Analyzing Blocking to Debug Performance Problems on Multi-Core Systems

Pierre-Marc Fournier, Michel R. Dagenais
École Polytechnique de Montréal
Département de génie informatique et génie logiciel
C.P. 6079, succ. Centre-Ville
Montréal, Québec, H3C 3A7, Canada
{pierre-marc.fournier, michel.dagenais}@polymtl.ca

ABSTRACT

Multi-core systems are rapidly becoming more prevalent. Consequently, developers frequently face performance bugs caused by unexpected interactions between parallel software components. The location of these bugs is difficult to identify with current tools. Indeed, the process exhibiting the slowness may be separated from the root cause of the problem by a blocking chain involving several other processes.

This article introduces a new approach for analyzing blocking on multi-core systems and reports on its implementation in the LTTV Delay Analyzer. It enables developers to quickly understand the dependencies among processes and see how the total elapsed time is divided into its main components. The LTTV Delay Analyzer was used to analyze and rapidly correct complex performance problems, something not possible with the existing tools. The Linux Trace Toolkit, LTTng, is used for most of the instrumentation and the trace recording, allowing the tracing of production systems with great accuracy and minimal impact. This approach uses solely kernel instrumentation and does not require the instrumentation or recompilation of processes. The analysis time is linear with respect to trace size.

1. INTRODUCTION

As systems become more parallel, developers face an increasing number of performance problems. These problems often have a system-wide scope and occur only on production systems. Since they do not produce execution errors or incorrect results, but only take more time than expected, traditional debugging tools are ineffective against them, making them challenging and time-consuming for developers.

The observable manifestation of a performance bug is that a process takes more time than expected to accomplish a given task – that is, to bring the system from one user observable state to another. In order to fix the underlying problem, the developer must work his way down to the root cause by first finding out how the total elapsed time was spent. This time might have been consumed in three ways.

1. The process was using the processor. Too much time spent this way indicates an intrinsically challenging problem with unavoidable high processor usage, or an algorithmic bug internal to the process.
2. The process was too frequently interrupted by external and often asynchronous events like IRQs, bottom halves[1] or preemption by the operating system.
3. The process was blocked on an operating system wait queue for an excessive amount of time, waiting for some hardware or another process to do something. For example, it might have been blocked in a `read()` system call waiting for data to arrive from a file descriptor, possibly pending a disk read access.

When encountering a performance problem on a small system, developers might guess in which of these states the system is spending too much time by a combination of the usage of operating system metrics (as displayed by tools such as `time`, `ps` or `strace`), the general feel of the system and instinct. On larger or more complex systems, however, doing so is much more difficult. For example, X Window is a complex application that involves many processes, communication among processes and with hardware, either directly or via the kernel. If X is slow to start on a system, finding out why is difficult with traditional tools. If the problem is hard to reproduce or lasts for very short periods of time, this difficulty is even greater. The proposed approach, however, makes debugging this type of problem much easier, as will be shown in section 2.4.

The way to fix a performance problem depends on its category. Problems of type 1) have been studied for a long time and can be further investigated with the traditional `gprof`[15] profiler, or, on a live system, with less costly profiling methods like `sysprof`[20] or `oprofile`[16]. The nature of problems of type 2) can be further understood by looking at operating system metrics such as the number of times each IRQ fired, the number of scheduling events related to the process, or the time other processes on the systems ran while the performance problem occurred. Means to investigate problems of type 3) (blocked process) are however much less readily available.

When a process spends too much time blocked, the problem can ultimately be tracked down to specific hardware not working fast enough. This means that some other processes in the system need to use the CPUs or that some hardware like a hard disk or a network card needs to produce data before the process can be unblocked. Although the process

[1] *Bottom halves* are also known as *second level interrupt handlers* or *software interrupts*.

itself can be directly blocked on this hardware resource, it is often not the case. Frequently, there is a chain of processes blocked on other processes for several levels until the direct blocking on hardware.

In this paper, we describe a method to debug performance bugs involving blocked processes. In order to do so, we define a set of instrumentation points to add to the operating system in order to get the necessary data in an execution trace. We propose a trace analysis method that enables the user to determine easily whether the performance problem he is facing is blocking-related. Finally, we contribute an algorithm to extract the chains of blocked processes, that lead down to hardware root causes.

The LTTV Delay Analyzer, which implements the proposed approach, is designed to run on production systems with minimal performance impact. Furthermore, the approach does not necessitate recompilation of code, as it relies on operating system instrumentation only. This black box model of processes enables the use of the method even if the source code to the processes involved is unavailable.

1.1 Previous Work

The DTrace dynamic tracer allows the debugging of blocking-related performance problems, as shown in the case study in [10]. The approach for debugging such problems with DTrace consists in initially writing a D script that instruments the symptom and procures more knowledge about the condition in which it occurs. This in turn allows to write other scripts that instrument that condition, walking the causality chain down to the root cause and writing numerous scripts along the way. Such a method requires advanced knowledge of the operating system in order to know where to instrument the code at each step. Also, the bug must be highly reproducible to allow for this iterative method. Finally, because DTrace works on a live system, a D script must contain code to filter out events not related to the bug, which can be a challenge.

