# 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 datacenter 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 micro-architectures 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

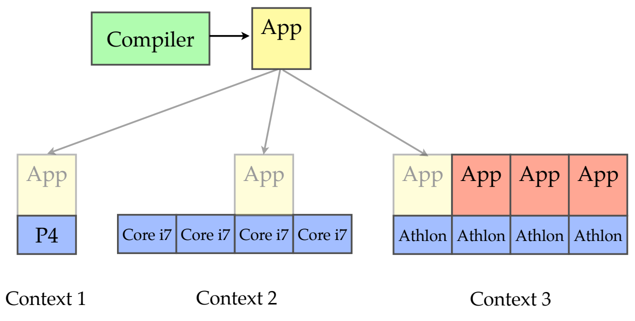

**Figure 1: A Single *Rigid* Binary Executing in Three Contexts**

## Keywords

cross-core interference, profiling framework, program understanding

## 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 micro-architectures that conforms to that ISA. The structure and layout of the binary code is determined statically, and consequently, it remains ridged across inputs, micro-architectures 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 micro-architectures 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 run-time 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 run-time 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 run-time 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 run-time 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*. In contrast to a *heavyweight* online adaptation technique that requires either instrumentation of the host application to enable monitoring or the dynamic translation of the applications 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 [28] 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 *lightweight* online adaptation 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 corunners 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 [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. 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 [38, 40].

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 corunners 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.

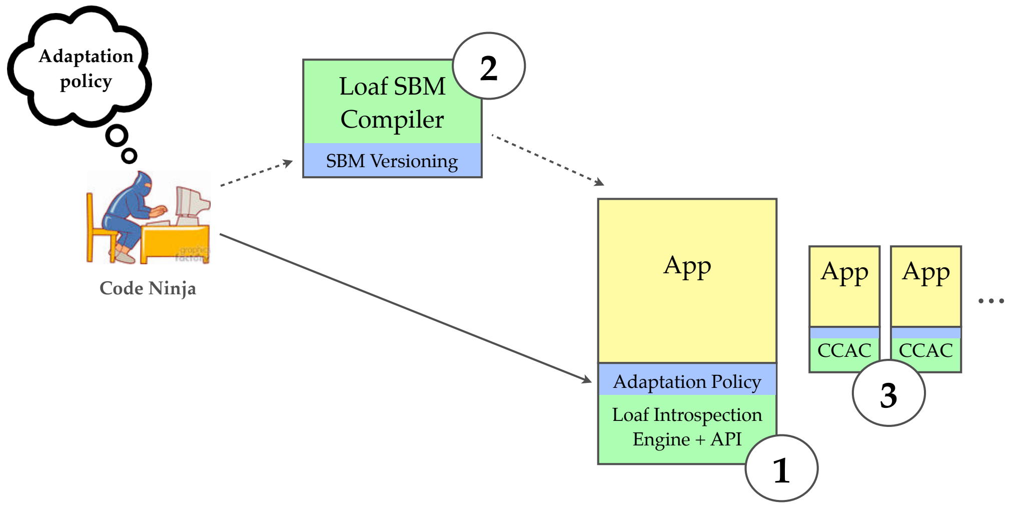

**Figure 2: Loaf Overview. (1) Lightweight Introspection (2) Scenario Based Multiversioning (3) Cross-Core Application Cooperation**

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**.

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. With a particular adaptation policy in mind, the user can leverage Loafs 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 2-1. 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 uninterrupted, 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, reset 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

```
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(c)
end
IssueTimerInterrupt(probe_interval)
```

**Algorithm 2:** Loaf Periodic Probes

```
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($probe\_interval$)
```

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.

Its 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 [49] among others. However, in this work, we present a general online adaptation framework for native binaries that leverages hardware performance monitors exclusively to provide a 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 reseting 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 **??** 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 **??** 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-

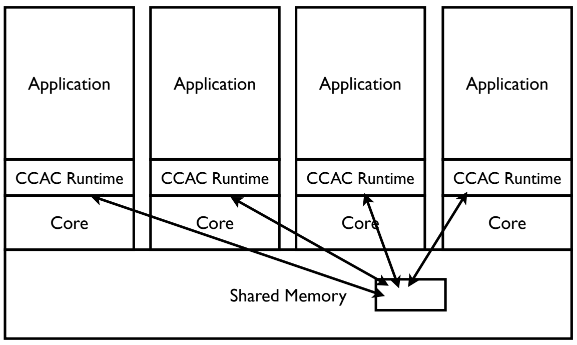

**Figure 3: Cross-Core Application Cooperation Runtime**

filing provided by GCC's GProf to identify the hottest functions of the application. We know from prior work that the top 2 to 8 functions most often covers the vast majority of the dynamically executed instructions across the SPEC 2006 benchmarks. Just the top 5 hottest functions can cover a significant portion of an applications execution, many times over 90%. Multiversioning these top functions leads to a very slight amount of code growth, for the SPEC2006 benchmarks less than 2% on average.

