Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations

Jason Mars
University of Virginia
Charlottesville, Virginia, USA
Email: jom5x@cs.virginia.edu

Robert Hundt
Google, Platforms Division
Mountain View, California, USA
Email: rhundt@google.com

***Abstract*—Online optimization allows the continuous restructuring and adaptation of an executing application using live information about its execution environment. The further advancement of performance monitoring hardware presents new opportunities for online optimization techniques. While managed runtime environments lend themselves nicely to online and dynamic optimizations, such techniques remain difficult to successfully achieve at the binary level. Binary level Dynamic optimizers introduce virtual layers which at the binary level produces overhead that is often prohibitive. Another challenge comes from the lack of source level information that can significantly aid optimization.**

In this paper we present a new static/dynamic collaborative approach to online optimization that takes advantage of the strength of static optimization and the adaptive nature of dynamic optimization techniques. We call this hybrid optimization framework *Scenario Based Optimization (SBO)*. Statically we multiversion and specialize functions for different dynamic *scenarios* that can be identified by monitoring micro-architectural events. Using these events to infer the current scenario, we dynamically reroute execution to the relevant code tuned to that scenario. We have implemented our static SBO infrastructure in GCC 4.3.1 and designed our Dynamic Introspection Engine using the Perfmon2 infrastructure. To demonstrate the effectiveness of our Scenario Based Optimization framework we have designed an SBO optimization we call the Online Aiding and Abetting of Aggressive Optimizations (OAAAO). Using SPEC2006 this optimization shows a speedup of 7% to 10% for a number of benchmarks and in our best case (h264ref) a 17% speedup over native execution compiled at the -O2 optimization level.

I. Introduction

Dynamic and online optimizations enable applications to dynamically reorganize and restructure the stream of instructions seen by the underlying architecture while executing. This allows the application to optimize itself based on the nature of its dynamic input, its resulting execution path, and its micro-architectural events. In addition, with the proliferation of multicore and manycore architectures, the set of programs running simultaneously alongside the executing application can also have an impact. This impact most often results from the added pressure on the system resources of the execution environment. Dynamic and online optimizations can also take advantage of these circumstances to improve the structure of the execution streams of a number of applications to reduce contention for resources. Because of the ever-changing nature of execution contexts, runtime optimization approaches can take advantage of these opportunities while static approaches simply can not.

Underlying hardware design is also evolving. Most processor architectures have begun, and are continuing, to add *performance monitoring hardware* (PMH) structures with increasing complexity [1], [2], [3]. These structures allow software systems to count and monitor the micro-architectural events of a chip, most commonly with negligible overhead. Performance monitoring hardware has been used for profiling, feedback directed optimization, application characterization, workload characterization, etc [4], [5], [2], [6]. In addition to these applications, the information provide by PMH presents a valuable opportunity to online optimization systems. Performance monitoring hardware provides real-time, and accurate descriptions of the execution context of an application or the entire system. An online optimization system can use these structures to collect fine grain, accurate information with low overhead to steer the restructuring of an applications execution stream. Although previous work investigates this path, much of the work that has been done either applies to managed runtime systems, proposes new hardware, or is focused on narrower problems such as optimization space pruning and iterative optimization [2], [4], [6], [7], [8], [9]. In this work we design a general framework for taking full advantage of the performance monitoring hardware of todays systems and source level information about the application to design new static/dynamic collaborative online optimizations for native binaries.

While managed runtime environments are well suited for dynamic and online optimizations, discovering and applying such optimizations at the binary level has proved to be quite difficult [10], [11], [12]. This can be attributed to two factors: a lack of source level information, and added overhead.

Many binary-level dynamic optimization frameworks do binary to binary transformations without source level information [12], [13], [14], [15]. This greatly limits the optimization opportunities available. The second challenge

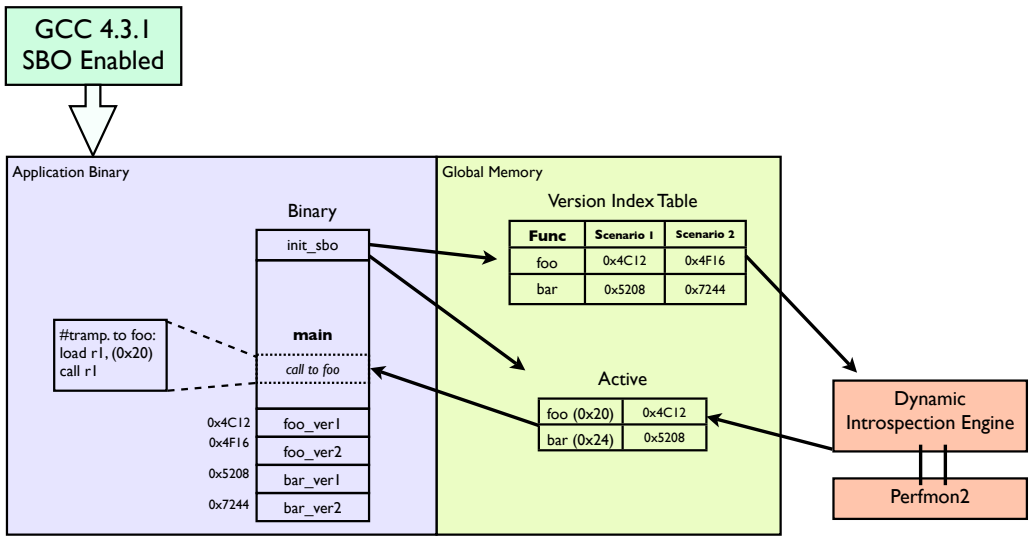

Figure 1. This is the main diagram of Scenario Based Optimizations.