Trace analysis tools with control flow views also allow debugging of blocking-related performance problems to a certain extent. Such tools include LTTV[2], the Wind River Workbench[8] and QNX Momentics[5]. Unlike DTrace, they work on a recorded trace which, if the instrumentation was sufficient, contains enough information to understand the problem without need for further instrumentation and tracing iterations. The trace can also be recorded with minimal resource usage for a long period of time, waiting for a rare bug to occur, using the flight recorder mode. The approach with these tools is however a manual one. The developer must inspect the Gantt chart of the control flow view in order to detect the unexpected behavior. This view contains a huge quantity of information and, unless one knows exactly what to look for, is mostly suited to help locate long states. Delays that result from the accumulation of short episodes are more difficult to identify visually. Another disadvantage is the absence of distinction between Blocked states with different causes on the screen. One must manually verify each one of them, which is a time consuming operation.

TIPME[14] helps identifying causes of user-perceived latency in interactive environments by summarizing the time passed in *running*, *runnable* and *blocked* states between a user input and the resulting graphical update. It does not however extract the chain of blocked processes.

Pip[18] generates a symbolic specification of the behavior of the program. This output can then be reviewed by the programmer to find anomalies. This specification can also serve as a reference that Pip can use to automatically compare a system behavior with. Alternatively, the programmer can manually write this specification. By including time constraints in it, Pip may be used to verify where the system spends an excessive amount of time. Such an approach makes a compromise between the detail level of the specification and the amount of bugs that can be detected. On a complex and evolving system, maintaining such a specification is challenging.

Magpie[9] traces the progression of a request through the various software components of a system as well as its resource usage. The path is deduced by correlating arguments of events generated by these components using an *event schema*. Magpie helps the programmer understand in what component a request that exhibited performance problems spent too much time. The detail level of this information is, however, related to the detail of the application instrumentation and event schema provided.

Paradyn[17] is a tool to find performance bottlenecks in parallel and distributed systems. It uses dynamic instrumentation that is activated and deactivated during the trace while testing various hypotheses. Paradyn tries to locate the performance problem under three axes: “why”, “where” and “when”. For each of these axes, a tree of possible causes is defined. Instrumentation is activated selectively for at most one node of a given depth at a time. Such an approach requires that the performance problem lasts for long enough and is dependent on the precision of the model.

DeBox[19] is a tool to provide system call performance behavior to the calling application. It does so by ”returning” a structure of metrics and other information after each system call. Among this information are details about each time the call blocked. These include the cause, the kernel source file and line, and sleep duration. While this approach proves useful for debugging single applications, it is limited for debugging complex systems when performance is affected due to a chain of blocked tasks.

In the subsequent sections, we first explain the architecture of the proposed analysis; we describe the instrumentation, the tracing method and the analysis algorithm itself. We then present the outputs of the Analyzer using a case study, followed by some results regarding the performance and memory usage of our technique. We conclude by discussing the advantages of the method and future work avenues.

2. ARCHITECTURE

The approach used consists in instrumenting the Linux kernel and tracing the system with LTTng. The trace is thereafter analyzed by a specially designed LTTV plugin. This analysis tool can execute on a machine whose architecture is different from that on which the trace was recorded. State

machines track blocking-related process data and make it available to report creation modules. These produce outputs that may be used by developers in order to understand blocking-related problems in the system.

2.1 Instrumentation

We instrumented the Linux kernel with the Linux Kernel Markers[11, 13], an in-kernel API for instrumenting its code. Markers offer virtually undetectable performance impact when deactivated and minimal performance impact when activated.

We instrumented scheduling changes (and whether they are caused by preemption or blocking), process wakeups, IRQ entry/exit, softIRQ entry/exit, trap entry/exit and process forking. Additionally, special *state dump* events are generated at the beginning of the trace that indicate in what control flow state (see section 2.3.1) each process is initially. All this instrumentation is kernel-based, no application instrumentation is necessary.

In order to display detailed information in reports, the Delay Analyzer also consumes events that give the following information.

- Mapping between IRQ number and name of hardware using it
- Mapping between softIRQ number and the function handling it
- Mapping between a process ID and its name
- Arguments of the `open()` system call, in order to save the mapping between file descriptor and file name
- Arguments of the `read()` system call, in order to know what file caused the system call to block
- What file descriptors caused a `poll()` system call to wake up

Work is under way to instrument yet more specific system calls.

2.2 Tracing

To trace the system, we use the LTTng[12] tracer. Its performance impact is very low, which allows recording traces on production systems. It provides accurate timestamping of events even on multi-processor systems, where the hardware permits. It is partly integrated in the Linux kernel and its integration is ongoing. Furthermore, it allows for an easy extension of its default instrumentation of the kernel.

A trace may be recorded in buffered write-to-disk mode or in *flight recorder* mode. The latter consists in recording the trace in circular buffers, in memory, without consuming their contents. This mode minimizes impact on performance. It is well suited to catch bugs that occur randomly or rarely and therefore require to record a trace for a long period. The tracing can go on for days if necessary; a specially designed script may then detect the first occurrence of the bug and stop the recording. Only then are the buffers transferred to disk. Their contents describe the system activity in the seconds or minutes preceding the bug occurrence, depending on the chosen buffer size. Hybrid approaches that use flight recorder for some buffers and write-to-disk for others are also available. Write-to-disk has the advantage of supporting traces larger than the memory.

2.3 Trace Analysis

An *a posteriori* analysis of the trace file produces reports that developers may use to study the performance problems at hand. We implemented this analysis in the form of a specially designed LTTV plugin[2] of about 2500 lines. LTTV is a trace viewing and analysis application. It is designed to handle efficiently traces many times larger than the workstation memory. LTTV may also read traces that were recorded on a system with different byte ordering and integer sizes than the analyzing system.