#### *2.1.3 Adapting the Environment*

To accommodate the adaptation of an application and its environment based on events that occur due to simultaneous co-scheduling on current multicore architectures, we use a *cross-core application cooperation* (CCAC) approach as shown in Figure 2-3. This approach is designed for problems that require the coordination of a number of processes or threads for a particular goal. The design of the CCAC enabled Loaf run-time environment is presented in Figure 3. In the scenario presented in the diagram, we have four applications running simultaneously on a quad core machine. In order to monitor and collect thread/core specific performance information on current hardware, we collect performance monitoring information on each core hosting the applications of interest and use a *shared communication table* to provide this information to other Loaf runtimes. Also, adaptation directives can be issued from one core to another through this shared communication table. We use the table to allow multiple CCAC enabled Loaf run-times to cooperate, respond, and adapt to each other.

We use shared memory to achieve this cross-core application cooperation (shown in Figure 3 as arrows pointing into the table). Performance information is gathered and added to the communication table intermittently using Loaf's *lightweight introspection*. It is also useful to record a window of multiple samples of performance information in the table as keeping a window of recent activity will allow us to observe trends in application behavior. To accommodate this communication protocal, we also develop an abstract primitive for each table entry which is supplied by our API.

## 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 design a contention aware execution environment that will 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 depends 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 Optimizatons*

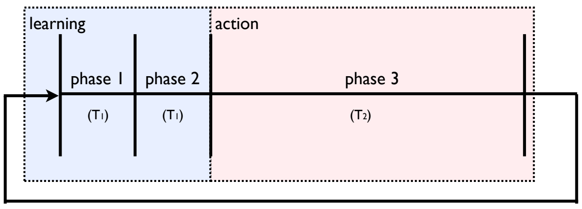

**Figure 4: This represents the three phase execution approach of OAAAO.**

The adaptation policy for the *online aiding and abetting of aggresive optimizatsons* (OAAAO), uses the *alternate versioning* scheme mentioned with an online three phase analysis. 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.

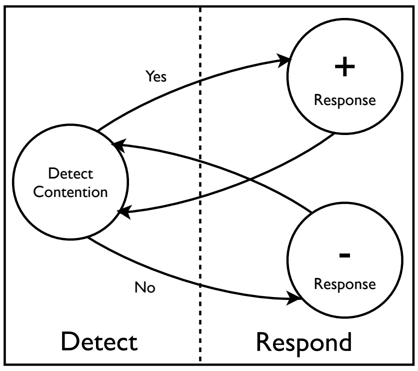

**Figure 5: Basic Detection Response**

### 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 [40].

As Figure 5 shows after detecting contention we transition into one of the response states, either *c-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

