Performance and environment monitoring for continuous program optimization

C. Caşcaval
E. Duesterwald
P. F. Sweeney
R. W. Wisniewski

Our research is aimed at characterizing, understanding, and exploiting the interactions between hardware and software to improve system performance. We have developed a paradigm for continuous program optimization (CPO) that assists in and automates the challenging task of performance tuning, and we have implemented an initial prototype of this paradigm. At the core of our implementation is a performance- and environment-monitoring (PEM) component that vertically integrates performance events from various layers in the execution stack. CPO agents use the data provided by PEM to detect, diagnose, and alleviate performance problems on existing systems. In addition, CPO can be used to improve future architecture designs by analyzing PEM data collected on a whole-system simulator while varying architectural characteristics. In this paper, we present the CPO paradigm, describe an initial implementation that includes PEM as a component, and discuss two CPO clients.

Introduction

The need for increased computing performance and functionality has resulted in a complex execution stack consisting of multiple hardware and software layers, each layer being complicated and challenging to understand. For example, hardware resources may be added to or removed from the system dynamically, current machines may run a varying set of applications (both scientific and commercial), and even an individual application may go through phases with different performance behavior and resource requirements. Therefore, understanding and tuning performance that involves multiple execution layers requires considerable expertise and sophisticated tools.

In this paper, we present a continuous program optimization (CPO) paradigm that assists in and automates the challenging task of performance tuning. The CPO paradigm may be conceptualized as two recurring phases of monitoring and optimization. Monitoring involves obtaining and analyzing performance information at varying granularities across layers of the system. Optimization involves using the information in a continual process of feedback-directed adaptation along two dimensions: adapting applications to their current execution environment and adapting the execution environment to enhance application performance. Examples include just-in-time (JIT) compiler optimizations [1], hot-swapping operating-system (OS) components [2], and redistributing application workloads. While CPO is conceptually divided into two phases, both are continually occurring in order to evaluate the effectiveness of tuning and to instantiate adjustments as needed.

We have developed an initial implementation of the CPO paradigm. An important component of the implementation, and the focus of this paper, is the monitoring functionality that captures system runtime behavior. This component is a vertically integrated performance- and environment-monitoring (PEM) framework that supports understanding the interactions

0018-8646/06/$5.00 © 2006 IBM

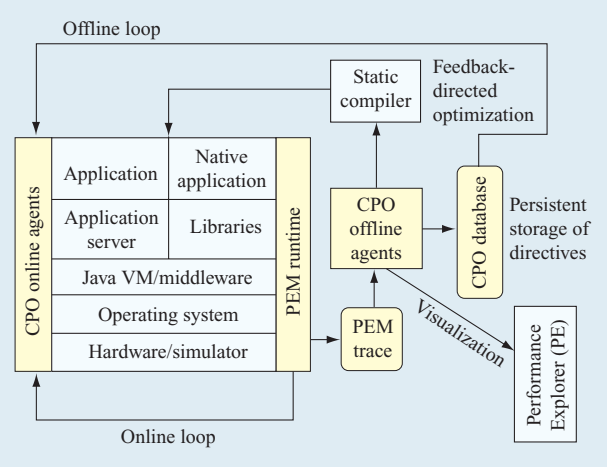


Figure 1

Continuous program optimization (CPO) architecture (CPO elements shaded in yellow).

among hardware and software and enables optimizations based on these interactions. For example, an optimization could take the form of an architectural design improvement achieved by varying hardware characteristics on a simulator, with PEM providing data on the total system performance and details of the effects on components of the entire execution stack. Another optimization approach supported by PEM is vertical performance tuning across execution layers. PEM provides data about the interaction of hardware and software, and of all the layers within the software stack. To be effective in a complex environment, performance tuning must be able not only to analyze data from all of the layers of the execution stack, but also to continuously tackle performance problems as they arise throughout the lifetime of an application and a system.

In this paper, we describe two CPO clients. The first is a visualization client that supports vertical performance visualization. The second is a page-size client that automatically optimizes large page use in an application.

The visualization client uses PEM to log monitored events and produce a PEM trace. The event trace is converted by a CPO offline agent by aggregating events into intervals that are more suitable for visualization and generating a new trace in the format expected by the Performance Explorer (PE) visualization tool [3].

Continuous program optimization

In CPO, performance tuning is viewed as a continuous process of feedback-directed adaptation. It involves dynamically identifying and characterizing performance problems, modeling application behavior and negotiating resources on behalf of the application, applying optimizations on the basis of resource availability, and validating the effectiveness of the applied optimizations.

The idea of feedback-directed optimization is not new. Previous approaches have focused on compiler optimizations. Our CPO paradigm goes beyond compiler optimizations and includes system and environment adaptation, such as the page-size client, which optimizes large page use in an application.