When debugging a performance problem, the developer wants to know why a process did not get in a certain state (the *target* state) more quickly, starting from a given time, at which it was in a different state (the *initial* state). For example, he might want to know why a reply from a server did not arrive more quickly, starting from the time the client sent the request. He could also want to know why an application was not started more quickly, starting from the moment he clicked on the icon to start it.

Normally, the recorded trace will cover the time span between these initial and target states. The trace will also generally include extra time spans at the beginning and at the end. Therefore, the developer must specify what these initial and target states are to the Analyzer[3]. He does so by specifying a timestamp that corresponds to the initial state, and a trace event that indicates the entry in the target state. The timestamp of this target state event is used as the time at which to end the analysis; moreover this event's process – the process in the context of which it occurred – is the process that will be analyzed. This process needs not be the process that is the root cause of the delay. It only needs to be a process that is exhibiting performance problems, mere symptoms. The analyzer will automatically follow the dependency chain down to the root cause of any blocking problem.

If this target process did not exist yet during a part of the time range that is analyzed, its closest parent is analyzed instead during that period. What the developer wants to know is indeed what the closest parent was doing that was preventing it from creating a closer parent (or the target process itself) earlier.

At this point, the start and end timestamps are used solely for the purpose of creating the reports. The part of the trace that precedes the analysis area is read fully, regardless of the requested initial and target states, in order to build the state of each process. In future versions, however, we plan

[2]This implementation is available in the LTTV repository at `http://lttng.org/`.

[3]In many cases however, this is not necessary because performance bottlenecks will often stand out even if there are extra time ranges that were traced at the beginning and at the end of the trace.

to integrate the state information into the LTTV checkpoint mechanism, which will require starting to read at the closest preceding checkpoint, rather than the beginning of the trace.

The architecture of the analysis consists of two state machines. Reports are created using the state inferred by these state machines. Figure 1 shows the interaction between them. The next sections explain the meaning of these states and how they are inferred.

2.3.1 First State Machine: Control Flow State Stack

During its execution, a process enters and leaves control flow states. The first column of Table 1 lists these states. Control flow states refer to the type of code that the control flow is executing at a given time, or to the fact that it is not executing. Control flow states can be nested in various ways. For instance, the kernel code that handles an interrupt may nest over a process running in userspace, in a system call, in a trap, in another interrupt, etc. When the execution of the kernel interrupt handler is finished, the kernel looks at the stack to find out what address to jump to, to return to the previous state. Because of this, the kernel instrumentation of the control flow state transitions of processes only yields events about relative state transitions. For example, an event might tell that a process just exited the IRQ context, but it will not tell what state it is going back to. Therefore, in order to be able to deduce the control flow state of a process at any time, when reading the trace, a model of the system must be maintained with a stack of these nested states.

This is the purpose of the control flow state stack. While the trace is being read, when an event indicating a state transition is encountered, the control flow state stack of the active process is updated. When an "entry" event occurs, a state is pushed onto the stack; when an "exit" event is encountered, the state at the top of the stack is popped and the new state is taken to be the one that was underneath. Each state has its specific entry and exit events. The context change event (event kernel_sched_schedule) is a special case. It indicates both that a process is being scheduled out (*Scheduled out* is then pushed on its stack) and that another is being scheduled in (*Scheduled out* is then popped from its stack) on a given CPU.

The *Scheduled out* state can occur in three different circumstances. 1. When a process is preempted by the operating system. In this case, it is pushed and later popped as a consequence of distinct kernel_sched_schedule events. 2. When a process gets scheduled out while blocked, waiting, via a wait queue, for a resource to become available. This state is pushed following a sched_schedule event, but popped following a sched_try_wakeup event that wakes it up. 3. Finally, the third case occurs just after (2): the process has been woken up but is still not running, it is waiting to be scheduled. Therefore, when a blocked process is woken (announced by a sched_try_wakeup event), a "blocked"-type *Scheduled out* is popped and a "waiting for schedule after blocking"-type *Scheduled out* is pushed on its control flow state stack.

Some control flow states are accompanied by additional important information. Some of this information is available immediately upon state entry. For example, the kernel_arch_syscall_entry (entry in a system call) event is accompanied by the syscall_id argument, which identifies the system call. Other information is not included directly in the event; it is rather delivered in later events. For example, the file name associated with an `open()` system call is announced in a distinct event that comes after the system call entry event. This is necessary because the events that announce entry into a control flow state must fire as close as possible to the actual state boundary in order to obtain a trace that reflects accurately the system state. At such an early point in the system call, its arguments have yet to be decoded.

Control flow state	Additional information
In userspace	–
In system call	syscall #, some syscall arguments
In trap	trap #
In IRQ	IRQ #
In softIRQ	softIRQ #
Scheduled out (not executing)	Type: "blocked", "preempted" or "waiting for schedule after blocking"

Table 1: Control flow states and the additional information kept with them.

2.3.2 Second State Machine: Working / Interrupted / Blocked (WIB) State

While the Control flow state corresponds to the real process stacks, the WIB (working/interrupted/blocked) state is a higher-level abstraction with no direct equivalent in kernel or user memory. It is more suitable for the study of the blocking behavior of the process. The WIB state is deduced from the Control flow state. Therefore, each time the Control flow state changes, the WIB state is updated if needed. The values it can take are the following.

