HAL open science

Extracting Threaded Traces in Simulation Environments

Weixing Ji, Yi Liu, Yuanhong Huo, Yizhuo Wang, Feng Shi

To cite this version:

Weixing Ji, Yi Liu, Yuanhong Huo, Yizhuo Wang, Feng Shi. Extracting Threaded Traces in Simulation Environments. 10th International Conference on Network and Parallel Computing (NPC), Sep 2013, Guiyang, China. pp.27-38, 10.1007/978-3-642-40820-5_3 . hal-01513764

HAL Id: hal-01513764

https://inria.hal.science/hal-01513764v1

Submitted on 25 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Extracting Threaded Traces in Simulation Environments

Weixing Ji, Yi Liu, Yuanhong Huo, Yizhuo Wang, and Feng Shi

Beijing Institute of Technology, Beijing 100081, China,
jwx@bit.edu.cn

Abstract. Instruction traces play an important role in analyzing and understanding the behavior of target applications; however, existing tracing tools are built on specific platforms coupled with excessive reliance on compilers and operating systems. In this paper, we propose a precise thread level instruction tracing approach for modern chip multi-processor simulators, which inserts instruction patterns into programs at the beginning of main thread and slave threads. The target threads are identified and captured in a full system simulator using the instruction patterns without any modifications to the compiler and the operating system. We implemented our approach in the GEM5 simulator and evaluations were performed to test the accuracy on x86-Linux using standard benchmarks. We compared our traces to the ones collected by a Pin-tool. Experimental results show that traces extracted by our approach exhibit high similarity to the traces collected by the Pin-tool. Our approaches of extracting traces can be easily applied to other simulators with minor modification to the instruction execution engines.

Keywords: program trace, full system simulation, multi-core processor

1 Introduction

Instruction trace characterizes a program's dynamic behavior and is widely used for program optimization, debugging and new architecture evaluation. Particularly, memory traces, which are subsets of instruction traces, are frequently used for new memory system evaluation. Program traces are also necessary for trace driven simulators, which is a well known method for evaluating new computer architectures. Prevailing tools for collecting application execution traces include tools built based-on Pin [1] and Linux-process-tracker provided by the full system simulator Simics [2]. There are also a number of simulators and emulators available to generate traces on some platforms [3, 4]. Theoretically, instruction traces can be extracted at virtually every system level, from the circuit and microcode levels to the compiler and operating-system levels [5]. However, existing trace collectors suffer from at least one of the following three limitations:

– Being highly dependent on operating systems and compilers and only available for one or two exiting platforms. It is difficult to add a new platform;

- Only supporting several existing ISAs, severely limiting the usage in new architecture exploration;
- Generating mixed instruction traces, which include instructions from other applications and operating system modules.

Tools built on Pin are efficient to collect traces for a single application, but it is only available for Windows and Linux running on IA-32 and x86-64. It is possible to port Pin to other platforms hosting a different operating system; and it is also possible to add a new back-end to Pin targeting a new processor family. However, this work would be time consuming and laborious. Pin-tools run as applications on existing platforms and this limits the usage when researchers are exploring new architectures. Some simulators, such as Simics features the functionality of single-process profiling only for Linux, while others, such as Solaris, are not supported at present. Besides, the techniques of instruction tracing in Simics are operating system dependent and the execution of scheduling module in the operating system triggers the tracing on and off. In case of an operating system update, the tracing functionality may lost their last straw to clutch at. The new simulator GEM5, which becomes increasingly popular in full system simulation, generates mixed instruction traces at present.

Uhlig [5] defines three aspects of metrics to evaluate the quality of traces and shows that the collected traces should be as close as possible to the actual stream of instructions made by a workload when running on a real system. In particular, the authors emphasize on the portability of trace collector. It should be easy to move the collector to other machines of the same type and to machines that are architecturally different. Finally, an ideal trace collector should be fast, inexpensive and easy to operate. In this paper, we propose a new approach to extract threaded traces in full system simulation environments that matches these criteria. This approach does not need to inspect the internal state of operating systems and does not need to change existing compilers.

