Annotatable Systrace: An Extended Linux ftrace for Tracing a Parallelized Program

Daichi Fukui Mamoru Shimaoka Hiroki Mikami Dominic Hillenbrand Hideo Yamamoto
Keiji Kimura Hironori Kasahara

Waseda University, Japan

{fukui, shimaoka, hiroki, dominic, hideo}@kasahara.cs.waseda.ac.jp, kimura@apal.cs.waseda.ac.jp, kasahara@waseda.jp

Abstract

Investigation of the runtime behavior is one of the most important processes for performance tuning on a computer system. Profiling tools have been widely used to detect hot-spots in a program. In addition to them, tracing tools produce valuable information especially from parallelized programs, such as thread scheduling, barrier synchronizations, context switching, thread migration, and jitter by interrupts. Users can optimize a runtime system and hardware configuration in addition to a program itself by utilizing the attained information. However, existing tools provide information per process or per function. Finer information like task- or loop-granularity should be required to understand the program behavior more precisely. This paper has proposed a tracing tool, Annotatable Systrace, to investigate runtime execution behavior of a parallelized program based on an extended Linux ftrace. The Annotatable Systrace can add arbitrary annotations in a trace of a target program. The proposed tool exploits traces from 183.equake, 179.art, and mpeg2enc on Intel Xeon X7560 and ARMv7 as an evaluation. The evaluation shows that the tool enables us to observe load imbalance along with the program execution. It can also generate a trace with the inserted annotations even on a 32-core machine. The overhead of one annotation on Intel Xeon is 1.07 us and the one on ARMv7 is 4.44 us, respectively.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Parallel programming

Keywords Automatic parallelization, Linux, ftrace, Systrace, Multicore

1. Introduction

Multicore processors have been employed in a lot of platforms from embedded devices to super computers. In order to utilize these multicore processors efficiently, it is important to parallelize a software appropriately.

Understanding the runtime behavior of a parallelized program is an essential task for performance tuning[1]. Widely used profiling tools can provide hot-spot information of a program. However, investigation of the runtime behavior along with the program execution is still a difficult work even using them. Trace information and its visualization tools have been used to analyze such kind of the runtime behavior. For instance, a Gantt chart of a parallelized program provides helpful information to analyze the program behavior and bottle-necks since it shows the progress of the tasks in the target program as a graph.

Android Systrace is one of the visualization tools for program traces, which captures program execution and system events like context switching and interruptions on an Android device[2]. The trace is shown as a Gantt chart. The trace information is exploited by Linux ftrace system[3][4]. Ftrace is a tool to trace Linux kernel events.

Intel VTune[5] and perf timechart[6] are also well known visualization tools. Both of them can trace system-level threads and draw Gantt charts. However, VTune works only on Intel processors, thus it cannot be applied to other processors. Perf timechart works on any Linux platforms and does not depend on their processor architecture, while it cannot trace task-level fine-grained information of a parallelized program. These tools can capture the information per thread or per function. However, finer information, such as loops and basic blocks, cannot be captured.

This paper has proposed a tracing tool, Annotatable Systrace, to analyze the parallelized program based on Linux ftrace. It can preserve the portability on different kinds of processor architectures. Here, a user can embed annotations in a program. They are sent to a newly developed kernel module through character device files. The kernel module receives the annotations and stores them in buffers temporar-

ily. Then, it sends them to the ftrace system. The ftrace system receives the annotations and integrates them with thread IDs, timestamps, and duration of each thread. After integrating them, the ftrace system creates a log file. Finally, an HTML file is generated from the log file. Reading the generated HTML file enables fine grain analysis of the behavior of tasks inside the parallelized program.

We evaluate the tracing tool with parallelized programs compiled by the OSCAR automatic parallelizing compiler. We also evaluate the overhead of the tool.

The main contributions of this paper are:

- Annotatable systrace helps to analyze runtime behavior of a task-level parallelized program in a fine grain manner. This is because annotations as strings in a trace are embedded using the extended Linux ftrace system.
- The evaluation result shows the proposed tracing tool is successfully applied up to 32 cores. It can be realized by preparing an interface for each core to avoid access contentions among cores.

The rest of this paper is organized as follows. Section 2 presents an overview of the Linux ftrace system. Section 3 gives a general concept of the proposed tracing tool. In Section 4, the OSCAR compiler is explained. Section 5 describes the evaluation environment and the target benchmarks. The proposed tool is applied to the OSCAR compiler to evaluate performance in Section 6. Finally, Section 7 gives conclusion.

