Loaf: A Framework and Infrastructure for Creating Online Adaptive Solutions

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

ABSTRACT
Achieving effective online adaptation for natively executed applications has proved quite challenging and to date has not been widely adopted. Traditionally, to enable online adaptation for native binary applications, a run-time layer is added that virtualizes the execution of the application by performing dynamic binary to binary translation. This virtual layer injects trampolines and instrumentation into the translated code to maintain control of the application. This approach adds significant overhead and complexity to the application, discouraging its use for online adaptation in commercial deployments and particularly in the modern data-center computing domain. In this work we present a new lightweight paradigm for online adaptation that leverages current microarchitectural advances to efficiently enable online monitoring and adaptation without the complexity of binary translation or fine-grain instrumentation. Our methodology takes advantage of the ubiquitous hardware performance monitors present in modern chip microarchitectures to dynamically monitor micro-architectural events and application behavior with negligible overhead. By leveraging these capabilities to develop an innovative lightweight online adaptation framework (Loaf) we are be able to address a number of important real-world online adaptation problems.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming—parallel programming; D.3.4 [Programming Languages]: Processors—run-time environments, compilers, optimization, debuggers; D.4.8 [Operating Systems]: Performance—measurements, monitors

General Terms
Performance, Design, Algorithms, Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EXADAPT ’11 June 5, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0708-6/11/06 ...$10.00.

1. INTRODUCTION
Traditionally, an application program is written by a programmer, then statically compiled to a binary executable file composed of instructions from a targeted instruction set architecture (ISA). This binary can then be run on a range of microarchitectures that conforms to that ISA. The structure and layout of the binary code is determined statically, and consequently, it remains ridged across inputs, microarchitectures and execution environments. It is well known that semantically equivalent variations in the code structure and layout can cause a wide range of variance in its performance and other properties [60, 22]. As a result, programmers and optimizing compilers are faced with the task of statically determining the optimal code layout and structure for application binaries. However, these code structure and layout decisions are impacted by changes in application input, system microarchitectures and execution environment. These execution contexts can change across application runs, and indeed during a single run. Figure 1 shows three execution contexts of a single application: in the first context the application runs alone on a single core machine, in the second context the same binary runs on a multicore machine, and in the third context the application’s execution environment is includes three other co-running processes on a quad core machine. The applications optimal code layout and execution behavior may vary across these contexts. To allow this flexibility, online adaptation techniques must be used.
To perform online adaptation, information that is only available at runtime must be used to restructure the application’s execution, its environment, or both. Online adaptation is composed of two key tasks. First, the application’s execution or execution environment must be monitored as it executes. Second, when a particular runtime characteristic or event is observed, the application’s execution or its environment is then restructured or adapted in some way to accommodate this behavior. To perform these two key tasks, online adaptation approaches require a runtime component to be present and executed in tandem with the host application.

However achieving online adaptation for native applications has proved quite challenging. The runtime layer necessary to perform the monitoring and dynamic restructuring of the binary application increases the amount of work required to execute the application. The benefit of adding runtime online optimization or adaptation must outweigh the penalty suffered from the added complexity. Effectively achieving online adaptation at the binary level has proved difficult and has, by-in-large, not been adopted for practical use in current industry and commercial domains. Current techniques fall into two categories: heavyweight approaches that provide too little benefit for the added complexity and approaches that propose novel hardware changes and are unable to be realized on current chip architectures.

In this work, we present a new paradigm for achieving online adaptation at the binary level that uses what we call lightweight introspection. Introspection is a heavy-weight online adaptation technique that requires either instrumentation of the host application to enable monitoring or the dynamic translation of the application’s binary instructions. A lightweight online adaptation technique uses no instrumentation and performs no binary to binary online translation. Our lightweight introspection approach takes full advantage of the performance monitoring hardware features that are ubiquitous in current micro-architectural design to perform all online monitoring with negligible overhead and minimal added software complexity. Using a technique we call periodic probing, monitoring is performed by taking snapshots of the hardware performance monitors using timer interrupts. In this work, we take advantage of this introspection methodology to design Loaf, the Lightweight Online Adaptation Framework. To effectively achieve online optimization on current microarchitectures, Loaf enables 1) the monitoring of the application and execution environment, 2) the dynamic restructuring of application code, and 3) the cooperative adaptation of the co-runners in an applications execution environment.