In summary, this paper makes the following contributions:

- A new approach to extracting program traces in simulation environments. It generates separated traces for each target thread and filters the noisy instruction sequences out.
- An efficient implementation of our approach in simulator GEM5 on x86-Linux. Evaluations on a suite of 7 benchmarks indicate that our approach is feasible and can be easily applied to other simulators.

The rest of this paper is organized as follows: Section 2 presents our new approach to extract thread level traces for applications in simulation environments. Section 3 shows our experimental results. Section 4 discusses the related work on trace extracting, and Section 5 concludes the paper.

2 Extracting Threaded Traces in Simulation Environments

2.1 Basic Idea

The basic idea of our approach lies in that a process is composed of one or more threads, and instructions executed on processors can be separated if all the threads of one process can be identified from the view of simulators. In general, thread IDs are software concepts and they are transparent to underline hardware in most implementations. Therefore, two issues should be addressed in this approach: (1) how to identify threads for a given process, and (2) how to find instructions executed by these threads.

In state-of-art operating systems, a process is a collection of virtual memory space, code, data and system resources. Although the internal presentation of threads and processes in operating system may differ from one to another, a process always has at least one thread of execution, known as the main thread or primary thread. Additionally, one or more threads can be created by the main thread and live within the same process. These slave threads share the same resources with the main thread. The running of a single process is represented as the execution of both main thread and slave threads. Namely, the traces of a process is actually the instruction sequences that are executed in these threads. One of the key issues is how to find the main thread and salve threads without the help of the operating system and compiler. We work out a solution to this problem using the stack pointer register (SP), which is explicit or implicit defined in most processors. Usually, at the creation stage, each thread is associated to a memory region referred to as thread stack. The primary purpose of this thread stack is used to store return address, pass parameters to the callee, store function local variables and so on. On the occasion of a function invocation, a new stack frame will be allocated for the execution of that function. When function returns, the caller or the callee is responsible to deallocated this stack frame by increasing or decreasing the stack pointer. Hence, the top of the stack dynamically changes from time to time and memory locations within the stack frame are typically accessed via indirect addressing.

For most operating systems and third-party thread libraries, a large memory block is usually allocated for each thread from the virtual memory space. The virtual address of the stack is not enough to distinguish the target threads from other threads created by other applications, as all the applications have the same size of virtual spaces starting from the same virtual address. However, these stack regions are mapped to different locations in the physical memory space. In general, the allocated virtual regions are large enough and programs do not change their location and size during runtime, and therefore threads can be distinguished using their stack positions in the physical memory space. However, this does not mean that all the physical addresses allocated to the stack can be considered as the unique identifiers of the target threads. The reason for this is that the virtual-to-physical mapping may change as the stack grows back and forth. Moreover, some of the physical pages may be swapped out and in

during execution. However, we find that the first page at the bottom of the stack is always in use during the whole lifetime of a thread and most of existing operating systems provide APIs to prevent that part or all of the calling process's virtual address space from being paged to the swap area. If the first page at the bottom of the stack is locked into RAM and it is not remapped during execution, then virtual-to-physical mapping will not change. It is natural that we can use the first physical pages of stacks to distinguish threads of running processes. So long as we know which thread is running on the core, instructions can be captured and directed into different output streams.

2.2 How to Obtain Traces

The following presents a step-by-step explanation of the methodology for trace extraction in a simulated full system environment.