2. Tracing a Parallelized Program using Linux ftrace

This section describes an overview of Linux ftrace and Android Systrace.

2.1 Linux ftrace

Linux ftrace is a tracing tool to capture kernel events. Ftrace becomes available by compiling the kernel with the following option enabled.

```
CONFIG_FUNCTION_TRACER
```

In order to access the ftrace system, `debugfs` file system must be mounted as following:

```
# mount -t debugfs nodev /sys/kernel/debug/
```

The traceable events by this tool are listed in

```
/sys/kernel/debug/tracing/events/
```

In this paper, the following kernel events are enabled to trace context switching:

```
# echo 1 > events/sched/sched_wakeup/enable
# echo 1 > events/sched/sched_switch/enable
```

These events are also enabled to trace thread migration.

2.2 Android Systrace

Android Systrace is included in Android SDK. This tool generates an HTML file from the captured kernel events by Linux ftrace to visualize them. A user can easily investigate the progress of the target program and other kernel events on Google Chrome web browser.

3. Annotatable Systrace

This section describes an overview of the proposed tracing tool, Annotatable Systrace. We extend Linux ftrace to trace arbitrary strings embedded in a user program together with other kernel events. In this paper, a target string to be traced is called 'annotation'. The runtime behavior of a parallel task created by a parallelizing compiler can be easily traced by embedding 'task name' as an annotation by the compiler.

Annotatable Systrace collects annotations from a parallelized program. Then, it sends them to the ftrace system in the kernel. The ftrace system integrates annotations with other kernel data such as context switching, timestamps and thread IDs. A Gantt chart is drawn in an HTML file from the integrated data. The HTML file includes thread IDs and the name of the program as well as parallel tasks. The architectural overview of Annotatable Systrace is described in Figure 1.

A new member is added to the `task_struct` structure in the Linux kernel. The data structure contains all the information about a process or a task. The new member keeps annotations sent from a user program.

Figure 1 shows a data flow of annotations from an application to the extended ftrace system in the kernel. A user program sends annotations embedded in it to character device files[7] through `write` system calls. The character device files are created under `/dev/` file system. A kernel module controls them. In the kernel module, a pointer to an instance of `task_struct` structure containing the sent annotations is passed to `trace_sched_switch` function. The `trace_sched_switch` function generates a trace log.

A parallelized application to be traced can consist of multiple threads. In addition, multiple applications can run on a processor simultaneously. Therefore, if there is only a single character device file, it can become a critical section in sending annotations from multiple threads or multiple applications. In order to avoid making a critical section, multiple device files are created as many as the cores on a processor. Buffers to store the annotations are also created inside the kernel module as many as the cores to avoid making the critical section.

A thread must determine the device file associated with the core on which it executes, since the device file is assigned on the core dynamically. The device file manager resolves association between the thread and the device file by providing the core id acquired from `sched_getcpu` function called by the thread. Note that even when multiple threads are assigned on a core, only one device file for the core sufficiently

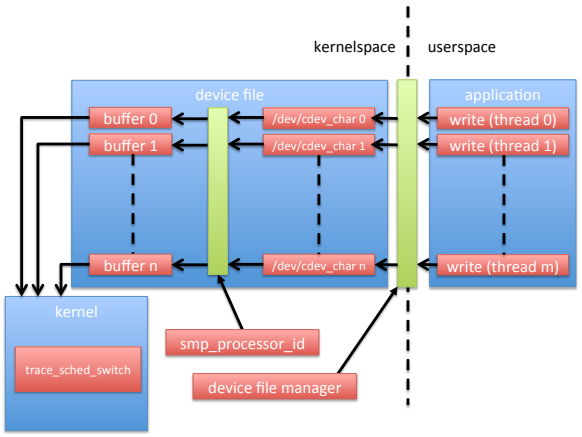


Figure 1. An overview of Annotatable Systrace

works without mutex-locks since only one thread is executed on the core at a time. The buffers inside the kernel module are also assigned to the cores by `smp_processor_id` macro.

4. OSCAR Multi-grain Parallelizing Compiler

This section introduces an overview of OSCAR multi-grain parallelizing compiler[8], which parallelizes target programs to be traced in this paper.