We then present two case studies showing how the Loaf infrastructure was used to design and construct practical lightweight adaptive solutions to two pressing problems in our field. The first problem is that of aggressive optimizations. These are optimizations that are risky, as they can significantly improve or degrade performance. As shown in previous work, the effect of applying this class of optimizations cannot be predicted statically, as it may depend on input size, micro-architectural events, and execution environment. In this work we show how the Loaf infrastructure is used to enact an adaptation policy to dynamically apply aggressive optimization only when there is benefit.

For the second case study we present how Loaf is used to adapt the environment to the application. The problem addressed in this case study is that of cross-core application interference. Contention for shared resources and cross-core application interference due to contention, pose a significant challenge to providing application level quality of service (QoS) guarantees on commodity multicore microarchitectures. The commonly used solution is to simply disallow the co-location of latency-sensitive applications and throughput-oriented batch applications on a single chip, sacrificing utilization. In this work, we show how to use Loaf’s ability to cooperatively adapt co-running applications to design an agnostic contention aware execution environment that will adapt an application’s environment to minimize cross-core interference due to contention, while maximizing chip utilization.

The main contributions of this work are:

- The design of Loaf, a holistic lightweight online adaptation framework for native binary applications that is both able to adapt an application to its execution environment, and also the execution environment to the application.
- The design and discussion of periodic probing for lightweight introspection, scenario based multiversioning for online code restructuring, and cross core application cooperation for coordinated adaptation across cores.

Two case studies demonstrating the process of leveraging the Loaf infrastructure to address real problems and the effectiveness of these solutions.

It is important to note that the primary focus of this paper is the Loaf framework and infrastructure. The details of the individual adaptation policies used for the two case studies have been published in our prior work. Loaf is a single general platform for the online adaptation for native applications that enables both the restructuring of the application code online without the complexity of performing any online code generation or binary rewriting, and the ability to cooperatively restructure the application’s co-runners to enact coordinated online adaptation policies. To the best of our knowledge there is currently no such holistic software-only general online adaptation framework with these capabilities.

2. LOAF OVERVIEW

In this section we first discuss the required functionality of Loaf, and then discuss our lightweight approaches to achieve this functionality.

2.1 Enabling Online Adaptation

In order to effectively enable online adaptation, Loaf must provide the capability to monitor online events and adapt the application or its environment to these events. To achieve these capabilities we have three key functionality requirements for Loaf which includes:

1. An efficient mechanism for the online monitoring of the application or its environment’s behavior.

2. An efficient mechanism to allow the dynamic restructuring of application code. One of the key methods used to adapt application behavior is to allow code restructuring in response to dynamic events.
3. An efficient mechanism to enable the adaptation of multiple co-running applications and threads in an application’s execution environment. Multicore architectures are ubiquitous in today’s computing environment, and an application can be affected by its simultaneously executing co-runners.

The underlying philosophy of our online adaptation approach is to achieve efficiency by remaining as lightweight as possible. Therefore, to achieve the tasks of online adaptation, observation and adaptation, with minimal application interference we use the following approaches to the three design goals mentioned above:

1. To achieve online monitoring, we use the lightweight approach of periodically probing the hardware performance monitors on current microarchitectures. We call this approach lightweight introspection.

2. To achieve the dynamic restructuring of application code, we use the lightweight approach of statically providing multiple code versions for regions of interest and allowing dynamic switching based on online monitoring. We call this approach scenario based multiversioning.