When a process is running in userspace, it is running code that makes it progress toward fulfilling its purpose; it is therefore **Working**. The same is true when running in a system call. Even though the code being executed was not designed by the application programmer, it is still helping the application progress toward its goal by manipulating hardware and kernel structures for it. The same applies to traps.

If an IRQ or a softIRQ occurs in the context of the process, then it is considered **Interrupted**. It is also the case if the process is preempted. Additionally, if the process was blocked, then woken up but not rescheduled yet, it is considered Interrupted.

If the process is scheduled out and is not runnable, it is considered **Blocked**. In this case, it asked for a resource and that request blocked. This can be either directly or accidentally. Direct requests are done via system calls (for example, a blocking file reading operation, a blocking wait for a network packet). Accidental requests are done via traps (for example, a page fault). A process that is in a trap or a system call is Blocked only while it is actually scheduled out. Otherwise, it is in the Working state.

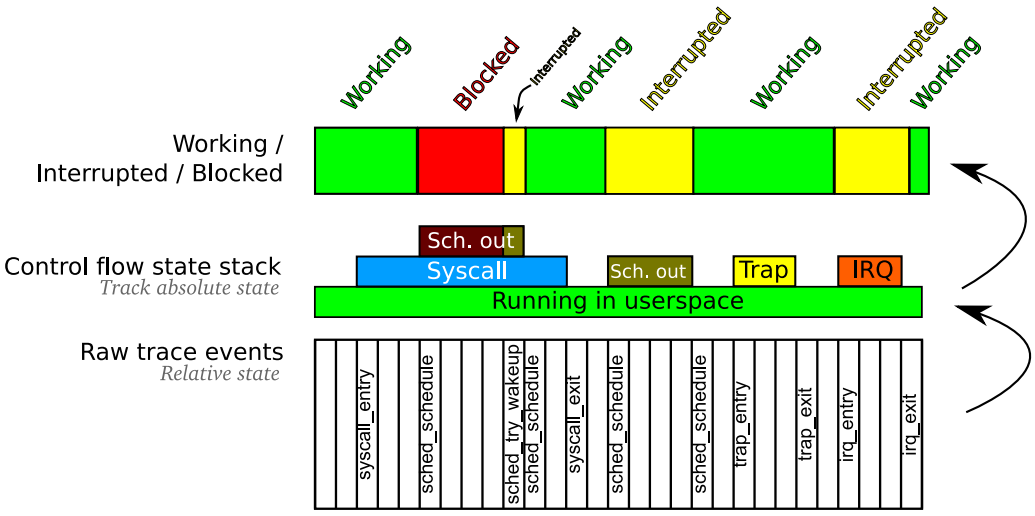

Figure 1: The entry and exit events from the raw trace (bottom) are used to infer the Control flow states, which are kept in a stack. These are in turn used to deduce the WIB states, which serve as main input to the report generation.

The time spent in any process p between times t_1 and t_2 is composed of the sum of time spans during which process p was working, interrupted and blocked. Therefore,

$$T_{p,t_1,t_2} = t_2 - t_1 = T^W_{p,t_1,t_2} + T^I_{p,t_1,t_2} + T^B_{p,t_1,t_2}.$$

These components themselves are the sum of each occurence of the state in the process. For instance,

$$T^W_{p,t_1,t_2} = \sum_{i=1}^{n} (t\_end^W_i - t\_start^W_i)$$

gives the total working time for process p by adding the time spans of the n occurences of the Working state the process was in. The same applies to the other WIB states (Interrupted and Blocked).

Unlike control flow states, WIB states are not stacked, as they can be deduced by examining the Control flow state stack. Table 2 shows how the WIB state is deduced from the Control flow state stack.

WIB states are also accompanied by additional information about the state of the process (Table 3). This information is extracted from the Control flow state stack and the additional information that accompanies the states in the stack.

The WIB state of each process serves as input to the report generation which is described in the next sections.

2.3.3 State Holdback

As explained previously, some information associated to the control flow state comes after the event that indicates the entry in that state. However, the WIB state must be updated at each change in the control flow state. If this is done as soon as the control flow state changes, some needed information will not be available for the WIB state. For example, a process enters an `open()` system call. When the kernel_arch_syscall_entry event occurs, the name of the file to be opened is not known yet. The process might block in this `open()`, and the file name may still not be know. In fact, the event that announces the file name arrives just before the event that announces the exit from the system call. The file name must, however, be stored with the Blocked state as its cause.

WIB State	Accompanying information
Working	start time, end time
Interrupted	start time, end time
Blocked	start time, end time, wakeup time, waking process, control flow state stack of waking process at wakeup time

Table 3: Information that accompanies each WIB state.

In order to compensate for this, our approach uses a hold-back mechanism. Some Control flow state push operations are not executed immediately on the stack. Rather, they are kept in a queue, if necessary as long as the event commanding the corresponding state pop does not arrive. When it does, its push and pop are processed and, in between, those of the states that were nested on top of it. Therefore, when the WIB state machine sees the control flow states, all their accompanying information is present. This mechanism is only used where necessary.

2.4 Reports and Case Study

The last part of the processing is the production of reports. Three reports are output by the LTTV Delay Analyzer. Each is described below through a case study.

