PiPA: Pipelined Profiling and Analysis on Multicore Systems

QIN ZHAO

Massachusetts Institute of Technology

IOANA CUTCUTACHE

Duke-NUS Graduate Medical School

and

WENG-FAI WONG

National University of Singapore

Profiling and online analysis are important tasks in program understanding and feedback-directed optimization. However, fine-grained profiling and online analysis tend to seriously slow down the application. To cope with the slowdown, one may have to terminate the process early or resort to sampling. The former tends to distort the result because of warm-up effects. The latter runs the risk of missing important effects because sampling was turned off during the time that these effects appeared. A promising approach is to make use of the parallel processing capabilities of the now ubiquitous multicore processors to speed up the profiling and analysis process. In this article, we present *Pipelined Profiling and Analysis* (PiPA), which is a novel technique for parallelizing dynamic program profiling and analysis by taking advantage of multicore systems. In essence, the application under examination is profiled using a dynamic instrumentation tool. Optimized instrumentation code outputs the profile information in a succinct format, that we call the REP format, to buffers. This lightweight trace compression minimizes the processing overhead impinged on the application whenever a buffer is full. Another thread recovers the required information from the REP buffer. The recovered full profile is then divided up and passed to multiple threads for further analysis. To achieve the best performance, the entire system has to be well-balanced. We have implemented prototypes of PiPA using two dynamic instrumentation systems, namely DynamoRIO and Pin, thereby demonstrating its portability. Our experiments show that PiPA is able to speed up the overall profiling and analysis tasks significantly. Compared to the more than $100\times$ slowdown of Cachegrind and the $32\times$ slowdown of Pin dcache, we achieved a mere $10.2\times$ slowdown on an 8-core system. In this paper, we will also describe the insights we gained in obtaining the balance needed for PiPA to perform optimally.

Categories and Subject Descriptors: D.2.11 [Software Engineering]: Software Architectures—*Patterns (pipeline)*; D.3.4 [Programming Languages]: Processors—*Run-time environments*

General Terms: Design, Experimentation, Performance

Correspondence Author: Q. Zhao MIT; email: qin.zhao@mit.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1544-3566/2010/12-ART 13 \$10.00
DOI 10.1145/1880037.1880038 <http://doi.acm.org/10.1145/1880037.1880038>

ACM Transactions on Architecture and Code Optimization, Vol. 7, No. 3, Article 13, Pub. date: December 2010.

Additional Key Words and Phrases: Analysis, dynamic instrumentation, multicore systems, parallel cache simulation, pipelining, profiling

ACM Reference Format:

Zhao, Q., Cutcutache, I., and Wong, W.-F. 2010. PiPA: Pipeline profiling and analysis on multicore systems. *ACM Trans. Architect. Code Optim.* 7, 3, Article 13 (December 2010), 29 pages
DOI = 10.1145/1880037.1880038 <http://doi.acm.org/10.1145/1880037.1880038>

1. INTRODUCTION

Knowing and understanding an application’s dynamic behavior is invaluable in many research areas such as computer architectural design, as well as in software development, including functional and performance debugging. However, collecting such information, especially at very fine granularities, is tedious. It often requires running the application under examination at a significantly slower speed. This is made worse by the fact that many analysis require runs over a large data set or a substantial native execution time to overcome initial start-up or transient effects. Furthermore, the information collected from such runs are often too large to be stored away for offline analysis. The reference input run of SPEC 2000’s 172.mgrid has about 4×10^{11} memory references. Assuming that 8 bytes (4 for the PC and 4 for the memory address) are needed for each reference, then more than 1 TB of storage would be required. The reference runs of SPEC 2006 are significantly longer. The only option is to perform the analysis “on the fly” while profiling, which further worsens the runtime overhead. In practice, it is often necessary to rerun the analysis over many different configurations or inputs, making the turnaround time of each individual run a serious concern. Sampling can be used to mitigate the overhead, but it trades off accuracy and also runs the risk of missing out on key events. Sampling is also not appropriate if a high degree of accuracy and fidelity is required by the analysis task at hand.

An effective means to build customized online profilers and analyzers is to use dynamic instrumentation. Unfortunately, dynamic instrumentation-based analysis contributes to the performance problem mentioned previously. Besides the slowdown arising from the dynamic instrumentation system itself, the overhead attributable to user-specific analysis can significantly worsen performance. For instance, the dynamic instrumentation system, Valgrind, causes an average $5\times$ slowdown due to its translation between x86 instructions and U-code compared to native execution. If a complete profile of the memory accesses by instruction fetching and data referencing is required, the runtime slowdown increases to $20\times$. When running Valgrind with a cache simulator (Cachegrind), the slowdown goes up further to more than $100\times$ on average.

The proliferation of commercial, off-the-shelves multicore systems has prompted researchers to explore new approaches for off-loading parts of the profiling and analysis tasks onto spare hardware execution cores in order to improve performance. Such systems allow multiple threads or processes to run simultaneously. One such proposal is *parallelized slice profiling* [Moseley et al. 2007; Wallace and Hazelwood 2007]. The application under examination first starts running without instrumentation. It then periodically forks off new

processes to execute slices of original application code. These slices are instrumented with profiling and analysis code, and execute in parallel with the main application. This approach comes with several technical challenges, including how to guarantee that the slices' executions are identical to the main application, how to handle multithreaded applications, and how to merge the final analysis results. In addition, this approach is only suitable for simple, independent tasks like instruction counting, but faces difficulties in performing more complex tasks such as branch prediction and cache simulation. This seriously restricts its usefulness.

In this article we exploit another form of parallelism when mapping online profiling and analysis to multicore systems, namely pipelining. We describe a novel technique for parallel profiling and analysis that we called PiPA¹ (*Pipelined Profiling and Analysis*), extending on the work presented in our previous conference paper [Zhao et al. 2008]. Essentially, in PiPA, threads form a pipeline for collecting and processing profiles. However, rather than a simple pipeline, a better analogy for PiPA would be the out-of-order instruction processing pipelines of superscalar processors. The application under examination acts as the source of the pipeline. The processing of the collected profile is further divided up into several pipeline stages. At some of these stages, there could be more than one thread simultaneously processing parts of the profile. It should be noted that for reasons such as the isolation of the memory spaces, threads may be replaced by operating system processes with interprocess communication acting as communication mechanism between pipeline stages.

PiPA has the same goal as the parallelized slice profiling approach in reducing the profiling overhead through parallelization. However, it has several advantages over the previous techniques. First, it is a straight-forward model making it easier for users to understand and build customized analysis tools. Second, it avoids many technical difficulties and heuristics that are required in the implementation of parallelized slice profiling (e.g., system call handling and signature detection [Wallace and Hazelwood 2007]). It is also easier to achieve some desirable properties in the profiling and analysis process such as preserving exact ordering in the profile. Furthermore, PiPA can handle multithreaded applications easily by having one pipeline for each application thread.

In this article, we describe the design and implementation of a prototype of PiPA in DynamoRIO [Bruening 2004] and Pin [Luk et al. 2005] so as to demonstrate and assess its portability. We evaluated the performance of PiPA on several multicore systems. The experimental results show significant speedup over traditional approaches. We summarize the contribution of this article as follows.

- We present PiPA, a novel approach for parallel program profiling and analysis;
- We introduce REP (Runtime Execution Profile), a compact profile format for storing detailed execution profiles that is amenable to fast recovery;

¹The pipa is an ancient Chinese string musical instrument.

- We investigate a set of optimizations for collecting runtime execution information that are useful for both serial and parallel profiling;
- We present a way to parallelize trace-driven cache simulation using PiPA.

The remainder of the article is organized as follows: Section 2 provides the background of dynamic instrumentation, program profiling and analysis. Section 3 presents the overview of the design and implementation of PiPA. Section 4 describes the format of REP, and the collection, optimization and recovery of REP in PiPA. Section 5 discusses how a trace-driven analysis like cache simulation can be parallelized in PiPA. Experimental results using PiPA are presented and discussed in Section 6. In Section 7 we will examine some of the limitations of PiPA. This is followed by the conclusion and future work in Section 8.