The architecture of our initial implementation of the CPO paradigm is shown in **Figure 1**. It has several components. The PEM runtime [4] allows vertically integrated performance-monitoring information to be gathered from all layers in the execution stack. CPO agents analyze this data and perform optimizations based on the analysis. The analysis consists of modeling salient aspects of system behavior using static information about different layers in the execution stack, and obtaining dynamic data from PEM. Some CPO agents execute online while others execute offline. Offline CPO agents are used to optimize applications between runs and are usually invoked once, unless the optimization fails the validation phase.

The CPO database acts as a repository for the history of past optimizations and their performance metrics and also as a unified place to resolve competing or conflicting optimizations. CPO agents store behavior information in the CPO database. There may be several CPO agents active concurrently, and their resource requirements may be conflicting. The common database and PEM provide the means to resolve potential conflicts by providing each agent with a view of performance events across the entire system. CPO agents can coordinate their activities by monitoring the system for performance events issued by other agents.

CPO clients are implemented through online and offline agents. On the basis of performance models, CPO agents negotiate resources that may either enhance performance directly or do so indirectly by enabling further code adaptations.

The validation of optimizations is a vital aspect of the CPO paradigm. It is crucial because modeling will not be able to capture all of the complex interactions in current or future systems. Therefore, it is critical to monitor the optimized execution stack continuously in order to validate that the modeled optimizations are having the intended effect and take action if they are not. Both optimization and validation occur automatically, with no programmer intervention.

Examples of the CPO paradigm include tuning of compiler optimization heuristics in a Java** JIT compiler, such as inlining policies, loop unrolling, and tiling parameters; tuning of linear algebra [5–7] and communication libraries to adapt to the actual

application usage; selecting different algorithms to solve a specific problem based on the characteristics of the input data set [8]; environment adaptation, such as the tuning of Java Virtual Machine (JVM) components (e.g., heap size and garbage collection policies), tuning OS resources, such as memory available for the JVM or changing the page size for different regions of memory allocated to an application [9]; and modifying the OS page replacement policy on the basis of application memory usage patterns [10].

PEM

PEM is a crucial component of CPO. **Figure 2** illustrates its architecture and the way it interacts with other CPO components. PEM comprises several components, discussed in the following subsections.

The PEM runtime provides an implementation of the event specification and the PEM Application Programming Interface (PEMAPI) on a given system [4]. We implemented PEM on the K42 open source research OS [11]. K42 already has a significant performance monitoring infrastructure [12] and provides an easy prototyping environment. Further, K42 was designed to be scalable and contains support for hot swapping [10], a useful feature for CPO. As we develop experience with the PEM runtime, the successful portions can be implemented on other operating systems. For example, we have recently ported to IBM AIX* the portions of PEM relevant to the page-size agent.

XML event specification

The first component of PEM consists of a repository of Extensible Markup Language (XML) event specifications that define the monitoring scope. An XML event specification defines the semantics of the event and may contain several fields to describe event attributes. It provides a unified location for encoding event semantics. For example, the fact that a send and a receive event in a library (such as Message Passing Interface) establish a communication pair based on a common communication port identifier constitutes semantic information. Any two such events will be linked, even though the ports may differ between pairs of events. **Figure 3** shows an example, in XML, of encoding the event of reading the contents of hardware performance counters. The XML specification defines the context of the `HW::HWPerf::HPMsAndPC` event with the `layerId`, `classId`, and `specifier` XML tags, and defines, with the `field` XML tag, the event attributes that will be stored with the event.

We have written an XML specification containing several hundred hardware, OS, and application-layer events. In addition to the benefits of readability and availability of XML parsers that can be used to build tools, we chose to use XML to satisfy several crucial design requirements:

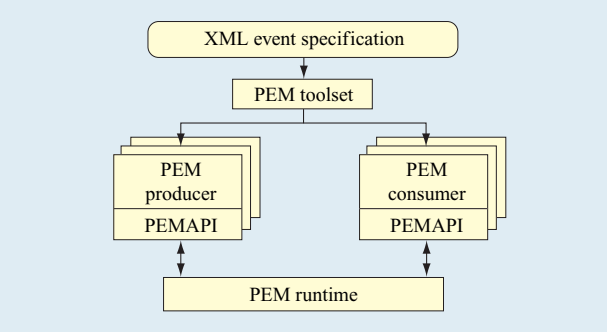


Figure 2

Overview of PEM.

```
<event name='HW::HWPerf::HPMsAndPC'
     description='Hardware counter events
     and program counter for threadId'>
  <layerId value='HW' />
  <classId value='HWPerf' />
  <specifier value='HPMsAndPC' />
  <fields>
    <field name='pc' type='uint64'
       description='program counter' />
    <field name='threadId' type='uint64'
       description='kernel thread id' />
    <field name='counterValues' type='list'
       eltType='uint64'
       description='counter values.' />
  </fields>
</event>
```

Figure 3

Example of an XML event specification.