On our dual-core 3 GHz system with 4 GB of RAM, starting Xorg and the KDE[1] desktop took 2.5 seconds. Although this is much better than some of our older systems, the fact that this machine had a very fast feel once the desktop was loaded left us under the impression that X was blocked at

Control flow state (top of stack)	Resulting WIB state
Running in userspace	Working
Running in system call	Working
Running in trap	Working
Running in IRQ	Interrupted
Running in softIRQ	Interrupted
Scheduled out (runnable)	Interrupted
Scheduled out (preempted)	Blocked
Scheduled out (waiting for schedule after blocking)	Interrupted

Table 2: Correspondence between control flow states and WIB states. The reason for which a state was scheduled out, which is enclosed in parentheses, accompanies each *Scheduled out* state in the memory structure.

some point during the starting sequence. We thus traced the starting of X and KDE.

On this system, running the `startx` script starts X and KDE. We started the trace just before running `startx`, in the same command. In order to know when the desktop was completely loaded, we put a script in the KDE Autostart directory (~/.kde/Autostart). Scripts in this directory are run as soon as the desktop is loaded, enabling us to locate quickly that point in the trace. The script ran a command to stop the trace. Although it was convenient to have the trace contain exclusively the interesting time span, it was not mandatory. We could have manually started the trace well before running `startx` and stop it well after the execution of the Autostart script.

We then ran the Delay Analyzer on the trace. As start and end points for the analysis, we used respectively an event at the beginning of the `startx` process and an event at the beginning of our Autostart script. As process to analyze, we chose the Autostart script, since the goal was to understand why it was not started earlier.

The Autostart script is run at the end of the trace, so the analyzer must look at its closest parent while analyzing the time range before its creation. This resulted in a chain of eight processes illustrated in Figure 2. The startup operation of X/KDE is relatively complex; other processes not involved in this parental chain run concurrently with it. One of them is `Xorg`, the process of the X server.

2.4.1 Report: WIB State Summary

The Delay Analyzer outputs a series of reports, the first of which is the WIB State Summary. Figure 3 shows an exerpt of the one that was obtained with this trace.

This summary indicates how much time was spent in each WIB state for a given process, during the time span being analyzed. The summary takes the form of a tree where each node is a state that is a subpart of its parent. The root represents the total elapsed time spent in all states. The Interrupted WIB state is further divided in "IRQ", "softIRQ", "preempted" and "waiting for schedule after blocking". Time passed in IRQs and softIRQs is further classified by its ID. As for the blocked time, it is further classified as having occurred within a system call or in userspace (in which a trap occurred), then even further by syscall ID or trap ID. Yet even further, the time blocked in some system calls is classified by special system call-specific criteria. For example, a Blocked state in the `read()` system call will be classified by the file that was being read.

In most cases, this report should give a first indication as to what caused a performance problem between the initial and the target states. The programmer can, by comparing the time spent in each of the three WIB states with estimations of normal values, deduce which one lasted too long.

The WIB State Summary for this critical path showed the usage of time in each of the processes of Figure 2 before it created the next one in the chain – that is, the time that delayed the execution of the Autostart script.

The first process we looked at was `xinit`, as it accounted for about half the start time. Figure 3, the part of the WIB State Summary that concerns `xinit`, shows that it basically spent the 1.3 seconds waiting in `select()` system calls.

2.4.2 Report: WIB State Instances

The WIB State Instances report shows, for each line of the WIB State Summary, the list of time spans during which the process was in that state. The labels (for example "<8>") in the state summary refer to such a list of state instances. These instances are ordered by decreasing duration because usually the longest ones will be the ones that a programmer will want to investigate first. This list can serve to locate instances of the states in the trace in order to study them with another LTTV view. The Control Flow View is most useful for this. The timestamps of each time span can also be used to find instances of blocking in the Blocking Causality report, which is described next.

Jumping to the WIB State Instances report for the `select()` calls in `xinit` gives Figure 4. Only one `select()` is responsible for 1.29 seconds of blocking. The Blocking Causality Report, described in the next section will reveal why this call blocked for so long.

2.4.3 Report: Blocking Causality

The Blocking Causality report shows the chronological list of time spans during which a process was blocked. It shows in what circumstances it was blocked, as well as how it was woken up (by what process, what IRQ or what softIRQ).

After the start event,
startx (pid 420) *ran for* **0.005269443 s**
before creating
xinit (pid 438) *which ran for* **1.309108247 s**
before creating
x-session-manager (pid 445) *which ran for* **0.116666736 s**
before creating
kdeinit (pid 500) *which ran for* **0.029303945 s**
before creating
kdeinit (pid 503) *which ran for* **0.240340942 s**
before creating
kdeinit (pid 519) *which ran for* **0.009239575 s**
before creating
kdeinit (pid 520) *which ran for* **0.806551758 s**
before creating
the autostart script (pid 542) *which ran for* **0.000002343 s**
before reaching the end event

Figure 2: Parental relationships from `startx` to the autostart script.

```
Process 438 [/usr/bin/xinit]
Total (1.309108247) <0>
  Blocked (1.294384393) <1>
    Syscall 23 [sys_select+0x0/0x16c] (1.294379908) <8>
    Syscall 57 [stub_fork+0x0/0x11] (0.000002859) <2>
    Syscall 59 [stub_execve+0x0/0xc0] (0.000001626) <4>
  Interrupted (0.009927037) <5>
    Scheduled out (0.009908719) <7>
    Waiting for schedule after blocking (0.000018318) <6>
  Working (0.004796817) <3>
```

Figure 3: WIB State Summary for xinit before it created x-session-manager.