2. BACKGROUND AND RELATED WORKS

2.1 Runtime Code Manipulation Systems

Runtime code manipulation is a powerful technique for runtime program introspection. There are many runtime code manipulation systems, and most of them have similar internal engines. Modified copies of the original application are executed in a code cache that preserves frequently executed blocks of code for future use. These runtime code manipulation systems can be categorized into different groups based on their applications. Dynamo [Bala et al. 2000], DynamoRIO [Bruening 2004], and ADORE [Chen et al. 2003] are dynamic optimization systems that are designed to speedup program execution by taking advantage of runtime information that is not available at compile time. Dynamic instrumentation systems such as Pin [Luk et al. 2005] and Valgrind [Nethercote 2004; Nethercote and Seward 2007] can be used to build customized program analysis tools. There are other applications of runtime code manipulation such as for dynamic translation [Ebcioglu and Altman 1997], security [Kiriansky et al. 2002], and reliability [Reis et al. 2005].

As an example of how such a system works we provide here a brief description of DynamoRIO's operation. DynamoRIO [Bruening 2004] is a dynamic instrumentation and optimization framework implemented for both IA-32 Windows and Linux. This system runs an application by copying it one basic block at a time into a code cache. After some modifications, the block is executed natively from there. Blocks in the cache are linked together via direct jumps or fast lookup tables so as to reduce the number of context switches to the DynamoRIO runtime system. In addition, DynamoRIO stitches sequences of hot code together to create single-entry multiple-exit traces that are stored in a separate trace cache for further optimization. DynamoRIO allows users to build clients using the APIs provided. These clients can manipulate the application code by supplying callback functions which are called by DynamoRIO before the code is placed in either caches.

2.2 Profiling

Profiling is a common technique used by compilers and application developers to understand the behavior of a program. Profiling may be done either by collecting data offline and then using it to guide the recompilation of the program, or by gathering the data at runtime and simultaneously performing optimizations. Some common types of profiling are path profiling, hot data stream profiling, and value profiling. The former [Ball and Larus 1996] collects sequences of executed blocks called *paths*, which capture the dynamic control flow of an application. The information about *hot* paths is commonly used to guide the optimizations.

Larus [1999] proposed a scheme called *Whole Program Path* (WPP) to capture the entire dynamic control flow in a compact fashion by using the Sequitur compression algorithm. The whole program paths are collected in two phases: first a trace of the acyclic paths executed by the program is determined, then this trace is compressed to a more compact form. Tallam et al. [2005] extended WPPs to also encode the memory dependencies, but their scheme incurs a large time and space overhead. Zhao et al. [2006] proposed the *Detailed Execution Profile* (DEP) as a more efficient method of collecting both control flow and memory reference information in a single pass. Instead of storing the memory reference addresses, DEP records and keeps track of the updates of the registers that are used for memory references. DEP reduces the profile size, but the process of reconstructing the memory reference information is more complicated, making it more suitable for offline analysis.

One of the major challenges in profiling is the substantial execution overhead caused by the extra profiling code. There have been a number of proposals to minimize this overhead. Sampling is a common technique that significantly reduces the runtime overhead. However, it fails when detailed continuous information is needed. Arnold and Ryder [2001] proposed a general framework that utilizes bursty tracing to reduce the profiling overhead. A similar idea was used in Ubiquitous Memory Introspection [Zhao et al. 2007], where a sampling-based method is used to select hot code regions for profiling and optimization. The insight is that frequently executed short memory profiles can be sufficient to reasonably approximate the real memory system's behavior. UMI was implemented on top of DynamoRIO [Bruening 2004], and it managed to maintain the overhead at an average of 14%, which is only 1% greater than the one incurred by DynamoRIO itself. However, sampling-based techniques does not yield full trace information provided by PiPA. Complete and detailed information is needed for tasks such as debugging, dependence analysis, parallelism discovery and architectural studies.

More recent research has focused on utilizing the increasingly popular multicore systems to reduce the overhead of profiling by off-loading profiling tasks to spare hardware cores. Shadow profiling [Moseley et al. 2007] runs the original uninstrumented application in parallel with instrumented slices to perform sampled profiling. SuperPin [Wallace and Hazelwood 2007] uses a similar approach, but tries to replicate the full program execution. Slices are periodically forked by the uninstrumented main process, either when a system call is

Fig. 1. The PiPA pipeline.

encountered or on a timeout. It then uses a signature heuristic to detect when such a slice should end so that it does not overlap with the next slice. This approach reduces the runtime overhead of profiling significantly. However, it is not suitable for analysis tasks that have state dependencies such as cache simulation and branch prediction simulation. These will require a large amount of communication between slices in order to maintain the dependencies between profiles.

Compared to shadow profiling and SuperPin, PiPA uses a different approach. It performs a very low overhead profiling in the same thread of application to produce compact profiles, and uses multiple threads as different stages of a pipeline to reconstruct the full profiles for analysis. Because the profiles are contiguous and processed in the same order as they are collected, PiPA is able to perform complex analysis that SuperPin cannot easily do. Furthermore, unlike shadow profiling, PiPA does not rely on sampling for achieving efficiency and, thus, provides 100% accurate analysis results.

3. PIPELINED PROFILING AND ANALYSIS

3.1 Design

Figure 1 depicts how PiPA works. The application under examination is instrumented with profiling code and executed in stage 0 of the pipeline. The collected profiles are passed to the thread at stage 1. This thread manipulates and reorganizes the profiles into specific formats for analysis in the later stages. In our example, it splits the profiles into subprofiles. These subprofiles are fed into several threads at stage 2 that perform the analysis in parallel.

Note that any of the threads can be easily replaced with operating system processes. For example, the threads in the first two stages can be in the same

process as the application that outputs the profile. The threads in stage 2 can be organized into one or more analyzer processes to perform parallel analysis.

There are three key challenges in PiPA design:

- Minimizing the profiling overhead in the application under examination,
- Minimizing the communication overhead between different pipeline stages,
- Coming up with efficient parallel analysis algorithms.

The speed at which the profiles can be produced is one of the most important determinants of PiPA's performance. No matter how good the parallel analysis algorithm is, it cannot run faster than the rate at which the profiles are produced. The latter is determined by the speed of the application and the overhead involved in profiling it. As the application under examination is given, in order to maximize the rate of profile production, the profiling overhead must be minimal. There are two keys for achieving low profiling overhead. The first is *double-buffering*. Profiling information is first collected in the first buffer with minimal processing. When this buffer is full, profile collection continues with a second buffer. Meanwhile, the first buffer is passed to the next stage of processing. Simple inlined code is used for filling the buffers. When a buffer is full, slightly more complex code is executed to hand the buffer over to the next processing stage. The second key is to have a profile format that is suitable for online profiling. An ideal profile format is able to reduce the profiling overhead by executing fewer instrumentation instructions. In the next section, we will introduce a profile format we called the *Runtime Execution Profile* (REP).

The second challenge is the reduction of the communication overhead when passing profiles between threads in different stages. Double-buffering also helps to reduce the number of synchronizations between threads. In addition, shared buffers are used so as to avoid data copying, further reducing the communication overhead. There are two buffers that are accessible to the producer and consumer threads in two consecutive stages. Accessibility to each buffer is controlled by a lock variable. The producer thread first fills one buffer while the consumer waits on that buffer lock. When the buffer is full, the producer continues on the other buffer, while the consumer obtains the first buffer lock and starts processing the filled buffer. In this way, the producer and consumer threads work on different buffers at the same time. The overhead of communication is therefore limited to acquiring the buffer locks. In the case where the producer is running significantly faster than the consumer, we can add more buffers and more consumers to parallelize profile consumption.