to binary-level online optimization is the fact that continuous monitoring and some form of analysis are required. This requirement adds further demand on system resources. When dealing with native binaries, this cost can impact application performance, sometimes causing the overhead introduced by the online optimization to be greater than the benefit of the optimization itself.

In this work we present a new framework for binary-level online optimizations that addresses both of these challenges, in addition to inspiring a new way to think about compile-time optimization. We call this new paradigm *Scenario Based Optimization* (SBO). SBO is a hybrid static/dynamic approach to optimization whose strength lies in the co-design between compiler and dynamic engine. A scenario can be described simply as an occurrence or set of occurrences in our execution environment. For example, as our application runs, a possible scenario could be that another program is launched on a neighboring core that causes thrashing between the two programs. Another example scenario may be the buses on-chip are oversubscribed. These scenarios will affect application performance and can occur at anytime or not at all.

Figure 1 shows an overview of our framework. The philosophy of Scenario Based Optimization is centered around the idea that code within an application can be optimized and tuned differently for particular dynamic scenarios. This is accomplished at function-level through multiversioning. A version for a function is statically generated by the compiler and specialized to each anticipated scenario. During runtime, scenarios are identified via a dynamic engine that uses performance monitoring hardware. When a scenario is identified, execution is dynamically rerouted to execute the appropriate version of the code. We have designed and implemented our Static Scenario Based Multiversioning (SSBM) in GCC 4.3.1. Our accompanying Dynamic Introspection Engine (DIE) is implemented as a library that hooks into both our SBO enabled binary and Perfmon2, an API to the performance monitoring hardware of the underlying architecture.

To demonstrate the potential we have designed and implemented a scenario based optimization called the Online Aiding and Abetting of Aggressive Optimizations (OAAAO).

Many optimizations may improve performance in some cases and degrade performance in others. We call these optimizations *aggressive*. This type of optimization may benefit from an adaptive online optimization approach. If we can detect the scenarios where aggressive optimizations are beneficial, we can exploit that knowledge dynamically. We show that, using our SBO framework, we can infer whether we are in a scenario that makes aggressive optimizations beneficial or not and reroute execution accordingly. Using the SPEC2006 benchmark suite, we show a performance boost ranging from 7%-8% for a number of benchmarks and up to 17% for `h264ref`.

In the next section we will discuss the static side of SBO. In Section III we discuss SBO's dynamic component. We then move on to discuss our Online Aiding and Abetting of Aggressive Optimization in Section IV. We present results in Section V. In Section VI we discuss related works. And finally in Section VII we discuss future work and conclude.

II. Static Scenario Based Optimizations

One of the major key insights of Scenario Based Optimizations is the fact that compiler writers can statically predict the possible runtime scenarios an application may face. This compiler writer can then discover and invent

new optimizations that take advantage of static compile-time optimization techniques and dynamic monitoring and execution routing to enact online policies. In this section we discuss the static aspects of the Scenario Based Optimization framework.

A. Philosophy

The underlying premise to SBO is that the benefit from an optimization may depend heavily on the execution context of the optimized code. The execution context of an application is always changing. Even when the same application routines are being executed and re-executed, the environment is being affected by may factors such as interrupts, tasks on neighboring cores, the demand on the memory bus, and many others. It is well known that code optimizations provide different benefits in different execution environments [16]. Therefore, these varying and ever changing execution contexts provide a unique opportunity.

The challenge is however that, traditionally at compile time there is only one opportunity to produce optimized code. Static compiler optimizations are tuned conservatively to be effective in all scenarios. When compilation is done, the optimization decisions are rigid, regardless of its execution environment or application phase changes. This is exactly what the Scenario Based Optimization framework aims to solve. To retain all of the capabilities of static compilation, SBO uses function level multiversioning to enable static compilation to achieve runtime flexibility.

B. Function Level Multiversioning

Function level multiversioning is an inter-procedural code transformation that has proven quite useful by prior work [17], [8], [9]. Within our SBO framework it provides a useful mechanism for generating the specialized versions of a function that target different scenarios. For SBO we extend this mechanism to provide an interface between the static binary and the dynamic introspection component. This is necessary to allow the dynamic component to reroute execution as the application is running.

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 SBO and ultimately have a negative impact on application performance. Therefore we must limit the number of functions we multiversion to only the hottest functions in the application.

1) Profiling: To efficiently multiversion our application we take advantage of some basic profiling. Profiling can provide some information about the runtime behavior of an application and has proven to be very useful for determining the hottest code in an application [1], [6], [7]. Therefore, our SBO framework uses simple profiling provided by GCC's GProf to identify the hottest functions of our application.

