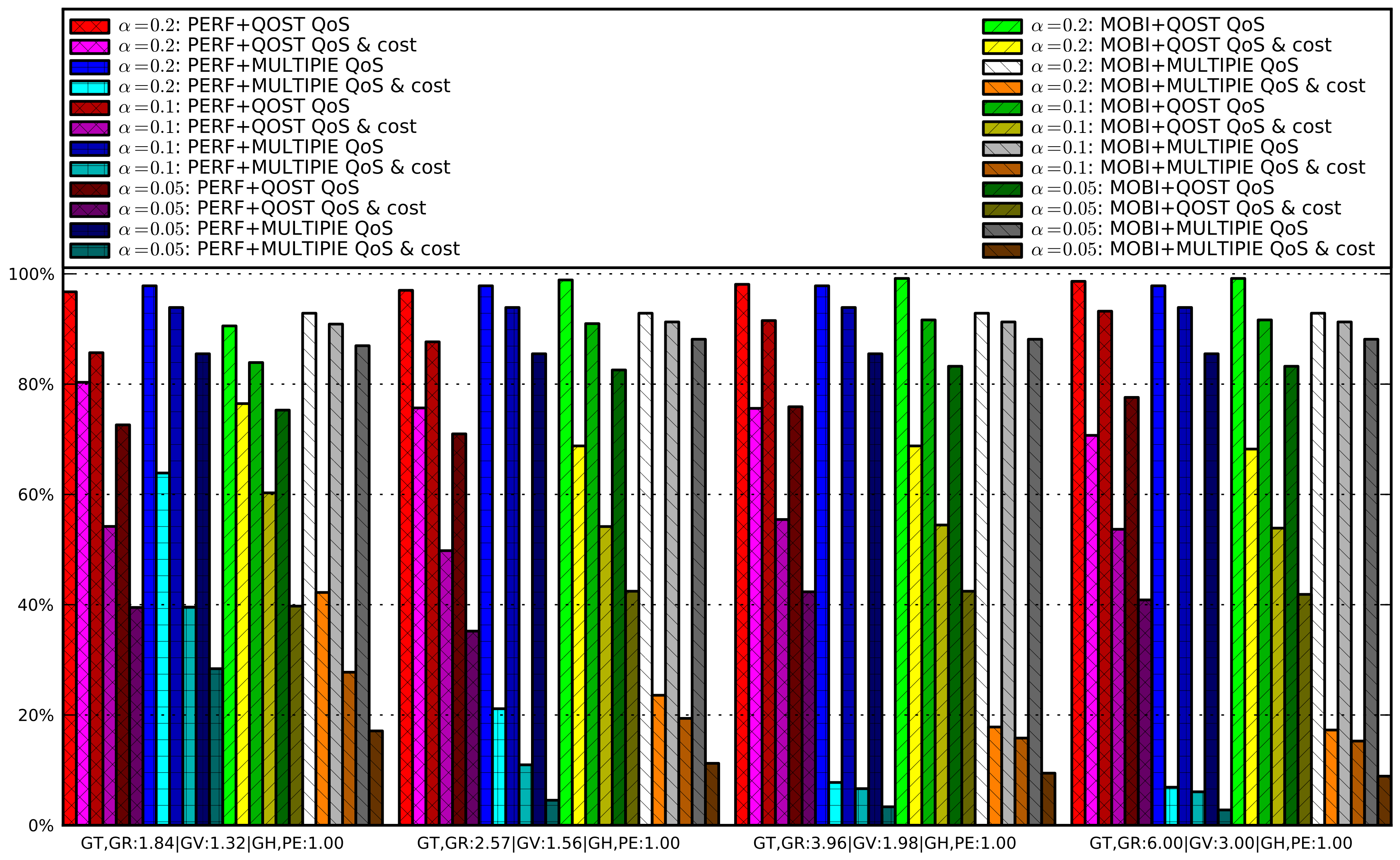

Figure 51: Average QoS met by QoST, PIE for different QoS ranges and asymmetric systems (α)

7.0 CONCLUSION

Today's computers feature multiple processors with several cores and are capable of huge amounts of parallel computation. Nevertheless, cores are a finite resource. An improper allocation of threads, cores, and/or choice of cores to a application will fail to meet performance objectives (e.g., throughput, QoS). These resources must be carefully and systematically adjusted to ensure that performance objectives will be met. However, the size of the configuration space is too large to explore online. Furthermore, the system's workload and performance objectives are dynamic, changing constantly and without warning. Even individual application behavior is diverse. Strictly online or offline techniques do not work due to the complex nature of modern computation.

