PiPA: Pipelined Profiling and Analysis on Multi-core Systems

Qin Zhao
Singapore-MIT Alliance
School of Computing
National University of Singapore
zhaoqin@comp.nus.edu.sg

Ioana Cutcutache Weng-Fai Wong
School of Computing
National University of Singapore
{ioana, wongwf}@comp.nus.edu.sg

ABSTRACT

Dynamic instrumentation systems are gaining popularity as means of constructing customized program profiling and analysis tools. However, dynamic instrumentation based analysis tools still suffer from performance problems. The overhead of such systems can be broken down into two components – the overhead of dynamic instrumentation and the time consumed in the user-defined analysis tools. While important progress has been made in reducing the performance penalty of the dynamic instrumentation itself, less attention has been paid to the user-defined component. In this paper, we present PiPA – *Pipelined Profiling and Analysis*, which is a novel technique for parallelizing dynamic program profiling and analysis by taking advantage of multi-core systems. We implemented a prototype of PiPA using the dynamic instrumentation system DynamoRIO. Our experiments show that PiPA is able to speed up the overall profiling and analysis tasks significantly. Compared to the more than 100x slowdown of Cachegrind and the 32x slowdown of Pin dcache, we achieved a mere 10.5x slowdown on an 8-core system.

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

Keywords

Profiling, Analysis, Dynamic Instrumentation, Pipelining, Parallel Cache Simulation, Multi-core Systems

1. INTRODUCTION

The knowledge of dynamic program behavior is invaluable in many research areas such as computer architectural design and software development. However, collecting the details of a program's execution is a tedious process that often requires running the application-under-examination at a significantly slower speed. Many analysis require runs over a large data set or a substantial native execution time in order to overcome initial start-up or transient effects. The collected profiles of such runs are often too large to be stored away for off-line 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.

Dynamic instrumentation systems have been proven to be very effective tools that allow users to construct customized dynamic program profiling and analysis tools. Unfortunately, dynamic instrumentation based analysis tools suffer from the performance problems mentioned above. 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 5x 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 20x. When running with a cache simulator (Cachegrind), the slowdown goes up further to more than 100x on average.

Multi-core systems allow multiple threads or processes to run simultaneously. The proliferation of such systems has encouraged researchers to explore new approaches for offloading parts of the profiling and analysis tasks onto spare hardware execution cores in order to improve performance. One such proposal is *parallelized slice profiling* [9, 14]. 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 novel approach comes with several technical challenges, including how to guarantee that the slices' executions are identical to the main application, how to handle multi-threaded applications, and how to merge the final analysis results. In addition, this approach is only suitable for independent tasks like instruction counting, but has difficulties in performing more

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.

complex tasks such as branch prediction and cache simulation. This seriously restricts the utility of this approach.

In this paper we exploit another form of parallelism, namely pipelining. We propose a novel technique for parallel profiling and analysis that we called PiPA[1] (*Pipelined Profiling and Analysis*). Essentially, in PiPA, threads form a pipeline for collecting and processing profiles. Rather than a simple pipeline, a better analogy to 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 (i.e. reducing the profiling overhead through parallelization), and can be considered a complementary approach to the latter. However, it has several advantages over the previous techniques. Firstly, it is a straight-forward model making it easier for users to understand and build customized analysis tools. Secondly, 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 [14]). Some desirable properties (for example preserving exact ordering in the profile), can be achieved much easier in PiPA because of these characteristics. Furthermore, PiPA can handle multithreaded applications without any difficulty by having one pipeline for each application thread.

In this paper, we describe the design and implementation of a prototype of PiPA in DynamoRIO [3]. We evaluated the performance of PiPA on several multi-core systems. The experimental results show significant speedup over traditional approaches. We summarize the contribution of this paper as follows:

- We propose 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;
- We propose 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 paper 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, which is followed by the conclusion and future work in Section 7.

[1]The pipa is an ancient Chinese string musical instrument.

2. BACKGROUND

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 [2], DynamoRIO [3] and ADORE [4] 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 like Pin [8] and Valgrind [10] can be used to build customized program analysis tools. There are other applications of runtime code manipulation such as for dynamic translation [5], security [6], and reliability [11]

As an example of how such a system works we provide here a brief description of DynamoRIO's operation. DynamoRIO [3] 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 DynamoRIO clients using the APIs provided. These clients can manipulate the application code by supplying hook 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. Some common types of profiling are path profiling, hot data stream profiling and value profiling.

