

**A Unified Approach to Adaptive Code Selection
for Modern Systems**

by

Joseph Blomstedt

B.S., University of Washington, 2005

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

2008

This thesis entitled:
A Unified Approach to Adaptive Code Selection for Modern Systems
written by Joseph Blomstedt
has been approved for the Department of Electrical and Computer Engineering

Professor Daniel A. Connors

Professor Manish Vachharajani

Professor Jeremy Sick

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Blomstedt, Joseph (M.S., Computer Engineering)

A Unified Approach to Adaptive Code Selection for Modern Systems

Thesis directed by Professor Daniel A. Connors

Code compilation and deployment is a much more complex affair today than it was in years past. Historically, the primary metric of compiled code was raw performance, and an individual simply compiled his code using the best available compiler and optimization flags to get the best resulting binary. Applications that required the utmost performance would take the step even further and utilize advanced techniques such as employing profile-guided optimizations. In recent years, however, additional metrics have arisen alongside performance, such as power consumption, temperature dissipation, and overall system throughput. To further complicate the matter, the trend in modern microprocessor design is increasingly focused on multi-threaded architectures; and, by virtue of many engineering decisions all existing and proposed multi-threaded designs include some number of hardware subsystems that are shared between the various contexts of execution. Contention over these shared resources provides an additional dimension to evaluate. Simply compiling a given application and evaluating its performance in isolation is no longer the optimal approach, since characteristics of the eventual dynamic co-scheduled workload will invariably have unevaluated effects.

To address this issue, this thesis presents ARCA – a compilation toolchain designed to generate numerous versions of a piece of code, profile and classify the various versions, and subsequently select between the different code versions at runtime based on dynamic workload characteristics. The primary contributions of the work are a version analysis and classification approach which allows a large set of candidate versions to be culled down to a useful subset, a set of three dynamic adaptation algorithms that are employed to guide the runtime selection between the available code versions, and

the demonstration of a unified framework capable of addressing a number of emerging issues.

In addition to these contributions, this work showcases the ARCA framework with two case studies: one focusing on performance improvement on multi-threaded systems, and one focusing on compromising temperature dissipation and performance. Additionally, this work presents future research opportunities that the techniques presented in this thesis are well suited to address.

Dedication

To wife, Katherine, and my childhood role model, Shigeru Miyamoto.

Acknowledgements

First and foremost, I would like to thank Professor Daniel Connors, my advisor, for his insight and support throughout this thesis as well as throughout my graduate career. Professor Connors' ability to keep me motivated and on task was a significant component leading to the completion of this thesis.

Second, I would like to thank Intel for providing the funding that allowed me to engage in this research full-time without additional distractions as well as employing me during the summer on a number of occasions. The contacts I have made at Intel such as Ramesh Peri and Youfeng Wu strengthen my research by providing insight into the real-world challenges often overlooked by academic work.

Lastly, I would like to thank my family. Particularly my wife, Katherine, who has endured my long hours at the lab and countless seven-day work weeks. Her support, sense of humor, and companionship are the foundations of my sanity. Also, my mother and brother who have suffered at the hand of my academic endeavors, and have seen little of me the past three years. I am forever grateful that all three of you understand that there is a light at the end of the tunnel, and that graduate school is a passing, albeit seemingly never ending, phase.

Contents

Chapter		
1	Introduction	1
2	Background and Motivation	5
	2.1 Multi-Threaded Hardware	5
	2.1.1 Temporal Multithreading	5
	2.1.2 Simultaneous Multithreading	6
	2.1.3 Chip-level Multiprocessing	7
	2.1.4 Examples of Real Hardware	7
	2.2 Motivation: Code Generation and Susceptibility	8
	2.3 Background and Related Work	10
3	The ARCA Framework	15
	3.1 An Overview of the ARCA Framework	15
	3.1.1 Code Generation and Analysis	15
	3.1.2 Dynamic Adaptation	18
	3.2 Version Analysis and Static Selection Methodology	20
	3.3 Dynamic Selection Techniques	22
	3.3.1 Greedy Selection	23
	3.3.2 Cooperative Selection	26
	3.3.3 Fractional Selection	29

3.4	System Overhead	31
4	Demonstration: Code Versioning and Adaptive Selection	33
4.1	Workload Description and Analysis	33
4.2	Case Study: Improving System Performance	34
4.2.1	Version Analysis and Available Opportunity	35
4.2.2	Experimental Results	39
4.2.3	Analysis	41
4.3	Case Study: Lowering CPU Temperature	42
4.3.1	Version Analysis and Available Opportunity	43
4.3.2	Experimental Results	44
4.3.3	Analysis	45
5	Future Work	47
5.1	System Scalability	47
5.2	High-Level Versioning and Selection	49
5.3	Emerging Research Areas	52
5.3.1	Complex Multi-threaded Systems	53
5.3.2	Runtime Adaption for Fault Tolerance	54
5.3.3	Heterogeneous, Distributed, and Virtualized Systems	54
6	Conclusion	56
	Bibliography	58

Tables

Table

3.1	The evaluated compiler and optimization flags.	16
4.1	The benchmarks evaluated.	33
4.2	The code versions selected when targeting performance adaptation. . . .	37
4.3	The average cycle count for the fastest code versions.	38
4.4	The estimated performance gains possible through adaptation.	39
4.5	The average observed core temperature at different adaptation ratios. .	45

Figures

Figure

2.1	An example of co-thread interference (400.perlbench).	9
3.1	The ARCA static compilation phase.	16
3.2	The ARCA run-time selection phase.	19
3.3	The code version execution and L2 access behavior for 400.perlbench.	21
3.4	The identified execution and L2 access behavior clusters for 400.perlbench.	22
3.5	The greedy selection algorithm	24
3.6	A synthetic demonstration of greedy selection.	25
3.7	The cooperative selection algorithm	27
3.8	A synthetic demonstration of cooperative selection.	28
3.9	The fractional selection algorithm	29
3.10	A synthetic demonstration of fractional selection.	30
3.11	The observed PMU sampling overhead.	32
3.12	The observed ARCA selection overhead.	32
4.1	The observed function invocations over time for each benchmark.	35
4.2	The impact of performance-focused adaptation (Greedy).	40
4.3	The impact of performance-focused adaptation (Cooperative).	40
4.4	The reduction in L2 misses from performance-focused adaptation.	41
4.5	The code version temperature and execution behavior for 433.milc.	44

4.6	The impact of temperature-focused adaptation (Fractional).	45
5.1	Demonstration of multi-application adaptation on a quad-core machine.	48
5.2	The impact of adaptation on an ATLAS DGEMM timing benchmark. .	52

Chapter 1

Introduction

The code compilation and deployment process is constantly adapting to changes in machine, programming language, and operating systems design and implementation. Historically, performance was the guiding metric used to judge compiled code and a developer simply generated code using the best available compiler and optimization flags. Furthermore, applications that required the utmost performance would make use of advanced techniques such as profile-driven optimizations [8, 11], iterative compilation [34, 4], and install-time code generation. In recent years, however, additional metrics – such as power consumption, temperature dissipation, and overall system throughput – have emerged to influence code generation decisions alongside the traditional performance considerations.

The recent trend in modern microprocessor design to increasingly focus on multi-threaded and multi-core architectures equally complicates the code generation issue. By virtue of engineering decisions (cost, performance, power), nearly all current and future architecture designs include some number of hardware resources that are shared amongst multiple execution contexts. For example, the Intel Core 2 architecture utilizes a shared L2 cache between processor cores, while processors enabled with Simultaneous Multi-Threading (SMT) share significant portions of the micro-architecture between executing threads. This contention over shared resources provides an additional dimension that must be considered. Specifically, there are multiple code generation decisions

that impact both the individual performance of an application as well as influence the system as a whole. Simply compiling a given application and evaluating its performance in isolation is no longer the optimal approach, since characteristics of the dynamically co-scheduled workload can invariably have unforeseen effects.

Traditional compiler optimizations primarily extract performance on modern processors through two means: exposing sufficient amounts of instruction parallelism to the hardware, and exploiting the cache hierarchy as much as possible in order to reduce costly main memory accesses. These aggressive optimizations, however, are often driven by the assumption that the underlying hardware resources are both dedicated to the target application as well as unchanging. On modern multi-threaded architectures, this assumption is often untrue. The effective size of a shared L2 cache available to a given application is typically smaller than the size the compiler optimized for, as well as a dynamic value that changes based on the cache usage of the co-executing applications. Without the ability to adapt to the other running threads on a multi-context system, an aggressively compiled application may unknowingly damage both its own performance as well as reduce the throughput of the entire system.

For example, there may be two or more different compilations for a given application that provide the best performance under different circumstances; such as, when running alone on a system versus running alongside an application that aggressively utilizes a shared L2 cache. In such a case, there is no single static compilation that will provide the best performance under all circumstances – the performance is inherently workload dependent.

To address this issue, this thesis presents ARCA – an adaptive compilation framework designed to generate numerous versions of code, profile and classify the various versions, and subsequently select between the different code versions at runtime based on dynamic workload characteristics. In particular, the run-time selection decisions are driven by observable hardware statistics such as performance counters and processor

temperature sensor readouts. Overall, the specific contributions of this thesis are as follows:

- (1) A code version analysis and selection methodology based upon profile-guided feedback and data clustering techniques. This methodology allows a large set of code versions to be distilled down to a subset of versions demonstrating interesting behaviors related to a desired improvement goal.
- (2) A set of three proposed dynamic adaptation algorithms that are applicable to different goals and situations, each with different strengths and weaknesses.
- (3) A demonstration of a unified and extensible system that can be utilized to address a wide range of emerging issues, accompanied by a discussion detailing the key features needed to design such a framework.

The rest of this thesis is organized as follows. Chapter 2 provides background on the types of existing multi-threaded systems as well as related work in this area – particularly, issues dealing with workload selection at both the hardware and software level; this chapter also includes a motivating example that demonstrates the case for dynamic code selection. Chapter 3 details the ARCA adaptive compilation system, presenting both the initial offline code generation and analysis phase as well as the subsequent dynamic adaptation phase. Furthermore, a set of synthetic benchmarks are used throughout to demonstrate the different adaptation techniques investigated. The chapter also provides an analysis discussing the overhead of the system. Chapter 4 uses a subset of the SPEC CPU2006 benchmark suite to demonstrate the ARCA framework from beginning to end. The chapter starts with an initial overview of the benchmarks evaluated, followed by a demonstration of two example applications of the ARCA framework: an attempt to improve overall system throughput and an attempt to reduce overall CPU temperature. Chapter 5 discusses possible avenues for future research built upon the contributions of this thesis, showcasing examples of emerging

issues that are well suited to code versioning and selection. Finally, Chapter 6 provides a summary of this work and concludes the thesis.

Chapter 2

Background and Motivation

This chapter provides a survey of existing multi-threaded hardware designs, including a discussion of existing microprocessors. The chapter then presents a motivational example demonstrating the impact that multi-threaded workloads have on code compilation decisions. Finally, previous work related to multi-threaded contention as well as dynamic selection and optimization is presented.

2.1 Multi-Threaded Hardware

The impact of co-executing threads upon each other is largely determined by the underlying architecture design. In particular, the primary factors are the way in which multi-threading is implemented as well as which hardware resources are shared amongst threads and to what extent this sharing occurs. This section discusses the various techniques used to implement multi-threaded hardware, and presents examples of real-world processors that utilize these techniques.

2.1.1 Temporal Multithreading

Temporal multithreading, also known as interleaved multithreading, is a design that allows a processor to execute instructions from different threads on a cycle-by-cycle basis. At present, all existing implementations that use temporal multithreading utilize in-order execution and are generally single-issue or VLIW style processors. The two

primary types of temporal multithreading, fine-grain and coarse-grain, are discussed in further detail below.

Fine-grain Multithreading. A design based on fine-grain multithreading switches between a set of threads each clock cycle. For example, if two threads were scheduled for execution, then every even clock cycle would be executing instructions from the first thread and every odd clock cycle from the second thread. This example assumes that there are instructions available from each thread and that neither of the two threads are designated to be skipped – a designation employed by some fine-grain multithreading designs in order to skip threads that are stalled waiting on long-latency events such as cache misses.

Coarse-grain Multithreading. Coarse-grain multithreading, on the other hand, continues to execute instructions from a single thread until a long latency event occurs, such as a miss in the L3 cache or a page fault. When the event occurs, the processor switches to another thread and remains executing that thread until another long latency event occurs.

2.1.2 Simultaneous Multithreading

Simultaneous Multithreading (SMT) is a multithreading technique built as an extension to superscalar processors. The technique is predominately employed to better utilize existing superscalar resources. For example, consider a superscalar processor that includes the ability to issue and execute four instructions each clock cycle. Without SMT, the processor will only be able to issue the full four instructions if the executing code has a significant amount of instruction-level parallelism (ILP). The addition of SMT to the design allows the processor to simultaneously issue instructions from multiple threads, thereby fully utilizing its 4-wide capabilities more often.

2.1.3 Chip-level Multiprocessing

Due to the ever-decreasing size of transistors, microprocessor manufactures are now capable of placing multiple processor cores within a single physical CPU package, many times placing multiple cores on the same die. This has lead to the proliferation of multi-core processors that support chip-level multiprocessing (CMP). At the basic level, a CMP design employs complete duplication between the various processor cores – with fully independent contexts, execution units, issue logic, and so forth. However, the most common designs always share some portion of the memory subsystem between, at the very least, a subset of the available cores.