1. An instruction pattern is defined for the target program, which is delicately designed and composed of several instructions supported by target processors;
2. A predefined small function is executed at the beginning of every target thread. This function is used to: (1) lock the first page of thread stacks in memory and (2) execute the predefined instruction pattern;
3. The source code of the program is compiled using an existing compiler and translated into machine binary;
4. At the very beginning of thread execution, the first page at the bottom of each thread stack is locked in RAM, so that the physical page will not be swapped out at runtime;
5. The simulator snoops the instruction stream of each processor core and captures the patterns that are inserted into the application;
6. When the exact instruction sequence defined in the instruction pattern is captured, the starting address $vaddrs$ and ending address $vaddre$ of current stack in the virtual space are calculated according to the content of SP and the stack size. The physical address $paddrs$ of $vaddrs$ is also obtained through virtual-to-physical translation in the execution context of the processor core. We pack $vaddrs$, $vaddre$ and $paddrs$ up in a structure and insert it into a target thread list(ttl).
7. For each instruction executed by a processor core, the simulator reads the register SP to find out the virtual address where the top of current stack is. Then we go through the ttl and test the content of SP against each virtual stack regions recorded previously. If we find that SP is pointing to one of the virtual stack regions, we translate the $vaddrs$ of the matched stack into its physical address $paddrs'$ according to current execution context. If $paddrs'$ is valid and equal to $paddrs$, which is obtained when the pattern is captured, then the instruction is included in the output trace.

In following discussion, we assume the starting address of a stack points to the location where the bottom of the stack resides, no matter which direction the

stack grows to. Therefore, the mapping from virtual address $vaddrs$ to physical address $paddrs$ does not change for a thread from the beginning to the end and this $paddrs$ can be used as the unique identifier of the target thread. Here, we need one more words for the calculation of $vaddrs$ and $vaddre$ before we proceed further. Given that the stack grows towards the lower address of the virtual space, these two addresses are calculated when the pattern is captured by:

$$vaddr_s = vaddr_{sp} - (vaddr_{sp} \bmod ps) + (k \times ps) - 1 \tag{1}$$

$$vaddr_e = vaddr_s + 1 - ss \tag{2}$$

where ps is page size and ss is stack size. k is an optional parameter and it is set to 1 in default. The reason why we can do this is that the instruction patterns are inserted at the beginning of both main and slave threads. We are certainly sure that the bottom of the stack is not far away from the address stored in the SP register when the pattern is captured. In general, there are 2 or 3 stack frames from the bottom to the frame that the pattern is capture. Hence, the distance from stack bottom to SP is no more than k pages. In most cases, the frame size of the thread function dose not exceeds the page size and k is set to 1. The starting address and the ending address of the stack can be calculated according to the size of the stack and the growing direction. The size of memory pages and the size of stacks are configurable and they can be easily figured out according to the version of the operating system and thread libraries.

To further present our idea clearly, a runtime scenario is given in Figure 1. As shown in the left part of this figure, two applications (App_1 and App_2) are started in the same operating system and run concurrently with 3 threads in total(2 for App_1 and 1 for App_2). In this case, the virtual stack of thread T_1 may overlap the virtual stack T_3 because both App_1 and App_2 have private virtual spaces, which start from the same address and spread out for the same size. However, the three virtual pages at the bottom of these stacks are mapped to different physical pages as threads do not share stacks. Given App_1 is our target application and instruction patterns will be captured by the simulator at the beginning of threads T_1 and T_2. At that moment, both SP_1 and SP_2 point to the first virtual page at the bottom of their stacks which are locked into RAM. Thus, $vaddr1_s$, $vaddr1_e$, $vaddr2_s$, and $vaddr2_e$ can be calculated using equation 1 and 2. Meanwhile, $paddr1_s$ and $paddr2_s$ are obtained by address translation using $vaddr1_s$ and $vaddr2_s$ respectively. Right now, the target thread list ttl looks like $\{\{vaddr1_s, vaddr1_e, paddr1_s\}, \{vaddr2_s, vaddr2_e, paddr2_s\}\}$. The right part of this figure shows the extracting context of an instruction. Every time when an instruction is executed by some core in the simulator, only SP_4 is read and it is used to search ttl. In the case of virtual stack overlapping, the first element $\{vaddr1_s, vaddr1_e, paddr1_s\}$ in ttl is tested and $vaddr1_s$ is translated into physical address $paddr1'_s$ in current execution context. However, it turns out that $paddr1'_s \neq paddr1_s$, which means current instruction is not belonged to any target threads. Therefore, the instruction is not included in the trace.