- *Language-independent support* for building tools to aid in the development of CPO agents, as explained in the next section.
- *Flexibility* to add and modify events. Additional execution layers that produce new events (e.g., new libraries) can easily be integrated into PEM by adding new event specifications into the repository.
- *Documentation* providing a single place for explicit field names and descriptions for entities stored in an event record, which encourages developers to better document event semantics.

PEM toolset

We have implemented a set of tools that take the XML specification as input and generate language-specific interfaces and stubs. We currently support the C, C++, Fortran, and Java languages. The PEM toolset generates

support for the functionality of the PEM runtime and the CPO agents.

The support for the PEM runtime includes interfaces used by the execution layers for instrumentation to invoke event notifications, and interfaces and stubs used for event processing. The event-processing interfaces react to event notification by emitting event records that can be either written to a trace file or consumed by a CPO online agent. The support for CPO agents generates code that tailors the XML event specifications to the specific needs of a client using the agent.

Both the PEM runtime and the CPO agents are concerned with event streams. However, the requirements for the way in which events in a trace should be structured in the event stream may differ between the PEM runtime, which produces the trace, and PEM clients, which consume the trace. To reduce system perturbation, event generation is best implemented as a raw event stream that minimizes the information contained in each event record so that events can be logged as quickly and as efficiently as possible with respect to space. For example, context information, such as the currently executing process and thread identification, is omitted because it can be recovered from previously logged context switch events.

In contrast, event consumption in PEM clients typically requires that the lost information be computed by interpreting the event stream. During interpretation, information that was omitted from individual raw event records at trace time is recovered on the basis of agent-specific semantic criteria. For example, consider our vertical performance visualization client, which uses the toolset to convert the raw event stream into a stream more suitable for visualization (see the visualization client section below).

PEMAPI

PEMAPI provides an interface between PEM producers and consumers. A PEM producer is an execution layer or a component in a layer that produces monitoring information as events. A PEM consumer processes and regulates the information that is monitored. CPO online agents are examples of PEM consumers that benefit from PEMAPI. Using PEMAPI, PEM consumers can be built to implement specific performance-monitoring tasks, such as logging events to a disk or online event processing. To facilitate the programming of PEM consumers, PEMAPI provides the following performance-monitoring abstractions:

- An `event` is a type of action taken by the system. Examples of events are cache misses, page faults, OS interrupts, garbage collection invocations, dynamic compiler invocations, and transaction completions. Events are defined in the XML event specification.
- An `event attribute` is data associated with an event. For example, a page fault event may have as an attribute the address that caused the page fault.
- An `eventSet` specifies a set of events. All events in a set can be handled as a single entity.
- A `context` specifies the state of the system. For purposes of PEMAPI, context is determined by the processor, process, and thread identifiers.

A PEM consumer can specify the context of an event in which it is interested. When an event occurs in a context that is specified by a PEM consumer, the consumer action is taken. In particular, any PEM consumer can create (private) statistics or log an event through PEMAPI.

PEM producers

A PEM producer is a hardware or software layer that generates events. In order for the PEM runtime to collect these events, a PEM producer is instrumented with PEMAPI event notification calls. This is the only modification to the PEM producer layer. The reaction to event notification calls in PEM is fully programmable through other PEMAPI functions. The PEMAPI provides event-specific notification interfaces that are automatically generated from the XML event specifications. An event-specific notification explicitly passes the event attributes as arguments. For example,

```
int notifyPageFault (attr_t tid,
  attr_t faultAddr, attr_t faultIAR)
```

specifies that a page fault has occurred with the attributes thread ID, faulting data address, and faulting instruction address.

If the event has not been enabled by any PEM consumer, event notification does nothing. If the event has been enabled by a PEM consumer for the current context, event notification takes the action specified by that PEM consumer.

PEM consumer

A PEM consumer uses a stream of events to model, analyze, or display behavior. The consumer must specify both the context and level of detail of an event in which the consumer is interested. The level of detail determines the amount of information available about an event. For example, a consumer might want an event and all of its attributes to be saved every time the event executes, while another consumer might only want to count the number of times the event occurs.