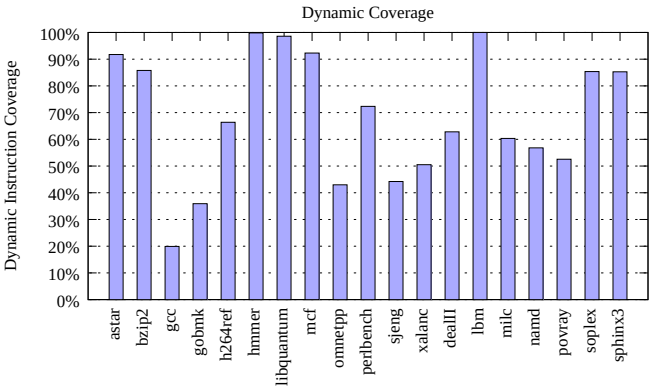

Figure 2. This graph shows the percent of execution time spent executing of the Top 5 hottest functions.

Across the SPEC2006 benchmarks, the top 2 to 8 functions most often covers the vast majority of the dynamically executed instructions. In Figure 2 we show the dynamic instruction coverage of the top 5 functions in the SPEC2006 benchmarks. This data was collected using GProf. As the graph shows, just the top 5 hottest functions can cover a significant portion of an applications execution, many times over 90%. Benchmarks such as gcc have less coverage because there are more distinct phases, however in benchmarks such as hmmer, libquantum, and lbm almost all execution is covered by the top 5 hot functions.

In Section V we show that multiversioning these top functions lead to a very slight amount of code growth.

2) Online Version Switching: To successfully achieve Scenario Based Optimizations we must provide an interface between the statically generated versions and the dynamic engine. This interface allows the dynamic engine to hook into the executing binary and reroute the execution via reseting the active versions of the functions. To accomplish this we have explored two designs.

We call the first design the *alternate versioning scheme* and the second the *n-version versioning scheme*. While both techniques require the use of a trampoline as the multiplexing mechanism there are differences. Figure 3 show the alternate version scheme. This scheme allows for a default and alternate version of a particular function. 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 either executes the default versions for all multiversioned functions or the alternative version. This provides a simple abstraction that a compiler writer can use to design Scenario Based Optimizations that do not require too much complexity.

Figure 4 shows the n-version versioning scheme. This scheme allows for any number of versions for any function. A global mapping table is maintained in memory for each

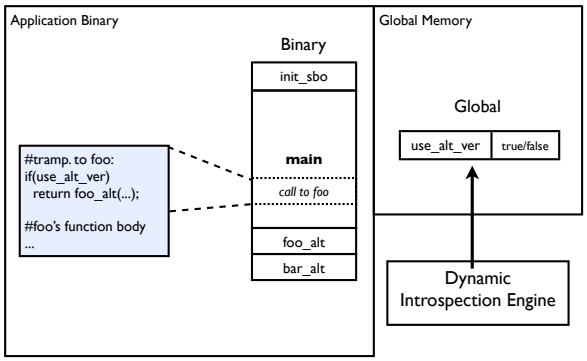

Figure 3. Alternate Versioning scheme

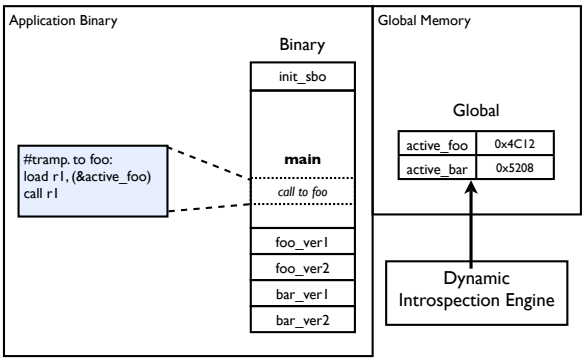

Figure 4. N-Version Versioning Scheme

function. Instead of a global switch, each call to a multiversioned function becomes 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 allows for much more complex SBO heuristics where multiple scenarios can occur at the same time.

C. Infrastructure: Technical Details

We have implemented our compile-time SBO infrastructure as a new pass in the GCC 4.3.1 compiler. The passes within GCC can be broken into four parts. First, there are the parsing passes where the text of the source code are processed. Second, we have the gimplification passes where GCC generates its Gimple intermediate representation on which optimizations can occur. Third, we have tree-SSA passes that optimize high level Gimple IR. Finally, we have the RTL passes where low level optimizations and code generation occurs.

Our new SBO pass has been placed right after GCC's earliest IR is generated as the first inter-procedural pass. This allows for maximum flexibility for compiler writers to design how the SBO function versions can be configured. For example, a function can be annotated to disable or enable any of the optimizations in later passes.

To specify which functions are to be multiversioned we have added a new command-line option to GCC, `-fmultiver_funcs=`. For example, if the functions `foo` and `bar` are to be multiversioned, invoking gcc with the command `gcc -fmultiver_funcs=foo,bar test.c` accomplishes this.