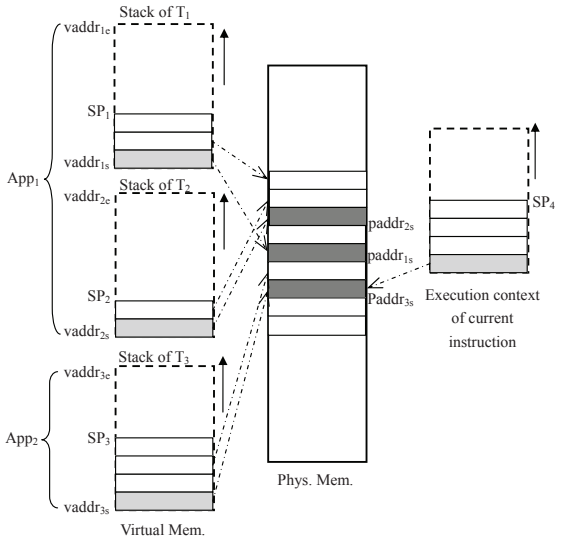

Fig. 1. Stack mapping and identification in simulation

2.3 Instruction Patterns

The instruction pattern should be delicately designed, as the same sequence of instructions may appear in the execution of both applications and operating system modules. However, if the instruction sequence in the pattern is long enough, then there will be a small possibility that the same sequence of instructions exists in the original binaries. For example, following instruction sequence in Figure 2 is an example designed for programs running on x86.

As multi-core processors are widely accepted and used now, applications running on an operating system tend to create multiple threads to fully utilize the hardware resource. It is important to have separated trace outputs for different threads, so that we get a precise instruction sequence for each thread. For this purpose, we pass the thread ID(*tid* in the last line of Figure 2) to the runtime in the pattern in a known register. This *tid* can be read from that register at the time when the pattern is captured by the simulator. Then it is stored in the *ttl* and used as an identifier for a target thread. Similarly, it is also possible to assign different thread IDs to different applications. In such a way, we can start and run multiple applications simultaneously in a system and have separated traces for all the threads created by all target applications. This feature is especially useful to analyze the dynamic execution interferences among multiple applications.

Our approach requires the user to insert instruction patterns into their source code and start the simulator with the pattern as an input. It can be implemented in most full system simulators without modifications to the compiler and the operating system, as the default size of thread stacks usually does not change. Even though the size is changed or the thread function has a very large stack frame, setting a new size in the configuration file will be able to extract correct

```
__asm__ __volatile__ ("move %0, %%eax\n\t"
                      "add $0x0, %%eax\n\t"
                      "add $0x0, %%eax\n\t"
                       : : "r"(tid)
);
```

Fig. 2. Instruction pattern for x86-Linux

traces. Our approach still work well in case of system updates, because it is not necessary to inspect the internal state operating systems.

3 Evaluation

3.1 Evaluation Method

GEM5 is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture [3]. GEM5 provides a highly configurable simulation framework, multiple ISAs, and diverse CPU models. Our experiments were performed on x86-Linux, as it is well supported and widely used. We lock some pages of a process's virtual address space into RAM, preventing these pages from being paged to the swap area. In our implementation, *mlock* is invoked at the very beginning of each thread to lock the identification pages at the bottom of thread stacks in RAM, so that each identification page is guaranteed to be resident in RAM and mapped to a same physical page before thread termination.

We also built an instruction tracing tool based-on the framework provided by Pin. In our experiments, we compared our traces to the ones obtained by the Pin-tool on x86-Linux. To further verify the effectiveness of our approach, we run multiple benchmarks together and generate separated traces for each application in GEM5. These traces are then compared with the traces generated by the Pin-tool. Taking 4 applications (a, b, c and d) for example, we run the applications in the simulator one at a time and get four traces(a_1, b_1, c_1 and d_1), then we partition the benchmarks into 2 groups((a,b), (c,d)), and start the two benchmarks in the same group at the same time in the simulator. Then we extracted another four traces(a_2, b_2, c_2 and d_2) from the hybrid instruction streams. After that, we compared the two traces of each benchmark with their Pin-tool counterparts to see how similar they are.