The PEMAPI provides two levels of details: logging and statistics. At the logging level, whenever the event occurs in the specified context, the event and its attributes are saved as a log record (typically in a trace). At the statistics level, whenever the event occurs in the specified

```
gcEnd(void *record) {
   timestamp = ((gcEndRecord *)record)->timestamp;
   stats.enable();
   stats.reset();
}
gcStart(void *record) {
   gcStartRecord *rec = (gcStartRecord *)record;
   long long *vals;
   stats.disable();
   vals = stats.read();
   logBetweenGCs(timestamp, rec->timestamp, vals);
}
context_t    context   = {UNRESTRICTED, myPid, myTid};
eventSet_t   events    = {pageFaultEvent, dataCacheMissEvent};
statistics_t stats     = registerStatistics(events, context, COUNT);
callback_t   gcStartCB = registerCallBack(gcStartEvent, context, gcStart);
callback_t   gcEndCB   = registerCallBack(gcEndEvent, context, gcEnd);
static long  timestamp = 0;
gcStartCB.enable();
gcEndCB.enable();

... // application execution

gcStartCB.unregister();
gcEndCB.unregister();
stats.unregister();
```

Figure 4

PEMAPI example to count number of page faults and data cache misses between garbage collection (GC) invocations.

context, an operation is executed that summarizes the event. The operation may count the number of times this event occurs or compute the maximum, minimum, or average values of one of the event attributes. The statistics level usually leads to the loss of the event attributes. For example, a count of page faults contains only the number of page faults, and not the set of faulting addresses.

Example

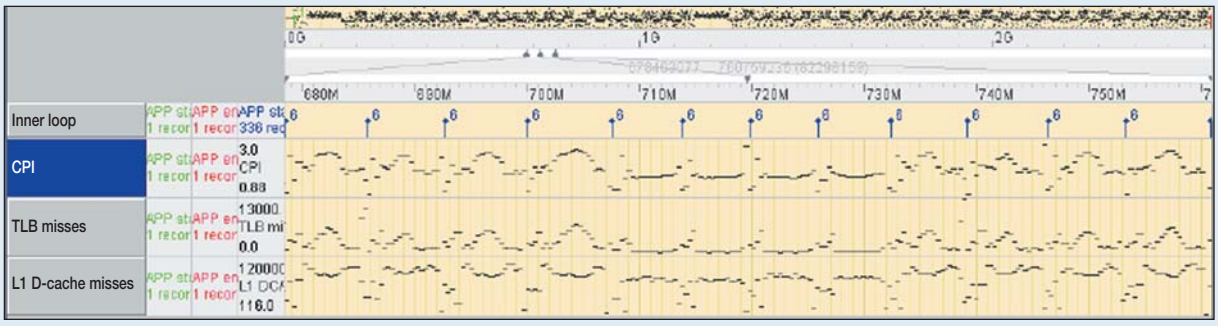
We illustrate the use of the PEMAPI with a specific example (**Figure 4**). In this example, a PEM consumer counts the number of page faults and data cache misses that occur between invocations of the garbage collector (GC) in a JVM.

A process-specific context (`context`) is created for this JVM. An event set (`events`) is defined to contain the two aggregated events. The event set is registered at the statistics detail level using the `COUNT` operation. A callback is registered for the end of a GC with the `gcEndEvent` event in the `context` context such that, when the JVM ends a GC, the callback routine `gcEnd` is invoked with the `gcEndEvent` log record passed as the first parameter. The `gcEnd` callback saves the timestamp at the end of GC in the static variable `timestamp` and enables and resets the `stats` handle, which counts the number of page faults and data cache misses. A second callback is registered for the start of a GC with the `gcStartEvent` event in the `context` context such that, when the JVM starts a GC, the callback routine `gcStart` is invoked with the `gcStartEvent` log record passed as the first parameter. This callback disables the statistics handle and logs the GC end and start timestamps and the number of page faults and data cache misses that occurred since the last GC. After both callback handles are enabled, one of the callbacks is triggered whenever this JVM starts or ends a GC.

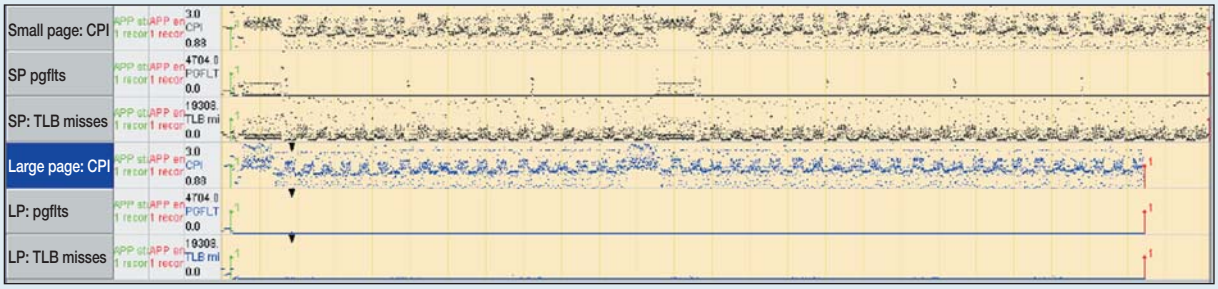
By unregistering the handles, the JVM stops counting the number of page faults and data cache misses that occur between GC events. The XML specifications for the events `pageFault`, `dataCacheMiss`, `gcStart`, and `gcEnd`, and the `logBetweenGCs` routine are not shown here.