3. To accommodate adaptation of the application and its environment based on events that occur due to simultaneous co-scheduling on current multicore architectures, we use the lightweight approach of sharing dynamic monitoring information across cores using a shared communication table, allowing multiple threads to cooperate during online adaptation. We call this approach cross-core application cooperation.

With a particular adaptation policy in mind, the user can leverage Loaf’s API in each of these components to enact the policy.

2.1.1 Online Monitoring

To achieve the necessary task of efficient online monitoring we use lightweight introspection as shown in Figure 2. The core intuition of this approach is to remain lightweight by leveraging periodic probing with the usage of hardware performance monitors. These hardware performance monitors provide realtime micro-architectural information about the applications currently running on chip. As the counters record this information, the program executes uninterupted, and thus recording this online profiling information presents no instrumentation overhead. These capabilities can be leveraged with one of the many software APIs, such as PAPI [35] or Perfmon2 [21]. In this work, we use Perfmon2 as it is one of the most robust and flexible PMU interfaces, and supports a wide range of micro-architectures.

The self introspection run-time employs a periodic probing approach, meaning information is gathered and analyzed intermittently. Using a timer interrupt the environment will periodically read the performance monitoring hardware, re-set the timer, and restart the performance monitoring counters. The algorithms that comprise our lightweight introspection engine is shown in Algorithms 1 and 2.

**Algorithm 1: Loaf LIE Initialization**

```plaintext
events_of_interest ← user_defined_events;
probe_interval ← user_defined_interval

foreach e in events_of_interest do
    active_counters ← PMUConfigure(e)
end

foreach c in active_counters do
    PMUBeginCounting(e)
end

IssueTimerInterrupt(probe_interval)
```

Figure 2 illustrates how a user of the Loaf infrastructure interacts with Loaf. Each of the three components mentioned above corresponds to the numbers in Figure 2 respectively. The blue sections of each component denotes the locations a user must touch to implement the desired adaptation policy. An adaptation policy is a specification of a desired response to some dynamic event or set of events.
Periodic probing is an efficient method for collecting information from hardware performance monitors. The overhead of this technique is determined by two factors: the frequency of probes (e.g., interrupts), and the complexity of the analysis and adaptation work done during those interrupts. These two factors are impacted by the nature of the desired adaptation policy. For the policies implemented in our case studies, the probe\_interval used is one every millisecond and the resulting overhead is negligible.

It's important to note that hardware performance counters are a ubiquitous hardware feature and has been used in prior works. There has been a significant amount of prior work discussing and using performance counters for particular applications such as selecting optimizations [10], enhancing operating systems [33], and in Java virtual machines [19] among others. However, in this work, we present a general online adaptation framework for native binaries that leverages hardware performance monitors exclusively to provide lightweight monitoring and introspection runtime that facilitates both the online adaptation of an application to its environment in addition to adapting the environment to the application itself.

Our Loaf lightweight introspection runtime serves as the core mechanism for the monitoring of application behavior, and the execution environment. The runtime can be attached to a host application in a number of ways including statically linking the run-time module into the application binary, dynamically linking in as a module, or as a third party virtual application host such as gdb. In this work we use the static linking approach.

### 2.1.2 Adapting the Application

To achieve the dynamic restructuring of application code as execution occurs we use a multiversioning technique we call scenario based multiversioning (SBM) as shown in Figure 2.2. The key insights of this scenario based multiversioning is to enable compiler writers to statically accommodate the various run-time scenarios and situations an application may face. Traditionally, static code layout and structure is rigid regardless of changes in its execution environment or application phase, this is exactly what the scenario based multiversioning is equipped to address. Using SBM, compiler writers can apply various optimizations and code layouts across multiple instances (or versions) of a code region, each specialized to particular dynamic situations and events. By taking advantage of these static compile-time optimization capabilities with lightweight introspection, execution can be rerouted dynamically to execute the desired code regions. In addition, SBM retains all of the capabilities and advantages of static compilation, such as the availability of high-level source information, while achieving run-time flexibility.