Larus [7] 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. Tallam *et. al.* [13] extended WPPs to also encode the memory dependencies, but their scheme incurs a large time and space overhead. Zhao *et. al.* [16] 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 off-line 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 [1] proposed a general framework that utilizes bursty

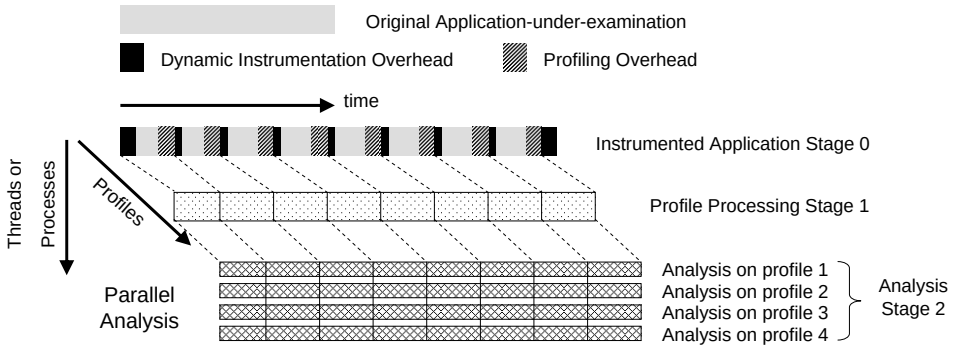

Figure 1: The PiPA pipeline.

tracing to reduce the profiling overhead. A similar idea was used in Ubiquitous Memory Introspection [15], 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.

More recent research has focused on utilizing the increasingly popular multi-core systems to reduce the overhead of profiling by off-loading profiling tasks to spare hardware cores. Shadow profiling [9] runs the original uninstrumented application in parallel with instrumented slices to perform sampled profiling. SuperPin [14] 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 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 on 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. Each horizontal bar represents a thread working as a unit in a pipeline stage. 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 re-organizes the profiles into specific formats for analysis in the later stages. In our example, it splits the profiles into 4 sub-profiles. These sub-profiles are fed into four threads at stage 2 that perform parallel analysis.

It should be noted that any of the threads can be easily replaced with an operating system process. For example, the threads in the first two stages can be in the same process as the application that outputs the profile in a desired format. The threads in stage 2 can be organized into one or several 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.

How fast the profiles can be produced is one of the most important constraints in the efficiency of PiPA. 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 named REP (*Runtime Execution Profile*).

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 paper, 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 sub-profiles independently. The results can be easily merged after the execution. Further details will be discussed in Section 5.

Profiling and analyzing multi-threaded 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 thread synchronization information may have to be recorded for analysis purposes.

3.2 Implementation

In order to demonstrate the effectiveness and efficiency of PiPA, we implemented a two-stage PiPA prototype using DynamoRIO. Note that it is entirely possible to use other dynamic instrumentation frameworks to implement PiPA.

When the application-under-examination starts executing under DynamoRIO, 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 application executes under DynamoRIO, profiling code is inserted into any application code that DynamoRIO 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 DynamoRIO's 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 directly. Alternatively, the recovery thread can write the reconstructed information into a shared buffer to be processed by another analyzer thread.

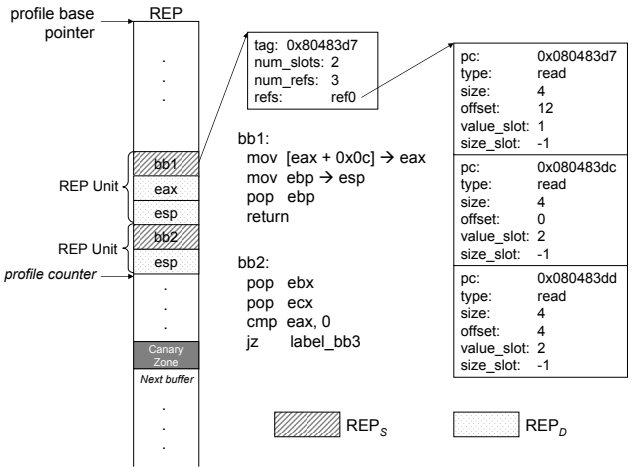

Figure 2: An example of REP.

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 naive raw format performs very poorly. In this section, we first describe a novel profile format REP (Runtime Execution Profile), then discuss how to efficiently collect REP and finally show 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.