2.1.4 Examples of Real Hardware

All existing and upcoming multi-threaded processors utilize one of the above techniques, with several designs using multiple techniques in the same processor. Examples of real-world multi-threaded architectures are:

- (1) **Intel Core 2 architecture.** The Core 2 architecture [43] is a CMP design that has been produced in both dual-core and quad-core iterations. The architecture is fundamentally a dual-core design, with two processor cores fabricated onto a single die along with a shared L2 cache. In a quad-core version, there are two sets of dual-cores. Each set contains a separate shared L2 cache, with communication between the two sets occurring over the processor's front-side bus.
- (2) **Sun UltraSparc T1.** The Sun UltraSparc T1 [24] architecture is an example of a system employing fine-grained multithreading. The T1's processor core is a single-issue, in-order design that executes exactly one instruction per cycle. After each cycle, the core switches to another thread which will supply the next instruction. The core is capable of maintaining and switching between

the contexts of up to four individual threads. As an overall system, however, the UltraSparc T1 is also a CMP, including eight individual cores in a single processor. The end result is a design that is capable of executing an instruction from up to eight threads in a given cycle while also interleaving instructions from up to 32 threads overall. Each core also provides an L2 cache which is shared over a crossbar switch with each of the other eight cores.

- (3) **Intel Dual-Core Itanium.** The Montecito edition of the Intel Itanium micro-processor [31] sold under the brand name Dual-Core Itanium utilizes a dual-core CMP design along with 2-way coarse-grain multithreading for each core. The processor as a whole therefore supports four threads with any two running simultaneously, one per each core. Unlike the other hardware examples presented above, the Dual-Core Itanium does not share any amount of the cache hierarchy between the two cores. Each core has its own L1, L2, and L3 cache. On the other hand, the caches are shared between multiple threads running on the same core.

2.2 Motivation: Code Generation and Susceptibility

One of the primary beliefs behind the need for a system such as ARCA is that different ways of generating code will produce code that exhibits different characteristics depending on the co-executing workload on a multi-threaded system. For example, it may be that the compilation which provides the fastest execution when running alone on a system turns out to be one of the slower versions when running as part of a workload consisting of other aggressively compiled applications.

To demonstrate this fact, the SPEC CPU 2006 benchmark 400.perlbench was compiled three times using three different compilers: gcc 3.4, gcc 4.1, and icc 10. In all three cases, the `-O3` optimization flag was used, along with flags that generate code

targeted at newer Intel microprocessors. Figure 2.1(a) shows the performance of each of the three compilations when running alone on an Intel Core 2 Duo E6600 microprocessor. The values in the figure are normalized so that the fastest version shows an execution time of 1. As previously discussed above, the Core 2 Duo processor is a dual-core design that shares the L2 cache between the two processor cores. As such, Figure 2.1(b) again shows the performance of the three compilations running on the same system; however, for these runs the applications are paired with a synthetic benchmark which continuously trashes the L2 cache. Given the shared L2 cache, this workload showcases one of the worst possible pairings. As shown in the two figures, the compilation which performs the best when running alone – the gcc 4.1 version – is not the fastest version when running alongside the synthetic benchmark. In fact, the gcc 4.1 version is a little over 5% slower than the icc version, the fastest compilation during the paired run.

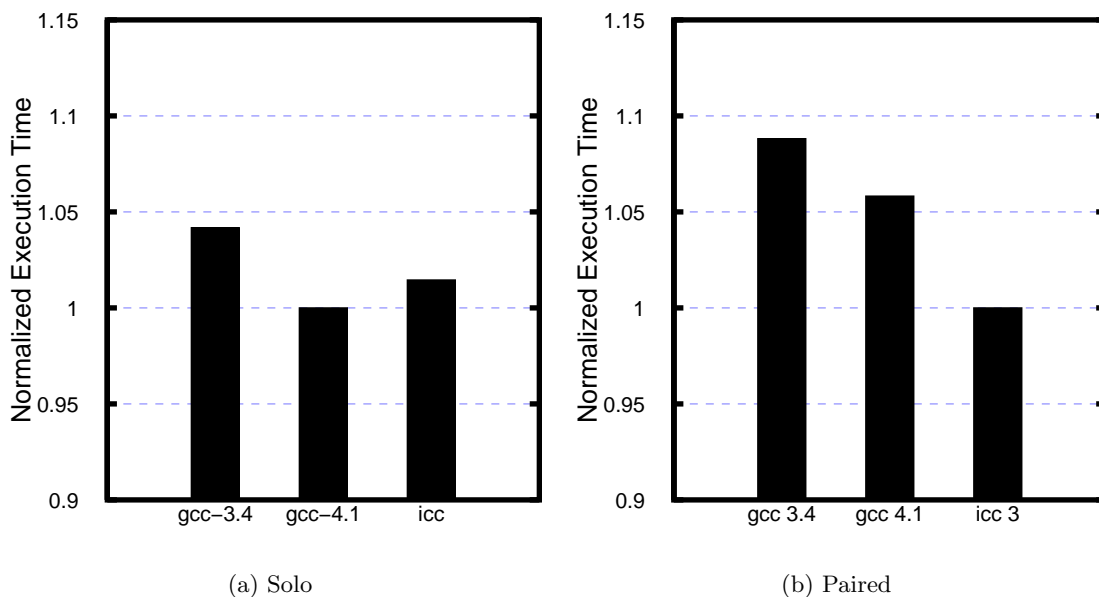


Figure 2.1: An example of co-thread interference (400.perlbench).

These results demonstrate that there is no single compilation that provides the best performance all the time. In particular, the performance of a given code version

running on a multi-threaded system depends largely on the characteristics of the other executing applications running simultaneously. While the workload shown consists of a pathological case – a synthetic benchmark which is designed to trash the L2 cache – the findings are themselves not completely pathological. Certain real-world workloads could very well exhibit similar behavior. Examples include streaming applications that continuously move large sets of data through the cache, as well as aggressively compiled code that exploits cache blocking and loop tiling in a way designed to completely fill the entire L2 cache.

Given a mixed workload that sometimes exhibits pathological behavior and many times does not, determining which compilation of `400.perlbench` should be deployed is difficult. The gcc 4.1 compiled code clearly performs the best when not paired with an application that aggressively uses the L2 cache, but subsequently performs poorly when paired with such an application. The best paired version, on the other hand, performs terribly when running alone. In a traditional environment, one would have to pick the version based upon the predicted ratio of good versus bad co-scheduled workloads. If it was more common to encounter the case with significant L2 cache contention, then the icc version would be the preferred choice; if such was less common, then the gcc 4.1 version is best. However, neither choice is optimal all the time. The aim of the ARCA framework is therefore to address issues such as these. By generating a hybrid binary that contains code from both the icc and gcc 4.1 compilations, and intelligently selecting between these versions at runtime, the system attempts to gain the best of both worlds.

2.3 Background and Related Work

Analyzing and addressing the problem of co-thread interaction is not a topic unique to this thesis. Moreover, portions of the ARCA framework are similar to a number of previous systems as well as utilize preexisting technologies such as hard-

ware performance counters. This section discusses both previous work in the area of contention in multi-threaded environments and provides background information on frameworks similar to ARCA as well as the technologies upon which ARCA is built.

Performance Monitoring Unit (PMU). As previously mentioned and discussed in further detail in the next section, ARCA dynamically selects between multiple versions of precompiled code based upon dynamic system characteristics at runtime. To gather this dynamic information, ARCA utilizes the performance monitoring unit, also known as hardware event counters, built into modern microprocessors. PMUs were originally designed as a debug mechanism built into processors in order to allow vendors to verify that the silicon was working as intended, as well as to provide detailed feedback that would allow vendors to further tune their designs in order to provide better performance. Eventually, the interface to these units was released by the various processors vendors to the public and published within a variety of software manuals [40, 21, 20].

While additional features have been added to the newest PMUs, the standard feature of a PMU is to provide a set of counters that can be configured to count a variety of hardware events as they occur. The types of events counted by these units are things such as the number of instructions retired, the number of cache accesses/misses for each level of the cache hierarchy, the number of integer or floating point instructions issued, the number of cycles stalled waiting on memory, and so on. These days, the number and types of events that can be counted include almost every statistic one would be interested in evaluating at runtime. As a reference point, the PMU included in first edition of the Intel Core 2 Duo microprocessor has 598 publicly accessible hardware events that can be counted. The standard interface that most PMUs provide is designed as a threshold based interrupt. Software initializes the counters designating which events to count as well as providing the threshold value, and then the counters continuously increment as events occur and generate an interrupt whenever the count reaches the threshold value. This interrupt approach allows PMU based profiling to in-

cur very little overhead. Furthermore, when the interrupt ultimately occurs the relative location of the instruction which generated the final event that surpassed the threshold is known; therefore giving the profile system a reasonable idea as to what region of code is responsible for triggering the events.

Shared Resource Contention. A substantial amount of previous work on shared resource contention dates back to early work on simultaneous multithreading (SMT) architectures. At the hardware level, Tullsen et al [41] explored instruction fetch policies designed to enable forward progress of both threads as well limit inter-thread interference. On the topic of code generation, Lo evaluated [28] the effects of various compiler optimizations on SMT performance and provided advice on how to best tune optimizations for code aimed at running on such architectures. This work is very similar to the goal of ARCA, except that ARCA is designed to work across a variety of architectures, not just SMT. Also, Lo’s work provided advice that resulted in code that was sub-optimal when running on a machine without any other co-executing threads. The work also focused on specific optimizations and how they should be adapted for use on an SMT system, whereas ARCA is based upon classifying code post-generation and aiming at selecting the right code given the circumstances.

Operating System Scheduling. There have also been a variety of papers published on OS scheduling for SMT architectures [38, 23, 32]. These works focused on observing and modeling interference between threads on SMT systems through the use of PMU data. The work was evaluated through the addition of a new scheduler to the Linux kernel that monitored PMU behavior and adjusted scheduling decisions accordingly.

Interference Modeling. In addition to attempting to deal with interference on multi-threaded architectures, there has been work published that aims to model the contention that might occur given a certain workload. One of the earlier works addressed modeling SMT resource contention [33] which was the basis for the one of

the above mentioned OS scheduling papers [32]. This work evaluated models based on linear regression as well as recursive partitioning decision trees, and demonstrated that for an SMT processor fairly simple models are capable of accurate interference predictions. Furthermore, the PhD thesis of Kihm from the University of Colorado [22] includes a detailed analysis of co-phase contention on both SMT and CMP-based Intel architectures. The focus of such work is to utilize neural network based modeling in the simulation of multi-threaded architectures, allowing only a subset of test cases to be fully simulated and the behavior of the remainder of the set to be predicted. This work was conducted on a Intel Pentium 4 based SMT system, as well as an Intel Core 2 Duo CMP system that is similar to the hardware used later in this thesis.

Dynamic Optimization Systems. The two main techniques used in past software-based dynamic adaptation research are dynamic optimization and dynamic selection. Examples of past dynamic optimization systems are Dynamo [2], DynamoRIO [7], Mojo [10], DELI [14], ADORE [29], rePLay [36], and Trident [47]. Additionally, several dynamic code translation frameworks also employ dynamic optimization techniques such as Intel’s IA-32 EL [45], Digital’s FX!32 [12], and Transmeta’s Code Morphing Software [13]. Dynamic optimization systems are capable of rewriting a binary at runtime and use that feature to intelligently modify the executing code to perform better based upon observed dynamic characteristics. For example, the ADORE framework monitors PMU events to determine regions that have large cache misses and then dynamically inserts software prefetching instructions into the code to improve data availability and reduce the number of misses. More sophisticated systems such as Dynamo and DynamoRIO employ a variety of optimization techniques, ranging from code inlining and dynamic loop unrolling to register reallocation and instruction strength reduction. The overall success of dynamic optimization has varied wildly from system to system. In general, the amount of overhead required to perform the online profiling and then subsequently implement the triggered optimizations is quite large. Therefore, to gain

a measurable performance improvement the implemented optimizations must improve performance enough to overcome both this additional overhead as well as perform better than the original, native code. Nevertheless, dynamic optimization is a powerful, fine-grained technique and in the cases where it works, it works extremely well.

Dynamic Selection Systems. A dynamic selection system selects between a set of precompiled code versions. The granularity of selection varies from system to system, although the most popular approaches are performing selection at function, dynamic trace, or loop boundaries. Examples of past dynamic selection systems include Dynamic Feedback [15] and ADAPT [42], as well as the work carried out by Fursin [18] and Lau [25]. The main strength of a dynamic selection system over a dynamic optimization system is that the candidate code versions are already precompiled. This allows the different versions to have a significant difference in composition as well as allows the versions to utilize several aggressive optimizations. On the other hand, since the code generation is handled online in a dynamic optimization system, the amount of code that can be altered as well as the optimizations that can be employed are limited in order to reduce the overhead of the approach. Another benefit of the dynamic selection approach is that the precompiled code versions can be profiled and categorized offline in order to better understand their overall behaviors, strengths, and weaknesses. This factor is the primary reason ARCA utilizes dynamic selection, as discussed in the next chapter. The fundamental weakness of dynamic selection systems is that the set of available optimizations is fixed and handled at a coarse-grain level. The approach does not allow for fine-grain tuning of small regions of code, nor subtle adaptation based upon a minor variations in observed dynamic events.