Parallelizing the analysis algorithm depends on what is done in the analysis. In this article, we will consider cache simulation. In particular, the simulation of set associative caches (the prevalent type of caches nowadays) can be parallelized by splitting the memory reference profiles into different sub-profiles based on the sets that each reference will access. Thus several cache simulators can execute in parallel to process different subprofiles independently. The results can be easily merged after the execution. Further details will be discussed in Section 5.

Profiling and analyzing multithreaded applications requires no significant change in PiPA's design, especially if the instrumentation system uses thread-private code caches. The profiling code is embedded in each thread of the application in order to extract the real execution trace of the thread, which is then fed to an analysis pipeline. Therefore, one pipeline would be used for each application thread. Some savings can be achieved if shared caches are used instead. However, this is entirely a design decision of the underlying instrumentation framework and does not affect the way PiPA works.

3.2 Implementation Overview

In order to demonstrate the effectiveness and efficiency of PiPA, we implemented a three-stage PiPA prototype using both DynamoRIO and Pin. It is entirely possible to use other dynamic instrumentation frameworks to implement PiPA.

When the application under examination starts executing under instrumentation, we first allocate $n > 1$ profile buffers, and spawn n *recovery threads*. These threads work as functional units in stage 1 of PiPA and have the task of reconstructing the profile from the information recorded in the buffers. Each thread is bound to one buffer, and in order to access this buffer it communicates with the application thread via two associated semaphores. This n -way buffering implementation is slightly different from the double-buffering design described above, but it simplifies the communication between the threads.

As the instrumented application executes, profiling code is inserted into any application code that the instrumentation engine copies into its basic block code cache. The profiling code records basic block and memory reference execution information into the current profile buffer. The instrumented code is optimized when frequently executed basic blocks are upgraded into the trace cache. In addition, a conditional check is also inserted to trigger a handler when the buffer is full. The handler releases the current buffer to the associated thread, and then tries to acquire the next empty buffer to act as the next active fill buffer.

As the application thread fills the buffers one at a time with profiling information, the recovery threads wait on their associated buffers' semaphore. When a buffer is released by the application thread, the corresponding recovery thread will reconstruct the information from it for analysis. After the entire buffer is processed, the recovery thread will release the buffer back to the application. If the analysis is simple (e.g., instruction counting), the analysis code can be implemented in the recovery thread. Alternatively, the recovery thread can write the reconstructed information into a shared buffer to be processed by another analyzer thread.

4. PROFILING

The efficiency of profiling hinges on a well designed profile format and carefully crafted instrumentation code. We will show in Section 6 that a naïve raw format performs very poorly. In this section, we first describe our novel profile format, REP, then discuss how to efficiently collect REP, and finally show

Fig. 2. An example of REP.

how full control flow and memory reference information can be extracted from REP.

4.1 Runtime Execution Profile

REP is a profile format designed for fast profiling, small profile size, and easy information extraction, making it suitable for online profiling and analysis. The key insight here is that a quick analysis of the code can yield information that minimizes the profiling overhead and the size of the profile information. Furthermore, at a higher cost of profile recovery, it is possible to compress the trace further. Trace compression reduces the need for the application thread (the producer) to synchronize with the consuming threads as buffers take a relatively longer time to fill up. Efficiency is achieved by passing the recovery cost to another thread. However, there is a need to balance the compression ratio achieved with the work needed to perform the compression as the latter will increase the overhead of profiling.

Figure 2 shows an example of a REP. A REP is pointed to by a base pointer. It consists of a number of contiguous *profile buffers* separated by special “canary” zones. These canary zones are initialized with the value 0xf0f0f0f0 and their purpose is to detect when the limit of a buffer is reached. Each profile buffer consists of a sequence of data *units* and each unit consists of a number of *slots*. A REP unit reflects the execution of a basic block and it stores the static and dynamic information associated to that execution using two types of slots: REP_S and REP_D , which are detailed below. A unit would start with a single REP_S slot followed by a variable number of REP_D slots. The next available unit is pointed to by a *profile counter*.

—The REP_S is a pointer to a data structure that stores static information about the associated basic block, including a tag that distinctly identifies the

basic block, the number of REP_D slots following the REP_S slot, the number of memory references in the basic block, and a pointer to a second level structure. This second level structure holds information regarding each memory reference, including the type of the reference, the size of the reference (if known statically), the constant offset, the slot number of the REP_D slot to be used in the address computation, and the REP_D slot number that holds the dynamic size of the reference.

- Each REP_D slot stores some dynamic information collected during the basic block's execution. These may include the contents of registers, memory reference addresses, and memory reference sizes that are not statically known. It should be noted that the same register may be saved multiple times in the same unit if it is used for different memory references and is overwritten between them. Because each basic block has a different amount of dynamic information to be stored, the number of REP_D slots varies.

The `size_slot` field in REP_S is used when the size of a reference can only be determined at runtime. In the case of the x86 architecture this happens for string instructions. For example, the instruction `rep movs` will move a number of bytes from the address `[esi]` to the address `[edi]`. The number of copied bytes is given by the value of register `ecx`. In this case, the value of `ecx` will be saved in a REP_D slot and `size_slot` will contain the slot number associated with it. If the size of a reference is known statically, this field will contain the value `-1`.

From REP_S and REP_D , we can reconstruct the full control flow and data access information of an execution instance of a basic block through a symbolic execution of the basic block. As an example, suppose we want to find the memory address referenced by the `pop` instruction in `bb1` of Figure 2. Following `bb1`'s REP_S , we find that the `pop` instruction corresponds to the second memory reference of the block. The field `value_slot` informs us that the value of the register to be used in the address computation is found in slot 2, where the `esp` was stored. The value in this slot is added with `offset` to get the memory reference address.

What we have described thus far is the actual REP format we used in our experiments. For different analysis, however, different kinds of information are required. Depending on the situation, REP_D will have to be customized for the analysis that one has in mind. For instance, when studying dynamic control flow information, REP_D is empty as REP_S is enough to reconstruct the entire dynamic control flow. When studying memory reference behavior, for example, a naïve approach is to use REP_D to store all of the memory reference addresses and reference sizes. Using this approach, it is easy to reconstruct the full memory reference information (i.e., `<pc, address, type, size>`), as follows: `pc`, `type` and `size` (if static) are obtained from REP_S , while `address` and any dynamic `size` can be read from REP_D . Alternatively, a smaller profile size can be achieved if we modify the way addresses are stored and recovered. For instance, to profile the instruction `mov 0 -> [eax+16]`, 16 is stored in `offset`, and we only need to store the value of register `eax` in REP_D . This removes the need to do address calculation in the instrumentation code, further

removing the need to steal an extra register (which would be needed for this computation).

There are some additional aggressive optimizations that can further reduce the size of REP_D . In the case where there are several references accessing different members of the same data structure, only the base address of the data structure needs to be recorded. Also, the memory reference addresses of a sequence of push or pop can be reconstructed from a single stack pointer value recorded in the REP_D . This last optimization is illustrated in the example given in Figure 2 where only one esp value was saved for the two stack references done by the pop and return instructions of bb1.

4.2 Instrumentation

There are five main tasks to do in the instrumentation code:

- (1) context switching so as to preserve the correctness of the execution of the application under examination (this consists of saving and restoring the values of the registers that are used by the instrumentation code);
- (2) calculating the address of a memory reference;
- (3) recording the address into a profile buffer;
- (4) updating the profile counter; and
- (5) checking if the buffer is full.

A carefully crafted instrumentation code for each of the above tasks can significantly reduce the profiling overhead. The next two subsections describe several optimizations that can be used for this purpose and were implemented in our DynamoRIO and Pin prototypes. The DynamoRIO implementation features more aggressive optimizations, as this instrumentation engine provides flexible APIs which allow a better control of the type of instructions used for instrumentation. Some of the proposed optimizations could not be implemented in the Pin version due to limitations in the provided API.