Additionally, it recursively shows what the processes that unblocked the process were doing in turn.

Figure 5 shows the report that was obtained. A line that is indented one level deeper than the previous line means it refers to a Blocked state that occurred within the waking process of the Blocked state described in the previous line. Blocked states at the same indentation level occurred in the same process sequentially.

Therefore, a developer wanting to explore a particular blocking type found in the WIB State Summary can use the WIB State Instances report to find the timestamps of the blocking instances, which will lead him to the right point in the Blocking Causality report. In it, the chain of blocked processes may be followed down to the root cause of the Blocked state. The root cause can either be a process (that is CPU-bound or waiting to be scheduled) or a busy hardware resource.

When the deepest cause of a long Blocked state is a Blocked state ultimately woken by a process, the system call or trap within which the last Blocked state occurred gives information about the cause. If this is not enough, the analysis can be run on the time span of that deepest Blocked state to get a WIB State Summary for its waker, that details what it was doing. This state summary will contain only Working and Interrupted states. On the other hand, when the deepest Blocked state is woken by an IRQ or softIRQ, then its ID indicates the hardware that was responsible for the delay.

The method used to produce this report is the following. The WIB states of the process are read sequentially until a Blocked state is found. Information about it is printed. The control flow state stack of the waking process, as it was at the moment it woke the process up, is examined. If it was in a softIRQ or IRQ, this fact is printed and the unblocking cause is considered hardware based and unrelated to the process in the context of which it occurred. Otherwise the process was Working and is responsible for the unblocking. In this case, the WIB states of the waking process are read from the time the Blocked state began to the time it ended. The exact same algorithm is applied to it, but the printing is nested, indented one more level. The search for Blocked states then resumes and so on.

The Blocking Causality report obtained (Figure 5) shows that the 1.29 second long `select()` was woken by Xorg, the X server process. We were surprised to see that during that time, Xorg was itself blocked on a series of more than 40 `nanosleep()` calls, each lasting between 0.01 and 0.03 seconds.

An examination of the raw trace events was done with LTTV near the timestamps of these Blocked states. It showed that between the `nanosleep()` calls, X reads and writes data to a file descriptor, which is opened just before the `nanosleep()` sequence starts. This file is `/dev/psaux`, the character device used in Linux to talk to PS/2 devices.

Looking at the Xorg configuration file revealed that it was

```
Node id: <8>
439924.976620220-439926.270851832 (1.294232)
439926.270856657-439926.270922847 (0.000066)
439926.271033308-439926.271045577 (0.000012)
439926.271070723-439926.271079202 (0.000008)
439926.271061869-439926.271068187 (0.000006)
439926.271231493-439926.271237701 (0.000006)
[...]
```

Figure 4: WIB State Instances report for blocking on `select()` in xinit.

```
Process 438 [/usr/bin/xinit]
[...]
Blocked in RUNNING, SYSCALL 59 [stub_execve+0x0/0xc0], (times: [...], dur: 0.000002)
Woken up in context of 3 [migration/0] in WIB state UNKNOWN
Blocked in RUNNING, SYSCALL 23 [sys_select+0x0/0x16c], (times: [...], dur: 1.294232)
        Blocked in RUNNING, SYSCALL 2 [sys_open+0x0/0x17], (times: [...], dur: 0.013940)
        Woken up by a SoftIRQ: SoftIRQ 8 [rcu_process_callbacks+0x0/0x47]
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.200006)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 23 [sys_select+0x0/0x16c], (times: [...], dur: 0.203262)
        Woken up by a SoftIRQ: SoftIRQ 1 [run_timer_softirq+0x0/0x21a]
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010002)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010001)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010002)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010005)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010002)
        [... 37 other nanosleep() calls lasting 0.01 to 0.03 seconds ...]
Woken up in context of 440 [/usr/bin/Xorg] in WIB state WORKING
[...]
```

Figure 5: Blocking Causality report showing the Blocked state of `xinit` caused by a `select()` call and the Blocked states it depends on (indented). Time ranges were removed for lack of space. Each "Blocked in..." line shows the contents of the control flow stack of the blocked process.

indeed setup to use `/dev/psaux` to communicate with the mouse. PS/2 is a serial communication protocol that requires to wait for a certain time between the transmission of each bit, thus explaining the `nanosleep()` calls. It so happens that the only mouse connected to this system is actually a USB mouse. Therefore the character device was only being used as an emulator to the much slower PS/2 protocol. Even worse, the X driver was configured for protocol auto-detection, which consumed even more time during its start sequence.

After changing the X configuration to use the more modern *evdev* interface to communicate with the mouse, X took around 1.5 seconds to start, a 40% improvement from the original value.

2.5 Implementation

In this section, we describe and discuss our experimental implementation.

The analyzer reads a trace once, from its beginning to the end of the range to analyze. The trace must be read from the beginning in order to build the initial system state at the point where the analysis must start. Afterwards, at each interesting event, the control flow state of the corresponding process is updated (possibly with holdback), and its WIB state is computed. If it changed, the new one is added to an in-memory list of the WIB states through which the process passed.

Once the trace is fully read and the WIB state list is complete, the reports are prepared. The creation of the WIB State Summary and of the WIB State Instances report is straightforward. It necessitates a single pass through the array of WIB states for each process. The creation of the Blocking Causality report requires to iterate through the WIB states of each process, printing each Blocked state. For each Blocked state that depends on another process, the corresponding range is found in the WIB state array of that process using a dichotomic search. The chains of blocking are explored this way, recursively.