Chapter 3

The ARCA Framework

The primary focus of ARCA is to dynamically adapt a program at runtime based on a variety of characteristics that change based on the running workload. As mentioned in section 2.3, the two main techniques used in past dynamic adaptation research are dynamic optimization and dynamic selection. ARCA is based upon the dynamic selection approach since the use of a fixed set of precompiled versions allows for a greater understanding of the possible interactions within a running workload. While an online dynamic optimizer is capable of making fine-grained changes that improve a single application, understanding the impact of such a change on an overall system workload is difficult and costly to do at runtime.

3.1 An Overview of the ARCA Framework

Conceptually, the ARCA framework has two primary phases: an initial code generation and classification phase and the subsequent dynamic adaptation phase. The initial phase is conducted only once for a given set of binaries, while the adaptation phase occurs every time the binaries are executed.

3.1.1 Code Generation and Analysis

The selection granularity of the ARCA system is function boundaries. In other words, an ARCA enhanced binary is created with numerous compiled versions of key

program functions and the specific versions executed at runtime are selected based on dynamic workload characteristics. Figure 3.1 showcases the overall approach used for code generation and analysis. Each of the primary components are discussed below.

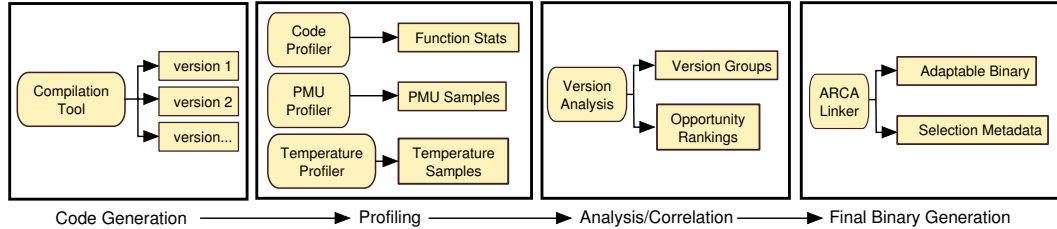


Figure 3.1: The ARCA static compilation phase.

Version Generation. Since ARCA is based upon the notion of dynamically selecting precompiled versions of code, a fundamental requirement of the system is that multiple compiled versions of a given application must initially be available. While any system that generates multiple code versions can be used, the ARCA framework itself contains a built-in compilation tool that compiles the supplied program source numerous times using different compilers as well as different optimization flags. The three compilers utilized include the 3.4 and 4.1 versions of gcc as well as version 10 of Intel’s icc. Table 3.1 below summarizes the different optimizations flags used with each compiler.

gcc 3.4	-O0	
gcc 4.1	-O1	-O1 -march=prescott
	-O2	-O2 -march=prescott
	-O3	-O3 -march=prescott
	-Os	-Os -march=prescott
icc 10.0	-O0	
	-O1	-O1 -xT
	-O2	-O2 -xT
	-O3	-O3 -xT
	-Os	-Os -xT

Table 3.1: The evaluated compiler and optimization flags.

The flags in the third column mirror those in the second, with additional flags

added that allow the compiler to generate code that uses features found only on the newer Intel microprocessors. In total, there are 27 code versions generated using the nine optimization levels across the three compilers.

Version Profiling. In order to classify the benefits and weaknesses of the different code versions, the 27 generated binaries are subsequently profiled. At the minimum, the ARCA system requires a code profile to be performed that breaks the code into slices of a fixed dynamic execution size as well as identifies the execution frequency of all functions within these code intervals. Beyond the base code profile, various dynamic characteristic profiles are conducted based upon the intended adaptation metric. For example, if the goal was to increase system throughput on an Intel Core 2 Duo processor than additional runs would be conducted that collect IPC and L2 cache events via the hardware performance monitor units (PMU). Such events would allow ARCA to determine the trade-offs between individual performance (IPC) and workload interference (L2 events, since the L2 cache is shared between cores on a Core 2 Duo processor).

Analysis and Correlation. After collecting the profile data, ARCA correlates the results back to the program slice intervals and then to each of the functions that executed within that slice. To correlate to the the individual functions, the overall slice value is linearly attributed to each function based upon execution weight. For example, the function that executes the longest in a slice is attributed as the producer of most of the observed values. While this approach is not completely accurate with respect to determining which function caused an observed behavior, the approach works well given the large sample size of the profile data. As more data is correlated, the results tend to converge to a reasonable approximation of the underlying behavior.

After the profile results are correlated back to each of the code versions, the versions are then clustered into categories of similar behavior and a subset of interesting code versions are selected from these depending on the adaptation goal. The version clustering and analysis technique is discussed in more detail in section 3.2.

Adaptable Binary Generation. The last stage of code generation component of ARCA is to create a new binary for the application that includes all of the selected code versions for key functions. To accomplish the generation of this new binary, ARCA utilizes a modified version of the GNU linker. The modified linker is capable of extracting compiled machine code at a function by function basis out of the object files generated for each version in the initial compilation phase. The linker fixes up the symbol references encountered across the extracted code as well as renames each copy of a function to avoid namespace conflicts. The linker also inserts a stub in place of each of the versioned functions which includes a `jmp` instruction to the version installed by default. This `jmp` instruction is then subsequently modified at runtime as the mechanism to select between the available versions.

3.1.2 Dynamic Adaptation

The aim of the dynamic adaptation component of ARCA is to observe a variety of system statistics and select function versions based upon these values and a desired criteria. ARCA is designed with support for two different system profiling frameworks. The primary framework is the OProfile system included as part of the Linux kernel. OProfile [35] is a system-wide performance monitoring framework that logs PMU events for all processes running on the system. In normal use, OProfile dumps log files to disk that are parsed offline in order to evaluate system performance. For use in ARCA, the standard OProfile system was modified to no longer dump log files but instead maintain an in-memory scoreboard that lists the top processes resulting in the most PMU events within a fixed time window. The scoreboard lists the processes in order as well as maintains a count of the events occurring at identical instruction addresses within the given processes. At the end of the time window, the scoreboard is cleared and processes once again make their way onto the now empty scoreboard as new PMU events occur.

In addition to the OProfile framework, ARCA also uses an in-house kernel module

designed to profile CPU temperature on the Pentium 4 and later Intel microprocessors. This module runs in conjunction with the OProfile setup and provides the temperature in a similar scoreboard style. For this scoreboard, however, the process rankings are based on cycles executed rather than anything related to the temperature read-out. Likewise, instead of maintaining an event sum, temperature scoreboard instead maintains a list of most recent temperature values for each of the top processes.

The actual selection engine of ARCA utilizes Pin[30], a dynamic instrumentation system developed by Intel, to take control of the versioned binary at runtime. To this end, Pin is used exclusively to read in the adaptation threshold values required by ARCA and install a monitor interrupt handler that is triggered via a timer interrupt. The dynamic instrumentation features of Pin are not used, and dynamic instrumentation is not a necessary requirement in general. In fact, the ARCA linker could itself directly install the initialization and monitor code into the adaptable binary if desired. A high-level overview of the dynamic system is depicted in Figure 3.2. The timer interrupt triggers the monitor code which then queries data from the system wide performance monitor. Depending on the observed state of the system, the monitor decides if it should alter any function versions. The `jmp` instruction in the relevant function stubs are then altered to point to the newly selected code versions. Three different selection mechanisms are supported in general and are discussed in section 3.3.

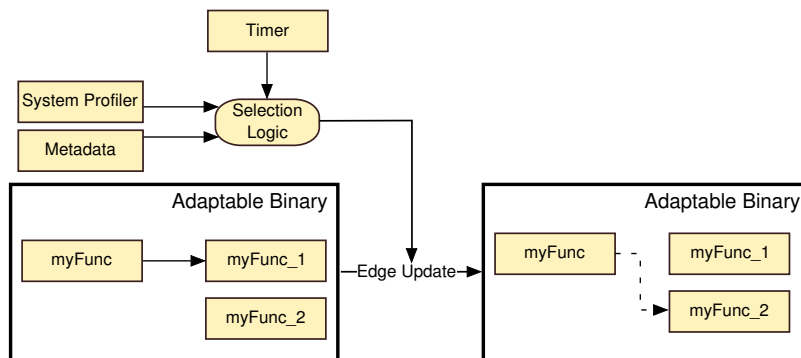


Figure 3.2: The ARCA run-time selection phase.

3.2 Version Analysis and Static Selection Methodology

In this section the version analysis and selection methodology of the ARCA framework is presented. The methodology is designed to identify unique behaviors within the large initial set of code versions and subsequently select candidate versions for inclusion in the hybrid, adaptable binary.

The classification is performed for each pair of two desired selection metrics, where one metric is considered the primary metric and the other the secondary metric. For example, one might want to classify versions using performance as the primary metric and L2 cache accesses as the secondary metric. For each pair of metrics, the data is clustered using a pivot clustering algorithm utilizing the Euclidean distance between the two metrics on a two-dimensional plane. Returning to the Core 2 Duo throughput example from above, a given cluster of versions might be characterized as having the best performance when running exclusively on the system but the worst performance when paired with a cache intensive application due to L2 interference. After clustering, each code version that maximizes the primary metric within each cluster is taken as the version representative of the behavior signified by that cluster. In practice the 27 starting versions tend to showcase only two or three statistically interesting behaviors: one that maximizes the primary metric, one that maximizes the secondary metric, and one that is a reasonable compromise between the two. After grouping, the representative versions are then sorted according to the desired selection metrics, with the first version maximizing the primary metric and each subsequent version trading a portion of the primary metric for a gain in the secondary metric. The system then chooses characteristic versions from each of the interesting behavior clusters depending on the adaptation goal.

In order to illustrate this version analysis methodology, consider the following example showcasing this technique using the 400.perlbench benchmark. For this example,

the primary metric is execution time and the secondary metric is L2 cache accesses. Figure 3.3 depicts the normalized average value for each of these metrics for the top six functions in 400.perlbench with respect to nine different compilations. Figure 3.4 shows the same data after behavior clustering has been applied and outlines the different behavior clusters.

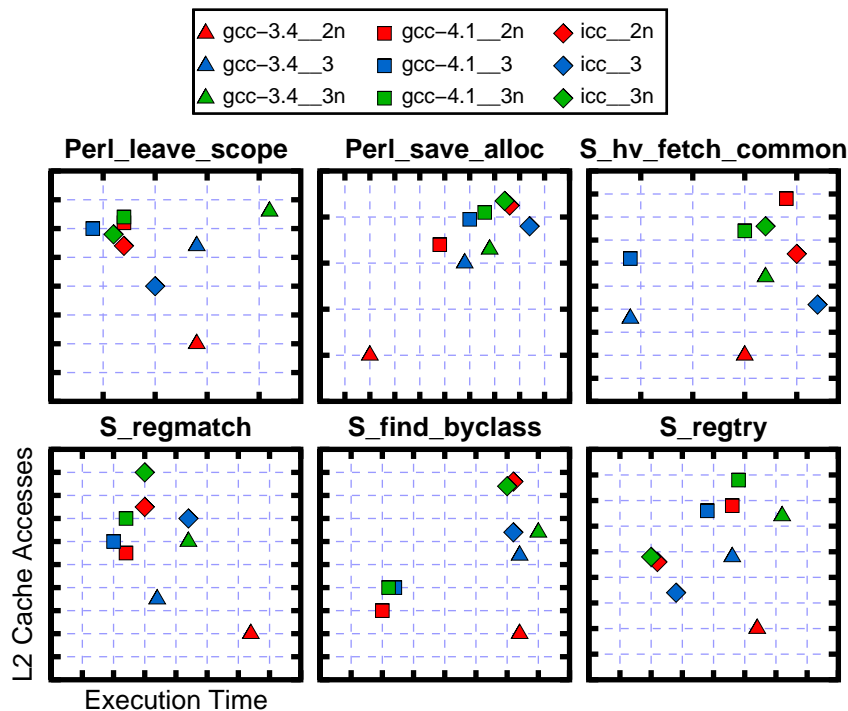


Figure 3.3: The code version execution and L2 access behavior for 400.perlbench.