Internally GCC provides a function and call graph cloning routine that is used or inter-procedural constant propagation. SBO uses this routine to clone the internal function data structures as many times as needed. We take the original function and rewrite its internals. This function now becomes a trampoline that the SBO dynamic component can manipulate via shared memory hooks. The way this new trampoline functions depends on whether we are using the alternate scheme or the n-version scheme.

For the alternate scheme we simply inject basic blocks into the function's head using GCC's internal basic block writing APIs. The logic of the injected Gimple basic blocks first checks a global, if it is set the calling parameters are then passed on to the alternate version and it is called using a direct call. Any values returned from the alternate version are then passed on to the original call site. If the global is not set we execute the default function code. The dynamic component controls this trampoline via this global.

For the n-version scheme we always trampoline out of the original function similarly to the case where the global is set in the alternate version. The primary difference is, with the n-version scheme, the function call is an indirect. The dynamic component controls this trampoline by writing the address of the target function in the address location the indirect call uses. The global variable and tables that are required to provide the interface to dynamic component are all injected into the binary through this SBO pass.

Finally, this pass injects one basic block into the head of the main function of the application. This basic block is composed of a single call to `init_sbo`. This call initializes and launches the dynamic component. The dynamic component is implemented as a library and contains the body to the `init_sbo` call. Any application compiled with SBO enabled must be linked with SBO's dynamic component.

III. The Dynamic Introspection Engine

The dynamic component is responsible for monitoring the execution context of the application and detect when a scenario may have begun. If this occurs the dynamic component is responsible for re-routing execution to only include the code best suited for the detected scenario.

A. Performance Monitoring

We take advantage of performance monitoring hardware to continually identify the current execution context of our host application. By using performance monitoring hardware we are able to collect this information about the execution environment while incurring negligible overhead. There are

a number of APIs available for taking advantage of performance monitoring hardware including OProfile, PAPI, and Perfmon among others.

We have chosen to use Perfmon2 [18] for the design and implementation of our dynamic introspection engine. The goal of the Perfmon2 project is to design and implement a general, standard Linux interface to architectural performance monitoring hardware. In addition to the kernel work, Stephane Eranian and the other Perfmon2 developers have also implemented user-level libraries and tools to facilitate development with Perfmon2. Perfmon2 supports most major architectures including core/core2, amd64, itanium, and powerpc. For these reasons we selected to build our dynamic infrastructure on Perfmon2.

B. Periodic Probing

```
//init_sbo is called at application startup
void init_sbo()
{
  //initialize performance counters
  set_up_performance_counters();

  //lauch the counters
  start_counting();

  //start the timer interrupt
  lauch_timer_interrupt();
}

//when the interrupt is thrown we handle it here
void interrupt_handler()
{
  //stop the counters, collect the information
  stop_counters();
  read_counters();

  //do the analysis required by
  //the scenario detection heuristic
  do_analysis();

  //switch the active versions of functions in
  //our application to match the detected scenario
  reroute_execution();

  //start the counters again after reseting them
  start_counters();

  //launch the timer
  lauch_timer_interrupt();
}
```

Figure 5. This is pseudo code for the general dynamic introspection component of SBO.

One thing to keep in mind is the dynamic component is modular and flexible. SBO statically generates binaries with specialized versions of hot functions and provides hooks for the dynamic component. The dynamic component can then use any heuristic to reroute execution via control through these hooks. How the dynamic component monitors execution is entirely up to the optimization designer and can vary in any way.

That being said our SBO infrastructure has a default design for the dynamic component. It is shown in Figure 5. To detect whether a scenario is occurring, SBO's dynamic component uses a timer interrupt approach. The dynamic component includes an `init_sbo` routine that is called once when the host application begins. When the `init_sbo` routine is called, performance counters are setup and the timer interrupt is started. When the timer interrupt has triggered, the interrupt handler executes. As shown in Figure 5 the counters are then stopped and read. Next, the scenario detection code executes. If a target scenario is detected, the dynamic engine will reconfigure the executing binary to execute the function versions tuned to that scenario. The counters are then reset and the timer launched again.

This interrupt driven periodic probing execution pattern executes continually as the application is running. The overhead of such a technique is determined by the frequency of the probing. The amount of runtime overhead incurred by our probing technique depends on two factors: the frequency of interrupts, and the complexity of the analysis due to those interrupts. These two factors are determined by the nature of the optimization hosted by our SBO framework. Using our default design for the Dynamic Introspection Engine, this overhead is negligible. For example the overhead of the optimization presented in the next section causes a slowdown of less than 0.5%.

IV. Online Aiding and Abetting of Aggressive Optimizations : An SBO Approach

To demonstrate the usefulness of our Scenario Based Optimization Framework we have designed an optimization we call Online Aiding and Abetting of Aggressive Optimizations (OAAAO). The key premise comes from the fact that many optimizations show benefit in some cases and a degradation in others [16]. We call these optimization aggressive optimizations. The intuition is that we should be able to detect the scenarios where aggressive optimizations are beneficial or not.

A. Motivation: Win Some, Loose Some

Aggressive optimization may increase performance in some contexts and decrease performance in others. For our OAAAO approach we have identified two such optimizations, software cache prefetching and loop unrolling. These optimization heuristics, both found in GCC 4.3.1 as optional optimizations, both improve performance in some cases and degrade performance in others.