SBM performs its multiversioning at the function level allowing the generation of specialized versions of a function to target different scenarios. As discussed in Section 5 function level multiversioning is a classic inter-procedural code transformation that has proven quite useful by numerous other compiler techniques and optimizations. However to achieve Loaf’s online code adaptation, traditional multiversioning must be extended to be driven by hardware performance counters via the lightweight introspection engine and provide a general platform able to accommodate a variety of adaptation policies. For SBM we provide an interface between the static binary and a lightweight introspection runtime component. This interface will allow the introspection engine to hook into the executing binary and reroute the execution via resetting the active versions of the functions. To accomplish this we have two designs.

We call the first design the alternate versioning scheme and the second the n-version versioning scheme. While both techniques requires the use of a trampoline as the multiplexing mechanism, there are differences. Figure 2.2 shows the alternate version scheme. For this scheme, we have a default and alternate version of particular functions. With the alternate version scheme there is a single global switch that the dynamic component interfaces to control which version the application uses. With this scheme the entire binary will either execute the default versions for all multiversioned functions or the alternative version. This provides a simple abstraction that a compiler writer can use to design SBM based techniques that do not require too much complexity. Figure 2.2 shows the design of the n-version versioning scheme. This scheme allows for any number of versions for any function and individual version switching. For this scheme, we maintain a global mapping table in memory for each function. Instead of a global switch, each call to a multiversioned function is transformed to an indirect call. During execution, the target address of the call is controlled by the dynamic component and any combination of versions can be active at anytime. This will allow for much more complex SBM techniques where multiple scenarios can occur at the same time.

One important consideration is that we cannot have multiple versions of every function in our application binary. This would cause an unacceptable amount of code growth, which would limit the applicability of SBM and ultimately have a negative impact on application performance. Therefore we limit the number of functions we multiversion to only the hottest functions in the application. To efficiently multiversion our application we take advantage of some basic profiling that has proven useful for determining the hottest code in an application [15][36][37]. SBM can use the simple pro-

---

**Algorithm 2: Loaf Periodic Probes**

```plaintext
PMUStopCounters();
foreach e in events_of_interest do
  e.value ← PMUReadCounter(e)
end
DoAdaptationAnalysis();
Adapt();
if new_events_of_interest then
  events_of_interest ← new_events_of_interest
end
if new_probe_interval then
  probe_interval ← user_defined_interval
end
foreach e in events_of_interest do
  active_counters ← PMUConfigure(e)
end
foreach c in active_counters do
  PMUBeginCounting(c)
end
IssueTimerInterrupt(probe_interval)
```
3. CREATING SOLUTIONS WITH LOAF

In this section, we demonstrate the practicality and effectiveness of our lightweight online optimization framework (Loaf), by showing how it is used to construct solutions to address two important pressing problems: one where the application must adapt to its environment, and the other where the environment must adapt to the application. First we use Loaf’s scenario based multiversioning to dynamically apply aggressive application optimizations to eliminate potential degradations and gain better application performance. Second we use Loaf’s cross-core application cooperation to dynamically detect contention on multicore chips and respond by adapting the applications environment to minimize the performance degradation due to cross-core interference while maximizing chip-wide utilization. The primary focus of this paper is the Loaf framework and infrastructure. The details of the individual adaptation policies used for these two case studies can be found in our prior work [38, 40].

3.1 Aggressive Optimizations Using Scenarios

To demonstrate Loaf’s ability to adapt the application to its environment we address the problem of aggressive optimizations. Many optimizations show benefit in some cases and a degradation in others [60]. We call these optimization aggressive optimizations. To address this problem, we use Loaf’s lightweight introspection and scenario based multiversioning to construct a technique we call Online Aiding and Abetting of Aggressive Optimizations (OAAAO). The intuition is that we should be able to detect the scenarios where aggressive optimizations are beneficial or not and react accordingly.