4.2.1 The DynamoRIO Prototype. DynamoRIO [Bruening 2004] features a rich API, which allows the user to manipulate existing instructions and generate new instructions by specifying the opcode and operands. Therefore, it permits the developer to use and insert specific cheaper instructions for instrumentation. Furthermore it provides routines for spilling registers to DynamoRIO's own thread-private spill slots, and for saving and restoring the arithmetic flags using an instruction sequence that is much faster than pushf/popf. Using this API, it is easy to implement aggressive optimizations that significantly improve the performance. Following is a description of the optimizations designed to lower the overhead of the five mentioned instrumentation tasks. It should be noted that some of these proposed optimizations specifically target the x86 architecture that was employed in our experiments.

First, in most cases, only register values are stored in the REP. This removes the need to perform memory address calculation. Therefore, in these cases, only one register is needed for holding the profile counter. Otherwise, an extra register is required for storing the computed memory reference address. In order to

be fast, such an address computation is done using the x86 `lea` instruction. This instruction computes efficiently the effective address of the source operand (a memory reference specified using one of the processor's addressing modes), and stores it in a destination register.

Second, the same `lea` instruction is used instead of `add` to update the profile counter. More specifically, `add reg update -> reg` can be replaced with `lea [reg + update] -> reg`. Unlike the `add` instruction, the `lea` instruction does not change the `eflags`. Doing away with the need to save and restore `eflags` significantly improves the overhead due to context switches.

Third, instead of modifying the profile counter on each profile update, all the changes are combined into a single update when recording the `REPS` data structure.

The buffer full check is performed when the profile counter is updated. As described in the previous section, the end of each profile buffer is guarded by a special canary zone consisting of the value `0xf0f0f0f0`. Because there are several buffers in use, to check for a full buffer using the profile counter would require first locating and fetching the corresponding buffer limit. Therefore, instead of checking the profile counter's value, it is more efficient to check if a canary value was hit. If so, then the buffer is full.

The buffer full handler performs two tasks. First the handler signals the recovery thread to start working on the filled buffer by performing a V-operation on the associated semaphore. Next, it switches to the next empty buffer, returning when it successfully acquired one. The buffers are switched by simply changing the profile counter's value.

There are several other well-known optimizations that can reduce the profiling overhead. First, to perform fast context switches, a one-time register liveness analysis for each basic block is performed to discover if there are registers that can be used without stealing. The second optimization combines profile updates. Several profile updates can be combined together if the register values that must be saved are not overwritten. The total number of instructions needed to steal the necessary registers can thus be reduced.

More aggressive optimization can be performed when DynamoRIO upgrades frequently executed basic blocks into traces. By taking advantage of the single entry, multiexits nature of a trace, the check for a full buffer in consecutive basic blocks of a trace can be removed. To do this, the size of the canary zone at the end of a buffer is chosen such that it is greater than the amount of information any one trace may write. This way, if a check at the beginning of a trace confirms that the buffer is not yet full, then no trace's execution will exceed the canary zone.

4.2.2 The Pin Prototype. Pin [Luk et al. 2005] is an efficient dynamic binary instrumentation framework designed at Intel. Compared to DynamoRIO, it has the advantage that it supports both binaries for IA32 and AMD64/Intel64 processors, but the main drawback is that it does not provide a mechanism for generating new instructions that can be inserted in the original code. What it offers is a call-based API, and the instrumentation can be done only by injecting calls to specific analysis routines. In order to improve the performance, Pin uses

a just-in-time compiler that optimizes the instrumentation code: it automatically performs register allocation, inlining, liveness analysis and instruction scheduling.

With the experience from the DynamoRIO implementation, it took us merely one to two man-months to reimplement PiPA in Pin. This is evidence of the portability of the main idea behind PiPA. The optimizations we implemented in the Pin PiPA are similar to the ones described in the previous section. However, due to the limitations of call-based instrumentation, we cannot use specific cheaper instructions as before and we do not have control over which registers are used in the instrumentation code. Due to the latter limitation, a register liveness analysis is not useful here. The following is a description of the differences between the two implementations.

The Pin implementation of REP records register values in a way similar to that for the DynamoRIO implementation explained in Section 4.1. In DynamoRIO PiPA memory addresses were sometimes computed directly and stored in REP. However in the Pin PiPA we never compute memory addresses while profiling as we cannot introduce a `lea` instruction in order to do this quickly. Therefore, this computation is always done in the recovery stage. In this implementation we also try to combine several profile updates if they record the value of the same register which was not modified in between the updates. In order to reduce the overhead of the calls inserted in the original code, we attempted to take advantage of the Pin's just-in-time optimizations and we designed our analysis routines (i.e., the routines that contain the injected instrumentation code) in such a way that they are inlined by Pin. These routines have the job of saving one or more values in the static or dynamic slots of the profile buffer. In order to be inlined the routines need to contain straight line code, without branches. Therefore we implemented several routines which can save from one up to nine values in the buffer (in the worst case we need to save all eight available registers plus the pointer of the static slot). Depending on how many values are saved at each update, we call the corresponding routine and we made sure that Pin inlines it.² The inlining can help significantly as the overhead of performing a procedure call is eliminated.

The buffer full check is done in a way similar to the DynamoRIO PiPA implementation. As Pin creates traces for all the executed code, we insert the checks only at the beginning of these traces and set the size of the canary zones large enough to contain an entire trace.

4.3 Profile Recovery

As mentioned in Section 3.2, every profile buffer is associated with a recovery thread that waits for it to be full. When a buffer is full and the instrumented application thread releases it, the recovery thread will start performing the reconstruction task by scanning the REP units in the buffer one by one.

²This check can be done by running Pin with the special flags `-xyzy -meson log_instrument`. The effect is that Pin will generate a log file that contains information about the instrumentation, including which analysis routines were inlined.

Let us use the recovery of memory references as an example. The recovery thread first gets the REP_S pointer to retrieve the static information of the basic block. For each memory reference, we are able to obtain the instruction program counter (pc), the reference type (read or write), the access size and the offset value. From the corresponding slot in REP_D , we obtain the dynamic value of the base register. In some cases of more complex addressing, the actual address is calculated during profiling and stored instead. The offset then would be zero. The addition of the offset and the REP_D value gives the memory reference address. Having recovered all the memory reference information for the current basic block instance, we move on to the next REP unit by using the stride information of the current unit.

After the entire profile buffer is processed, the canary zone is reset, and the buffer is released back to the application thread. Before the application under examination exits, it will notify all the recovery threads by writing a special word at the end of each buffer. Each recovery thread will process whatever that is in its buffer and exit.

5. PARALLEL CACHE SIMULATION

In this section we will describe the implementation of cache simulation as an example to show how parallel analysis algorithms can be realized using PiPA.

Minimizing interthread communication is an important way of obtaining good performance in parallel programs. In the context of cache simulation, one simple approach to achieve this is to split the address trace into different groups that are independent. The dependence here refers to the dependencies between updates of the cache simulator's state. For instance, in a set associative based cache, two memory references that access two different sets of the cache are not dependent on each other. This simple observation gives an effective way to parallelize a cache simulator: the sets of the cache are partitioned and simulated by independent simulators. Each of these simulators are fed from address subtraces obtained by segregating addresses in the main trace using their set indexes. The simulators do not need to communicate with one another except at the end of the simulation when their results have to be combined.

In order to evaluate the benefits of such parallelization, we implemented a parallel cache simulator as a stand-alone process that communicates with stage 1 of PiPA via semaphores and shared memory. This parallel cache simulator consists of several independent simulators implemented using different threads. The PiPA recovery threads at stage 1 are modified accordingly to segregate memory references into the shared memory buffers.

We use double buffering in order to reduce the overhead of the communication between the recovery threads at stage 1 and the simulators at stage 2. A doublebuffer consists of two shared memory buffers and, as described in Section 3.1, it allows a producer recovery thread and a consumer simulator to work in parallel. Each shared memory buffer has a maximum size of 2MByte, limited by the OS. The access to each shared buffer is controlled by two associated semaphores. The total number of shared memory buffers is dependent on the number of recovery threads and the number of cache simulator threads. As an

example, if we use 8 recovery threads and 8 cache simulators, we need 8 double-buffers (16 memory buffers) for each cache simulator, and therefore a total of 128 shared buffers. Assuming 2MByte buffers we would allocate 256MByte of shared memory in total.