Figure 6 shows the impact these optimizations have on performance for 12 of the SPEC2006 benchmarks. These experiments were run on the Core 2 Quad 6600 running Linux 2.6.25. This graph show the speed up that results from applying either or both of these optimizations.

With notable exception of lbm, in most cases, software prefetching has a negative impact on performance. This is

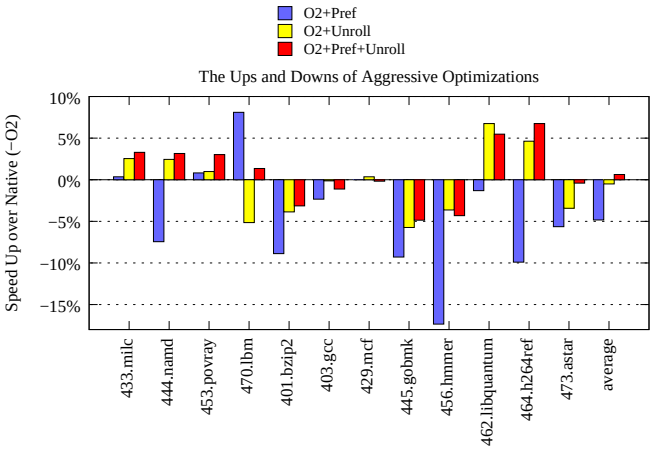

Figure 6. This graph shows the speedup in execution time when *aggressive* optimizations are applied. Note that sometimes there is a benefit other times we see a degradation.

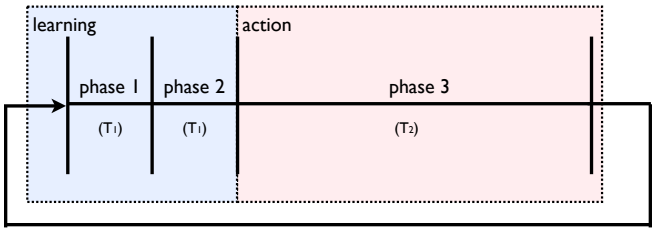

Figure 7. This represents the three phase execution approach of OAAAO.

most likely due to the already present hardware prefetching structures on the Core 2. If the hardware prefetching is already doing the work, having the explicit prefetch instructions simply adds an extra burden to the architecture. We can clear see that software prefetching is an aggressive optimization. While software prefetching improves lbm's performance by 8%, in the case of hmmer, the degradation due to adding the software prefetching is over 15%.

Loop unrolling is also an aggressive optimization, here also we see a performance improvement in some cases and a degradation in others. Note that both software prefetching and loop unrolling degrades much more severely when applied individually and less so when applied together. This must come from some interaction with its micro-architectural environment.

This observation brings us to our hypothesis: Using Scenario Based Optimizations we should be able to improve the performance of these aggressive optimizations. Using the dynamic introspection engine we should be able to detect the scenarios when aggressive optimizations are improving or degrading performance. We can then reroute execution accordingly.

```
void catch_alarm(int sig_num)
{
  int i;
  //stop and read the counters
  pfm_stop(ctx_fd);
  pfm_read_pmds(ctx_fd, pd, inp.pfp_event_count);

  //execute the code for current phase
  //and move to the next phase
  if(phase==0) {
    phase=1;
    ver1_stat=ver2_stat=0;
    __mv_version_switch=0;
  }
  else if(phase==1) {
    phase=2;
    ver1_stat=pd[0].reg_value;
    __mv_version_switch=1;
  }
  else if(phase==2) {
     phase=0;
    ver2_stat=pd[0].reg_value;
    if(ver1_stat>ver2_stat)
      __mv_version_switch=0;
  }

  //clear and restart counters
  for (i=0; i < inp.pfp_event_count; i++) {
    pd[i].reg_value=0;
  }

  pfm_write_pmds(ctx_fd, pd, inp.pfp_event_count);
  pfm_start(ctx_fd, NULL);

  //launch the timer for next signal
  if(phase==0) alarm(10);
  else alarm(1);
  //renter executing application
}
```

Figure 8. This is the core three phase code to the dynamic component of the OAAAO algorithm.

B. Three Phase Execution

For the design and implementation of OAAAO, we use the *alternate versioning* scheme. Statically we generate code for two scenarios. Firstly we generate code for the scenario that aggressive optimizations would degrade performance. This is a function without software prefetching or loop unrolling; we call this the non-aggressive version. For the scenario that aggressive optimizations would improve performance, we generate a function that has software prefetching and loop unrolling; we call this the aggressive version.

The dynamic component of our OAAAO approach enforces three phases. Figure 7 shows our three phase design. The first two phases compose the learning and monitoring part of OAAAO, the third phase composes the action part of OAAAO. During execution these phases continually loop until the host application terminates.