Visualization client

This section presents our first experience using PEM with a vertical performance visualization client that enables users to understand the correlation of events across layers in the execution stack. We have collected PEM event traces for UMT2K [13], a scientific application, running on K42 on an Apple Power Mac** G5 [14] with two IBM PowerPC* 970FX processors. The UMT2K application, written in a mix of C and Fortran, is a three-dimensional



(a)



(b)

Figure 5

(a) PE view of UMT2K across execution layers for small pages. (b) PE view of UMT2K comparing metrics for small- and large-page executions. (SP = small page; LP = large page.)

deterministic multigroup photon transport code for unstructured meshes. We collected events across the following execution layers:

- *Application layer:* The core of UMT2K takes place in a loop nest, and we instrumented the code to emit phase marker events that indicate the iterations in the loop nest. Although we instrumented the application manually, the insertion of phase markers at code boundaries, such as loops and subroutines, can easily be automated in a compiler.
- *OS layer:* K42 was instrumented to emit a variety of OS events, including context switches, system and kernel calls, locking and synchronization events, and page faults.
- *Hardware layer:* The PowerPC 970FX provides a rich set of hardware performance counter events. Among others, we collected cycles per instructions completed (CPI), L1 data cache load misses (L1 D-cache misses), and data translation lookaside buffer (TLB) misses.

We used the PEM toolset to build a CPO offline agent that acts as a bridge between the raw event stream coming from the PEM runtime and the event requirements of the PE visualizer. PE expects complete context information to be available in every record in the visualization event stream. The semantic information about what constitutes the event context is fully contained in the XML event specification. Thus, we were able to write an agent-specific PEM tool to automatically generate the code for the CPO offline agent, in this case, as Java classes.

We used the vertical performance visualization client to investigate the effects that different memory page sizes have on performance. The PowerPC 970FX supports two page sizes: small (4 KB) and large (16 MB). Using large pages generally increases application performance by reducing TLB misses and page faults.

First, we produced a trace for UMT2K with a baseline small-page allocation. **Figure 5(a)** shows a view of the UMT2K event trace in PE. The *x*-axis represents time and the *y*-axis represents four event plots that show data for a selected interval. The overview plot at the top of the figure shows the entire run for the highlighted CPI plot, and the top arrows indicate the selected interval in the overview plot. Through the vertical integration of events in the trace, we can analyze, within the same view, application, library, virtual machine (if applicable), OS, and hardware events. In this view we are analyzing application and hardware events. The *Inner loop* strip shows the application markers for an innermost loop

(phase 6). Each marker indicates the beginning of a loop iteration. The remaining strips show frequency plots of hardware events, with each line segment indicating the frequency of events for a four-million-cycle interval. The first frequency plot shows CPI. In general, the more cycles that are required to execute an instruction, the worse the application performance. Thus, the lower the CPI line segment, the more performance improves. The next frequency plot shows TLB misses, and the last frequency plot shows L1 D-cache misses. Figure 5(a) is just one of the multiple views that PE provides, and we show only a sample of the data events that can be collected with PEM.

The second observation is the correlation of the behavior within each periodic phase (i.e., within each loop iteration). Figure 5(a) shows a strong correlation between TLB misses and CPI. As TLB misses rise during an iteration, CPI also rises, suggesting that TLB miss behavior is a strong factor in overall application performance. In contrast, the L1 D-cache misses both correlate and inversely correlate with CPI. When L1 D-cache misses correlate with CPI, such misses are a factor in overall application performance. A metric that inversely correlates with CPI indicates that it is not a factor in overall application performance. For example, when CPI rises and L1 D-cache misses fall, it is less likely that L1 D-cache misses are causing CPI to rise.

To explore the effect of large pages on performance, we ran a second experiment in which we mapped the heap data of UMT2K to large pages. **Figure 5(b)** shows a view of the event traces for both small and large pages. For the large page run, the number of page faults is reduced by 99.34%, and the number of TLB misses is reduced by 98.55%. The overall execution time is reduced by 6.6%.

We used the visualization client together with other tools to understand which situations benefit most from large pages. We then applied the results of this investigation in our second CPO client, which automatically predicts and exploits the benefits of large pages.

Page-size client

Our second CPO client automates the manual process used in the visualization client for exploiting large-page benefits. This client consists of an offline and an online CPO agent that cooperate to optimize large-page usage. The offline agent models page allocation behavior using data gathered from the hardware, OS, and application events. The predictive model produced by the offline agent allows the online agent to optimize page usage across different applications and varying inputs. Details of the page-size modeling can be found in [15]. Here we provide an overview.

The first time an application is run on the system, the online agent programs PEM to collect a trace consisting of data address samples and TLB misses using hardware counters, page-fault information from the OS, and memory allocation events from the runtime library. This trace is used by the offline agent to model the behavior of the application and predict the relative benefits of using large pages for different application data structures. The analysis results are stored as a set of page-size directives in the CPO database. For subsequent invocations of the application, the online agent examines the database and the number of large pages currently available and determines which data structures to back with large pages. Consistent with the CPO paradigm, these subsequent runs are monitored to validate that the chosen mapping had the predicted benefits.