3.1.1 Problem Description

Aggressive optimizations may increase performance in some contexts and decrease performance in others. For our OAAAO approach we use two such optimizations, software cache prefetching and loop unrolling. These optimization heuristics are found in GCC as optional optimizations and, as shown in previous work [60] [22], the effect of applying this class of optimizations cannot be predicted statically, as it may depend on input size, micro-architectural events, and execution environment. Our hypothesis is that lightweight introspection and scenario based multiversioning should be able to improve the performance of these aggressive optimizations. Using lightweight introspection we detect the scenarios when aggressive optimizations are improving or degrading performance. We can then reroute execution accordingly; using our scenario based multiversioning mechanism.

3.1.2 Adaptation Policy for Aggressive Optimizations

![Figure 4: This represents the three phase execution approach of OAAAO.](image)

The adaptation policy for the online aiding and abetting of aggressive optimizations (OAAAO), uses the alternate versioning scheme mentioned with an online three phase analy-
sis. Statically we generate code for two scenarios. First, we generate code without software prefetching or loop unrolling for the scenario that aggressive optimizations would degrade performance; we call this the non-aggressive version. Then, for the scenario that aggressive optimizations would improve performance, we generate code for that same function that has software prefetching and loop unrolling; we call this the aggressive version.

The dynamic component of our OAAAO approach has three phases as shown in Figure 4. The first two phases compose the learning and monitoring part of OAAAO, and the third phase composes the action part of OAAAO. During execution these phases continually loop until the host application terminates. The full details of this heuristic are presented in our prior work [38].

3.2 Addressing Cross-Core Interference Using Loaf

To demonstrate Loaf’s ability to adapt the environment to the application we address the problem of contention for shared resources on chip. Current multicore chip design in commodity hardware is composed of private and shared caches. For example, Intel’s Core 2 Duo architecture has 2 cores, each with a private L1 cache and a single L2 of 4mb shared between the two cores, Intel’s new Core i7 (Nehalem) architecture has 4 cores, each with private L1 and L2 caches and a single 8mb shared L3 cache for all 4 cores [28]. The shared last level cache presents the first level of possible contention and can cause significant cross core performance interference. For latency-sensitive applications, and other QoS requirements, this cross-core interference is not acceptable. We take advantage of Loaf’s lightweight introspection and cross-core application cooperation to detect and respond to cross core performance interference as it happens dynamically. We call this online adaptation approach the contention aware execution run-time (CAER).

3.2.1 Problem Description

When more than one application is using the shared last level of cache heavily, and the data is not shared, contention occurs. One way to address this problem is to increase the size and associativity of the cache, and although cache sizes have been increasing with every generation of processors, they still remain far behind the demands of today’s application workloads. When simply co-locating two SPEC2006 applications on state-of-the-art quad core chips, we often see a performance degradation exceeding 30% and up to 2x overhead.

Application priority and quality of service requirements often cannot withstand unexpected cross-core interference. For example, applications commonly found in the web service data center domain such as search, maps, image search, email and other user facing web applications are latency-sensitive. These applications must respond to the user with minimal latency, as having high latency discourages the user.

Data centers for web services classify applications as either being latency-sensitive or as throughput-oriented batch applications, where latency is not important. To avoid cross-core interference between latency-sensitive and batch applications, web service companies simply disallow the co-location of these applications on a single multicore CPU. Using this solution may leave the CPU severely underutilized, and is a contributing reason to the server utilization of these data centers often being 15% or less [34]. Low utilization results in wasted power and lost cost saving opportunities. We use Loaf’s lightweight introspection and cross-core application cooperation to create a contention aware execution run-time (CAER) to detect and respond to contention to minimize cross-core interference and maximize utilization.

3.2.2 Adaptation Policy for Contention Detection

Figure 5 presents our CAER approach. As shown in the figure, before CAER can react to contention in the shared cache, it must first detect that the applications are indeed contending. We have developed two heuristics for this detection task, a burst shutter approach, and a rule based approach. Both heuristics run continuously throughout the lifetime of CAER to detect and respond to contention using Loaf’s periodic probing. The full details of these heuristics are presented in our prior work [19].