Figure 8 shows the pseudo code of our design. During the first phase we set the active version for the binary to non-aggressive. The dynamic engine then starts the counters

to look at the absolute number of instructions retired. The application then executes for T_1 time. The number of instructions that successfully executed are then saved. During the second phase we set the active version for the binary to aggressive. We then do the same; we record the number of instructions that executed for this T_1 time. Before the third phase begins we compare the number of instructions retired for both phases 1 and 2. We select the version with the highest number of instructions retired to be executed in the third phase which lasts for T_2 time. Essentially we are selecting the version that has exhibited the lower average CPI for T_1 time. This ad-hoc performance metric conveys whether the scenario is well suited for aggressive optimization. After T_2 seconds of executing the winning version we enter phase one and restart the process.

We have chosen 1 second for our T_1 in phases 1 and 2 to allow enough time to allow execution to enter the hot functions. We do not want to base or decision on code that executes outside these hot functions. We have chosen 10 seconds for our T_2 in phase 3 because we want to have the dynamic component re-learn at a rate that keeps the intuition accurate, while allowing sufficient benefit to warrant the analysis. We have arrived at these particular parameters for our heuristic from hand tuning although one can imagine using more adaptive heuristics. We reserve further investigation into a self tuning approach and other heuristics for future work.

In the next section we present the results of our OAAAO SBO approach.

V. Results

In this section we present the data for a number of experiments evaluating the effectiveness of our OAAAO optimization built on our Scenario Based Optimization framework. The goals 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.

A. Execution Time

Figure 9 shows the impact on execution time when applying aggressive optimizations with and without the Online Aiding and Abetting of Aggressive Optimizations. The data shown in this graph has been normalized to the baseline, optimization level -O2. Anything greater than 100% marker shows a degradation anything lower than this marker shows a speedup.

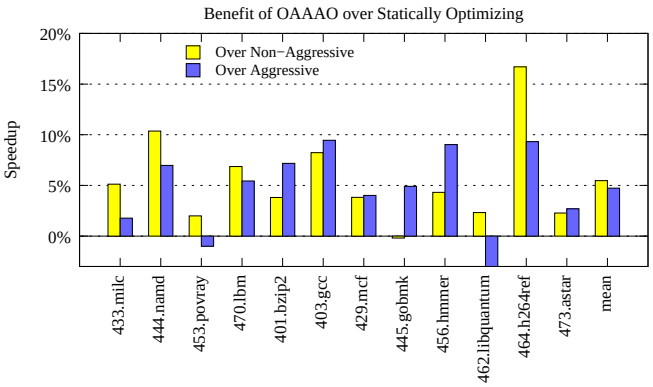

Figure 10. Here we show the benefit of using OAAAO to dynamically select the right version for a scenario versus using only the code for either scenario for the entire run.

One of the major goals of OAAAO is to eliminate the degradations incurred by aggressive optimizations. As the data in figure 9 shows, only when OAAAO is applied we see only 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 it exceeds the benefit of aggressive optimizations, in most cases more than doubling the performance boost.

B. Effect of Dynamic Switching

One very important question that arises is whether there is much switching occurring dynamically. If there is not much dynamic switching going on, there may be no need to continually probe the counters and redo analysis. To address this question we present Figure 10. Here we show the speedup of having dynamic OAAAO approach adaptively switch the active version between aggressive and non-aggressive compared to only having one version execute for the duration of application execution.

In Figure 10 the first bar shows OAAAO over only having the non-aggressive version, the second bar shows having OAAAO over having only the aggressive version. In this figure we highlight the fact that only two benchmarks, povray and libquantum, it is better to have the statically assigned aggressive version.

C. Degradation Reversal

In Figure 11 we highlight one of the brightest contributions of OAAAO. That is the fact that OAAAO actually makes loop unrolling and software prefetching show benefit

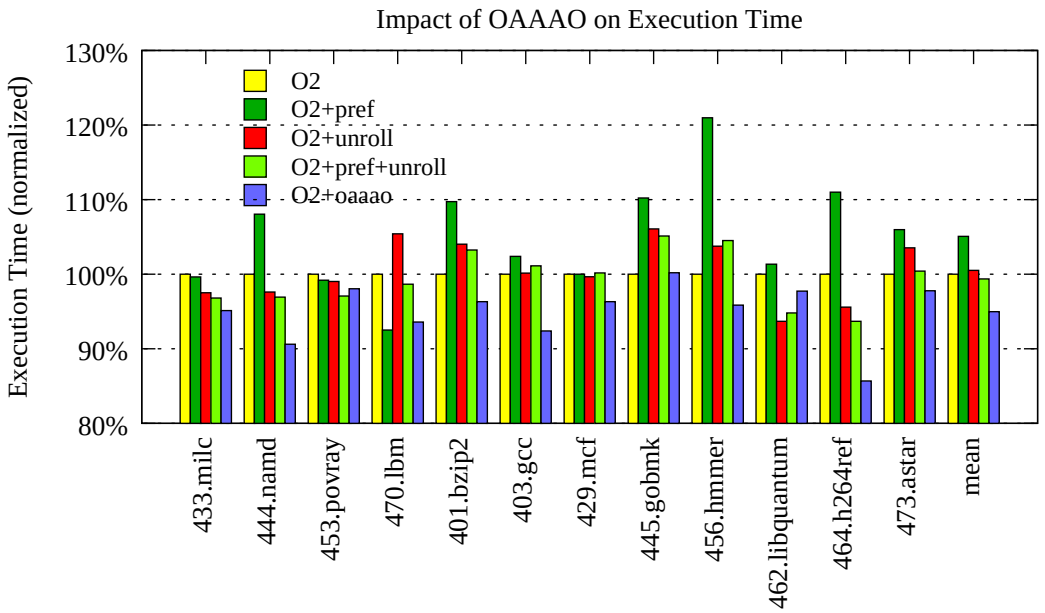

Figure 9. This is the execution time after applying the aggressive optimizations statically compared to applying the same optimizations using OAAAO. (lower is better)