The offline agent takes as input a stream of memory references and a stream of memory allocation events from the collected PEM trace. The agent transforms the memory reference stream into a page reference stream using the memory allocation events and then applies reuse distance analysis [16] to compute the number of TLB misses and page faults for different mappings of data structures to large pages. The result of the analysis is a ranking of individual mappings based on relative benefits over the baseline all-small-page mapping.

The ranking is expressed as *percentage of maximal benefits* (PMB). The mapping with the greatest reduction in TLB and page-table misses has a PMB ranking of 100%. Other mappings will have a fraction of the maximal benefits, between 0 and 100%. An example of the benefit ranking for the SPECfp**2000 benchmark `galgel` is presented in [15]. The benefit ranking shows the percentage of maximal benefits against the number of large pages that are needed for five different mappings. The five mappings map all, none, or a certain subset of the categories to large pages. The figure shows both the PMB ranking predicted by the model and the actual ranking that results from execution times.

We evaluated the page-size agent using the SPECfp2000 [17] benchmark suite and two large scientific applications, UMT2K [13] and RF-CTH [18], by comparing the predicted PMB ranking with the actual PMB ranking. We compute the ranking error per mapping to be the distance between the predicted and the actual PMB rankings. The overall ranking error for a benchmark is determined as the average ranking error across all mappings. For example, in **Figure 6**, the ranking error is 6.59%. The geometric mean of the ranking error over the entire set of benchmarks is 3.72%, indicating that the CPO page-size agent makes accurate benefit predictions and is effective in making optimal page-size mapping decisions. Our initial implementation of the page-size client was on the K42 operating system, and we have recently ported the client to IBM AIX.

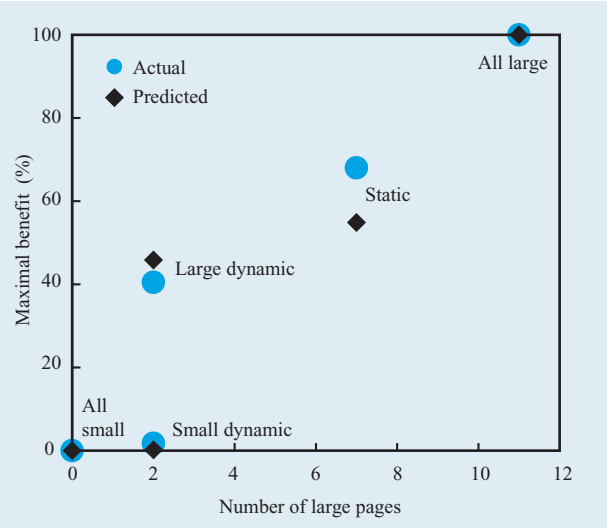


Figure 6

Benefit ranking for `galgel`. ©2005 IEEE. Reprinted from [15] with permission.

Related work

There is a significant amount of work related to feedback-directed and dynamic optimization. A comprehensive survey of dynamic JIT compiler optimizations in virtual machines can be found in [19]. In the context of virtual machines, the term *continuous compilation* has been used to describe an approach in which compilation is overlapped with program interpretation and native execution [20]. Dynamic optimization has also been applied to native binaries without compiler support [21].

There are a number of previous approaches that, like CPO, emphasize the aspect of continuity in the optimization process. The work by Kistler and Franz [22] describes an approach to continuous program optimization that has been applied to data layout optimizations in object-oriented programs based on online feedback. Childers et al. [23] describe an approach to continuous compilation that applies aggressive code optimization at all times, from static optimization to online dynamic optimization. The continuous compilation framework has been demonstrated for adapting application code in embedded systems and has also been used to describe a hardware dynamic optimization mechanism to optimize an application instruction stream [24].

There has been considerable work in tracing for understanding kernel behavior—LTT [25], KernInst [26], and K42 [12], and for understanding application behavior—Paradyn [27] and SvPablo [28]. In addition, DTrace [29] and CrossWalk [30] allow event collection across system call boundaries. OProfile is a profiler for Linux** for systemwide sampling of hardware performance counters [31]. Most of the previous work focuses on offline data collection and postmortem visualization. The main contribution of this paper is an infrastructure for vertical monitoring and online optimization. In addition to previous work, we enable data collection seamlessly across multiple layers of hardware and software. Our system is also designed to allow data to be processed online. If offline traces are desired, we showed how a simple client can be written to extract the data from the collection stream.