The OSCAR compiler exploits multi-grain parallelism from a C or Fortran program. Multi-grain parallelism consists of coarse grain parallelism, loop iteration level parallelism, and statement level near-fine grain parallelism. In exploiting the parallelism, the compiler first divides a sequential program into coarse grain tasks called macro tasks (MTs), which can be a basic block, a loop, or a subroutine. The compiler then analyzes both of control and data dependencies among the MTs, and a macro tasks graph (MTG) is generated after the analysis. The compiler generates parallelized code by assigning MTs statically when there are only data dependencies in the MTG. When there are control dependencies in the MTG, the compiler embeds a dynamic scheduling code in a parallelized program. When the parallelized program starts, OpenMP threads are created as many as target cores. The threads are synchronized via shared memory. They are joined in the end of the program.

The compiler embeds annotations to trace MTs and synchronizations.

5. Evaluation Environment

We evaluate the Annotatable Systrace on HA8000/RS440 with Intel Xeon and Nexus7 2013 with ARM multicore processor. Specifications of evaluated machines are shown in Table1 and Table2, respectively. As target applications, 183.equake, 179.art from SPEC CPU2000[9], and mpeg2enc from mediabench[10] are traced for the evaluation. They are parallelized by the OSCAR compiler.

Table 1. Specification of HA8000/RS440

Name	Hitachi HA8000/RS440
OS	Ubuntu 12.04.2 LTS (64bit, Linux 3.2.52)
CPU	Intel Xeon X7560 (2.27 GHz)
# of cores	32 (8-core x 4-socket)
L2 Cache	256KB/core
L3 Cache	24MB/core
Compiler	GCC-4.6.3
RAM	32GB

Table 2. Specification of Nexus7 2013

Name	Nexus7 2013
OS	Android 4.3 (64bit, Linux 3.4.0)
CPU	Qualcomm Snapdragon S4 Pro (1.7 GHz)
# of cores	4
L2 Cache	2MB
Compiler	arm-linux-gnueabihf-gcc-4.6.3
RAM	2GB

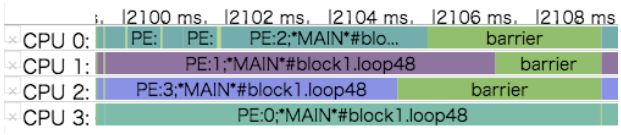


Figure 2. Load imbalance of parallelized 183.equake on Nexus7

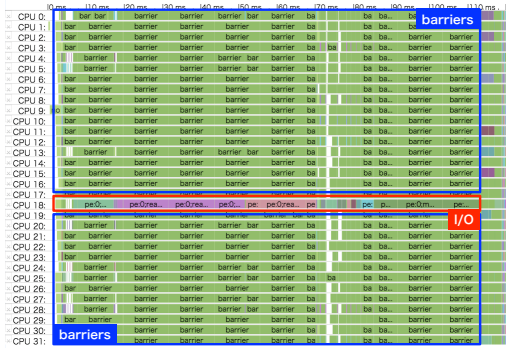
6. Evaluation Result

The evaluation is carried out to demonstrate the capability of the Annotatable Systrace using two scenarios: detecting load imbalance and parallelization on 32-core platform. The overhead of write system calls for annotations is also measured.

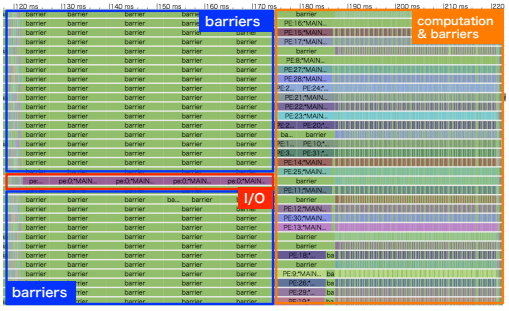
6.1 Detecting Load Imbalance

Figure 2 shows the execution trace of 183.equake parallelized for four cores, from CPU0 to CPU3, on Nexus7. 'loop48' in the figure is an MT, each of which is a sub-chunk of a parallelized loop. A green block labeled 'barrier' shows that the core is in a barrier synchronization.

The starting points of the barrier synchronizations on CPU0, CPU1, and CPU2 are different from each other according to the trace result. This situation shows the load imbalance occurred among those cores. This is because while-



(a) The first half of the program behavior of 183.equake on 32 cores



(b) The second half of the program behavior of 183.equake on 32 cores

Figure 3. The trace of 183.equake on 32 cores

loops in 'loop48' have fluctuations in the number of loop iterations.