Therefore, modern systems need to allocate resources dynamically and efficiently. This thesis introduced AUTO, a resource allocation framework for modern computers. AUTO enables previously impractical, but always necessary, resource allocation policies. It proposes and makes use of the inflate/deflate programming model to dynamically adjust application thread counts in response to the allocation policy and/or user objectives.

This thesis developed three techniques of AUTO: AUTO-TC, AUTO-FINITY, and AUTO-GENEOUS. AUTO-TC controls application thread counts in the prescence of corunners, AUTO-FINITY selects application affinities, and AUTO-GENEOUS selects application thread count and core type on asymmetric systems. Each technique was evaluated and shown to meet QoS and/or increase system throughput. Results show that AUTO enables diverse resource allocation policies and efficiently solves the problem of resource allocation on modern CMP computers.

7.1 SUMMARY OF CONTRIBUTIONS

This thesis made the following contributions:

1. **The AUTO framework:** This thesis introduced a framework to dynamically adjust resource allocations: application thread count, core count, and choice of cores. Using automatically generated models, the framework makes predictions about the effects of resource allocations on system performance as measured by an allocation policy. Through the use of the framework, system policies can efficiently evaluate the effects of resource allocations and, consequently, better meet or exceed user performance objectives.
2. **Inflation/deflation-capable applications:** This thesis introduced the inflate/deflate programming model. Inflate/deflate allows applications to dynamically change their thread counts (i.e., in response to a new resource allocation). Existing applications were modified to support inflation/deflation. The modification experience was discussed, in order to better analyze inflate/deflate. Developers were also given guidance to help them understand how to modify their applications and add inflate/deflate support. Adding inflate/deflate support to applications is worthwhile and can be relatively straightforward.
3. **Thread count allocation for one or more co-running applications (AUTO-TC):** This thesis developed and used the AUTO framework to select thread counts for one or more co-running applications. The framework's predictive models considered application scalability (or lack thereof), negative interference, and the importance of the corunner used during model training. A policy, DYCA, was developed to use the models to maximize system throughput subject to meeting QoS. Results demonstrate model accuracy and utility: the policy met QoS 22% more often and had 6% higher throughput than traditional policies.
4. **Application affinity selection (AUTO-FINITY):** This thesis developed and used AUTO to select an application's preferred class of affinity. The lightweight selection process is independent of thread count and works without training on the application ahead of time. It requires a single, short sample of hardware performance counters. AUTO-FINITY was used by a demonstrative policy, MAGNET, to boost application performance through proper affinity selection. In 12 of 15 cases, performance was within 1% of or better than the per-application best fixed

affinities. In two cases, it outperformed the fix assignments. This thesis also introduced a second set of techniques to build models that predict application performance given an affinity. A policy using these models, COMPASS, resulted in application performance being improved by up to 39% as compared to the best application-agnostic policy.

5. **Asymmetric system thread count and core type selection (AUTO-GENEOUS):** Asymmetric chip multiprocessors are becoming increasingly popular due to their energy efficiency. This thesis used AUTO-GENEOUS to select the core count and type that an application's threads should be executed on. To minimize training, AUTO-GENEOUS used piecewise basis models alongside simpler inductee models. The inductee models transform application behavior from one processor type (the basis processor) to another (the inductee). A comprehensive evaluation showed AUTO-GENEOUS's accuracy. When used by a policy, QoST, AUTO-GENEOUS resulted in the system meeting QoS constraints 88% of the time while also minimizing power costs in 53% of cases.