We have not found a way to build the Blocking Causality report incrementally while reading the trace in one pass, freeing the old WIB states. These old states cannot be freed because it is always possible that a Blocked state that lasts since the beginning of the trace will be unblocked. The old states of the waker will therefore need to be accessible to print the chain of blocking. Unfortunately, the waker is not known until the waking time.

Our implementation builds WIB state arrays for all processes because the process being analyzed can be unblocked by any process, and we will then want to know what that process was blocked on in order to display it in the Blocking Causality report.

This approach to the analysis has the advantage that its run time is roughly linear with respect to the size of the trace. However, its memory usage also grows linearly with respect to the trace size because every WIB state is kept in memory. This limits its ability to analyze long traces taken on very busy systems.

We plan to make the memory usage close to constant by not saving these states in memory. Instead, the algorithm will read the trace for only one process at a time – initially, the process containing the *target state* – and create the reports incrementally while reading the trace. For the Blocking Causality report, when a Blocked state is found in the process, the trace will be read until the end of the state is found. When it is, and the waking process is known, the analyzer will seek in the trace to the beginning of the Blocked state and read events related to its waking process, providing the indented content for that state in the report. If further nested Blocked states are found, another seek will occur, and so on, until the analyzer is back to the initial process and done reading it.

This improved implementation will use much less memory, but will likely take more time, as the trace will be read more than once. However, the analysis time will be bounded by the nesting level of Blocked states, which is quite low on the traces we saw. Furthermore, it will permit analysis of huge traces on any recent workstation.

3. RESULTS

3.1 tbench - Analysis Time

We recorded several traces on a system running the tbench[6] benchmark. Tbench generates a workload of network traffic between an arbitrary number of client and server processes. Both the tbench server and client processes were run locally, on the traced system, as the goal was to produce Blocked states between local processes, rather than real network traffic. All traces were recorded and analyzed on a dual-core 3 GHz system with 4 GB of RAM.

Figure 6 shows the analysis time with several trace sizes, recorded both with one and two processors. For the single-processor traces, one of the cores was deactivated using the Linux CPU hotplug facility. The analysis itself was always run with both cores enabled, but is single-threaded and therefore used only one core. Figure 7 shows similar measurements, but with a varying number of tbench clients.

Figure 6 indicates that with either one or two processors, the analysis time grew linearly with the size of the trace. The nth point for one processor corresponds to a trace of the same duration as the nth point for two processors. The clusters from left to right represent traces of 1 to 8 seconds, at 1 second intervals. For a given trace duration, the analysis time doubles when the number of processors doubles. This is not surprising, because the system can accomplish twice as much work, therefore the trace size doubles.

Figure 7 indicates that the analysis time grows linearly with the trace size, with different numbers of tbench clients. The analysis time appears independent of the number of clients.

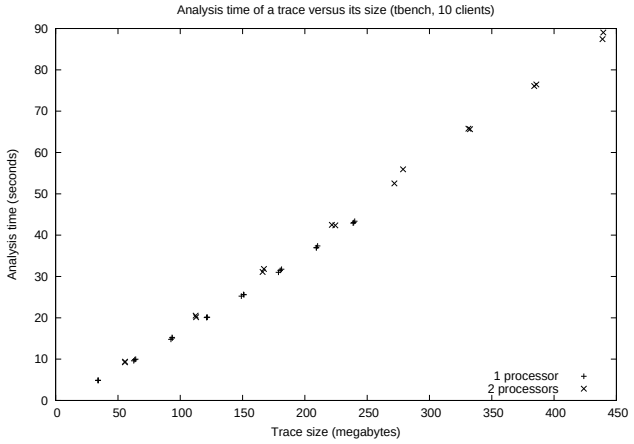

Figure 6: Analysis time versus trace size, system traced while running tbench, with one and two cores. Clusters, from left to right, correspond to traces that lasted 1 to 8 seconds.

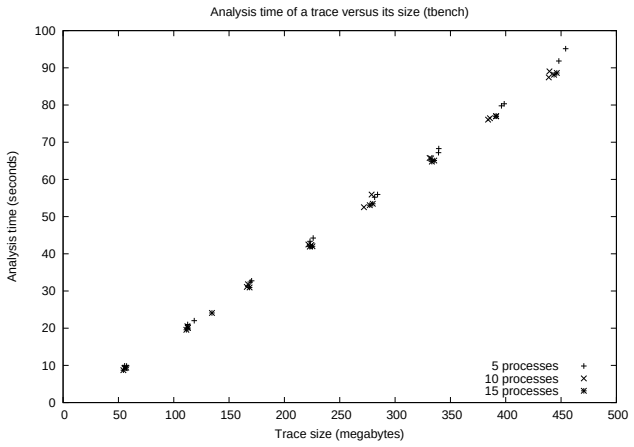

Figure 7: Analysis time versus trace size, system traced while running tbench, with various numbers of tbench client processes. Clusters, from left to right, correspond to traces that lasted 1 to 8 seconds.

3.2 tbench - Memory Usage

Figures 8 and 9 show the memory usage of the analyzer while it is analyzing 8 second traces recorded on a system loaded with tbench. The memory usage is quite high for such short traces because of the correspondingly high rate of system calls done by tbench. Figure 8 shows two traces, one recorded on the same dual-core machine as before, the other recorded on the same system, with one core disabled. Figure 9 shows similar measurements done with both cores active, but with a varying number of tbench clients.