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.

For different analysis, different kinds of information are required. REP_D can be customized for the analysis 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 naive 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 tasks to do in the instrumentation code: (1) context switching so as to preserve the correctness of the execution of the application-under-examination; (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. This section presents several optimizations that can be used for this purpose. Some of the proposed optimizations specifically target the x86 architecture that was employed in our experiments.

Context switching consists of saving and restoring the values of the registers that are used by the instrumentation code. One of the goals of our optimizations is to minimize the number of these registers in order to reduce the overhead.

Firstly, 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.

Secondly, 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.

Thirdly, instead of modifying the profile counter on each profile update, all the changes are combined into a single update when recording the REP_S 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 multi-exits format 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 in the buffer. This way, if a check at the beginning of a trace tells us that the buffer is not yet full, then no trace's execution will exceed the canary zone.

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.

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

This section uses cache simulation as an example to discuss how parallel analysis algorithms can be implemented using PiPA.

Minimizing inter-thread 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.

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 the PC value of branch instructions.

In order to evaluate the benefits of such parallelization, we implemented a parallel cache simulator as a stand-alone process that communicates with PiPA via semaphores and shared memory. The simulator works in a master-slave mode. The master thread communicates with PiPA to obtain the segregated memory reference profiles, and dispatches them to slave threads. Each slave thread is an independent cache simulator that simulates a partition of the cache. The experimental results are presented in Section 6.3.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of PiPA. We ran the experiments on the three different multi-core systems listed in Table 1. We used the full SPEC CPU2000 [12] 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.

No. of Cores	2	4	8
CPU Model	1 x Dual Core Intel Pentium D	2 x Dual Core AMD Opteron	2 x 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

Table 1: Multi-core systems used in experiments.

6.1 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 optimizations described in Section 4.2, while 'un-optimized profiling' refers to collecting REP without any of the proposed optimizations. Our optimizations clearly improve the profiling overhead significantly.

Table 2 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

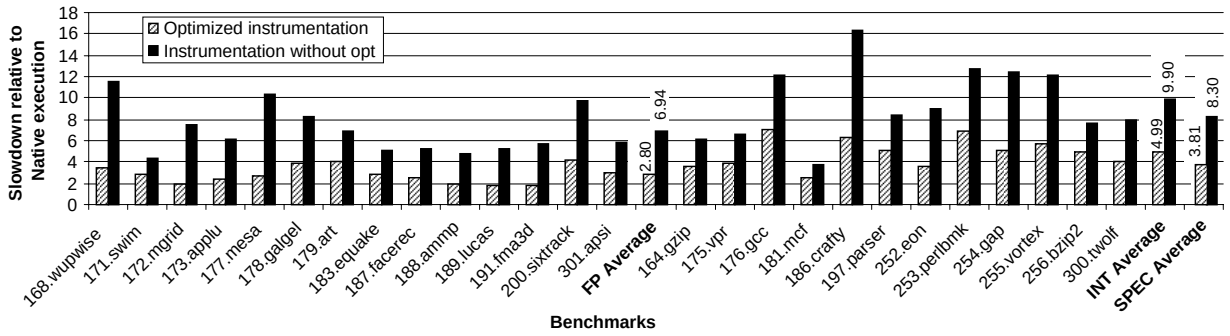

Figure 3: Profiling overhead on 8-core system with and without instrumentation optimization.

Systems	SPECint		SPECfp		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

Table 2: Profiling performance with and without optimization.

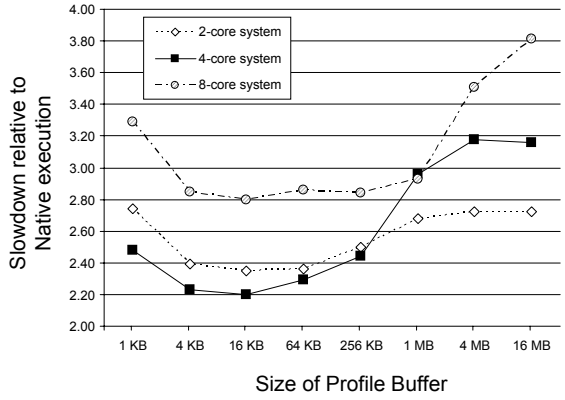

Figure 4: Performance on the various multi-core systems when profile buffer size is varied.

can be observed. 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.