Even though the simulated x86-Linux is different from the host machine in many aspects, we expect highly similar traces for a same executable binary. This is because the difference in microarchitecture dose not changes the order of instructions in the same thread; however, in effect, they are not exactly the same due to thread scheduling and synchronization. The operating system images and compilers are all provided by the GEM5 team and we didn't make any modifications to these system software. All the benchmarks are compiled with the "-static" options to make sure that no differences will be introduced

into the traces by using different dynamic loaded libraries. Meanwhile, the instruction patterns are labeled by the *#pragma OPTIMIZE OFF* and *#pragma OPTIMIZE ON* pair to turn off GCC optimizations in these regions. This optimization restriction prevents the instructions in the pattern being removed or reordered, otherwise, the patterns can not be captured in the simulator while applications are running.

We use the standard multi-thread benchmarks to evaluate our approach, and all of the programs are selected from SPLASH2 [14]. For each application, only three lines of codes are inserted into the original sources: 1) including a head file defining the patterns, 2) inserting the pattern at the beginning of the main thread, and 3) inserting the same pattern at the beginning of slave threads. After that, all the benchmarks are compiled and each binary is executed on the host machine and the simulated full system in GEM5 separately. All the benchmarks we used are listed in table 1.

Table 1. Benchmarks

Benchmark	Description	Group
fft	A 1-D version of six-step FFT algorithm.	1
lu-non	Dense matrix factoring kernel.	1
radix	Integer radix sort kernel.	2
lu-con	Dense matrix factoring kernel.	2
fmm	Body interaction simulation.	3
ocean	Large-scale ocean movements simulation.	3
ocean-non	Large-scale ocean movements simulation.	3

3.2 Results

Figure 3 and 4 show the calculated similarity between two traces collected by hacked GEM5 and Pin-tool respectively. Call traces are reduced instruction traces that only contain instructions of function calls and returns. All the GEM5 traces are collected in solo-runs in the simulated full system. It shows that our traces are much similar to the ones collected by the Pin-tool. Though the the similarity of call traces is as high as up to 90% for most benchmarks, the instruction traces exhibit a relative low similarity around 85%. The reason for this is that the hacked GEM5 only starts to generate traces after instruction patterns are captured. Hence, the instructions executed at the startup phase before the main function are not included in the GEM5 traces, which add up to about 10 thousands in total. Meanwhile, we found that the difference is slightly enlarged as the number of threads increases. In order to find out the differences for the two call traces, the edit sequence for the two traces is rebuilt. From the edit sequence, we found the benchmarks with multiple threads spent much longer time at the synchronization points than their sequential versions. The threads,

which run faster than others are scheduled out and in from time to time, execute a number of instructions to check the status of barriers.