7.2 FUTURE WORK

Future work for this thesis includes:

1. This thesis proposed inflate/deflate applications and guidelines on how to create inflate/deflate-capable applications. Nevertheless, future work could further assist developers with modification of their application to become inflate/deflate-capable. For example, a library to enable inflate/deflate applications would reduce modification difficulty and promote acceptance of inflate/deflate. The library could provide a class that an application extends. The class could provide common inflate/deflate functionality. Developers would override such functionality on an as-needed basis. Because work stealing is the most flexible way to enable inflate/deflate functionality, the base class would accomplish work via work stealing and could itself be an extension of existing work-stealing libraries (e.g., Intel Thread Building Blocks [79]).
2. This thesis also introduced the concept of sketches. Sketches are designed to inform the Runtime Configurator about an application's behavior (e.g., cache usage, scalability). However, this thesis did not formally define sketches. This was because, in part, profiling can determine

most of the relevant behavior. Although the costs of profiling are acceptable, future work could examine how to further reduce profiling costs. A formal definition of sketches, their properties, requirements, and interactions would be a large step towards the reduction or elimination of profiling.

3. This thesis examined application performance on different core types. Digital voltage and frequency scaling (DVFS) can be represented in AUTO as multiple core types (e.g., a core type per frequency available). However, the cost of building a model for each core type (frequency setting) could possibly be reduced by using DVFS-aware models. Identical cores set to different frequencies are more similar than dissimilar (e.g., identical caches, identical functional units). Future work can examine how to convert between one frequency and another with less profiling than current approaches.
4. This thesis treated threads within an application identically. Another extension could to add support for heterogeneous threads within a application. Currently, each thread is treated as behaving identically to every other thread. While this is acceptable from an application correctness viewpoint, additional performance could be achieved with more detailed, per-thread properties. For example, pipeline parallel applications will benefit from stage-aware performance predictions. Heterogeneous thread-aware models must consider more allocation possibilities and interactions. Techniques should be developed to take advantage of the similarities between processors, in order to reduce profiling.
5. This thesis focused on CPU-bound or memory-bound applications. Although these applications are the type that most benefit from this thesis's techniques, additional classes of applications may also benefit, and therefore, could be considered in future work. I/O-bound applications will require models that consider I/O-contention on a per-device basis (e.g., hard drive, network). Additional types of parallel applications could also be considered. For example, client-server processes might interact and be dependent on each other's performance. Some applications might run periodically (e.g., cron jobs). Future work could consider how to predict system behavior and preemptively, instead of reactively, adapt.
6. This thesis examined the effects of contention on the scalability of CPU-bound or memory-bound applications given the thread count of corunner(s). Future work could develop techniques to model interference caused by affinity choices and choice of core type.

8.0 BIBLIOGRAPHY

[1] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate: enabling autonomic applications on the grid. In *Autonomic Computing Workshop*, June 2003.

[2] S. B. Ahmad. On improved processor allocation in 2D mesh-based multicomputers: controlled splitting of parallel requests. In *Proc. of the 2011 Int'l Conf. on Comm., Computing & Sec.*, ICCCS '11. ACM, 2011.

[3] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu. An Opportunistic Prediction-based Thread Scheduling to Maximize Throughput/Watt in AMPs. In *Int'l. Conf. on Parallel Architecture and Compilation Techniques*, 2013.

[4] Architecture Review Board. Openmp application program interface v3.0. http://www.openmp.org/mp-documents/spec30.pdf.

[5] ARM. Advances in big.little technology for power and energy savings. http://www.arm.com/files/pdf/Advances_in_big.LITTLE_Technology_for_Power_and_Energy_Savings.pdf.

[6] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[7] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Computing frontiers*, 2006.

[8] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[9] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 290–301, New York, NY, USA, 2009. ACM.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[11] C. Bienia and K. Li. Fidelity and scaling of the parsec benchmark inputs. In *Workload Characterization (IISWC), 2010 IEEE Int'l Symp. on*, dec. 2010.

[12] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *Proc of the 2011 USENIX Conf on USENIX Annual Tech Conf*, USENIXATC'11, Berkeley, CA, USA. USENIX Assoc.

[13] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 240–249, New York, NY, USA, 2008. ACM.