As Figure 6 shows after detecting contention we transition into one of the response states, either -negative or c-positive. In these states the CAER run-time environment can respond by dynamically modifying and adapting the batch application under which it runs. CAER reacts to contention by enforcing a fine grained throttling of the execution of the batch application to relieve pressure in the shared cache. An example of this is a red-light green-light approach which, as the name implies, stops or allows execution for a fixed or adaptive number of periods based on the outcome of our contention detection phase.

4. EVALUATING LOAF

We have presented the design and implementation of the Loaf online adaptation framework. However, Loaf is effective only if it can be successfully used to address important problems by both allowing the adaptation of the application to its environment, and the adaptation of the environment to the application. We evaluate Loaf for the tasks of Online Application of Aggressive Optimizations and Contention Aware Execution on current commodity multicore processors with the SPEC2006 benchmark suite as follows:

- **Online Application of Aggressive Optimizations**: To evaluate the effectiveness of using Loaf for the online application of aggressive optimization we demonstrate how Loaf can be harnessed to implement the application adaptation policy and evaluate the resulting overall improvement of application performance. That
is if our application with our online adaptation based solution performs better than statically applying aggressive optimizations we conclude that our Loaf-based approach is effective.

- **Contention Aware Execution Run-time**: The goal of our contention aware execution run-time environment built on Loaf is to both maximize utilization and minimize the performance penalty from cross core application interference for latency sensitive applications. As we discuss in Section 3.3 simply disallowing the collocation of latency sensitive applications and other applications gives full performance isolation but eliminates all utilization provided by neighboring cores. Simply allowing the collocation gives full utilization of neighboring cores however causes the maximum penalty from cross-core interference. If our Loaf-based approach allows us to have a very small performance penalty for significant utilization gain we consider it effective.

### 4.1 Effectiveness of Loaf-based Solutions

In this Section we present key results from these two applications of the Loaf framework. First we present results from leveraging Loaf’s scenario based multiversioning for the online application of aggressive optimizations. We then present results of our contention aware execution run-time environment based on Loaf’s cross-core cooperation architecture.

#### 4.1.1 Using SBM for Aggressive Optimizations

The goal of our OAAAO optimizations is to eliminate the degradations of aggressive optimization while reaping the benefits. We also hypothesized that we would be able to exceed the potential benefits of applying and using aggressive optimizations statically. All of our experiments were performed on a machine with the Intel Core 2 Quad 6600 architecture and 2gb of ram. We used a selection of benchmarks from the SPEC2006 v1.1 suite and ran them on their reference inputs to completion. We used the GCC 4.3.1 compiler to compile these benchmarks. The benchmarks were all compiled with optimization level -O2, and tuned to the Core 2 architecture (compiler option -march=core2). All experiments were run on Ubuntu Linux Kernel 2.6.25 patched with Perfmon2.

The fourth bar in Figure 6 shows the impact on execution time when applying aggressive optimizations with and without the Online Aiding and Abetting of Aggressive Optimizations. As the data in Figure 6 shows, only when OAAAO is applied do we see performance improvements with exception of gobmk where the degradations are effectively eliminated. In addition to eliminating the degradations and leaving only performance improvement, in the large majority of the benchmarks the performance improvements significantly exceed those produced by any combination of aggressive optimization without OAAAO. In 9 out of the 12 benchmarks presented OAAAO exceeds the benefit of aggressive optimizations, in most cases more than doubling the performance boost.

#### 4.1.2 Using CCAC to Detect and Respond to Contention

We aim to minimize the interference penalty (overhead of the latency-sensitive application due to contention) and maximize the utilization of the chip. We demonstrate the effectiveness of our contention aware execution run-time (CAER) environment by showing a considerable reduction in this interference penalty when allowing co-location, while achieving a significant increase of chip utilization (60% on average) compared to disallowing co-location.