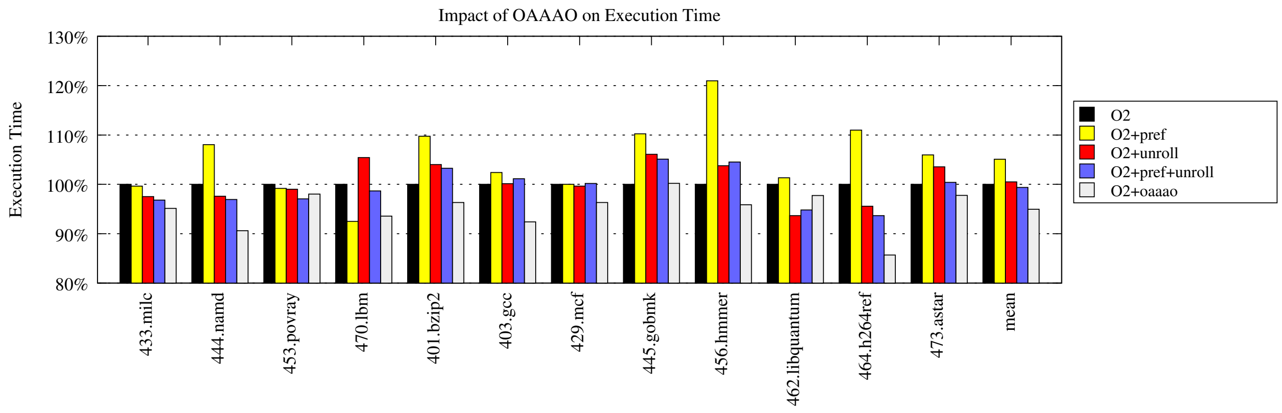

**Figure 6: This is the execution time after applying the aggressive optimizations statically compared to applying the same optimizations using OAAAO. (lower is better)**

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 disucss in Section 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 colocation 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 exceeds 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. In

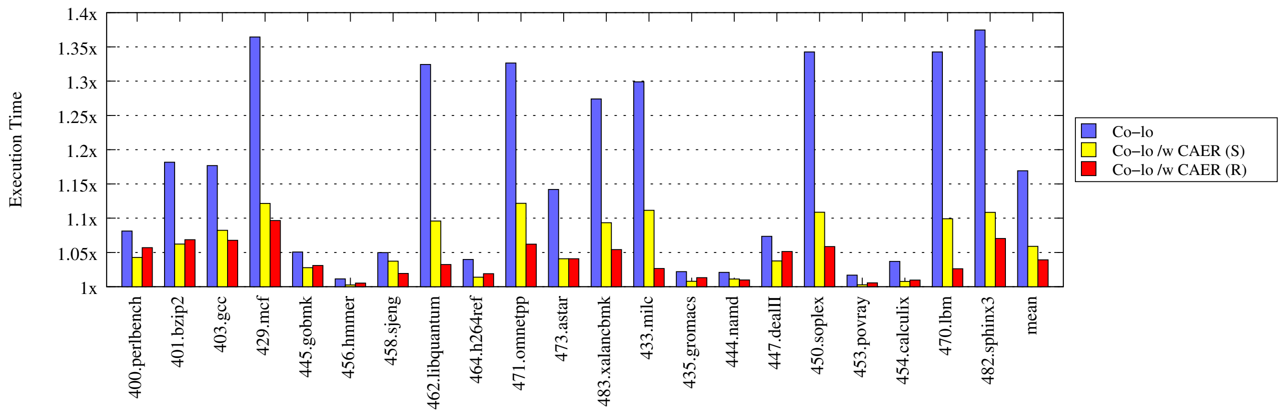

**Figure 7: Investigating the reduction in interference penalty.**

Figure 7 we show 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 multicore chip. The second bars shows 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 [44, 53, 2] 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 [57, 56] 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 [4], DynamoRIO [6], and Strata [51]. 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. [4] 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 [6] and was unable to achieve significant improvement. A similar effort was made with the Strata [51] 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 [26].

Research attention has also been paid to online optimization approaches using multicore architecture and novel hardware techniques. The Adore infrastructure has been used by Lu et al. [36] 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 [37]. Zhang et al. proposed Trident [58, 59], 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. [14]. 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 [45, 47, 24]. 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 [3]. 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 [8, 18, 7, 9, 17, 20, 55]. It was originally conceived for classic optimizations such as inter-procedural constant propagation (IPCP) [8]. It has also been been used by Carini et al. for flow insensitive IPCP [9] and Cierniak 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 synchronization mechanism online. 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 mutliversioning has been used in a number of works by Fursin et al. as a mechanism to provide dynamic machine-learning testbeds for evaluating optimization configurations and performing online optimization space pruning [22, 23]. In this work we take advantage of function cloning and multiversioning to provide a general, flexible and lightweight approach to enable online code adaptation.

## 5.4 Cache Contention

When two application are running on neighboring cores, contention for the shared cache can affect application *Quality of Service* (QoS) and can negatively affect overall throughput and scheduling fairness. QoS and Fairness techniques have received much research attention [25, 41, 32, 29, 30, 42, 43, 52]. These works propose QoS and fairness models, as well as hardware and platform improvement to enable QoS and fairness be enforced. Rafique et al. investigates micro-architectural extensions to support the OS for cache management [46]. There has been a number of works aimed at better understanding and modeling cache contention [5, 12] and job co-scheduling [31, 16]. Other hardware techniques to enable cache management have also received research attention [54, 27, 48, 13]. Suhendra [54] proposes partitioning and locking mechanisms to minimize unpredictable cache contention. Cache reconfiguration [48] has also been proposed as a mechanism to enable cache partitioning. Although these works show promising future directions for hardware and system designers to take when addressing these problems, unfortunately current commodity micro-architectures cannot support these solutions as they do not meet the micro-architectural assumptions made these works. Another very promising direction based on what is likely to be future hardware capabilities, is to leverage core specific dynamic voltage scaling as is presented by Herdirch, Illikkal, Iyer, et al [25].