[14] S. Chandra, X. Li, T. Saif, and M. Parashar. Enabling scalable parallel implementations of structured adaptive mesh refinement applications. *J. Supercomput.*, 39, February 2007.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. OOPSLA '05. ACM, 2005.

[16] G. Chen and M. Kandemir. Optimizing embedded applications using programmer-inserted hints. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 157–160, New York, NY, USA, 2005. ACM.

[17] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Design Automation Conf.*, 2009.

[18] J. Chen, L. K. John, and D. Kaseridis. Modeling program resource demand using inherent program characteristics. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS, pages 1–12, New York, NY, USA, 2011. ACM.

[19] H.-D. Cho, K. Chung, and T. Kim. Benefits of the big.LITTLE Architecture. White Paper, ARM/Samsung, February 2012.

[20] S. Cho and S. Demetriades. Maestro: Orchestrating predictive resource management in future multicore systems. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 1–8, 2011.

[21] J. Cleary, O. Callanan, M. Purcell, and D. Gregg. Fast asymmetric thread synchronization. *ACM Trans. Archit. Code Optim.*, 9(4):27:1–27:22, Jan. 2013.

[22] J. Cong and B. Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Int'l. Symp. on Low power Electronics and Design*, 2012.

[23] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *Micro, IEEE*, 30(2), march-april 2010.

[24] Y. Cui, Y. Wang, Y. Chen, and Y. Shi. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. *ACM Trans. Archit. Code Optim.*, 9(4):44:1–44:25, Jan. 2013.

[25] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and

M. Roth. Traffic management: a holistic approach to memory placement on numa systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[26] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. pages 76–86, April 2011.

[27] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. Bws: balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 365–378, New York, NY, USA, 2012. ACM.

[28] S. Diwan and D. Gannon. Adaptive resource utilization and remote access capabilities in high-performance distributed systems: The open hpc++ approach. *Cluster Computing*, 3:1–14, January 2000.

[29] A. Dorta, C. Rodriguez, and F. de Sande. The OpenMP source code repository. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conf on*, Feb.

[30] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 389–400, Washington, DC, USA, 2012. IEEE Computer Society.

[31] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, and F. Zambonelli. A roadmap towards sustainable self-aware service systems. In *SEAMS '10*. ACM.

[32] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 83:1–83:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[33] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 362–373, New York, NY, USA, 2011. ACM.

[34] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Int'l. Symp. on Computer architecture*, 2011.

[35] Y. S. et al. 28nm high-k metal-gate heterogeneous quad-core CPUs for high-performance and energy-efficient mobile application processor. In *IEEE Int'l. Solid-State Circuits Conf.*, 2013.

[36] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro*, Mar. 2011.

[37] P. Greenhalgh. Big.LITTLE Processing with ARM CortexTM-A15 and Cortex-A7. White Paper, ARM, September 2011.

[38] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[39] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, Nov.

[40] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[41] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz. Juggle: proactive load balancing on multicore computers. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 3–14, New York, NY, USA, 2011. ACM.

[42] T. Instruments. OMAP 5 mobile applications platform. http://focus.ti.com/pdfs/wtbu/OMAP5_2011-7-13.pdf, July 2011.

[43] E. Ipek, B. de Supinski, M. Schulz, and S. McKee. An approach to performance prediction for parallel applications Euro-Par 2005 parallel processing. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, chapter 24, pages 627–628. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[44] M. Itzkowitz and Y. Maruyama. HPC profiling with the sun studio performance tools. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, chapter 6, pages 67–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[45] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 223–234, New York, NY, USA, 2012. ACM.

[46] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric cmps. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 154–165, New York, NY, USA, 2013. ACM.

[47] M. Ju, H. Jung, and H. Che. A performance analysis methodology for multi-core, multithreaded processors. *Computers, IEEE Transactions on*, PP(99):1–1, 2012.

[48] M. Kandemir, S. P. Muralidhara, S. H. K. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. In *Proceedings of the 42nd Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pages 505–516, New York, NY, USA, 2009. ACM.

[49] O. Khan and S. Kundu. A self-adaptive scheduler for asymmetric multi-cores. In *Great Lakes Symp. on VLSI*, GLSVLSI '10, 2010.