6.2 Profile Recovery Overhead

In the second set of experiments, we evaluated the full PiPA framework performance in which both profiling and recovery are done in parallel on the multi-core systems. The

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

Table 3: Normalized profiling and recovery overhead on 8-core system with different profile buffer size.

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

Table 4: Normalized profiling and recovery overhead on 8-core system with different number of recovery threads.

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 `<pc, addr, size, type>` tuple, which is then copied into a thread-private non-local data structure. This copying is necessary to avoid 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 available 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 3 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

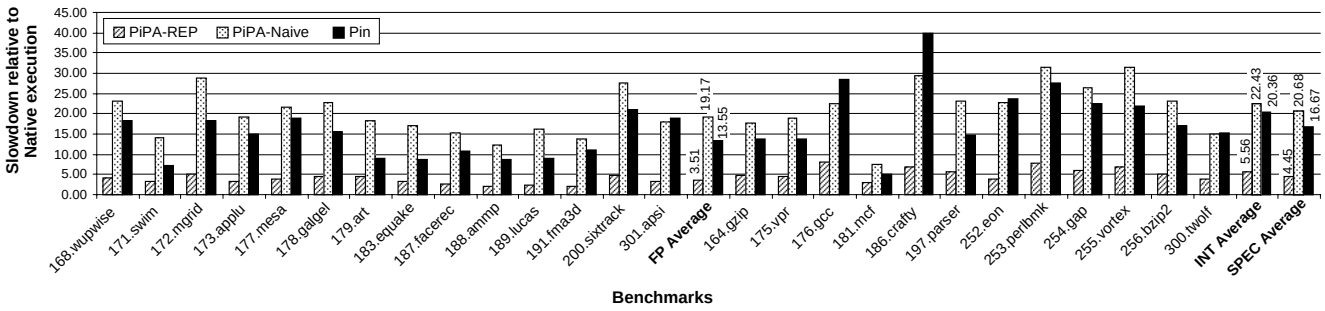

Figure 5: Runtime overhead of three profiling approaches on 8-core system.

	Normalized to native exec.			Normalized to profiling		
# cores	2	4	8	2	4	8
SPECInt	7.01	5.02	5.56	1.95	1.20	1.11
SPECFP	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

Table 5: Profiling and recovery overhead with different number of CPU cores.

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 4. 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 multi-core systems. In this experiment, we used 8 recovery threads, and 16M profile buffers, to make sure they can take advantage of any available hardware resources in all of the three systems. On the left of Table 5, 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 SPECint, 12.56 for SPECfp, 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 in PiPA. We implemented a version of PiPA to collect the naive memory reference information tuple (i.e. `<pc, addr, size, type>`). We also used Pin to collect this tuple format. Again, memory references are counted, and the profile is copied to a thread-private non-local data structure.[2]

[2]We 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.

As shown in Figure 5, PiPA using REP performs significantly better than the PiPA or Pin using the naive format. It should be noted that PiPA-REP and PiPA-Naive were implemented in DynamoRIO. There are two reasons why REP performs much better than the naive profile. First, as mentioned before, REP enables several optimizations that lower the profiling overhead compared to the naive 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 other reason is that REP is more compact. For each memory reference, there are around 4 bytes on average used, in contrast to the naive 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 naive profile collection by PiPA is even slower than the serial naive profile collection of Pin.

6.3 Cache Simulation

Finally, we evaluated the effectiveness of PiPA in parallel cache simulation by comparing it against the Pin `dcache` simulator. To be fair, we used the same cache simulator 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.

We tested the simulations on the 4-core and 8-core systems. Figure 6 shows the normalized execution times of PiPA cache simulator and Pin dcache simulator on the 8-core system, while Figure 7 shows the speedups of 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 5x. In general the 8-core achieves slightly better speedups than the 4-core. However, there are a few benchmarks which 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 simu-