## 6. CONCLUSION

Many of the important problems that exist for computer systems and applications are those that only arise during application execution. Some of these are predictable and occur during every run, while others are non-deterministic and cannot be detected statically. To enable a robust ability to address these problems, an online adaptation mechanism must be used. 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

[1] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 267–276, New York, NY, USA, 2004. ACM.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm.

In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[3] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000.

[5] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[7] M. Byler, M. Wolfe, J. R. B. Davies, C. Huson, and B. Leasure. Multiple version loops. In *ICPP*, pages 312–318, 1987.

[8] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *SIGPLAN Not.*, 39(4):155–166, 2004.

[9] P. R. Carini and M. Hind. Flow-sensitive interprocedural constant propagation. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 23–31, New York, NY, USA, 1995. ACM.

[10] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, Mar 2007.

[11] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[12] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[13] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.

[14] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[15] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[16] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[17] M. Cierniak and W. Li. Interprocedural array remapping. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, page 146, Washington, DC, USA, 1997. IEEE Computer Society.

[18] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Comput. Lang.*, 19(2):105–117, 1993.

[19] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[20] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, May 1997.

[21] S. Eranian. Perfmon2. http://perfmon2.sourceforge.net/.

[22] G. Fursin, A. Cohen, M. F. P. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Trans. on High Performance Embedded Architectures and Compilers*, 1(1):13–31, Jan. 2007.

[23] G. Fursin and O. Temam. Collective optimization. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] R. Gupta, D. A. Berson, and J. Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 358–368, Washington, DC, USA, 1997. IEEE Computer Society.

[25] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 479–488, New York, NY, USA, 2009. ACM.

[26] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic

translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[27] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM.

[28] Intel Corporation. *IA-64 Application Developer's Architecture Guide*. Intel Corporation, Santa Clara, CA, USA, 2009.

[29] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM.

[30] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2007. ACM.

[31] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[32] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[33] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54 – 66, 2008.

[34] S. Lohr. Demand for data puts engineers in spotlight. *The New York Times*, 2008. Published June 17th.

[35] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *14th Conference on Parallel and Distributed Computing Systems*, August 2001.

[36] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of run-time data cache prefetching in a dynamic optimization system. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

[37] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[38] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[39] J. Mars and M. L. Soffa. Multicore adaptive trace selection. Appeared at STMCS '08: Third Workshop on Software Tools for MultiCore Systems, March 2008.

[40] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *CGO '10: Proceedings of the 2010 International Symposium on Code Generation and Optimization*, pages 257–265, New York, NY, USA, 2010. ACM.

[41] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. Flexdcp: a qos framework for cmp architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, 2009.

[42] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[43] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2007. ACM.

[44] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 150–162, Washington, DC, USA, 2007. IEEE Computer Society.

[45] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM.

[46] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM.

[47] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 106–116, New York, NY, USA, 2002. ACM.

[48] R. Reddy and P. Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 198–207, New York, NY, USA, 2007. ACM.

[49] F. Schneider, M. Payer, and T. Gross. Online optimizations driven by hardware performance monitoring. *PLDI '07: Proceedings of the 2007 ACM*

*SIGPLAN conference on Programming language design and implementation*, Jun 2007.

[50] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 373–382, New York, NY, USA, 2007. ACM.

[51] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[52] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

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

[54] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

[55] M. Voss and R. Eigemann. High-level adaptive program optimization with adapt. *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, Jul 2001.

[56] M. Wegiel and C. Krintz. The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 91–102, New York, NY, USA, 2008. ACM.

[57] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed run-times. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 289–300, New York, NY, USA, 2009. ACM.

[58] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded 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.

[59] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.

[60] M. Zhao, B. R. Childers, and M. L. Soffa. An approach toward profit-driven optimization. *ACM Trans. Archit. Code Optim.*, 3(3):231–262, 2006.