Each PiPA recovery thread reads the profiles collected in its associated REP buffer, reconstructs the memory reference information as described in the previous section, and puts it into one of the 8 double buffers according to the cache set it accesses (assuming that 8 simulators are used in stage 2). When a recovery thread finishes processing all the units from its REP buffer, it releases this buffer back to the profiling thread at stage 0, also writes special ending characters in the 8 double-buffers and releases them to the cache simulator. In this way, it notifies the parallel simulator to move on and communicate with the next recovery thread after processing the remaining data.

The parallel cache simulator works in a master-slave mode. The master thread communicates with PiPA to obtain the segregated memory reference profiles, and dispatches them to the slave threads. Each slave thread is an independent cache simulator that simulates a partition of the cache.

The master thread starts by communicating with the first recovery thread to obtain the shared buffers. The received buffers are dispatched to the slave threads to feed the cache simulation, and then returned back to the associated recovery thread after processing. The master and the slave threads continue working on the 8 double-buffers of the first recovery thread until encountering the ending characters. Then, the master thread switches to the second recovery thread, and obtains another 8 double-buffers for parallel cache simulation. Therefore, the simulator communicates with the PiPA recovery threads in a round-robin manner until the end of the simulation.

There are two major advantages of using this master-slave model. First, it simplifies the communication between stage 1 of PiPA and the cache simulator, and minimizes the dependency between the two stages' implementations. For instance, we can easily adjust the number of slave threads without modifying stage 1's implementation. Second, this mode allows dynamic workload balancing. For example, before the simulator switches to work on the 8 double-buffers of the next recovery thread, the master thread can check if the workload of the slave threads are balanced or not. If not, it can easily reconfigure the slave cache simulator threads and balance the workloads.

Most memory reference analysis, for example memory dependence analysis, can benefit from similar parallelization techniques. Branch prediction simulation can also be parallelized by appropriately segregating branch instructions according to their PC values.

5.1 The Design Space

As with any parallel applications, obtaining the optimal performance is often about getting the scheduling and load balancing correct. The current implementation reported in this paper consists of three parts, namely the profiling the application, REP recovery, and the analysis. A thread is used for each of the former two. For the analysis (i.e., cache simulation) part, a number of threads

Table I. The Multicore Systems Used in the SPEC2000 Experiments

No. of Cores	2	4	8
CPU Model	1 × Dual Core Intel Pentium D	2 × Dual Core AMD Opteron	2 × Quad Core Intel Xeon
CPU Frequency	3.2 GHz	2.4 GHz	1.8 GHz
L2 Cache Size/Core	1 MB	1 MB	4 MB
Memory Size	2 GB	4 GB	4 GB
Operating System	32-bit Linux 2.6.17	64-bit Linux 2.6.15	64-bit Linux 2.6.21

are used to simulate different partitions of the cache. This is appropriate when there are spare cores and the profiling and analysis efforts are comparable. However, when the analysis substantially dominates the overall performance, then one may consider using naïve profiling, which will reduce the need for a recovery thread. In the other extreme where the analysis is relatively light weight, the task of REP recovery can be combined with analysis. The correct configuration will largely depend on the tasks at hand as well as the resources available.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of PiPA using two suites of benchmarks: SPEC CPU2000 [SPEC 2000] and SPEC CPU2006 [SPEC 2006]. In Section 6.1 we present the initial experimental results obtained for SPEC CPU2000 using the DynamoRIO prototype and we study the impact of different PiPA parameters on the overall performance. Section 6.2 shows the results obtained for both DynamoRIO and Pin prototypes using the larger SPEC CPU2006 benchmarks. For our implementations we used DynamoRIO version 0.9.4 and Pin kit release 17236.

6.1 SPEC CPU2000 Results

We ran the experiments described in this section on three different multicore systems (listed in Table I) using the DynamoRIO PiPA prototype. We used the full SPEC CPU2000 suite of benchmarks, which were compiled with gcc 4.0 using the ‘-O3’ flags. All the runs were conducted using the reference input sets of the respective benchmarks.

6.1.1 Profiling Overhead. In the first set of experiments we assessed the runtime overhead of collecting REP profiles. There are two major factors that impact the profiling performance, namely the execution of the instrumented code and the size of the profile buffer.

We first fixed the profile buffer size to 16MB, and evaluated the profiling overhead of our profile code optimizations. The results for runs on the 8-core system are shown in Figure 3. The bars show the execution times of un-optimized and optimized profiling normalized to that of native execution. By “native execution” we mean running the benchmark binaries as they are without profiling, DynamoRIO or any instrumentation. “Optimized profiling” means using the

Fig. 3. The profiling overhead on the 8-core system with and without optimized instrumentation.

Table II. Profiling Performance with and without Optimization

Systems	INT2000		FP2000		SPEC2000	
	opt	w/o opt	opt	w/o opt	opt	w/o opt
2-core	3.60	7.28	1.98	5.13	2.27	6.13
4-core	4.20	8.27	2.27	5.79	3.16	6.94
8-core	4.99	9.90	2.80	6.94	3.81	8.30

optimizations described in Section 4.2.1, while “unoptimized profiling” refers to collecting REP without any of the proposed optimizations. Our optimizations clearly improve the profiling overhead significantly.

Table II shows the average profiling performance on the three different systems. The profiling code runs in the same thread as the application under examination. Differences in the slowdown shown in the table are therefore mainly due to the different CPU models, not the number of cores. The results are normalized to the native execution time. For all experiments described from here on, we will use the optimized instrumentation code.

In the next experiment, we studied how the profiling overhead is affected by the size of the profile buffer. We varied the profile buffer size from 1KB to 16MB. The average normalized execution times for different profile buffer sizes on the three systems are shown in Figure 4. A similar pattern can be observed for all three systems. As the profile buffer size is increased, performance initially improves. This is due mainly to the reduction in the total number of invocations of the buffer full handler, and hence the number of buffer switches. Performance stabilizes after the buffer size is increased to 16KB. After certain points, (64KB in systems with 1M cache, and 256KB in the system with 4M cache), performance degrades. We attribute this to cache effects. Most likely, the large buffer interfered with the working set of the application. After significantly exceeding the L2 cache’s size, the performance again becomes stable since the situation cannot get any worse.

6.1.2 Profile Recovery Overhead. In the second set of experiments, we evaluated the full DynamoRIO PiPA framework performance in which both profiling and recovery are done in parallel on the multicore systems. The application thread collects the REP profiles. The recovery threads count the total number of memory references, and reconstruct the detailed reference information into the form of a $\langle pc, addr, size, type \rangle$ tuple, which is then copied into a

Fig. 4. The performance on the various multi-core systems when the profile buffer size is varied.

Table III.
The normalized profiling and recovery overhead on the 8-core system with different profile buffer sizes

	small (64k)	medium (1M)	large (16M)
INT2000	8.43	7.52	5.56
FP2000	5.08	4.39	3.51
SPEC2000	6.62	5.84	4.45

thread-private non-local data structure. This copying is necessary to keep the recovery code from being optimized away by the compiler.

We studied three factors that may affect performance, namely the size of the profile buffer, the number of recovery threads, and the number of CPU cores.

We first assessed how buffer size changes can affect the performance on the 8-core system. The number of recovery threads is set to 8 in this experiment to make sure there is enough parallelism. We chose three buffer sizes, one small (64KB), one medium (1MB), and one large (16MB). The total buffer sizes are therefore 512KB, 8MB and 128MB, respectively. The results in Table III show that the larger the buffer size, the better the performance is. That is because large buffer sizes allow the recovery threads to spend more time on consuming the profiles in buffers and less time on communication and synchronization.