The clusters demonstrate code versions with similar observed behaviors with respect to the two metrics. For example, the function `S_find_byclass` has a cluster near the origin that encompasses code versions which minimize both execution time and L2 cache accesses. In contrast, the function `Perl_leave_scope` showcases three distinct and interesting behaviors: a cluster that minimizes execution time, a cluster that minimizes L2 cache accesses, and a cluster that serves as a compromise in between the two metrics. Depending on the adaptation goal, a set of code versions from these clusters

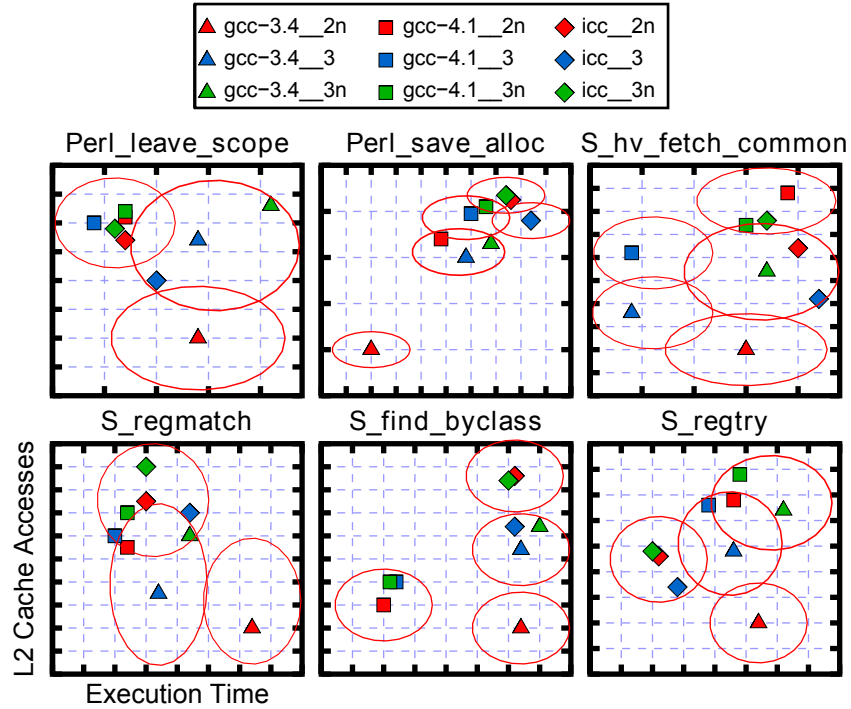


Figure 3.4: The identified execution and L2 access behavior clusters for 400.perlbench.

will be selected for inclusion in the hybrid binary. Section 4.2 provides a demonstration of adaptation focused on selecting between execution-favor and cache-favored code versions and discusses in further detail which behaviors are selected.

3.3 Dynamic Selection Techniques

The dynamic selection component of the ARCA framework provides three different selection algorithms which are discussed and evaluated below. These three algorithms are each targeted at different circumstances and designed with different goals in mind.

3.3.1 Greedy Selection

The first technique is aimed at providing adaptation that allows a given application to achieve the best performance possible regardless of the co-scheduled workload. The focus of the technique is on selecting code versions that maximize the individual application's primary metric without regard to the system as a whole. Given this goal, this particular technique is titled greedy selection. The greedy selection approach favors remaining in the highest performing code versions at all times. The approach will switch to lower performing code versions only when the dynamic situation suggests that the normally lower performing versions will perform at a higher rate at that instant. For example, if the intended focus is improving performance on a shared L2 cache system, then a greedy selector will favor having the fastest code versions installed and will only switch to a slower code version if the selector believes the new version will actually run faster given current system characteristics. Such characteristics might be an abnormally high amount of L2 cache misses compared to the standard observed for the application in isolation. This factor would suggest large interference in the L2 cache and therefore the selector would favor a slightly slower version if such a version exhibits fewer L2 cache accesses than the primary version. Moreover, all deviations from the default version by a greedy selector are temporary. After a fixed time interval, a function always reverts back to its default version. A high-level pseudo-code implementation of a greedy selector is shown in Figure 3.5. The implementation assumes the versions are ordered with the best default version first and each additional version giving up portions of the primary metric for gains in the secondary metric.

3.3.1.1 Synthetic Demonstration

A workload of two synthetic benchmarks was created to demonstrate the greedy selection approach, and ARCA in general. The first synthetic benchmark includes a

Figure 3.5 The greedy selection algorithm

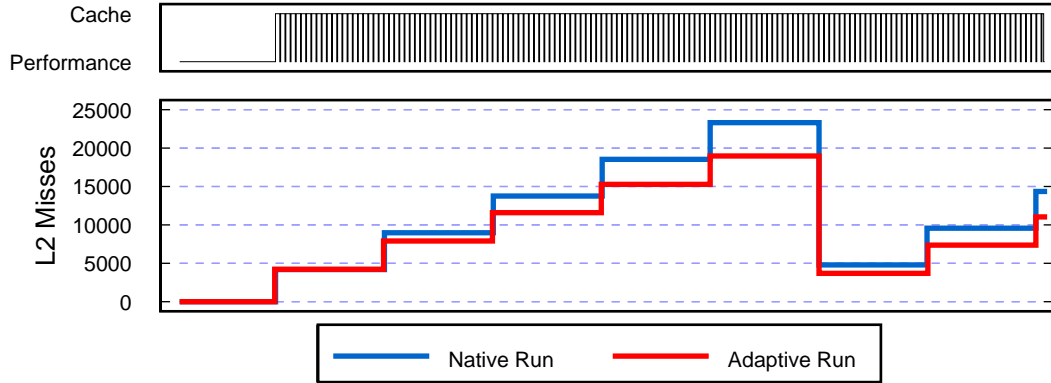
```

my_samples ← FILTER(recent_samples, pid)
for all (func, val) ∈ my_samples do
  if func is versioned and val ≥ threshold then
    NEXT_VERSION(func)
    APPEND(revert, (func, TIME + delay))
  end if
end for
for all (func, reset_time) ∈ revert do
  if reset_time ≤ TIME then
    RESET_VERSION(func)
  end if
end for

```

function which computes a set of complex hash values. This function has two different versions. The first version precomputes the entire set of hash values and stores them in memory. Whenever a value is requested, the value is fetched from memory and returned. The second version skips the precomputation and computes the hash value each time it is requested. The hash computation is designed to take more cycles to compute than the time it takes to retrieve a precomputed value from the L2 cache. Furthermore, the precomputed array is designed to fit within the size of the L2 cache on the Core 2 Duo E6600 microprocessor used for this demonstration. By design, the version exploiting precomputation is the fastest version when running alone on the system. The second synthetic benchmark is the same L2 cache trashing benchmark mentioned in section 2.2. It is assumed that the pairing of the hash benchmark with the L2 trashing benchmark will provide a favorable environment for the non-precomputation version to be the better performing version. Figure 3.6(a) provides a timeline showcasing the overall system L2 cache miss behavior for both a native and adaptation-enabled run of the two benchmarks as well as a timeline depicting the version selection choices made during the adaptive run. Figure 3.6(b) summarizes the overall impact of the adaptation with respect to both the execution duration of the benchmarks as well as the observed L2 miss behavior.

The results show that the runtime of the L2 cache trashing benchmark is im-



(a) Adaptation Timeline

Hash Speedup	-5.57%
CFlush Speedup	3.13%
Average Reduction in L2 Misses	22.58%
Median Reduction in L2 Misses	21.25%

(b) Impact Summary

Figure 3.6: A synthetic demonstration of greedy selection.

proved, with the actual hash benchmark that is being manipulated by ARCA losing a small amount of performance. Furthermore, there is a 22.58% reduction in average L2 cache misses. The loss in performance for the target application is largely dominated by the fact that the greedy selector is constantly switching back and forth between the two code versions, as shown in the version timeline. Since the greedy algorithm is opportunistic and always reverts back to the primary code version after a fixed interval of time, the approach is weak to system workloads that express a constant and unchanging threatening environment.

3.3.1.2 Approach Benefits

- (1) The primary benefit of the greedy approach is that it works exclusively based off profile information related to the adapting application, without requiring

system wide knowledge. In effect, the greedy selector works much like a cellular automation, where the technique works diligently to improve the performance of the target application yet implicitly benefits the system as a whole.

- (2) Given the feature above, the greedy approach is the most scalable and portable of the workload adaptive techniques. On a system without shared resources, there will be no observed contention at the target application regardless of the behavior of the rest of the workload. Equally, on a system with a complex shared resource composition, the target application will only notice behavior that affects it. By design, the greedy selector works without knowledge of makeup of the underlying system.

3.3.2 Cooperative Selection

The second selection technique is aimed at improving the performance of the system as a whole, without regard to the adapting application's performance. Such a technique is useful in situations where a given application is less important than other applications on the system, and therefore sacrificing the performance of the adapting application for an overall improvement throughput is acceptable. Based on this goal, this technique is titled cooperative selection. The cooperative selector considers all PMU events rather than just those events related to the target application. While the greedy selector's adaptation threshold was based on a value that was believed to be the point at which slower versions performed better than the default code version, the adaptation threshold provided to the cooperative selector instead includes an estimation of the typical behavior the target application should see when running as a part of a co-scheduled workload. If the system overshoots this estimated baseline, the selector will kick in and select versions that attempt to reduce the trend. For example, if the system is exhibiting an abnormally large amount of L2 cache misses, the cooperative scheduler

will choose code versions that have fewer cache accesses. In contrast to the greedy selector, the cooperative selector never reverts its selection decisions until after the system behavior falls within range of the target baseline. A pseudo-code implementation of the cooperative selector is shown in Figure 3.7. The implementation assumes the versions are sorted in order of trade-off between the primary metric and the secondary metric.

Figure 3.7 The cooperative selection algorithm

```

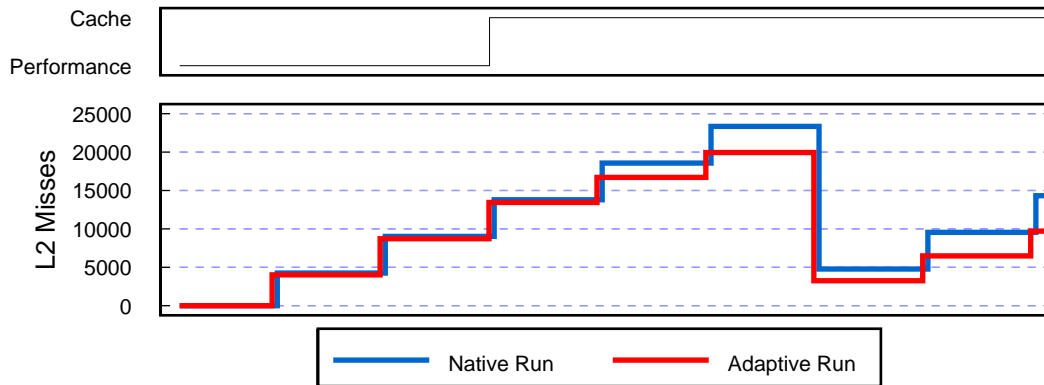
if sample_total  $\geq$  threshold then
  my_samples  $\leftarrow$  FILTER(recent_samples, pid)
  for all (func, val)  $\in$  my_samples do
    if func is versioned then
      NEXT_VERSION(func)
      APPEND(revert, func)
    end if
  end for
else
  for all func  $\in$  revert do
    RESET_VERSION(func)
  end for
end if

```

3.3.2.1 Synthetic Demonstration

To demonstrate the cooperative selector, the same workload and experiment discussed in section 3.3.1.1 was performed. Figure 3.8(a) showcases the L2 miss behavior timelines for both runs as well as the version selection timeline for the adaptive run. Figure 3.8(b) summarizes the impact of the cooperative-driven adaptation.

The results demonstrate that the cooperative selector is capable of improving the performance of the L2 cache trashing benchmark while slowing down the adapting hash benchmark. The adaptation also results in a 33.46% reduction in average L2 cache misses. Also of interest is the fact that the cooperative selector in this example immediately switches to the cache-favored code version early on and never reverts back to the primary code version. As mentioned in 3.3.1.1, and demonstrated by the cache



(a) Adaptation Timeline

Hash Speedup	-6.39%
CFlush Speedup	3.89%
Average Reduction in L2 Misses	33.46%
Median Reduction in L2 Misses	31.72%

(b) Impact Summary

Figure 3.8: A synthetic demonstration of cooperative selection.

miss timelines, the synthetic benchmark expresses a constant and unchanging hostile environment. While the greedy approach ignored this fact and constantly switched between code versions, the cooperative selector remains in the version best suited to the observed level of cache interference.

3.3.2.2 Approach Benefits

- (1) The main benefit of the cooperative selector is its ability to remain in a state that aims to benefit the overall system, which often results in a better improvement of the workload as a whole. For example, in the above synthetic demonstration the cooperative technique impacted the number of L2 misses more than the greedy approach discussed previously.
- (2) Furthermore, the cooperative selector avoids constantly switching between ver-

sions during an execution where the system environment is consistently characterized by interference. Avoiding this state may lead to greater improvements for some applications.

3.3.3 Fractional Selection

The third selection technique is titled fractional selection. The fractional selection approach is designed to ignore workload characteristics altogether and simply toggle between two available versions at a fixed ratio. The technique is aimed less at multi-threaded adaptation and more so towards compromising between two desired metrics such as performance and power, or performance and fault tolerance. In this thesis, the fractional selector is only evaluated in section 4.3 as the preferred selector for the temperature-motivated adaptation case study. This choice stems from the fact that the evaluation conducted in that section focus on compromising between performance and temperature dissipation for an application that is running alone on the system. A pseudo-code implementation of the fractional selector is shown in Figure 3.9. This code assumes that there are exactly two code versions, with version 0 favoring the primary metric and version 1 favoring the secondary metric.

Figure 3.9 The fractional selection algorithm

```

if counter = 0 then
  for all func ∈ versioned_functions do
    CHANGE_VERSION(func, 1)
  end for
else if counter > threshold then
  for all func ∈ versioned_functions do
    CHANGE_VERSION(func, 0)
  end for
end if
if counter = max then
  counter ← 0
end if

```

3.3.3.2 Approach Benefits

- (1) The fractional selector does not require runtime profiling and therefore can be used across a variety of systems as well as utilized in cases where there are no relevant dynamic statistics.
- (2) The fractional approach provides a means to compromise between multiple desired characteristics. The approach is therefore beneficial even for solo execution as well as execution on single threaded machines.