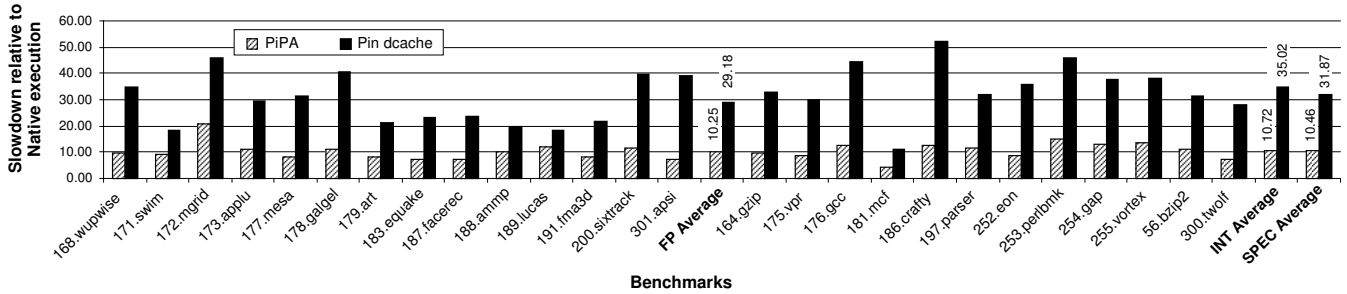

Figure 6: Cache simulation overhead for PiPA and Pin dcache on 8-core system.

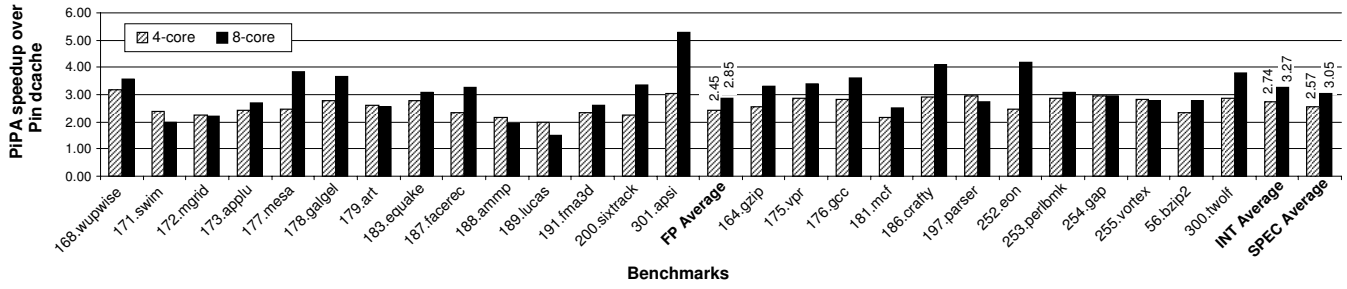

Figure 7: PiPA speedup over Pin dcache on 4-core and 8-core systems.

lator 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 8x 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.ammp` 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 2x to 5x speedup relative to a sequential cache simulator (i.e. Pin dcache). Still, as a whole, the average slowdown for cache simulation was reduced from 32x to 10.5x.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced PiPA, a technique for performing parallel program profiling and analysis that takes advantage of multi-core systems and drastically reduces the program analysis time. We implemented PiPA using the dynamic instrumentation framework DynamoRIO. In addition, we proposed REP, a novel profile format. For maximal efficiency, the design of the trace buffer data structure is crucial. This data structure has to (i) require the minimal instrumentation code thereby reducing the profiling overhead; (ii) be compact so that the buffer holds as much of the trace as possible before filling up; and (iii) make it easy for the next pipeline stages to recover the full trace. REP meets these requirements. We also 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 have conducted a comprehensive set of experiments to assess PiPA's performance on actual multi-core 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 the success of PiPA opens up a new approach for parallel 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. By carefully crafting the API this communication can be hidden away 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. Because 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 monitoring. A balanced workload is important for achieving good performance in PiPA. However it is hard to discover if the workload

is balanced, locate bottle-necks, and dynamically re-balance 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.

8. REFERENCES

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, 2000.

[3] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Sep. 2004. http://www.cag.csail.mit.edu/rio/.

[4] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the First International Symposium on Code Generation and Optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[5] K. Ebcioğlu and E. R. Altman. Daisy: dynamic compilation for 100In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 26–37, New York, NY, USA, 1997. ACM Press.

[6] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[7] J. R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, 1999.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[9] T. Moseley, A. Shye, V. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, 2007.

[10] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007. ACM Press.

[11] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Proceedings of the Third International Symposium on Code Generation and Optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite, 2000. http://www.spec.org/osg/cpu2000/.

[13] S. Tallam, R. Gupta, and X. Zhang. Extended whole program paths. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 17–26, 2005.

[14] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 209–220, 2007.

[15] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 299–311, 2007.

[16] Q. Zhao, J. E. Sim, W.-F. Wong, and L. Rudolph. DEP: Detailed Execution Profile. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 154–163, 2006.