Next, we fixed the buffer size to 16M, and varied the number of recovery threads. This set of experiments was executed on the 8-core system which had the maximum amount of hardware resources. The number of recovery threads is set to 0, 2, 4, 6, and 8. In the case of zero recovery threads, the profiling thread also has to perform recovery using a single buffer. In the other cases, there is exactly one buffer for each recovery thread. The results are shown in Table IV. As more recovery threads are added, performance improves due to the parallelism. From 6 to 8 threads, marginal utility sets in as the speed of profile production limits the overall performance.

We also studied how well PiPA performs on different multicore systems. In this experiment, we used 8 recovery threads, and 16M profile buffers, to make

Table IV.
The normalized profiling and recovery overhead on the
8-core system with different number of recovery
threads

# thr	0	2	4	6	8
INT2000	19.35	13.76	6.64	5.99	5.56
FP2000	13.39	9.63	4.36	3.76	3.51
SPEC2000	16.14	11.54	5.41	4.79	4.45

Table V. The Profiling and Recovery Overhead with
Different Number of CPU Cores

# cores	Normalized to native exec.			Normalized to profiling		
	2	4	8	2	4	8
INT2000	7.01	5.02	5.56	1.95	1.20	1.11
FP2000	4.45	2.96	3.51	2.25	1.30	1.25
SPEC2000	5.63	3.91	4.45	2.07	1.24	1.17

sure they can take advantage of any available hardware resources in all of the three systems. On the left of Table V, the execution times normalized to native execution are shown. The sequential version of recovery on the 2-core system has an average slowdown of 16.60 for INT2000,³ 12.56 for FP2000, and 14.42 overall. In contrast, the parallel version of PiPA running on two cores halved the execution time. Surprisingly, the 4-core system did better than the 8-core system. However, because the two systems are different, the profiling overhead on each is also different. If we normalize execution time against profiling time, rather than native execution time, we can see that on the 8-core system, because of more available parallel hardware, recovery can be done more efficiently.

We next studied the impact of using REP as opposed to using a naïve profile format that records the 4-tuple `<pc, addr, size, type>` for each memory reference. We implemented a version of PiPA that collects in the latter format. We also used the Pin atrace tool to collect in this tuple format. Again, memory references are counted, and the profile is copied to a thread-private, nonlocal data structure.⁴

As shown in Figure 5, PiPA using REP performs significantly better than the PiPA or Pin atrace that use the naïve format. It should be noted that PiPA-REP and PiPA-Naïve used here were implemented in DynamoRIO. There are two reasons why REP performs much better than the naïve profile. First, as mentioned before, REP enables several optimizations that lower the profiling overhead compared to the naïve profile collection, even though several common optimizations were applied to both. The lower profiling overhead allows profiles to be produced much faster than the recovery threads can consume them. The

³We shall use ‘INT200x’, ‘FP200x’, and ‘SPEC200x’ to refer to the integer, floating point, and combined SPEC200x benchmarks, respectively.

⁴We also tried to run this experiment using SuperPin, but we failed to get it to work properly with the released Pin tool set. Therefore, we were unable to compare PiPA with SuperPin.

Fig. 5. The runtime overhead of the three profiling approaches on the 8-core system.

Fig. 6. The cache simulation overhead for DynamoRIO PiPA and Pin dcache on 8-core system.

Fig. 7. The DynamoRIO PiPA speedup over Pin dcache on 4-core and 8-core systems.

other reason is that REP is more compact. For each memory reference, on average around 4 bytes are used, in contrast to the naïve format which requires 16 bytes. The smaller profile size allows for more references to be stored in a given profile buffer, thereby improving the recovery threads' computation to communication ratio. The cache effects caused by the profile buffers meant that naïve profile collection by PiPA is even slower than the serial naïve profile collection of Pin atrace.

6.1.3 Cache Simulation Overhead. Finally, we evaluated the effectiveness of the DynamoRIO PiPA in parallel cache simulation by comparing it against the Pin dcache simulator. To be fair, we used a cache simulator similar to the one provided in the Pin toolkit, with some modifications to handle our profile format. We used 8 recovery threads in PiPA, 8 slave cache simulator threads, 16MB of profile buffer, and two pieces of 2MB shared memory to feed each slave thread. In total PiPA used 256M of shared memory and 128MB of profile buffer.

We tested the simulations on the 4-core and 8-core systems. Figure 6 shows the normalized execution times of DynamoRIO PiPA cache simulator and Pin dcache simulator on the 8-core system, while Figure 7 shows the speedups

of DynamoRIO PiPA over Pin dcache on both 4-core and 8-core systems. It can be easily observed that PiPA outperforms Pin dcache significantly on both systems. On average, PiPA reduces the slowdown by a factor of three. In the best case (301.apsi), PiPA's speedup over dcache exceeds $5\times$. In general, the 8-core achieves slightly better speedups than the 4-core. However, there are a few benchmarks that exhibit a lower speedup when run under PiPA on the 8-core system compared to the 4-core. We believe this is due to the architectural differences between the two experimental systems, especially the size of L2 cache. Pin dcache does not use large buffers as PiPA does, and thus is more sensitive to the size of this cache. Therefore, some benchmarks (e.g., 171.swim, 189.lucas, 197.parser) ran with Pin dcache have a better cache locality on the 8-core. In contrast, the same benchmarks running under the PiPA based cache simulator benefit little from the large L2 cache, and thus PiPA's speedup over sequential dcache is lower on this machine.

Ideally, the cache simulator should be speeded up by $8\times$ on a 8-core system. However, there are several reasons that prevent the PiPA cache simulator from achieving that speedup. Firstly, PiPA introduces extra work in filling and reading the profile buffers and shared buffers. Secondly, we have already noted the cache effects that come with large buffer sizes. Another cause for this phenomenon is workload imbalance, encountered in the case when the profiles are biased towards some partitions. In such cases the corresponding cache simulators will have a heavier workload than the others. This is the case in 188.amm and 189.lucas. Also, different benchmarks may attain different profiling speeds, and, in some, profile production cannot keep up with the consumption by the cache simulators. As a result, most benchmarks can only achieve a $2\times$ to $5\times$ speedup relative to a sequential cache simulator (i.e., Pin dcache). Still, as a whole, the average slowdown for cache simulation was reduced from $32\times$ to $10.5\times$.

It is also worthwhile to note that if sampling is used, the simulation results can be quite inaccurate. In our experience with sampling [Zhao et al. 2007], cache misses computed via sampling can be significantly higher than the actual ones due to cold starts and exaggerated temporal nonlocality. PiPA, on the other hand, is 100% accurate. In Zhao et al. [2007], however, it is the *relative* importance of the load instructions causing the misses that mattered, and not the absolute miss numbers. Zhao et al. [2007] shows that the former can be correctly deduced by sampling.

6.2 SPEC CPU2006 Results

In this section we present the results of the experiments ran using the SPEC CPU2006 suite of benchmarks (compiled with gcc 4.1 using the -O3 flags). For these experiments we used the three multi-core systems described in Table VI and we evaluated both the DynamoRIO and the Pin PiPA prototypes.

As we already tuned most of the PiPA parameters in the SPEC2000 experiments, for the SPEC2006 experiments we used the best parameter values found in the previous experiments.