Other performance visualization tools include Intel VTune** [32], SGI SpeedShop** [33], Apple Shark [34], and Paraver [35]. Kimelman et al. [36] present Performance Visualizer, which focuses on presenting temporal information from multiple levels of the system. Hauswirth et al. [3] show that for understanding modern, object-oriented systems, a vertical approach to performance understanding across system layers is important, and thus we used the PE visualizer in this work.

As early as the 1990s, Romer et al. [37] acknowledged the potential usefulness of large pages. The same work also states that ... *good policies for superpages have been elusive [because] a cost benefit analysis is required to determine if the overhead of constructing a superpage is outweighted by its benefit*. Their work, as well as more recent work [9], migrates data to large pages reactively, with analysis performed at the OS level using data gathered by the system. In contrast, we developed a predictive model and use information gathered from across the execution stack. Current operating systems, such as Linux, AIX, and SunOS**, support restricted use of large pages. Commonly, a fixed pool of large pages is fixed at boot and may or may not be modified during system execution. If modification is allowed, it is typically an expensive operation. Thus, the ability to predict the benefits obtained from using large pages and the number of large pages required is helpful.

Future work

Work is ongoing to further strengthen the CPO framework and build additional CPO agents for other client optimizations. We have started work on a CPO agent that monitors Message Passing Interface (MPI) performance and attempts to tune MPI use and parameters. Once a sufficiently large number of concurrent CPO agents is available, we plan to address the challenge of coordinating multiple concurrent agents. With multiple optimizations taking place concurrently, there is an increased potential for conflict and adverse side effects from agent activity. We plan to address these potential conflicts by using PEM as a communication

channel so that concurrent agents coordinate their activities through established protocols.

Conclusions

We have presented a continuous program optimization (CPO) architecture to allow automatic tuning of application performance in today's complex hardware and software environment. The focus of this paper was on the performance- and environment-monitoring (PEM) component of CPO and how we use PEM to build CPO agents for performance tuning. PEM vertically integrates events from the various layers of the execution stack and thereby provides an integrated whole-system view of performance to CPO agents. We presented our XML specification for describing PEM events and our implementation of the PEM runtime on K42, a prototype open-source research OS.

We illustrated the use of PEM in two client applications. The first was a vertical performance visualization client that uses an offline agent to aid in program understanding and manual performance tuning. The second was a page-size client that uses both an offline and an online agent to automatically tune system performance by selecting the most beneficial data structures to map to large pages.

Acknowledgments

Many people contributed to the realization of the CPO vision and the PEM runtime: the entire K42 team and, in particular, Maria Butrico, Michal Ostrowski, and Bryan Rosenburg supported us throughout the implementation of the large-page scenario with OS features and machines. Reza Azimi implemented hardware counter support for the PowerPC 970FX in K42. Matthias Hauswirth, the creator of Performance Explorer, made extensive modifications to support visualizing PEM traces. Marina Biberstein helped with data visualization. This work was supported by Defense Advanced Research Project Agency Contract No. NBCH30390004.

References

1. J. Aycock, "A Brief History of Just-in-Time," *ACM Computing Surv.* **35**, No. 2, 97–113 (2003).
2. J. Appavoo, K. Hui, M. Stumm, R. W. Wisniewski, D. Da Silva, O. Krieger, and C. A. N. Soules, "An Infrastructure for Multiprocessor Run-Time Adaptation," *Proceedings of the 1st Workshop on Self-Healing Systems,* 2002, pp. 3–8.
3. M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical Profiling: Understanding the Behavior of Object-Oriented Applications," *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004, pp. 251–269.
4. R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Caşcaval, and R. Azimi, "Performance and Environment Monitoring for Whole-System Characterization and Optimization," *Proceedings of the PAC2 Conference on Power/Performance Interaction with Architecture, Circuits, and Compilers*, 2004, pp. 15–24.
5. M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *J. High Perform. Computing & Appl.* **18**, No. 1, 25–41 (2004).
6. R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," *Technical Report UT CS-97–366*, Computer Science Department, University of Tennessee, Knoxville, TN 37996, 1997.
7. M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, 1998, pp. 1381–1384.
8. X. Li, M. J. Garzaran, and D. Padua, "A Dynamically Tuned Sorting Library," *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, p. 111–124.
9. J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," *ACM SIGOPS Oper. Syst. Rev.* **36**, No. SI, 89–104 (2002).
10. C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, "System Support for Online Reconfiguration," *Proceedings of the USENIX Technical Conference*, 2003, pp. 141–154.
11. IBM Corporation,The K42 Operating System; see *http://researchweb.watson.ibm.com/K42/*.
12. R. W. Wisniewski and B. Rosenburg, "Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems," *Proceedings of Supercomputing*, 2003; see *http://researchweb.watson.ibm.com/people/b/bob/papers/sc03.pdf*.
13. The UMT Benchmark Code; see *http://www.llnl.gov/asci/purple/benchmarks/limited/umt*.
14. Apple Computer Inc., Power Mac G5; see *http://www.apple.com/powermac*.
15. C. Caşcaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Multiple Page Size Modeling and Optimization," *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques,* 2005, pp. 339–349.
16. R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.* **9**, No. 2, 78–117 (1970).
17. SPEC CPU2000 V1.3, Standard Performance Evaluation Corporation; see *http://www.spec.org/cpu2000*.
18. E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington, "CTH: A Software Family for Multi-Dimensional Shock Physics Analysis," *Proceedings of the 19th International Symposium on Shock Waves,* 1993, pp. 377–382.
19. M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A Survey of Adaptive Optimization in Virtual Machines," *Proc. IEEE* **93**, No. 2, 449–466 (2005).
20. M. P. Plezbert and R. K. Cytron, "Does 'just in time' = 'better late than never'?," *Proceedings of the 24th ACM Annual Symposium on Principles of Programming Languages,* January 1997, pp. 120–131.
21. E. Duesterwald, "Design and Engineering of a Dynamic Binary Optimizer," *Proc. IEEE* **93**, No. 2, 436–448 (2005).
22. T. Kistler and M. Franz, "Continuous Program Optimization: A Case Study," *ACM Trans. Program. Lang. & Syst.* **25**, No. 4, 500–548 (July 2003).