Such the trace result can lead us to further optimizations, such as dividing MTs into finer sub-chunks to reduce load imbalance. Clock gating or power gating can be applied to the cores during barrier synchronizations if the target processor has a capability of power control.

6.2 Parallelization on 32-core Platform

Figure3(a) and Figure3(b) show the execution trace of 183.equake parallelized for 32 cores on HA8000/RS440. The first half of the program is mainly used for the initialization of the program including file I/O. There is a parallelized computational part on 32 cores at the second half of the program. These figures clearly show that the Annotatable Systrace can successfully generate an execution trace with arbitrary annotations even with 32 cores by providing a dedicated character device file and its associated buffer for each core.

6.3 Overhead

We also measure the overhead of the Annotatable Systrace. The measurement result for calling one write system call for an annotation is 1.07us on HA8000/RS440, and 4.44us on Nexus7.

Table 3. Total execution time of each program with and without tracing on RS440[s]

	systrace	ftrace	w/o trace
183.equake	0.101	0.076	0.076
179.art	1.13	0.717	0.714
mpeg2enc	0.442	0.187	0.185

Table 4. Total execution time of each program with and without tracing on Nexus7[s]

	systrace	ftrace	w/o trace
183.equake	1.47	1.44	1.47
179.art	2.23	1.90	1.83
mpeg2enc	1.15	0.289	0.291

In order to understand the impact of the overhead for a write system call in a parallelized program, the total execution time of each benchmark program with and without tracing on HA8000/RS440 and Nexus7 is measured as in Table3 and Table4. The benchmark programs are parallelized for 32 cores for HA8000/RS440, and for four cores for Nexus7. In the tables, 'systrace' stands for tracing with Annotatable Systrace, 'ftrace' stands for tracing with original ftrace, and 'w/o trace' stands for without tracing, respectively.

From Table3, 183.equake shows the lowest overhead on HA8000 /RS440. 'systrace' takes 1.32x execution time comparing with 'w/o trace'. On the other hand, mpeg2enc shows the highest overhead, such as 2.39x execution time for 'systrace' comparing with 'w/o trace'.

Similarly, from Table4 for Nexus7, 183.equake has little overhead, while mpeg2enc takes 3.95x execution time for 'systrace' comparing with 'w/o trace'.

The overhead of the Annotatable Systrace depends on the frequency of the write system calls for annotations. This heavily depends on the task granularity of the program. This is the main reason for the differences of the overheads among benchmark programs. A user should consider the relationship between task granularity and the overhead.

7. Conclusion

This paper has proposed the Annotatable Systrace to trace a parallelized program based on Linux ftrace. Annotations inserted in a parallelized program are integrated with other kernel events. The experimental evaluation was carried out on HA8000/RS440 and Nexus7 with benchmark programs parallelized by the OSCAR compiler. The evaluation result showed the proposed tool can successfully capture load imbalance along with the progress of tasks. Furthermore, the tool can trace even on the 32 core machine by providing its own dedicated annotation interface for each core.

References

[1] Eileen Kramer, and John T. Stasko: The Visualization of Parallel Systems: An Overview, Journal of Parallel and Distributed Computing, 1993

[2] Google: Android Systrace, http://developer.android.com/tools/help/systrace.html, 2015

[3] Jake Edge: A look at ftrace, http://lwn.net/Articles/322666/, 2009

[4] Steven Rostedt: Debugging the kernel using Ftrace - part1, http://lwn.net/Articles/365835/, 2009

[5] Intel Corporation: Intel VTune Amplifier XE 2015, https://software.intel.com/en-us/intel-vtune-amplifier-xe

[6] Stephane Eranian, Eric Gouriou, Tipp Moseley, Willem de Bruijn: Tutorial - Linux kernel profiling with perf http://perf.wiki.kernel.org/index.php/Tutorial, 2015

[7] Ariane Keller: Kernel Space - User Space Interfaces, http://people.ee.ethz.ch/ arkeller/linux/kernel_user_space_howto.html

[8] Obata M., Shirako J., Kaminaga H., Ishizaka K., Kasahara H.: Hierarchical parallelism control for multigrain parallell processing, LCPC 2002

[9] SPEC CPU 2000: https://www.spec.org/

[10] Chunho Lee, Miodrag Potkonjak, William H. Mangione-Smith: MedhiaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, 1997