Table VI. The Multicore Systems Used in the SPEC2006 Experiments

No. of Cores	2	4	8
CPU Model	1 × Dual Core Intel Core 2	2 × Dual Core AMD Opteron	2 × Quad Core Intel Xeon
CPU Frequency	2.66 GHz	2.4 GHz	1.86 GHz
L2 Cache Size/Core	4 MB	1 MB	4 MB
Memory Size	2 GB	8 GB	4 GB
Operating System	32-bit Linux 2.6.20	64-bit Linux 2.6.20	32-bit Linux 2.6.22

Table VII. The Profiling Overhead of PiPA Measured in Slowdown Normalized to Native Execution

			2 cores	4 cores	8 cores
DynamoRIO PiPA	FP2006	Max	4.41	4.51	4.40
		Min	1.30	1.03	1.15
		Ave	2.53	2.55	2.43
	INT2006	Max	5.54	6.12	5.50
		Min	2.23	1.92	2.13
		Ave	3.96	3.89	4.04
	Spec2006	Ave	3.14	3.12	3.12
Pin PiPA	FP2006	Max	12.26	7.58	12.80
		Min	1.31	1.18	1.24
		Ave	5.27	3.96	5.37
	INT2006	Max	13.89	9.87	13.57
		Min	3.95	2.85	3.81
		Ave	9.08	6.56	9.42
	Spec2006	Ave	6.84	5.04	7.04

6.2.1 Profiling Overhead. In these experiments we evaluated the overhead of collecting REP using 16MB profile buffers. Table VII shows the slowdowns measured on the three systems when using the DynamoRIO PiPA, and Pin PiPA.

The DynamoRIO PiPA achieves on average a $3.1\times$ slowdown on all three systems, and this is similar to the overheads observed when using SPEC 2000 benchmarks. However, the Pin PiPA has a considerably higher overhead for profiling. The average slowdowns are $6.84\times$ on the 2-core system, $5.04\times$ on the 4-core and $7.04\times$ on the 8-core. As explained in Section 4.2.2, some of our optimizations could not be implemented in Pin because of its API limitations, and this is the reason why the Pin PiPA profiling is slower. For these experiments we use only stage 0 of PiPA which is single-threaded, therefore the differences in the slowdowns on the 3 systems are due only to the different CPU models and cache sizes.

As SPEC2006 runs are considerably longer, we also used them to assess how compact REP is compared to a naïve profile format that records the 4-tuple $\langle pc, address, type, size \rangle$ for each memory reference. We measured two profile sizes: the average size needed for 1 million instructions and the total profile size. For these experiments we used the DynamoRIO PiPA and the results are shown in Figures 8 and 9. The results clearly prove the compactness

Fig. 8. The average profile size for 1 million instructions using REP and naïve formats.

Fig. 9. The total profile size using REP and naïve formats.

of the REP format. On average, for 1 million instructions, REP needs 1.6MB of profile buffer, while the naïve format requires more the 8MB. The total profile size also confirms this significant difference, on average the REP profile size being 8 times smaller than the naïve one. The latter format may reach over 60TB of profile size in the case of 416.gamess benchmark, while REP needs at most 9TB for the same benchmark.

6.2.2 Profile Recovery Overhead. Next, we evaluated the performance for the first two stages of PiPA, in which both profiling and recovery are done in parallel. As in the case of the SPEC 2000 experiments, the recovery threads count the total number of memory references, and reconstruct the detailed reference information into the form of a `<pc, addr, size, type>` tuple, which is then copied into a thread-private non-local data structure. For these experiments we used 16MB of profile buffers and 8 recovery threads in stage 1 of PiPA. Table VIII shows the slowdown of recovery relative to profiling on the three systems using the DynamoRIO PiPA and the Pin PiPA, respectively.

A similar tendency can be observed for both implementations: as expected, a larger number of available cores improves the overall performance due to the increased parallelism. The Pin PiPA exhibits a slightly lower slowdown compared to the DynamoRIO one. For Pin PiPA, on average the measured slowdowns are $1.99\times$, $1.34\times$, and $1.06\times$ on the dual core, quad core, and eight core respectively. In the case of the DynamoRIO PiPA the average slowdowns

Table VIII. The Slowdown of Profile Recovery Relative to Profiling

			2 cores	4 cores	8 cores
DynamoRIO PiPA	FP2006	Max	4.50	2.66	1.59
		Min	1.64	0.93	1.02
		Ave	2.73	1.48	1.22
	INT2006	Max	2.74	1.49	1.19
		Min	1.40	0.91	1.06
		Ave	2.09	1.10	1.12
	Spec2006	Ave	2.47	1.32	1.18
Pin PiPA	FP2006	Max	5.55	2.81	1.49
		Min	1.21	1.05	0.87
		Ave	2.37	1.53	1.07
	INT2006	Max	1.99	1.33	1.10
		Min	1.17	0.82	1.01
		Ave	1.47	1.07	1.04
	Spec2006	Ave	1.99	1.34	1.06

Fig. 10. The speedup of Pin PiPA over Pin atrace.

are $2.47\times$, $1.32\times$, and $1.18\times$. It should be noted that the overall overhead compared to native execution is larger for Pin PiPA because of the higher profiling overhead shown in the previous section.

We also compared the parallel Pin PiPA with the sequential Pin atrace tool which collects the same 4-tuples for each memory reference. As shown in Figure 10, the parallel PiPA performs significantly better than the sequential Pin atrace. On average, Pin PiPA is $1.8\times$ faster than Pin atrace on the 8-core system.

6.2.3 Cache Simulation Overhead. Finally, we evaluated the effectiveness of the entire cache simulation PiPA using SPEC 2006. As in the case of the SPEC 2000 experiments, we compared PiPA with the sequential Pin dcache simulator. Both PiPA and dcache use similar set-associative caches characterized by the same parameters. In the initial experiments, we used for PiPA 8 recovery threads at stage 1 and 8 cache simulators at stage 2. The communication between these two stages is done using two shared buffers of 2MB for each cache simulator.

The slowdowns obtained for the DynamoRIO PiPA are shown in Table IX, and they are similar to the numbers got for SPEC 2000 benchmarks; on the 8-core system PiPA is 10.2 times slower than the native execution. Table X shows the speedups obtained over Pin dcache. The DynamoRIO PiPA manages

Table IX. The Cache Simulation Overhead of PiPA
Measured in Slowdown Normalized to Native Execution

			2 cores	4 cores	8 cores
DynamoRIO PiPA	FP2006	Max	27.72	27.72	16.58
		Min	8.70	3.71	4.47
		Ave	18.69	14.08	11.09
	INT2006	Max	28.04	19.77	16.38
		Min	6.87	2.60	3.59
		Ave	18.19	10.84	8.94
	Spec2006	Ave	18.48	12.74	10.20
Pin PiPA	FP2006	Max	32.64	28.57	18.78
		Min	9.94	4.63	6.65
		Ave	21.52	15.97	12.19
	INT2006	Max	40.15	23.36	17.21
		Min	8.34	3.51	5.07
		Ave	23.45	13.84	12.26
	Spec2006	Ave	22.32	15.08	12.22

Table X. The Speedup of PiPA over Pin Dcache

			2 cores	4 cores	8 cores
DynamoRIO PiPA	FP2006	Max	1.95	2.20	4.92
		Min	1.50	1.27	1.69
		Ave	1.77	1.67	2.91
	INT2006	Max	2.17	2.28	4.56
		Min	1.53	1.30	2.54
		Ave	1.85	1.94	3.79
	Spec2006	Ave	1.80	1.78	3.27
Pin PiPA	FP2006	Max	1.70	2.45	4.26
		Min	1.35	1.03	1.30
		Ave	1.52	1.50	2.54
	INT2006	Max	1.73	1.51	3.22
		Min	1.17	0.95	2.15
		Ave	1.45	1.23	2.69
	Spec2006	Ave	1.49	1.39	2.60

to be more than three times faster than a sequential cache simulator on an 8-core system.

We ran the same experiments using the Pin PiPA, and the results can be seen in Tables IX and X. As expected, because of the large profiling overhead that is incurred in stage 0, Pin PiPA is slower than the DynamoRIO PiPA. On average, when run on the 8-core system Pin PiPA presents a $12.2\times$ slowdown compared to native execution. However, Pin PiPA achieves a $2.6\times$ speedup over Pin dcache.