23. B. Childers, J. W. Davidson, and M. L. Soffa, "Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation," *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003, pp. 205.1.
24. B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, "Continuous Optimization," *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* 2005, pp. 86–97.
25. Opersys inc., Linux Trace Toolkit; see *http://www.opersys.com/LTT/index.html*.
26. A. Tamches and B. P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 117–130.
27. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer* **28**, No. 11, 37–46 (November 1995).
28. L. A. De Rose and D. A. Reed, "SvPablo: A Multi-Language Architecture-Independent Performance Analysis System," *Proceedings of the International Conference on Parallel Processing*, 1999, pp. 311–318.
29. B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," *Proceedings of the USENIX 2004 Annual Technical Conference*, 2004, pp. 15–28.
30. A. V. Mirgorodskiy and B. P. Miller, "CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary," *Proceedings of the International Conference on Parallel Computing*, 2003, pp. 745–752.
31. W. E. Cohen, "Multiple Architecture Characterization of the Linux Build Process with OProfile," *Proceedings of the IEEE International Workshop on Workload Characterization*, 2003; see *http://people.redhat.com/wcohen/wwc2003/wwc2003a.ps*.
32. Intel Corporation, Intel VTune Performance Analyzers; see *http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm*.
33. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Proceedings of the ACM/IEEE Supercomputing Conference*, 1996, p. 16.
34. Apple Computer Inc., Shark; see *http://developer.apple.com/tools/performance/*.
35. European Center for Parallelism of Barcelona, Paraver; see *http://www.cepba.upc.es/paraver/*.
36. D. Kimelman, B. Rosenburg, and T. Roth, "Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior," *Proceedings of the Conference on Visualization,* 1994, pp. 172–178.
37. T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," *Proceedings of the 22nd Annual International Symposium on Computer Architecture,* 1995, pp. 176–187.

Received June 21, 2005; accepted for publication August 3, 2005; Internet publication March 1, 2006

Călin Caşcaval *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (cascaval@us.ibm.com)*. Dr. Caşcaval is a Research Scientist and manager of the Programming Models and Tools for Scalable Systems at IBM Research. He received his Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign. He has worked on system software for the Blue Gene and PERCS (DARPA HPCS) projects. Dr. Caşcaval's interests are in programming models and languages for scalable systems, performance analysis, and optimization tools and compilers for large parallel systems.

Evelyn Duesterwald *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (duester@us.ibm.com)*. Dr. Duesterwald is a Research Staff Member in the Emerging System Software Department. She received M.S. and Ph.D. degrees in computer science from the University of Pittsburgh. Her research interests include programming language implementation, with a focus on runtime systems for adaptive and dynamic optimization. She has published in the areas of dynamic and static compiler optimization, program analysis and software development, and performance analysis tools. Dr. Duesterwald is coinventor on numerous patents filed on dynamic optimization technology.

Peter F. Sweeney *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (pfs@watson.ibm.com)*. Mr. Sweeney is a Research Staff Member. He received B.S. and M.S. degrees in computer science from Columbia University. His research interests include understanding the behavior of object-oriented programming languages to reduce their space and time overhead. Mr. Sweeney works in automating performance analysis and tuning.

Robert W. Wisniewski *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (bobww@us.ibm.com)*. Dr. Wisniewski is a Research Scientist. He received a Ph.D. degree from the University of Rochester. His research interests include scalable parallel systems, first-class system customization, performance monitoring, and using performance-monitoring information for continuous programming and system optimization. Dr. Wisniewski is working on the K42 OS and is exploring scalable, portable, and configurable next-generation operating systems.