3.4 System Overhead

Since ARCA is a runtime system which itself is part of the co-executing workload, understanding ARCA's impact on the system is another factor that must be considered. Specifically, in order to receive the most benefit, the overhead of the framework must be reasonable. There are two primary overheads associated with ARCA: the overhead of the system-wide PMU profiler, and the overhead of the ARCA selection logic itself.

Sampling Overhead. Sampling performance counter and temperature values is a fundamental requirement of ARCA. Since the sampling frameworks used by ARCA are external projects, the primary control the framework has over the impact of the sampling technology is selecting the rate at which the hardware counters overflow. Figure 3.11 shows the results of an experiment measuring the overhead of the OProfile sampler at different event overflow values. As a whole, the impact of sampling is less than 1%. Since the overhead values observed are likely within the margin of measurement error, the figure is composed of numerous samples at each rate level. Linear regression was then used to compute the trend line shown, and the error bar in the center of the figure depicts the the overall mean and standard deviation of the results. In general, the sampling overhead is range of 0.2% to 0.6% regardless of the overflow rate. Based on this data, ARCA is designed by default to use an overflow value of 2500 events.

Selection Overhead. The selection logic lies primarily in the selection interrupt handler mentioned above. The overhead of the adaptation framework is therefore influenced by the frequency of the timer that triggers the selection handler. Figure 3.12 shows the results of another overhead experiment that explores the overhead of the selection system as a function of the selection timer frequency. This observed overhead is additional overhead up top of the baseline PMU overhead discussed above. As shown in the figure, the overhead is within the range of 0.01% to 0.1%. With both the selection and sampling overheads considered, the ARCA framework has a total overhead of less than 1% in all cases.

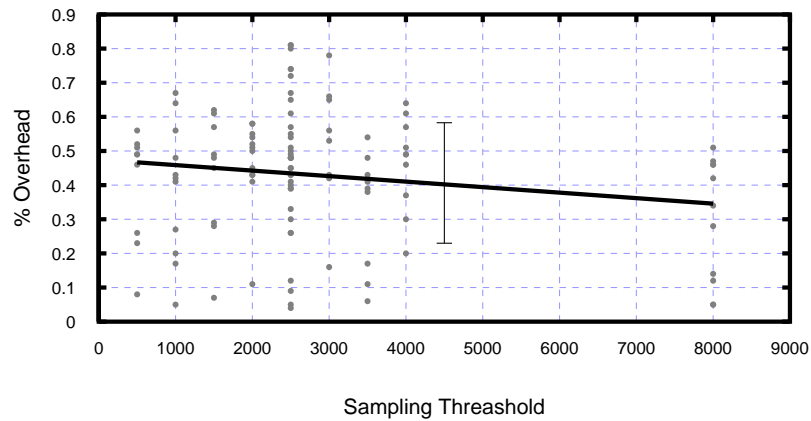


Figure 3.11: The observed PMU sampling overhead.

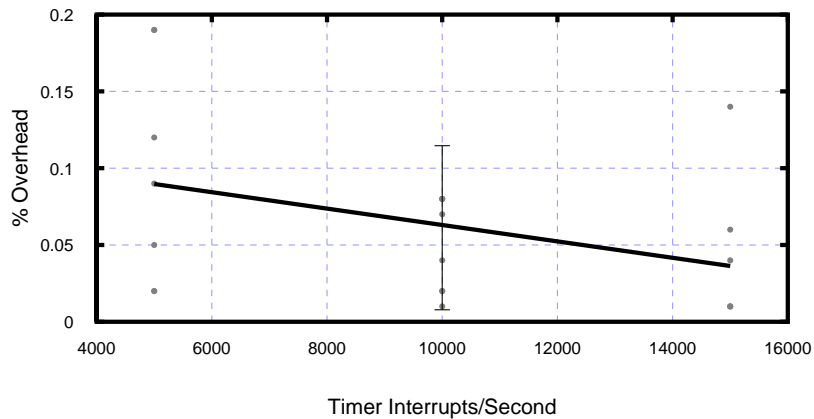


Figure 3.12: The observed ARCA selection overhead.

Chapter 4

Demonstration: Code Versioning and Adaptive Selection

This chapter demonstrates the complete code versioning and adaptive selection capabilities of ARCA with respect to two different goals: improving multi-process system performance, and compromising between temperature dissipation and performance in the case of a single application.

4.1 Workload Description and Analysis

A subset of the SPEC CPU2006 benchmark suite is used to evaluate the case studies in this chapter. Since ARCA generates 27 versions of each benchmark and performs several levels of profiling on each version, it was impractical to evaluate the entire SPEC CPU2006 suite. Furthermore, the ARCA linker has some unresolved issues dealing with binaries generated from C++ code, and therefore the benchmarks were restricted to C only selections. The following seven benchmarks were chosen:

Benchmark	Application Area
400.perlbench	Scripting Language
401.bzip2	Compression
403.gcc	C Compiler
429.mcf	Combinatorial Optimization
433.milc	Physics / Quantum Chromodynamics
462.libquantum	Physics / Quantum Computing
470.lbm	Fluid Dynamics

Table 4.1: The benchmarks evaluated.

After performing the initial profiling and analysis stages, the 27 versions were reduced to approximately nine versions that demonstrated statistically interesting behaviors. In each case, the nine versions corresponded to the both the generic i386 and Core 2 optimized -O3 binaries as well as the Core 2 optimized -O2 binary generated by all three compilers. The remainder of the chapter therefore focuses solely on these nine versions. When identified, the versions are named according to the compiler along with a suffix denoting the optimization flags used. In cases where the suffix includes the letter *n*, the suffix corresponds to the shown optimization level compiled with the flags in the third column of Table 3.1 – in other words, code for the newest Intel processors.

Since ARCA is implemented as a dynamic selector that switches between versions by altering the function entrance stub, the amount of selections the system can perform is fundamentally limited by the number of times a given function is called during execution. For example, it is impossible for the adaptive selector to alter a function that is called once and never returned from as the program will never again pass through the function stub. To understand the absolute selection potential consider Figure 4.1 which shows the number of function invocations occurring across each benchmark over time. Code regions where the number of invocations is greater than 10,000 are likely capable of taking full advantage of ARCA’s adaptation mechanism. Cases with fewer invocations will gain less benefit, losing out on the fast, fine-grained switching that the system is capable of providing. Of the seven benchmarks, 470.lbm and 429.mcf are the only two with problematic regions, arising from the fact that both benchmarks are heavily dominated by long executing compute kernels.

4.2 Case Study: Improving System Performance

As demonstrated in section 2.2, the key motivating factor behind the design of ARCA is that various code versions exhibit different characteristics when running solo versus when run as part of a aggressive co-scheduled workload. For example, consider

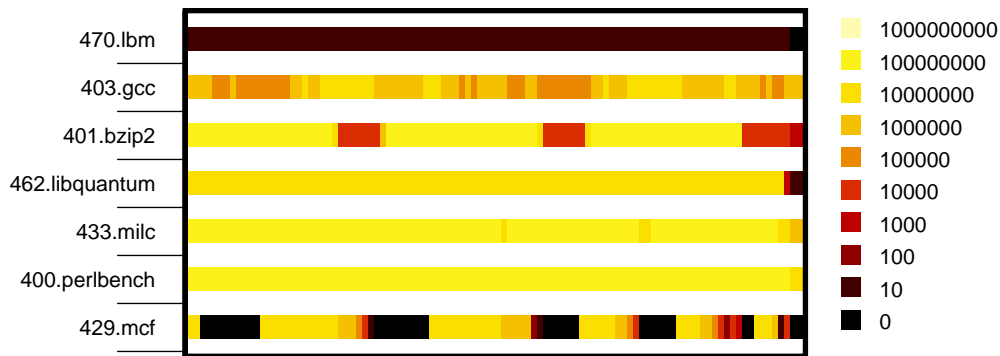


Figure 4.1: The observed function invocations over time for each benchmark.

the data previously shown in Figure 3.3 for the benchmark 400.perlbench which plots the nine different code versions for the top six most executed functions in 400.perlbench with respect to execution duration on the x-axis and L2 cache accesses on the y-axis. In many cases, the fastest version is also the version with the fewest cache accesses. However, this is not universally true. Given that the L2 cache is the primary element shared between cores on a dual-core Intel Core 2 Duo processor, the number of cache accesses serves as a first order estimate of how a version will perform as part of a paired workload. When paired with another application that is aggressively using the L2 cache, selecting versions which have fewer cache accesses may provide better performance even if those versions are normally slower when running without cache interference. The motivational example presented in section 2.2 serves as further support for this claim given that it demonstrated that the fastest overall compilation of 400.perlbench performs worse than a normally slower version when executing alongside a program with high L2 interference.

4.2.1 Version Analysis and Available Opportunity

In an attempt to address this performance issue, ARCA was used to analyze and cluster the workload mentioned in previous section with respect to individual execution

performance as the primary metric and L2 cache accesses as the secondary metric. Table 4.2 lists the function that ARCA selected as versioning candidates. For each function, the table lists the percentage of the total benchmark execution time that is spent inside that function, as well as the average characteristics of the different versions included in the generated hybrid binary. The numbers shown for each metric represent the absolute difference of the given version with respect to the best version for the metric. The benchmark 471.libquantum is omitted since every function identified for versioning consisted of less than 1% of the benchmark’s execution time, and therefore are not worth versioning.

Given the results presented in the table, an analysis can be performed that estimates the adaptation opportunity available to ARCA. The first issue to consider is the execution frequency attributed to each of the versioned functions. Based on Amdahl’s Law, the overall expected improvement is greatly affected by how much of the total execution can be improved. The benchmark 429.mcf, for example, spends less than 5.74% of the time within versioned functions. As such, the maximum theoretical improvement that could be achieved through adaptation is a gain of exactly 5.74% assuming infinite speedup. While calculating the theoretical maximum improvement is useful to establish an upper bound, determining a realistic approximation of the available opportunity is also necessary. In this scenario, the goal is to switch between execution-favored versions and cache-favored versions; therefore, a realistic approximation of the speedup potential can be generated by estimating how much benefit a cache-favored version provides in pathological workload scenarios. According to [27], the Core 2 micro-architecture takes 12 cycles on average to fetch data from the L2 cache and 165 cycles to fetch data from memory that is not in the cache, for a difference of 153 cycles. The estimated performance improvement of a cache-favored version in a pathological situation can therefore

				Δ Cycle	Δ L2
400.perbench	<1%	Perl_pp_nextstate	icc_3n icc_3	0.00 757.84	57.84 0.00
	1.07%	Perl_regexec_flags	gcc-4.1_3 icc_3	0.00 6471.14	434.34 0.00
	1.11%	Perl_runops_standard	icc_3n icc_3	0.00 2803.55	181.67 0.00
	1.70%	S_hv_fetch_common	gcc-3.4_3 gcc-3.4_2n	0.00 16790.47	97.13 0.00
	3.80%	Perl_leave_scope	gcc-4.1_3 gcc-3.4_2n	0.00 29593.54	1496.37 0.00
	7.98%	S_regtry	icc_3n gcc-3.4_2n	0.00 123419.71	2566.87 0.00
	12.96%	S_find_byclass	gcc-4.1_2n gcc-3.4_2n	0.00 219556.57	2248.91 0.00
	49.13%	S_regmatch	gcc-4.1_2n gcc-3.4_2n gcc-3.4_3	0.00 831095.75 259662.69	9689.35 0.00 3493.45
401.bzips2	<1%	uncompressStream	icc_2n gcc-4.1_2n	0.00 73.78	6.09 0.00
	9.23%	generateMTFValues	icc_3 icc_3n	0.00 314672.29	1500.42 0.00
	18.33%	BZ2.decompress	icc_2n icc_3n	0.00 217699.92	57.68 0.00
403.gcc	<1%	add_noreturn_fake_exit_edges	gcc-4.1_2n gcc-3.4_2n	0.00 27.28	4.24 0.00
	1.06%	for_each_rtx	gcc-4.1_3n gcc-4.1_2n	0.00 100079.38	8157.40 0.00
	1.42%	note_stores	gcc-3.4_3 icc_3	0.00 8955.82	109.01 0.00
	2.36%	htab_traverse	icc_3n gcc-3.4_3	0.00 1031135.41	86578.95 0.00
429.mcf	<1%	read_min	gcc-3.4_2n gcc-4.1_3n	0.00 1531.41	2.76 0.00
	4.74%	primal_iminus	gcc-3.4_2n gcc-4.1_2n	0.00 53934.33	2958.71 0.00
433.milc	<1%	su3_adjoint	gcc-4.1_3n icc_3	0.00 222265.68	4991.84 0.00
	1.35%	mult_adj_su3_mat_4vec	icc_2n gcc-3.4_3	0.00 963791.74	42987.75 0.00
	1.35%	scalar_mult_add_su3_vector	gcc-3.4_2n gcc-3.4_3	0.00 231691.68	963.78 0.00
	1.36%	path_product	gcc-3.4_3 gcc-3.4_2n	0.00 555.14	69.68 0.00
	2.89%	uncompress_anti_hermitian	gcc-3.4_3 gcc-4.1_2n	0.00 121028.87	685.03 0.00
	4.05%	mult_su3_an	gcc-3.4_2n gcc-3.4_3	0.00 2926.02	3811.82 0.00
	7.62%	scalar_mult_add_su3_matrix	gcc-3.4_3 gcc-3.4_2n	0.00 77283.30	4406.36 0.00
	15.45%	mult_su3_nn	gcc-4.1_2n gcc-3.4_3	0.00 89734.09	9314.59 0.00
470.lbm	<1%	LBM_initializeSpecialCellsForLDC	icc_3 gcc-4.1_3	0.00 480145.58	18903.08 0.00
	99.46%	LBM_performStreamCollide	gcc-3.4_2n gcc-3.4_3	0.00 731950.76	15758.91 0.00

Table 4.2: The code versions selected when targeting performance adaptation.

be estimated using the following equation:

$$S = \frac{153(L_{\Delta})(I) - C_{\Delta}}{C}$$

where S is the calculated % speedup of a cache-favored version of a particular function, L_{Δ} is the $\Delta L2$ value from Table 4.2 for the execution-favored version, I is an arbitrary value between 0 and 1 which estimates the interference of a given workload, and C_{Δ} is the $\Delta Cycle$ value from Table 4.2 for the cache-favored version, and C is the actual cycle count of the execution-favored version which is enumerated in Table 4.3.