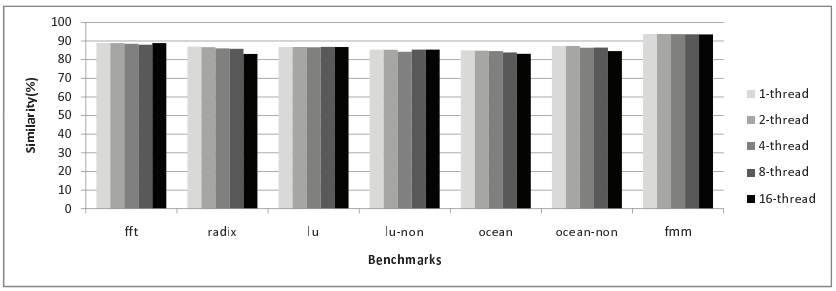

Fig. 3. Instruction trace similarity for solo-runs

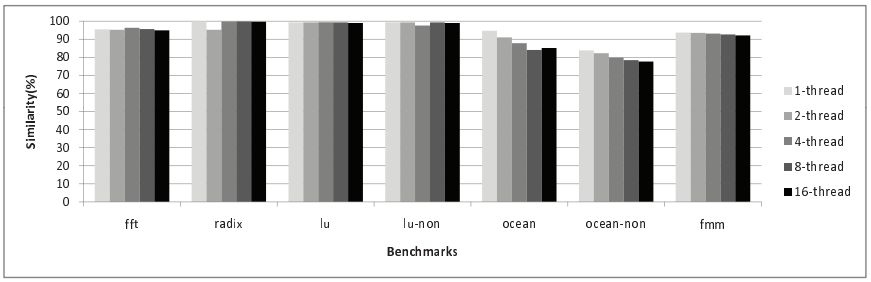

Fig. 4. Call trace similarity for for solo-runs

Even though multiple applications are started to run at the same time, our approach is capable of distinguishing one application from another with different thread IDs. We partition all the benchmarks into 3 groups. Each group has 2 or 3 benchmarks and all the benchmarks in one group are started together in the same simulated full system. The hacked simulator generated separated traces for each application. The partition of groups is given in table 1 and the calculation results for the two platforms are shown in Figure 5 and 6. Note that each instruction and call trace extracted from GEM5 in co-runs are compared with their counterparts collected by the Pin-tool in solo-runs. All the benchmarks in the same group were started with 1-16 threads and the simulated system was configured with 4 physical cores. Hence, target threads are swapped in and out frequently at runtime because the number of threads is much larger than that of available cores. The calculated results for instruction traces are very close to each other.

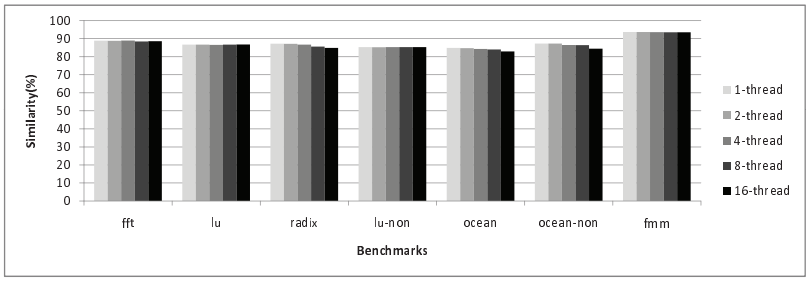

Fig. 5. Instruction trace similarity for co-runs

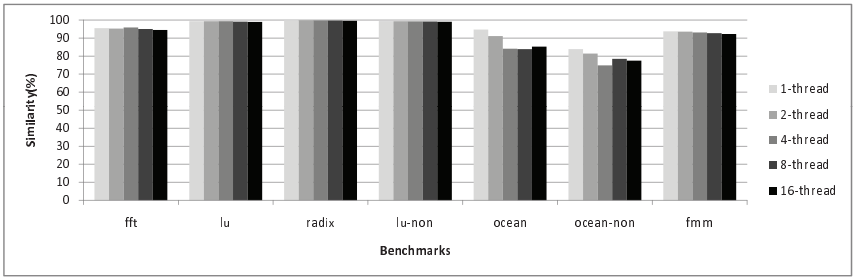

Fig. 6. Call trace similarity for co-runs

4 Related Work

Many approaches have been in use for obtaining low level instruction traces for applications, including dynamic instruction instrumentation, exploiting of hardware performance counters, utilization of the hardware monitor and instruction simulators or emulators [5].

Anita et. al. built a system for generating and analyzing traces based on link-time code modification [6], which makes the generation of a new trace easy in early days. Their system was designed for use on RISC machines and on-the-fly analysis removes most limitations on the length of traces. Binary dynamic instrumentation tools such as Pin [7], and other similar tools can collect application instruction traces by modifying the application instruction stream when running. Some tracing tools can be easily built with Pin and it widely used by researchers to obtain traces on IA-32 and x86-64.