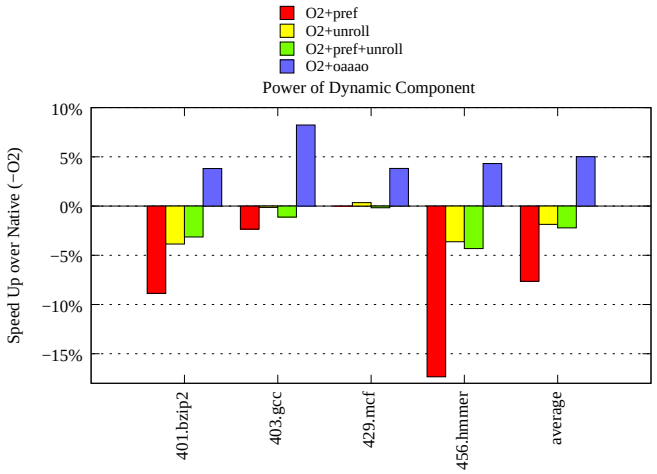

Figure 11. This graph highlights the power of a Scenario Based dynamic approach. These benchmarks all degrade or show no benefit when applying aggressive optimizations statically.

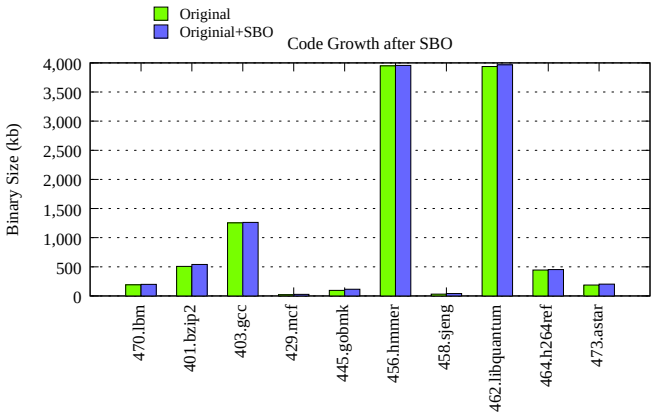

Figure 12. Here we show what percentage of the binary is occupied by code added by prefetching and unrolling in addition to that added by OAAAO.

where it would otherwise not. In the benchmarks presented in this figure, software prefetching and loop unrolling simply does not work without OAAAO. Regardless of whether they are applied individually or simultaneously they show degradations. However when governed by OAAAO they show significant improvements, going from degradations to speedups.

D. Code Growth

As we designed our SBO framework careful attention was paid to other types of overhead such as the impact on code size. Scenario Based Optimizations requires the duplication of functions, however it is not necessary to multiversion every function. As mentioned in section II we only multiversion the top 5 hottest functions.

Figure 12 shows the code growth due to the SBO framework. This particular instance of SBO was configured with the alternate version scheme used for OAAAO and includes its complete implementation. The size of the added dynamic component is included in these measurements. The first bar shows the size of the original binary, the second bar shows the size of the binary compiled with 2 versions of its top 5 hottest functions and the OAAAO dynamic component

linked in. We see that the final code size of the binary is largely unaffected by SBO. This is due to the fact that the increase in code size ranges from 3kb to a mere 12kb. For many benchmarks the absolute sizes of the binary are in the hundreds and thousands of kilobytes.

VI. Related Work

There is a wealth of prior related work that primarily comes from three areas of study. These areas include binary-level dynamic optimization, applications of performance monitoring hardware, and function cloning and multiversioning.

A. Binary-Level Dynamic Optimization

There has been prior work employ binary-level dynamic optimization techniques with limited success. One of the seminal works that has inspired many future projects was the work by Bala et al. [12] on Dynamo. Dynamo is a binary to binary translator and dynamic optimizer that works at the basic block and trace levels. Dynamo was one of the only approaches of its class to achieve performance gains. This has mostly been attributed to the intricacies of the PA-RISC platform on which it was implemented. Attempts have been made on other architectures and the results shown in the Dynamo work has not yet been duplicated. Bruening et al. reimplemented the Dynamo infrastructure for x86 with the DynamoRio project [11] and was unable to achieve significant improvement. A similar effort was made with the Strata [14] 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 work focused on optimizing the dynamic optimizer itself [10].

Other efforts have been made to achieve binary-level dynamic optimization, most of which has focused on cache prefetching. The work by Chilimbi et al. [19] used bursty tracing to achieve profile sampling to enact complex prefetching patterns. The Adore infrastructure has been used by Lu et al. [6] 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 [7]. Zhang et al. proposed Trident [20], a new dynamic optimizer framework that requires new hardware support. This work shows promising potential, but depends on new hardware to be developed.