400.perlbench	Cycles
Perl_pp_nextstate	31839.74
Perl_regexec_flags	88362.25
Perl_runops_standard	110080.31
S_hv_fetch_common	165987.37
Perl_leave_scope	300879.91
S_regtry	681425.13
S_find_byclass	973977.21
S_regmatch	3728027.82
401.bzip2	Cycles
uncompressStream	1246.9
generateMTFValues	9604572.55
BZ2_decompress	7618735.0
403.gcc	Cycles
add_noreturn_fake_exit_edges	721.73
for_each_rtx	759729.33
note_stores	39307.6
htab_traverse	4130060.95
429.mcf	Cycles
read_min	102264.04
primal_iminus	2212669.4
433.milc	Cycles
su3_adjoint	6221843.71
mult_adj_su3_mat_4vec	6974056.91
scalar_mult_add_su3_vector	7810033.8
path_product	531388.57
uncompress_anti_hermitian	1660723.08
mult_su3_an	6365859.56
scalar_mult_add_su3_matrix	5596311.83
mult_su3_nn	11078916.44
470.lbm	Cycles
LBM_initializeSpecialCellsForLDC	4939434.83
LBM_performStreamCollide	13734558.94

Table 4.3: The average cycle count for the fastest code versions.

Furthermore, the following equation can be used to estimate the overall opportunity for a given benchmark:

$$O = \sum_{i=0}^n S_i * F_i$$

where n is the number of versioned functions in the benchmark, S_i is the calculated speedup for a given cache-favored function version as calculated above, and F_i is the execution frequency of the function as listed in Table 4.2. Based on this methodology,

the Table 4.4 can be generated which lists the estimated performance opportunity for each benchmark given different values of I .

Benchmark	Interference Constant (I)										
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
400.perlbench	-0.16	-0.13	-0.09	-0.06	-0.03	0.00	0.04	0.07	0.10	0.14	0.17
401.bzip2	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01
403.gcc	-0.01	-0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
429.mcf	-0.00	-0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01
433.milc	-0.01	-0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.03	0.04	0.04
470.lbm	-0.05	-0.04	-0.02	-0.00	0.02	0.03	0.05	0.07	0.09	0.10	0.12

Table 4.4: The estimated performance gains possible through adaptation.

Based on these values, it is reasonable to state that this workload will exhibit little benefit from dynamic adaptation unless the system interference level is at least 40% or higher. Moreover, at 40% the predicted gain is at most 3% before subtracting out the adaptation overhead discussed in section 3.4. At an interference level of 60% or more, the workload does appear to have reasonable opportunities. Concerning specific benchmarks, 401.bzip2 is estimated to always lose performance through adaptation, while 400.perlbench, 403.gcc, 433.milc, and 470.lbm are the benchmarks which show some the most promise. Also, as previously shown in Figure 4.1, 470.lbm has so few function invocations that the benchmark has little opportunity to adapt, despite the encouraging predictions in Table 4.4.

4.2.2 Experimental Results

Figure 4.2 shows the normalized execution time compared to native execution for the various possible paired workloads given our seven benchmarks when run under ARCA using the greedy selection approach. Figure 4.3 show similar results for the cooperative selection approach. In both cases, dynamic adaptation was enabled only for the first benchmark while the second benchmark ran natively. The results are shown both for the adapting binary itself as well as the overall paired workload. The execution time of the overall workload is defined as the time of the longest running application

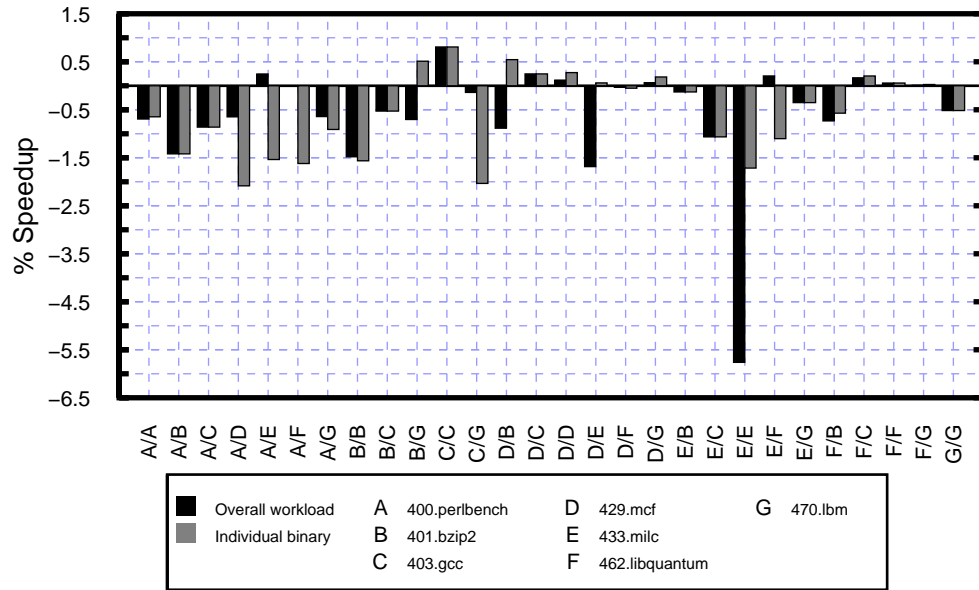


Figure 4.2: The impact of performance-focused adaptation (Greedy).

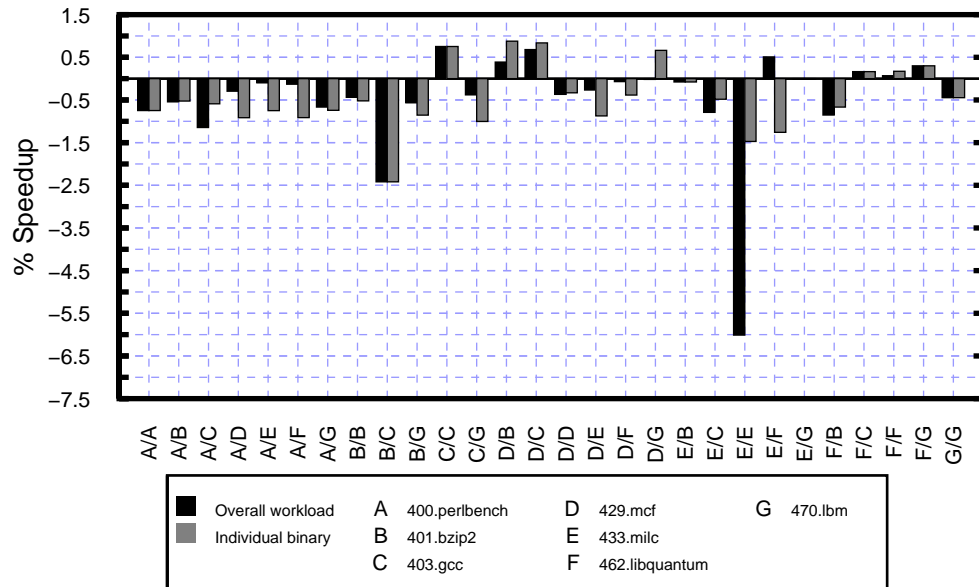


Figure 4.3: The impact of performance-focused adaptation (Cooperative).

in the workload, be it the adapting binary or not. Figure 4.4 shows similar results showcasing the the reduction in the total L2 cache miss rate between the adaptive run and native execution for all the possible paired workloads. Since the total L2 miss

rate is being considered rather than the individual rate for each application, only the cooperative selection technique is shown as such is most likely to impact the observed system-wide metric.

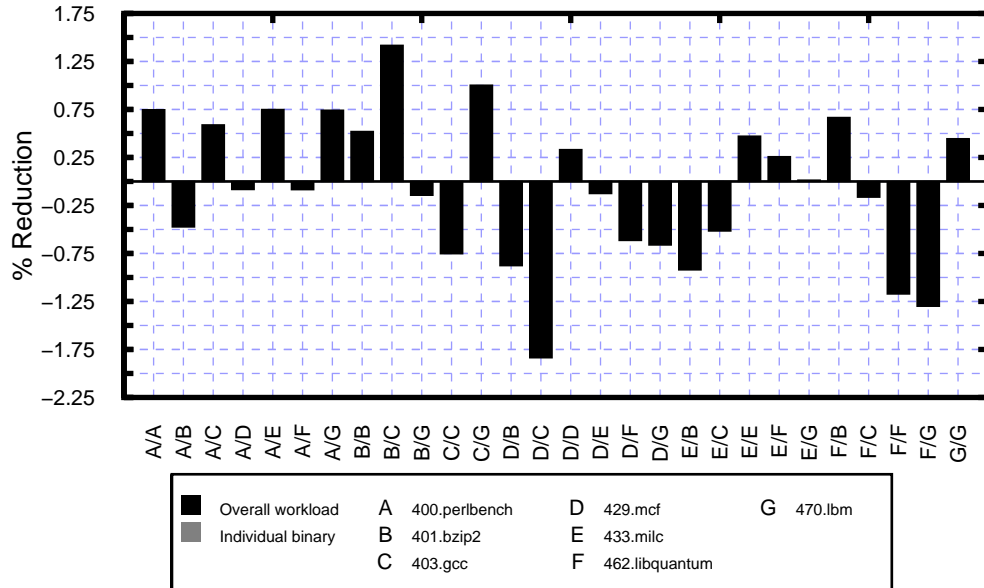


Figure 4.4: The reduction in L2 misses from performance-focused adaptation.

4.2.3 Analysis

Overall, the results demonstrate that the workloads running with dynamic adaptation enabled end up showcasing a slight slowdown rather than a notable speedup. For the greedy selection runs, the adaptive binary showcases an average speedup of -0.56% with a minimum of -2.08% and a maximum of 0.88%. The related values for the overall workload are -0.58%, -5.76%, and 0.80% respectively. For the cooperative selection runs, the adaptive binary showcases an average speedup of -0.44% with a minimum of -2.41% and a maximum of -0.88%. Similarly, the overall workload values for the cooperative case are -0.48%, -6.01%, and 0.75% respectively.

As previously predicted in Table 4.4, 401.bzip2 had fairly terrible performance throughout, while 403.gcc showed marginal gains in a few instances. On the other

hand, there are a number of results which disagree with the previous predictions. For example, 433.milc was predicted to have around 2% of available adaptation opportunity yet ultimately displayed very poor performance in many instances. Likewise, 429.mcf was predicted to have little available opportunity but ended up being the benchmark with the most observed positive impact.

On the issue of L2 cache traffic, the results in Figure 4.4 do indeed demonstrate a reduction from the adaptation technique. As a follow up to these findings, it would be interesting to run these same two case studies on a system which is much more sensitive to cache misses than the aggressive out-of-order Core 2 processor. For example, on an in-order processor where there is no chance of hiding cache miss latency, the reductions shown in the L2 miss rate may conceivably result in notable performance gains. Any future work in this area should stress such architectures in addition to the standard, high-performance out-of-order machines.

4.3 Case Study: Lowering CPU Temperature

Another system characteristic that has become more relevant in recent years is ambient CPU temperature. Since less heat requires less cooling, reducing the temperature produced by a CPU allows for smaller devices as well as less power wasted on cooling solutions such as fans. Most existing processors that are targeted at temperature sensitive markets are simply designed such that they always operate within a given temperature window. In general, this means clocking the processor at a speed which makes it impossible to surpass a max temperature regardless of the running code. New processors are more intelligent, allowing the processor to run higher frequencies by default and then dynamically scaling down the clock if the temperature passes a given threshold. Such a feature of newer processors makes things interesting with respect to code generation. For example, one code version might perform faster than another but this version might also result in a CPU temperature that triggers a frequency reduction.

Running this version at the lower clock frequency may, in fact, be slower than running the initially slower code at the faster frequency.

4.3.1 Version Analysis and Available Opportunity

To this end, ARCA was used to analyze the version differences with respect to observed change in system temperature. The temperature metric was calculated by converting the temperature results gathered during a profile run into temperature deltas showing the change in temperature from one code slice to another. The temperature deltas were then linearly attributed to the functions composing the slice based upon function execution percentage. The approach mirrors the general approach taken by ARCA for traditional PMU events, as discussed in section 3.