[50] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. autopin automated optimization of thread-to-core pinning on multicore systems. In P. Stenstrm, editor, *Trans on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Comp Sci*. Springer Berlin / Heidelberg, 2011.

[51] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *Int'l. symposium on Microarchitecture*, 2003.

[52] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Int'l. Symp. on Computer Architecture*, 2004.

[53] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, Jan. 2013.

[54] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 25:1–25:12, New York, NY, USA, 2009. ACM.

[55] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[56] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th annual*

international symposium on Computer architecture, ISCA ’10, pages 270–279, New York, NY, USA, 2010. ACM.

[57] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC ’09, pages 522–527, New York, NY, USA, 2009. ACM.

[58] L. F. Leung, C. Y. Tsui, and W. H. Ki. Minimizing energy consumption of multiple-processors-core systems with simultaneous task allocation, scheduling and voltage assignment. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC ’04, pages 647–652, Piscataway, NJ, USA, 2004. IEEE Press.

[59] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, 2009.

[60] LuxRender Team. Luxrender v0.8. http://www.luxrender.net, 2012.

[61] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, ISMM ’11, pages 11–20, New York, NY, USA, 2011. ACM.

[62] Z. Majo and T. R. Gross. Matching memory access patterns and data placement for numa systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO ’12, pages 230–241, New York, NY, USA, 2012. ACM.

[63] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 ’11, pages 248–259, New York, NY, USA, 2011. ACM.

[64] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Perfor-*

mance and Embedded Architectures and Compilers, HiPEAC '11, pages 167–176, New York, NY, USA, 2011. ACM.

[65] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, Sept. 2009.

[66] M. C. Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA, 2006. ACM.

[67] D. S. McFarlin, C. Tucker, and C. Zilles. Discerning the dominant out-of-order performance advantage: is it speculation or dynamism? In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 241–252, New York, NY, USA, 2013. ACM.

[68] MediaTek. MediaTek Enables ARM big.LITTLE Heterogeneous Multi-Processing Technology in Mobile SoCs. Technical report, 2013.

[69] MediaTek. MediaTek True Octa-Core. Technical report, 2013.

[70] Message Passing Interface Forum. Mpi: A message-passing interface standard version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[71] R. W. Moore and B. R. Childers. Inflation and deflation of self-adaptive applications. In *Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, New York, NY, USA, 2011.

[72] T. Moseley, D. Grunwald, J. L. Kihm, and D. A. Connors. Methods for modeling resource contention on simultaneous multithreading processors. *Computer Design, International Conference on*, 0:373–380, 2005.

[73] NAS Parallel Benchmarks Team. NAS parallel benchmarks 3.3.1, 2009.

[74] K. J. Nesbit. *Virtual private machines: a resource abstraction for multicore computer systems*. PhD thesis, Madison, WI, USA, 2009.

[75] D. Nikolopoulos, G. Back, J. Tripathi, and M. Curtis-Maury. Vt-asos: Holistic system software customization for many cores. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE Int'l Symp. on*, 2008.

[76] J. Parekh, G. Kaiser, P. Gross, and G. Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9, 2006.

[77] V. Petrucci, O. Loques, D. Mosse, R. Melhem, N. Gazala, and S. Gobriel. Thread assignment optimization with real-time performance and memory bandwidth guarantees for energy-efficient heterogeneous multi-core systems. In *Real-Time and Embedded Technology and Applications Symp.*, 2012.

[78] S. Phadke and S. Narayanasamy. Mlp aware heterogeneous memory system. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, 2011.

[79] C. Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23, April 2008.

[80] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM.

[81] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. Power-performance modeling on asymmetric multi-cores. In *Int'l. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2013.

[82] K. Pusukuri, R. Gupta, and L. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Workload Characterization (IISWC), 2011 IEEE Int'l Symp. on*, nov. 2011.

[83] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread tranquilizer: Dynamically reducing performance variation. *ACM Trans. Archit. Code Optim.*, 8(4):46:1–46:21, Jan. 2012.

[84] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread to strand binding of parallel network applications in massive multi-threaded systems. In *Proc of the 15th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, PPoPP '10. ACM.

[85] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 107–116, New York, NY, USA, 2011. ACM.