B. Profiling and Hardware Performance Monitors

Profiling has become the cornerstone for understanding our applications behavior and can play an important part in compiler optimizations as shown in the work by Chang et al. [21]. This work introduces a compiler design to support 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. This has lead to many new kinds of optimizations [22], [23], [24]. However these compiler optimizations remain rigid and thus aims to best fit the program's entire execution and does not allow accommodating particular scenarios the application may encounter.

Performance counters has 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. [25]. 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. [5] 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. [1] and Mars et al. [26] 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. [2] and Adl-Tabatabai et al. [4]

C. Function Cloning and Multiversioning

Function cloning and Multiversioning is an inter-procedural code transformation that is used by a number of optimizations. It was originally conceived for classic optimizations such as inter-procedural constant propagation (IPCP) [27]. It has also been been used by Carini et al. for flow insensitive IPCP [17] and Cierniak et al. for inter-procedural array remapping [28].

More recently it 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 [8], [9]. Our work contrasts Fursin's in that instead of pruning the optimization space using machine learning techniques, we present a framework for the design of novel optimizations techniques with a particular dynamic scenario in mind.

VII. Future Work and Conclusion

A. Future Work

There is much future work we wish to investigate. As this work focuses on the SBO framework, we plan to do a thorough investigation and generalization of the OAAAO technique in future work. We can further develop the three phase and other heuristics. Currently our three phase heuristic has rigid settings. We plan to look into developing a self tuning OAAAO that can independently learn and respond to individual slices of code. In addition, we wish to see if what other heuristics are necessary for the OAAAO version switching decision maker to target parallel applications. Similarly, we can look at what other aggressive optimizations can be applied OAAAO to. We also would like to do further

analysis and evaluation of the switching frequencies and the limits of the opportunity of OAAAO.

Beyond this we also plan to extend the dynamic component of SBO to run as a separate thread. This will allow for monitoring at a finer grain as the host application and the dynamic component can execute simultaneously in addition to enabling more complex analyses. Lastly, we are investigating other novel Scenario Based Optimizations.

B. Conclusion

In this work we have presented Scenario Based Optimization, a new paradigm for using runtime information to steer online optimizations. Our framework allows compiler-writers to take advantage of the strengths of static compile-time optimizations while focusing on the ever-changing execution environment. In general, collecting and using runtime information about an application's execution environment for dynamic optimizations at the binary-level has proved difficult. However using this hybrid static/dynamic collaborative optimization paradigm, compiler writers can arrive at new, clever optimizations that would previously be impossible. We have described the design and implementation of this framework and demonstrated its effectiveness by designing a new SBO optimization: Online Aiding and Abetting of Aggressive Optimizations.

This SBO optimization takes two traditional static optimizations, cache prefetching and loop unrolling, restructures them into reactive dynamic optimizations, and significantly improves their usefulness. Moreover, in a number of cases OAAAO turns degradations into speedups. OAAAO improves performance in all benchmarks with exception to gobmk where it breaks even. These performance boost ranges from 4% to 11% in many cases, and is 17% for h264ref.

REFERENCES

[1] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 79–90.

[2] 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*. New York, NY, USA: ACM, 2007, pp. 373–382.

[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*. New York, NY, USA: ACM, 2005, pp. 101–110.

[4] 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*. New York, NY, USA: ACM, 2004, pp. 267–276.

[5] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 185–197.

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

[7] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 93–104.

[8] 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*, vol. 1, no. 1, pp. 13–31, Jan. 2007.

[9] G. Fursin, C. Miranda, S. Pop, A. Cohen, and O. Temam, "Practical run-time adaptation with procedure cloning to enable continuous collective compilation," in *Proceedings of the GCC Developers' Summit*, July 2007.

[10] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 61–73.

[11] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275.

[12] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2000, pp. 1–12.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[14] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 36–47.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 89–100.

[16] M. Zhao, B. R. Childers, and M. L. Soffa, "An approach toward profit-driven optimization," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 3, pp. 231–262, 2006.

[17] 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*. New York, NY, USA: ACM, 1995, pp. 23–31.

[18] S. Eranian, "Perfmon2," http://perfmon2.sourceforge.net/.

[19] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 199–209.

[20] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 87–98.

[21] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu, "Using profile information to assist classic code optimizations," *Softw. Pract. Exper.*, vol. 21, no. 12, pp. 1301–1321, 1991.

[22] 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*. New York, NY, USA: ACM, 1990, pp. 16–27.

[23] 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*. New York, NY, USA: ACM, 2002, pp. 106–116.

[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*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 358–368.

[25] 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*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 292–302.

[26] J. Mars and M. L. Soffa, "Multicore adaptive trace selection," Appeared at STMCS '08: Third Workshop on Software Tools for MultiCore Systems, March 2008. [Online]. Available: http://www.cs.virginia.edu/ jom5x/papers/mats.pdf

[27] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *SIGPLAN Not.*, vol. 39, no. 4, pp. 155–166, 2004.

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