Evaluating all seven of the considered benchmarks returned mixed results. Nearly all of the benchmarks demonstrated that the code versions with the least impact on temperature were also the fastest executing versions. The findings are logical since the fastest code versions are fast predominately because the compiler was more adept at removing redundant code. Thus, the faster versions tend to be shorter and with fewer instructions to execute the impact on temperature is reduced. On the other hand, 433.milc serves as a case where there is a trade-off between performance and temperature. Figure 4.5 depicts the trade-offs by plotting the average temperature delta for each code version versus the corresponding execution time for the top six functions in 433.milc.

Even when running alone, 433.milc appears that it might gain from an adaptation approach that attempts to average out the overall temperature over time without deviating from the best performing code very often. Since there is no need to profile dynamic characteristics as 433.milc is running along and in no way hindered by other applications, the fractional selection approach makes the most sense. As mentioned in 3, the fractional selector is designed to routinely switch between two different code

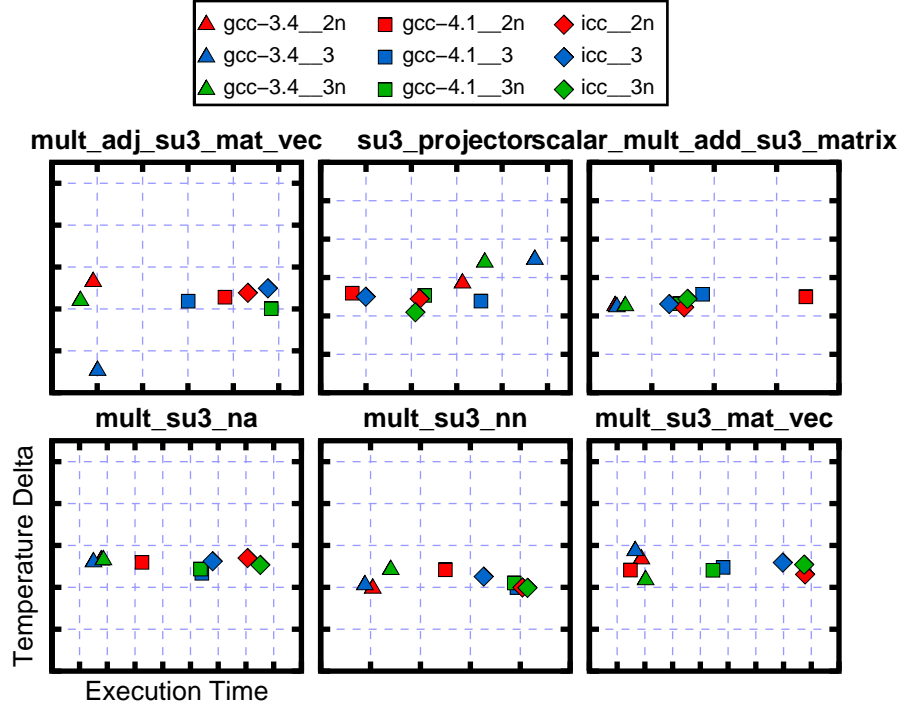


Figure 4.5: The code version temperature and execution behavior for 433.milc.

versions in order to achieve a compromised behavior.

4.3.2 Experimental Results

Figure 4.6 showcases the normalized execution time as well as the normalized temperature mean for several different adaptive runs of 433.milc with the intention of achieving an improvement in temperature without a major impact on performance. Each of the multiple runs uses the fractional selector with a different selection ratio, which determines how frequently to switch between the temperature-favored and execution-favored code versions. The ratio is plotted along the x-axis, with increasing numbers representing more time spent in execution-favored code. At 10%, the system is selecting temperature-focused versions 90% of the time.

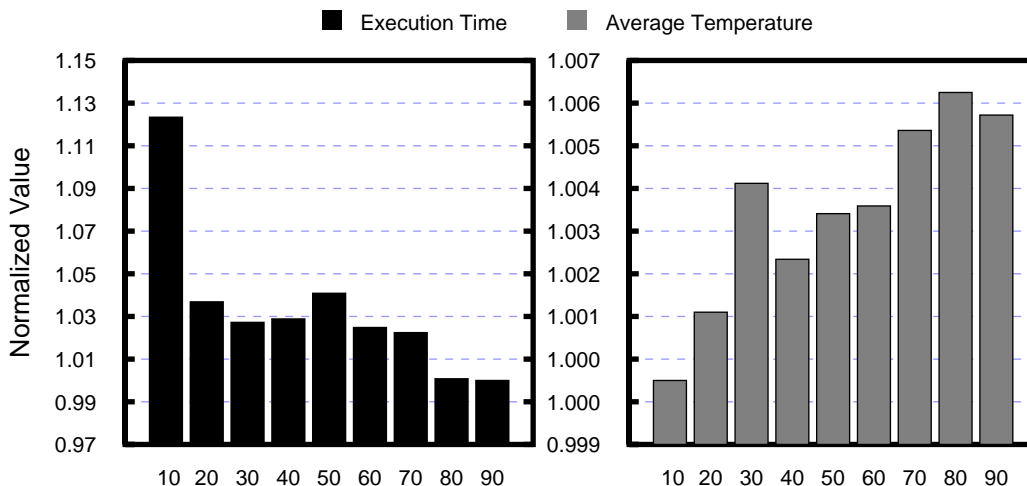


Figure 4.6: The impact of temperature-focused adaptation (Fractional).

4.3.3 Analysis

The results demonstrate that there is a general trend for the performance to drop as the system favors more performance-centric code versions. Likewise, the temperature results also demonstrate the desired trend. As execution functions become more favored, the temperature results increase. On the other hand, the variation in temperature between the different ratios is very small. Table 4.5 lists the actual observed temperature values that were used to construct the normalized view in Figure 4.6.

Ratio	Temperature ($^{\circ}\text{C}$)
0%	56.75
10%	56.30
20%	56.39
30%	56.56
40%	56.46
50%	56.52
60%	56.53
70%	56.63
80%	56.68
90%	56.65

Table 4.5: The average observed core temperature at different adaptation ratios.

The values listed differ by less than half a degree regardless of the selection ratio.

For 433.milc, the very minor reduction in temperature through adaptation does not justify the much larger performance degradation. Another issue is that the thermal sensor included on the Core 2 microprocessor has a quantization error of $\pm 0.5^{\circ}\text{C}$. The data in Table 4.5 shows that most of the values are within 0.10°C of one another. Given a sensor error of $\pm 0.5^{\circ}\text{C}$, the results as a whole tend to be well within the margin of error.

Chapter 5

Future Work

One of the primary contributions of this thesis is that the techniques presented allow for a unified framework that can run on a variety of systems as well as address a number of different adaptation goals. This chapter presents the following miscellaneous issues and topics related to this notion of flexibility and extensibility:

- (1) A preliminary demonstration of the potential scalability of the techniques presented in this thesis. An experiment is shown that evaluates multi-application adaptation across a four-application workload on an Intel quad-core microprocessor.
- (2) A brief discussion of the limitations of the compiler centric approach utilized in this work, followed by a preliminary analysis targeted at using this work's techniques and framework to address algorithmic selection.
- (3) A presentation of future research topics that serve as a natural progression of the ideas presented in this thesis.

5.1 System Scalability

The demonstrations presented in chapter 4 showcased workloads with only one of the two paired applications utilizing dynamic adaptation on a dual-core Core 2 Duo microprocessor. In fact, the majority of this thesis is based on dual-core workloads with

single application adaptation. However, the techniques and methodologies presented in this work are not limited to dual-core machines or single program adaptation. To provide some insight into this issue, an experiment was performed to showcase the ability of the ARCA framework to work on quad-core machine as well as the capability of providing more than single application adaptation.

The experiment was built around the SPEC benchmarks evaluated in chapter 4, however the focus was on workloads consisting of four applications rather than two. In order to evaluate multi-application adaptation, each workload was ran four times with each additional run enabling adaptation on a new binary. In other words, during the first run only one application out of the four is adapting while the other three are running natively; while during the second run, two applications are adapting with only two running natively. Figure 5.1 showcases the results of this experiment with the x-axis displaying the number of adapting applications and each data point representing the average speedup across all four applications in a particular run of a given workload.

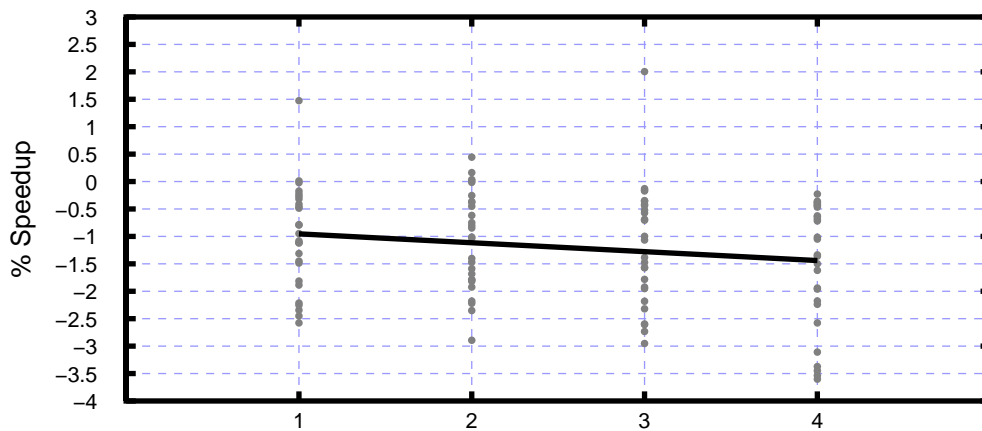


Figure 5.1: Demonstration of multi-application adaptation on a quad-core machine.

The quad-core results demonstrate a general trend towards degraded performance as the number of adapting applications increases. On the other hand, the actual speedup values are very similar to the values observed in section 4.2; ranging from a slowdown of

roughly 4% to a speedup of 2%. The fact that the results mirror those of the dual-core case is to be expected as the Intel Core 2 Quad processors are simply two Core 2 Duo processor dies placed into a single package, and therefore the quad-core runs behave more or less the same as two paired runs. Since the results of the paired runs for the SPEC benchmarks are poor, so are the quad-core results.

5.2 High-Level Versioning and Selection

The code versioning approach presented in this work focuses on generating a number of code compilations by using a number of different compilers and compilation flags. Such an approach is fundamentally limited by the amount of variation provided by the different code generation decisions that a compiler is capable of making. An adaptation campaign focused on selecting between execution-favored and cache-favored versions such as the demonstration in section 4.2 is crippled by the fact that most traditional compiler optimizations do not result in decisions selecting between these two metrics. In fact, most compiler optimizations will favor both metrics since one of the primary means by which a compiler improves performance is by reducing memory accesses by better utilizing registers and identifying and propagating previously computed values and sub-expressions.

The most benefit from an adaptation system such as ARCA is going to come from higher level algorithmic decisions that compilers typically do not address. For example, the synthetic hash benchmark in section 3.3 is designed around two hand-coded versions of a function with the primary difference being the high-level decision of exploiting pre-computation or not. One of the only high-level decisions that a compiler will make is related to code generation of `switch` statements. One option is to generate code built around a number of conditional statements, while another option is to generate code that utilizes a jump table. The first approach tends to be slower for a large amount of cases since the approach forces each case to be checked sequentially, yet the approach

has fewer memory accesses than the equivalent jump table solution. On a system with significant cache and memory interference, the code-based version may perform better than the normally faster jump table version. Nonetheless, even with this consideration towards `switch` statement generation, compilers provide little interesting variation to exploit. After all, very few applications spend the majority of their time executing `switch` statements. Several common cases of high-level coding decisions do, however, potentially lend themselves to online adaptation.

Parser Generation. Parsing is a common computing task with uses in everything from compilers to web browsers, and is therefore a well understood and researched field. In practice, the average engineer does not write a parser by hand but instead utilizes a parser generation system. Modern parser generators can produce code that implements the parsing logic in several different ways. The main consideration is if the produced code should be a table-driven automation or a recursive-decent parser that mirrors the approach more traditionally utilized in hand-written code. There are a number of strengths and weaknesses to both approaches in the parsing field, but with respect to an adaptation framework such as ARCA the two different techniques serve as ideal candidates for online selection. Theoretically, the table-driven approach will be more sensitive to system L2 cache interference than the recursive-decent implementation. The techniques and contributions of this thesis could therefore potentially be applied to a future hybrid, mixed-implementation high-performance parser.

Cache Tuning in Scientific Applications. In both the theoretical computer science and scientific computing communities, considerable research has been performed investigating tuning code for system cache structures. There is research on tuning scientific kernels specifically for a given set of architectures [19, 9, 37], developing cache oblivious algorithms that perform well without tuning [6, 17, 3], as well as studies comparing the performance of cache oblivious and cache aware techniques [5, 46]. The overall summary of this work is that well tuned cache aware algorithm will perform bet-

ter than a cache oblivious approach, however the cache oblivious approach will perform better when the cache aware algorithm is poorly tuned or wrong in its assumptions. The problem with making assumptions and tuning to a specific cache size increases in importance on multi-threaded systems with shared caches. An application tuned to a specific cache size may degrade in performance when another co-executing application occupies a significant portion of the shared cache.