Our CAER prototype supports two applications, one deemed latency-sensitive and the other a throughput-oriented batch application. We use the SPEC2006 benchmark (C/C++ only) and run all to completion using their reference inputs. We use the Intel Core i7 (Nehalem) 920 Quad Core architecture to perform our experimentation. This processor has three levels of cache: the first two private to each core, the third shared across all cores. The sizes of the L1 and L2 caches are 16kb and 256kb respectively. The L3 cache is 8mb and inclusive to the L1 and L2. The system used has 4gb of main memory, and runs Linux 2.6.29.

In the experiment shown here, the lbm benchmark served as our batch application and was co-located on a neighboring core. The main benchmark is assumed to be the latency-sensitive application. Lbm was chosen as our batch application because it presents an interesting adversary as it makes heavy usage of the L3 cache.

We evaluate the reduction in the interference penalty due to contention when running on our CAER environment.
Figure 7: Investigating the reduction in interference penalty.

Figure 7 shows the slowdown in execution time due to contention when we co-locate the latency-sensitive and batch applications. The first bars show the interference penalty when co-locating the native applications directly on a multicores. The second bars show the interference penalty when co-locating the native applications on CAER with the burst shutter heuristic. The last bars show this co-location on CAER with the rule based approach.

As Figure 7 shows we significantly reduce the cross-core interference penalty for the wide range of SPEC2006 benchmarks. Our burst shutter contention detection technique uses the red-light green-light response with a response length of 10 periods. The impact threshold in for the burst shutter detection is set to 5%, meaning if the batch application burst causes a spike of 5% or more in last level cache misses of the latency-sensitive application we assert contention. Using this approach CAER brings the overhead due to contention from 17% down to 6% on average, while gaining close to 60% more utilization of the processor over running the latency-sensitive application alone.

5. RELATED WORK

Current online dynamic optimization approaches can be separated into two categories: those that deal with managed run-time systems targeting bytecode and those that apply to native application binaries. The majority of online optimization frameworks that target bytecode work at the function granularity. These optimizers detect frequently executed methods and identify them as hot. These hot methods are then just-in-time compiled and recompiled at higher levels of optimization, depending on how often they are executed. Other bytecode online optimization and adaptation approaches address memory and other issues. However this work is concerned more with online adaptation at the binary level.

5.1 Online Adaptation at the Binary Level

This work deals with the class of online optimizers and optimization frameworks that deal with native binaries directly such as Dynamo, DynamoRIO, and Strata. These current dynamic optimization techniques have had limited success. One of the seminal works that has inspired many future projects was the work by Bala et al. on Dynamo. Dynamo is a binary to binary translator and dynamic optimizer that works at the basic block and trace levels. Dynamo was the only online optimizer of its class to achieve consistent performance gains. This has mostly been attributed to the intricacies of the PA-RISC platform for which it was implemented. Attempts have been made to reproduce this performance benefit on other architectures but have been largely unsuccessful. Bruening et al. reimplemented the Dynamo infrastructure for x86 with the DynamoRio project and was unable to achieve significant improvement. A similar effort was made with the Strata infrastructure and was also unable to achieve performance gains. One major challenge these three approaches face is the added overhead from virtualizing the application and maintaining control of the executing binary. In fact there has been much work focused on optimizing the dynamic optimizer itself, in particular the handling of indirect branches.

Research attention has also been paid to online optimization approaches using multcore architecture and novel hardware techniques. The Adore infrastructure has been used by Lu et al. to achieve dynamic software prefetching via the use of helper threads and performance monitoring hardware. A similar technique was also later applied to SUN’s UltraSparc Architecture. Zhang et al. proposed Trident, a new dynamic optimizer framework that requires hardware support. This work proposes that trace selection occurs entirely in hardware and uses a number of hardware extensions. This work shows promising potential, but currently cannot be applied as it depends on novel micro-architectural features to be developed.