Full system simulators and emulators, such as Simics [2], GEM5 [3], and QEMU [4], have the ability to collect instruction traces. But limitations can be found as well. The Linux-process-tracker is a Simics-provided module that allows tracking user-specified processes by either process id (pid) or file name in Linux [8], which inspects the simulated operating system and calls the callback functions when interesting things occur. The GEM5 simulator is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture [3]. It is widely adopted in ar-

chitecture research and does offer detailed instruction trace functionality mainly for debugging purpose. QEMU is a fast processor emulator using a portable dynamic instruction translator [4]. It supports many ISA(x86, ARM, MIPS,etc) both on host and guest sides and also can run in full system emulation mode. QEMU has the ability to generate memory traces.

Researchers in application profiling and performance optimization have proposed several tools and frameworks that exploit hardware devices such as hardware performance counters to collect performance data [9]; while others built hardware devices [10] [11] [12] [13] to accomplish this work. With the help of these tools, researchers can collect application traces fast and accurately. Unfortunately, these equipments are either expensive or complicated to be set up and therefore cannot be widely used.

5 Conclusion

Traces record all the information about a program's execution in the form of instruction sequences. In this paper, we propose a new approach to extract threaded traces for applications in full system simulation environments. Traces of each application are extracted from the instruction stream blended with instructions from operating system modules and other applications. Each thread in a given application is identified by an instruction pattern without inspecting the internal state of the operating system. Our approach can be applied to existing full system simulators with no changes to compilers. We implemented our instruction extraction approach in the simulator GEM5 and performed a number of experiments on the simulated full system x86-Linux. Experimental results show that traces extracted by our approach exhibit high similarity to the traces collected by a Pin-tool.

References

1. Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing Parallel Programs with Pin. Computer, 43(3), 34-41, 2010.
2. Virtutech. Simics User Guide for Unix 3.0,Virtutech, 2007.
3. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The GEM5 simulator. SIGARCH Computer Architecture News, 39(2), 1-7, 2011.
4. Fabrice Bellard. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference. Berkeley, CA, USA, 41-41, 2005.
5. Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. ACM Computing Surveys, 29(2), 128-170, 1997.
6. Anita Borg, R. E. Kessler, and David W. Wall. Generation and analysis of very long address traces. SIGARCH Computer Architecture News, 18(3a), 270-279, 1990.

7. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. New York, NY, USA, 190-200, 2005.
8. Xueling Chen. SimSight: a virtual machine based dynamic call graph generator. Technical Report TR-UNL-CSE-2010-0010, University of Nebraska at Lincoln, 2010.
9. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In Proceedings of the 2000 ACM/IEEE conference on Supercomputing. Washington, DC, USA, Article 42, 2000.
10. Ashwini Nanda, Kwok-Ken Mak, Krishnan Sugarvanam, Ramendra K. Sahoo, Vijayaraghavan Soundararajan, and T. Basil Smith. MemorIES3: a programmable, real-time hardware emulation tool for multiprocessor server design. SIGARCH Computer Architecture News, 28(5), 37-48, 2000.
11. Chalainanont, N. and Nurvitadhi, E. and Morrison, R. and Su, L. and Chow, K. and Lu, S.L. and Lai, K. Real-time l3 cache simulations using the programmable hardware-assisted cache emulator. IEEE International Workshop on Workload Characterization, 86-95, 2003.
12. Hyung-Min Yoon, Gi-Ho Park, Kil-Whan Lee, Tack-Don Han, Shin-Dug Kim, and Sung-Bong Yang. Reconfigurable Address Collector and Flying Cache Simulator. In Proceedings of the High-Performance Computing on the Information Superhighway. Washington, DC, USA, 552-556, 1997.
13. Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. HMTT: a platform independent full-system memory trace monitoring system. In Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. New York, NY, USA, 229-240, 2008.
14. Woo, Steven Cameron, et al. The SPLASH-2 programs: Characterization and methodological considerations. ACM SIGARCH Computer Architecture News. 23(2), 24-36, 1995.