The following experiment was performed to provide a preliminary evaluation of this problem as well as attempt to address the problem using the techniques in this thesis. The experiment consisted of using the ARCA framework to create a hybrid binary for a timing benchmark from the Automatically Tuned Linear Algebra Software (ATLAS) project [44, 1]. The ATLAS project generates architecture-specific tuned versions of a key kernel functions from the Basic Linear Algebra Subprograms (BLAS) framework [26, 16], a framework used by a variety of scientific computing applications that provides standard building blocks for performing basic vector and matrix operations.

The focus on the experiment was on the DGEMM, Double-precision General Matrix Multiply, kernel of ATLAS as DGEMM is one of the most commonly utilized functions out of BLAS. The ATLAS framework performs several levels of code tuning. The main features of the system are that the generated code is designed to be optimally scheduled for the specific architecture in question and is also tuned to best utilize the L1 cache. By default, the system uses a semi-cache oblivious approach to address the L2 cache, but further tuning can be applied to generate L2 cache aware code. To perform the tuning the ATLAS system provides a tool that runs a benchmark several times with different values for the L2 cache tuning parameter and identifies the value that provides the best performance. The system is then recompiled using this best case value. When this tuning tool was used on the Core 2 Duo E6600 processor used throughout this thesis, the tool provided an optimal L2 tuning parameter of 2 MB when the tool was run alone. However, when the tool was run alongside the L2 cache interfering

benchmark used in sections 2.2 and 3.3, the optimal tuning values was reported at 512 KB. Therefore, both tuning parameters were used to generate two separate code versions for the ATLAS DGEMM kernel and the ARCA framework was used to create a hybrid binary containing both versions, with the 2 MB version installed by default. Figure 5.2 shows the normalized execution in both the solo and paired case for the three versions of the timing benchmark: 2 MB version, 512 KB version, and the hybrid version with adaptation enabled.

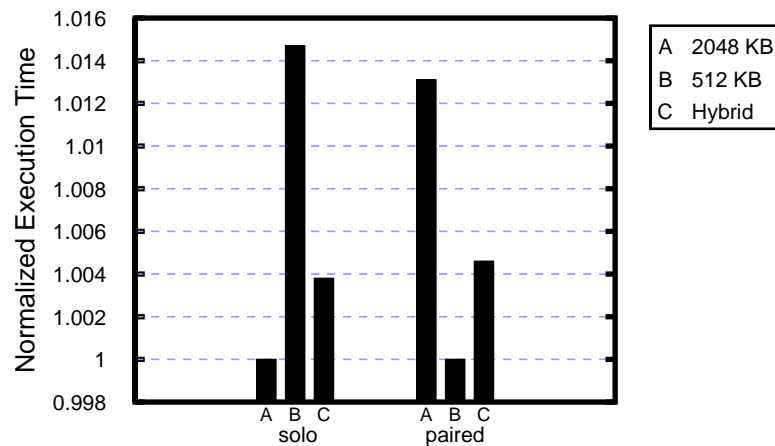


Figure 5.2: The impact of adaptation on an ATLAS DGEMM timing benchmark.

The ATLAS results demonstrate that the two statically tuned versions do indeed perform the best in the cases they were tuned for, while showing degraded performance in the alternate case. On the other hand, the ARCA generated hybrid version with adaptation-enabled performs reasonably well in both cases.

5.3 Emerging Research Areas

This section provides insight into future research topics that are a natural progression of the ideas and contributions presented in this thesis.

5.3.1 Complex Multi-threaded Systems

The entirety of this thesis was evaluated on microprocessors based on the Intel Core 2 micro-architecture. While the Core 2 architecture is the most prevalent consumer-class processor at the time of writing, the design is only mildly interesting as the basis for adaptive selection research. This reality stems from the fact that the only heavily utilized component shared between multiple threads is the L2 cache, which leads to a workload-centric adaptation campaign focused almost exclusively on memory and cache contention issues. Much of the previous work presented in chapter 2 such as OS scheduling methodologies, modeling techniques, and hardware scheduling focused on SMT designs which share many fine-grained structure and therefore lead to a variety of interesting challenges.

Future architectures such as the Intel Nehalem microprocessor, due for release at the end of 2008, utilize both CMP and SMT multi-threading technology in addition to providing Intel's standard, out-of-order execution core. Microprocessors such as those based on Nehalem as well as those coming out from other vendors such as IBM's Power7 architecture and Sun's Rock will be so entrenched in multi-threading that the motivating issues behind this thesis will be potentially more relevant.

Code Scheduling. One potential future area of research is using runtime code selection as a means to augment traditional OS scheduling on complex multi-threaded designs. For example, on a system using both CMP and SMT techniques, the OS scheduler is ultimately responsible for deciding how a workload is split between physical real processing cores and virtual SMT cores. After deciding on a schedule, however, the OS is unable to provide any service aimed at improving the schedule beyond its normal behavior. On the other hand, dynamic code selection could step in and select between two versions of code targeted at different scheduling choices. One version might work best when schedule to a physical core, while another might work better when sharing

a core with a particular type of application. An example might be falling back to non-vector instructions if dynamic feedback demonstrates that the target problem is sharing the core with another application that utilizes vectorized code.

5.3.2 Runtime Adaption for Fault Tolerance

An emerging area of research [39] is currently underway that aims to identify software issues that both increase and decrease an application's susceptibility to transient hardware faults. This research ranges from looking at software algorithmic choices to low-level code-generation decisions. The ARCA framework could be extended to provide a means to select between a set of code versions with known susceptibility characteristics depending on the running workload. For example, at mission critical times of the day the system could select highly tolerant versions which are then replaced by higher performing, but more susceptible versions during non-peak hours.

5.3.3 Heterogeneous, Distributed, and Virtualized Systems

Another trend in modern systems is the move towards distributed and virtualized computing platforms and grid infrastructures. Such environments are yet another area in which a hybrid, adaptable binary could provide additional flexibility and improvement. Consider, for example, a binary that would adapt depending on if it was running directly on a physical node versus running on top of a virtual machine layer. The virtual machine aware code could make use of specialized APIs provided by the virtual machine that provides better performance than native system libraries that have to ultimately be handled by the VM layer anyway. The same holds true for heterogeneous and distributed deployment. On a homogeneous distributed compute cluster, the physical location of resources becomes a factor. Employing different code generation techniques depending on if the code has fast access to a certain resource or slow access, due to being physically distant from the resource, is one such possibility. A prime motivation factor

for the use of an ARCA-style framework stems from the notion that the virtualized and distributed systems of today already support live migration of applications and data. An application can be executing on a given system and transparently moved to another system without being aware of the change. Manually deploying and launching code targeted at different systems become suboptimal once automatic migration and other similar features are employed. ARCA, however, is specifically designed to monitor runtime behavior and adapt a program accordingly. Dealing with these issue would be a natural use of the system.

Chapter 6

Conclusion

The multi-threaded systems of today and tomorrow exhibit dynamically, changing and often times unpredictable workload characteristics that can affect applications in ways not considered at traditional compile-time. This thesis proposed code versioning and adaptive code selection as a means to address this and other emerging issues.

Within the area of code selection, this thesis provided three contributions. The first contribution is a code analysis and classification technique that provides the ability to identify and extract code versions representing unique behaviors out of a large initial version set. The analysis methodology presented correlates multiple dynamic characteristics to the different code versions and then clusters the results using the Euclidean distance between the desired evaluation metrics.

The second contribution is a set of three dynamic adaptation algorithms that can be used in order to determine how to select between the various code versions at run time. The three techniques presented – greedy, cooperative, and fractional – have different strengths and weaknesses and are each applicable to different adaptation goals.

The third contribution is a demonstration that a unified framework can be built that addresses multiple problem domains and adaptation goals. The key features needed to design such a framework are as follows: a flexible analysis and selection technique such the first contribution discussed above; a domain agnostic adaptation approach such as the second contribution discussed above; a scalable runtime system; and an abstract

dynamic monitoring framework such as the scoreboard approach utilized in ARCA and presented in section 3.1.2.

This thesis also presented two case studies showcasing the ARCA framework from beginning to end when applied to two different goals: improving system behavior on a multi-threaded machine, and compromising between power and performance for a single binary. Finally, a discussion along with two preliminary experiments were provided that suggest that future research can build upon the contributions of this thesis in order to address a number of emerging issues.

Bibliography

- [1] Automatically Tuned Linear Algebra Software (ATLAS). Web site: <http://math-atlas.sourceforge.net/>.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, pages 1–12, June 2000.
- [3] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, page 399, 2000.
- [4] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In Proceedings of the Workshop on Profile and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques, October 1998.
- [5] Gerth Stolting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, pages 307–315, 2003.
- [6] Gerth Stolting Brodal, Rolf Fagerberg, and Kristoffer Vinther. Engineering a cache-oblivious sorting algorithm. J. Exp. Algorithmics, 12:2.2, 2007.
- [7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In CGO '03: Proceedings of the international symposium on Code generation and optimization, pages 265–275. IEEE Computer Society, 2003.
- [8] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. Software Practice and Experience, 21(12):1301–1321, December 1991.
- [9] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In CGO '05: Proceedings of the international symposium on Code generation and optimization, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

- [10] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3), December 2000.
- [11] B. Cheng. A profile-driven automatic inliner for the impact compiler. Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.
- [12] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. Fx!32: A profile-directed binary translator. IEEE Micro, 18(2):56–64, 1998.
- [13] James C. Dehnert, Brian Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing - software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In Proceedings of the 1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA, March 2003.
- [14] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. DELI: A new run-time control point. In 35th Annual International Symposium on Microarchitecture, December 2003.
- [15] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pages 71–84, 1997.
- [16] J. Dongarra. Basic linear algebra subprograms technical forum standard. 16(1):1–111, 2002.
- [17] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, page 285, 1999.
- [18] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005).
- [19] Bruce Greer. The most important technical library in the world. SIGPLAN Fortran Forum, 17(3):25–32, 1998.
- [20] IBM. PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual, 1999.
- [21] Intel Corporation. Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization, 2004.
- [22] Joshua Kihm. Simulation and Characterization of Inter-Process Interference on Multithreaded and Multicore Architectures. PhD thesis, Electrical and Computer Engineering, University of Colorado, Boulder, CO.

- [23] Joshua Kihm and Daniel Connors. Implementation of fine-grained cache monitoring for improved smt scheduling. In Proceedings of The 22nd International Conference on Computer Design, 2004.
- [24] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparcc processor. IEEE Micro, 25(2):21–29, 2005.
- [25] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. In PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pages 239–251, 2006.
- [26] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. ACM Transactions on Mathematical Software, 5(3):308–323, 1979.
- [27] David Levinthal. Cycle Account Analysis on Intel Core 2 Processors. Technical report, Intel Corporation.
- [28] Jack Lo, Susan Eggers, Henry Levy, Sujay Parekh, and Dean Tullsen. Tuning compiler optimizations for simultaneous multithreading. In Proceedings of the 30th Annual International Symposium on Microarchitecture, pages 114–124, December 1997.
- [29] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. In Journal of Instruction-Level Parallelism 6(2004), pages 1–24, April 2004.
- [30] C-K. Luk and et al. Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, June 2005.
- [31] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread titanium processor. IEEE Micro, 25(2):10–20, 2005.
- [32] Tipp Moseley. Adaptive Thread Scheduling for Simultaneous Multithreading Processors. PhD thesis, Computer Science, University of Colorado, Boulder, CO.
- [33] Tipp Moseley, Joshua Kihm, Dan Connors, and Dirk Grunwald. Methods for modeling resource contention on simultaneous multithreading processors. In International Conference on Computer Design, 2005.
- [34] A. P. Nisbet. GAPS: Iterative feedback directed parallelisation using genetic algorithms. In Proceedings of the Workshop on Profile and Feedback-Directed Compilation, Paris, France, October 1998.
- [35] OProfile. Web site: <http://oprofile.sourceforge.net/>.
- [36] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic program optimization. Technical Report CRHC-99-16, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, December 1999.

- [37] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pages 115–126, 2005.
- [38] Alex Settle, Josh Kihm, Andy Janiszewski, and Daniel A. Connors. Architectural support for enhanced smt job scheduling. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, 2004.
- [39] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In Proceedings of the International Conference on Dependable Systems and Networks, June 2007.
- [40] Brinkley Sprunt. Pentium 4 performance-monitoring features. In IEEE Micro 22(4), pages 72–82, 2002.
- [41] D.M. Tullsen, S.J. Eggers, J. Emer, H.M. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In Proceedings of the 23rd International Symposium on Computer Architecture, May 1996.
- [42] Michael J. Voss and Rudolf Eigemann. High-level adaptive program optimization with adapt. In PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, pages 93–102, 2001.
- [43] O. Wechsler. Inside Intel Core Microarchitecture: Setting New Standards for Energy-Efficient Performance. Technology@Intel Magazine, 2006.
- [44] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience, 35(2):101–121, February 2005. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>.
- [45] Chaohao Xu, Jianhui Li, Tao Bao, Yun Wang, and Bo Huang. Metadata driven memory optimizations in dynamic binary translator. In VEE '07: Proceedings of the 3rd international conference on Virtual execution environments, pages 148–157, New York, NY, USA, 2007. ACM.
- [46] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pages 93–104, 2007.
- [47] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multi-threaded dynamic optimization framework. In PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.