5.2 Extracting Run-time Information

The usefulness of information about an application’s runtime behavior and dynamic micro-architectural impact has also shown to be quite important. Profiling has become the cornerstone for understanding an application’s behavior and can play an important part in compiler optimizations as shown in the work by Chang et al. This seminal work introduces compiler support for profile feedback directed compiler optimizations. The compiler executes the application on a number of canned inputs, profiles it, and recompiles the application using this information. Using profiling information has lead to many new kinds of optimizations. However these compiler optimizations remain rigid and thus tends to be applied conservatively.
Performance counters have shown to be a great tool to enable low overhead profiling of micro-architectural events. Moreover, these hardware structures are becoming more complex as is seen in the work by Dean et al. [19]. Azimi et al. presents a technique to use limited performance counters to simultaneously profile numerous events via sampling [6]. In recent work by Cavazos et al. [11], performance counters and machine learning are used together to find better compiler optimization settings for applications. These performance counters are also being used for more than just profiling. In the works by Chen et al. [15] and our prior work [39], performance monitoring hardware are used to form dynamic hot traces without slowing down the running application. We also see performance counters used in Java VMs and JITs to steer optimization in the works by Schneider et al. [50] and Adl-Tabatabai et al. [1].

5.3 Function Cloning and Versioning

Function cloning and Multiversioning is an inter-procedural code transformation that is used by a number of optimizations dating back to the earliest works on compiler optimization [5, 15, 5, 17, 20, 53]. It was originally conceived for classic optimizations such as inter-procedural constant propagation (IPCP) [8]. It has also been used by Carini et al. for flow insensitive IPCP [9] and Ciernia et al. for inter-procedural array remapping [17].

Multiversioning approaches have also been used by Diniz et al. [20] and Voss et al. [55]. In the work by Diniz et al. multiversioning is used in the context of a parallelizing compiler for object-based languages to provide a mechanism to dynamically switch the implementation of a particular language. This work presented a technique to use limited performance counters to present information to steer version switching dynamically. Although the concept of dynamic feedback is discussed in this work, a general mechanism to achieve online code adaptation using multiversioning is not explored. In addition, the mechanisms used in this work to gather information to steer version switching is significantly limited in comparison to the scenario based multiversioning approach presented in this work. The multiversioning approach provided by Diniz’s dynamic feedback relies entirely on a timing approach, only allowing for variants of the sampling/production phase heuristic presented in their work. However our scenario based multiversioning provides much more general monitoring capabilities in that our approach is guided by the ability to identify scenarios based on a collection of the information available through our lightweight introspection interface. In addition the idea of having a number of co-running application adapting in cooperation is also not explored.

The work by Voss et al. [55] discusses the idea of switching regions of executing code dynamically, however multiversioning is not performed statically. Versions are generated continuously by compiler tools and optimizers running on sockets and machines separate to the executing applications. In addition users of their system must learn a new domain specific language, and the flexibility of potential adaptation policies is limited to what can be expressed in this language. Requiring this new language presents a significant amount of complexity, which is contrary to the goal of Loaf. Also the fact that a new language must be learned to use their system may further deter users from adopting this approach.

More recently multiversioning has been used in a number of works by Pursin et al. as a mechanism to provide dynamic and lightweight approach to enable online code adaptation. Our online adaptation framework is lightweight, practical and flexible. It can be used to enable online code adaptation. In this work, we present the design and implementation of an efficient online adaptation framework capable of addressing many of these dynamically occurring problems. Our online adaptation framework is lightweight, practical and flexible. It can be used to enable online adaptation solutions. In this work, we have described in detail three key lightweight approaches to achieve online adaptation, lightweight introspection, scenario based multiversioning, and cross-core application cooperation. In addition, we use our online adaptation framework to provide novel solutions to two pressing problems using each of our lightweight approaches.

7. REFERENCES