In the next set of experiments we used the 8-core system and evaluated how the number of cache simulators affects the overall performance. For this, we first used the DynamoRIO PiPA and varied the number of threads used in stage 2 for cache simulation from 4 to 64. In the case of 64 threads we were not able to use 2MB buffers for the communication with stage 1, as the total amount of allocated shared memory exceeded the system's limits. Therefore, in this case we used just 1MB buffers. The slowdowns obtained for SPEC INT2006 and SPEC FP2006 are shown in Figure 11. From this

Fig. 11. The slowdown of DynamoRIO PiPA cache simulation on a 8-core system using different numbers of simulators.

Fig. 12. The speedup of DynamoRIO PiPA over Pin dcache when using 16 and 32 simulators (8-core system).

results we can easily see that 16 and 32 simulators perform better than 8 simulators. The DynamoRIO PiPA with 32 simulators manages to reduce the overall slowdown to $7.4\times$. The reason for this improvement is probably the fact that the workloads got partitioned more evenly between the simulators and more parallelism was achieved. However, when switching to 64 simulators the performance degrades. There are two possible explanations for this behavior: the total number of threads is too large compared to the number of available cores, and the reduced buffer size increases the communication costs. Figure 12 shows the achieved speedups of DynamoRIO PiPA over Pin dcache when using 16 and 32 simulators. On average, using 32 simulators, the DynamoRIO PiPA manages to achieve a considerable $4.3\times$ speedup over the sequential dcache. We expect that a dynamic balancing of the simulators' workloads will bring about even more performance improvements.

We also evaluated how much the performance can be improved for the Pin PiPA when increasing the number of cache simulators to 32. Figure 13 shows a comparison of the speedup obtained by Pin PiPA over Pin dcache when using 8 simulators versus using 32 simulators. On average the speedup increases from $2.6\times$ to $3.4\times$. The maximum speedup, obtained for 410.bwaves, reached $5.8\times$.

7. LIMITATIONS

The current version of PiPA has only very limited support for parallel applications. It does not record synchronizations needed for accurate depiction of thread interactions. Also, all threads are started at the beginning of the application. Therefore, thread forks and joins cannot be supported. PiPA essentially obtained good performance at the expense of additional resources. For this

Fig. 13. The speedup of Pin PiPA over Pin dcache when using 8 and 32 simulators (8-core system).

article, we reported the best partitioning and mapping for the cache simulation application. In Section 5.1, we discussed alternative points in the design space that may be appropriate for different applications and under different scenarios. However, it remains true that if there are insufficient spare resources, the performance of PiPA will seriously degrade. For example, if a parallel application already consumes all the cores and most of the memory and bus resources available, then it would be hard for PiPA or, for that matter, any other tool, to profile and analyze the application efficiently.

Another limitation with the current implementation is that it lacks an exception handling capability. On a return from an exception, it is not possible to recreate the register state correctly. Thus applications have to be exception free. However, one of the authors in a recent work has solved this problem [Zhao et al. 2010]. The same solution should work in PiPA.

8. CONCLUSION AND FUTURE WORK

In this article, we described and investigated PiPA, a technique for performing parallel program profiling and analysis that takes advantage of multicore systems and drastically reduces the program analysis time. In PiPA, a very lightweight profiler written by means of dynamic instrumentation is used to produce the trace information in shared buffers. These buffers are handed off to analysis threads so as to take advantage of multi-cores.

An important aspect of our work is the data representation in the buffers. A verbose representation significantly reduces the work required in the profiling code. However, it will cause buffers to fill up quickly, increasing the need for expensive synchronization with the analysis threads. A compact representation will reduce the latter, but it will require more work on the part of the profiling code. This may in turn become the bottleneck, starving the analysis threads of work. We believe that we have achieved the right trade-off in REP, a novel profile format that has several advantages: it can be collected with minimal profiling overhead, it is very compact and therefore allows a better usage of the profile buffers, and it makes it easy for the next pipeline stages to recover the full profile trace.

As a case study of the kinds of analysis PiPA is capable of, we described an approach for parallelizing trace driven analysis, and demonstrated it using a parallel cache simulator. Other types of program analysis, such as memory

dependence analysis and branch prediction simulation can be parallelized in a similar manner.

We implemented PiPA using two dynamic instrumentation frameworks: DynamoRIO and Pin. This demonstrates the portability of PiPA across different instrumentation infrastructures, even though we obtained better performance with DynamoRIO due to the optimization opportunities available. We conducted a comprehensive set of experiments to assess PiPA's performance on actual multicore systems. These include assessing the efficiency of REP profiling, memory reference recovery, and parallel cache simulation using PiPA. The experimental results show that PiPA improves on the performance of both profiling and analysis, and, therefore, is an effective technique for parallelizing program analysis in practice. We believe that PiPA offers a new paradigm in parallelizing runtime program analysis.

The next step is to extend PiPA by designing APIs and library templates for certain types of usage models. For instance, the communication and synchronization between different pipeline stages is similar for most types of analysis. The API should hide this communication from the programmer, who should only be concerned with the instrumentation and analysis that must be done in each stage. This would significantly ease the task of developing new PiPA tools.

To further improve the efficiency of PiPA there are two major future directions that can be explored. The first is parallel profiling. In the current implementation, profiling is performed in the same application thread, a bottleneck can occur if the profiles cannot be produced fast enough to satisfy the demands of the parallel recovery threads. This will limit the scalability of the recovery and analysis processes especially on many-core systems. We would like to explore other approaches like SuperPin for parallel profiling. The second direction is workload balancing. A balanced workload is important for achieving good performance in PiPA. However it is hard to discover if the workload is balanced, locate bottlenecks, and dynamically rebalance it. More research is needed for automatic approaches that dynamically monitor the workload and the progress of each thread so that adjustments can be made at runtime to balance the system.

REFERENCES

- ARNOLD, M. AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 168–179.
- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. 1–12.
- BALL, T. AND LARUS, J. R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'29)*. 46–57.
- BRUENING, D. 2004. Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, Massachusetts Institute of Technology. <http://www.cag.csail.mit.edu/rio/>.
- CHEN, H., HSU, W.-C., LU, J., YEW, P.-C., AND CHEN, D.-Y. 2003. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 79–90.
- EBCIOĞLU, K. AND ALTMAN, E. R. 1997. Daisy: Dynamic compilation for 100. In *Proceedings of the 24th International Symposium on Computer Architecture*. ACM Press, New York, NY, 26–37.

- KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, 191–206.
- LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. 259–269.
- LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LONEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 190–200.
- MOSELEY, T., SHYE, A., REDDI, V., GRUNWALD, D., AND PERI, R. 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. 198–208.
- NETHERCOTE, N. 2004. Dynamic binary analysis and instrumentation. Ph.D. thesis, University of Cambridge. <http://valgrind.org/>.
- NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 89–100.
- REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. 2005. Swift: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 243–254.
- SPEC. 2000. SPEC CPU2000 benchmark suite. <http://www.spec.org/osg/cpu2000/>. Standard Performance Evaluation Corporation.
- SPEC. 2006. SPEC CPU2006 benchmark suite. <http://www.spec.org/osg/cpu2006/>. Standard Performance Evaluation Corporation.
- TALLAM, S., GUPTA, R., AND ZHANG, X. 2005. Extended whole program paths. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 17–26.
- WALLACE, S. AND HAZELWOOD, K. 2007. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. 209–220.
- ZHAO, Q., CUTCUTACHE, I., AND WONG, W.-F. 2008. Pipa: Pipelined profiling and analysis on multicore systems. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. ACM, New York, NY, 185–194.
- ZHAO, Q., RABBAH, R., AMARASINGHE, S., RUDOLPH, L., AND WONG, W.-F. 2007. Ubiquitous memory introspection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. 299–311.
- ZHAO, Q., SIM, J. E., WONG, W.-F., AND RUDOLPH, L. 2006. DEP: Detailed Execution Profile. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. 154–163.
- ZHAO, Q., BRUENING, D., AND AMARASINGHE, S. 2010. Umbra: Efficient and Scalable Memory Shadowing. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'10)*. To appear.

Received March 2009; revised January 2010; accepted May 2010