[86] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *SEAMS '10*. ACM, 2010.

[87] Renesas. R-Car H2 Product Specifications. Product cut sheet, `http://www.renesas.com/press/news/2013/news20130325_s.jsp`, Renesas, March 2013.

[88] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *European conference on Computer systems*, 2010.

[89] H. Sasaki, Y. Ikeda, M. Kondo, and H. Nakamura. An intra-task dvfs technique based on statistical analysis of hardware events. In *Proceedings of the 4th international conference on Computing frontiers*, CF '07, pages 123–130, New York, NY, USA, 2007. ACM.

[90] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 107–116, New York, NY, USA, 2012. ACM.

[91] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *Oper. Syst. Rev.*, 43, April 2009.

[92] Y. Shin, K. Shin, P. Kenkare, R. Kashyap, H.-J. Lee, D. Seo, B. Millar, Y. Kwon, R. Iyengar, M. su Kim, A. Chowdhury, S.-I. Bae, I. Hong, W. Jeong, A. Lindner, U. Cho, K. Hawkins, J. C. Son, and S. H. Hwang. 28nm high- metal-gate heterogeneous quad-core cpus for high-performance and energy-efficient mobile application processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 154–155, 2013.

[93] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proc. of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12. ACM, 2012.

[94] Sniper multi-core simulator. Available from web site, http://snipersim.org/w/index.php?title=The_Sniper_Multi-Core_Simulator&oldid=527, 2013.

[95] F. Song, S. Moore, and J. Dongarra. Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, 2009.

[96] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer. Efficient interaction between OS and architecture in heterogeneous platforms. *Oper. Syst. Rev.*, Feb. 2011.

[97] Y. Su, D. Ye, and J. Xue. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In *Int'l Conference on High Performance Computing*, 2013.

[98] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, July 2005.

[99] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international*

conference on Architectural support for programming languages and operating systems, ASPLOS '09, pages 253–264, New York, NY, USA, 2009. ACM.

[100] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proc of the 2nd ACM SIGOPS/EuroSys European Conf on Comp Systems 2007*, EuroSys '07.

[101] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for qos in warehouse scale computers. In *Int'l. Symp. on Code Generation and Optimization*, 2012.

[102] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 283–294, 2011.

[103] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein. Data and thread affinity in openmp programs. In *Proc of the 2008 workshop on Memory access on future processors: a solved problem?*, MAW '08. ACM.

[104] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *Proc of the ACM Int'l Conf on Object oriented programming systems languages and applications*, OOPSLA '10. ACM, 2010.

[105] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[106] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE). In *Int'l. Symp. on Computer Architecture*, 2012.

[107] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the UltraSPARC t2 processor. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 481–492, New York, NY, USA, 2009. ACM.

[108] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *Architectural support for Prog. Lang. and Operating Syst.*, 2010.

[109] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. Qscores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Int'l. Symp. on Microarchitecture*, 2011.

[110] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and practice of parallel programming*, PPoPP '09, pages 75–84, New York, NY, USA, 2009. ACM.

[111] Z. Wang, M. F. P. O'Boyle, and M. K. Emani. Smart, adaptive mapping of parallelism in the presence of external workload. In *Int'l. Symp. on Code Generation and Optimization*, 2013.

[112] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[113] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 2–11, 2011.

[114] X. Wu and V. Taylor. Performance modeling of hybrid mpi/openmp scientific applications on large-scale multicore supercomputers. *Journal of Computer and System Sciences*, 79(8):1256 – 1268, 2013.

[115] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache conscious task regrouping on multicore processors. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 603–611, 2012.

[116] X. Xiang, C. Ding, H. Luo, and B. Bao. Hotl: a higher order theory of locality. In *Proceedings of the eighteenth international conference on Architectural support for programming*

languages and operating systems, ASPLOS '13, pages 343–356, New York, NY, USA, 2013. ACM.

[117] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36, May 2008.

[118] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Proc of the 15th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, PPoPP '10. ACM.

[119] J. Zhao, H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2013.

[120] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, volume 45 of *ASPLOS '10*, pages 129–142, New York, NY, USA, Mar. 2010. ACM.