In Figure 8, the memory usage increases linearly as the trace is read. The analyzer working on the one-core trace stops increasing its memory usage about twice as fast as its counterpart. This is because the 8 second trace taken on a two-core system is twice as big as the one taken on the single-core system, because the system was able to do twice as much work. Its parsing takes twice as long. Afterwards, both have

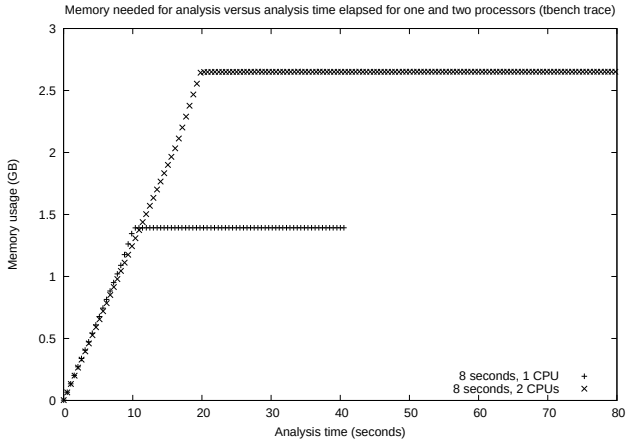

Figure 8: Evolution through time of the analyzer memory usage while processing 8 second traces recorded on a system running tbench on one and two cores.

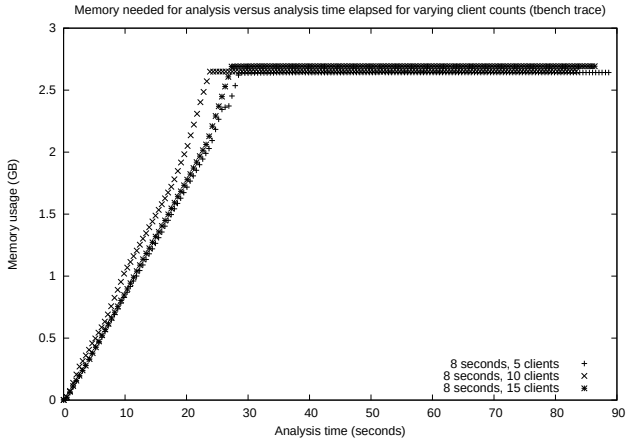

Figure 9: Evolution through time of the analyzer memory usage while processing 8 second traces recorded on a system running tbench with varying client counts.

a constant memory usage for some time while the reports are being generated. The largest portion of the report creation time is for the Blocking Causality report. Creating this report for the two processor trace takes about twice as long because there are about twice as many WIB states.

Figure 9 shows that the memory usage is about the same for all client counts. In all cases, the two processors were saturated by a number of clients greater than the number of CPUs, resulting in traces roughly equal in size and in parsing time.

3.3 Web Server

To get an impression of the performance of the analyzer under a real-life load, we traced a web server.

We hosted a copy of the Tracing Wiki[7] on the same system as before. We used the MediaWiki[3] engine 1.13.2 with

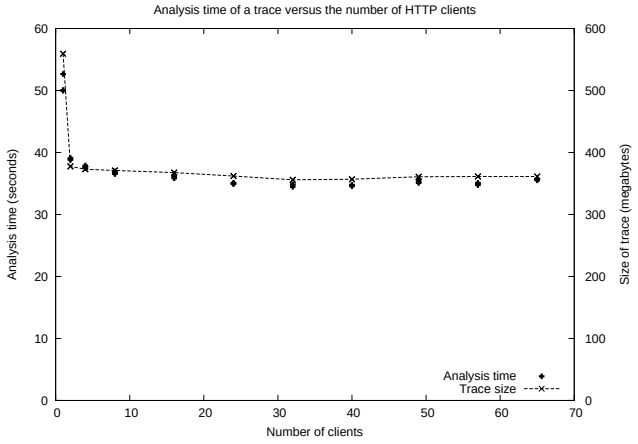

Figure 10: Analysis time and trace size versus number of web clients.

memcached[4] 1.2.2, that we hosted with Apache 2.2.9 and MySQL 5.0.51a.

We traced the server while two other machines were downloading each a full copy of the wiki (pages, images, documents). We used `wget` as client. For each run, we varied the total number of `wget` clients (half on each server) between which the full download was spread.

Figure 10 confirms that the analysis time does not vary with the number of clients but is rather constant. The abnormally high value for two clients accompanies a correspondingly large trace size. This is due to the fact that the system is not fully used with only two clients. It therefore takes longer for the downloads to complete, resulting in a larger trace that takes longer to analyze.

Figure 11 shows the increase of analysis time versus the number of events in a trace. To obtain this graph we used one of the traces recorded for Figure 10. We ran the analysis simulating the end of the trace after varying event counts. As expected, the analysis time seems to grow linearly with the number of events read.

4. CONCLUSION

We have described a method for analyzing performance problems on multi-core systems using LTTng and the LTTV Delay Analyzer. The instrumentation required and the method used for recording traces were explained. Afterwards, we described the method used for analyzing the trace and producing reports that allow the programmer to understand performance problems exhibited by a system. Finally, we presented performance measurements.

Our current implementation may exhibit memory usage problems on extremely large traces, since it increases slowly but linearly with the trace size. However, modifications that would result in nearly constant memory usage while retaining a linear execution time increase, with respect to trace size, were described.

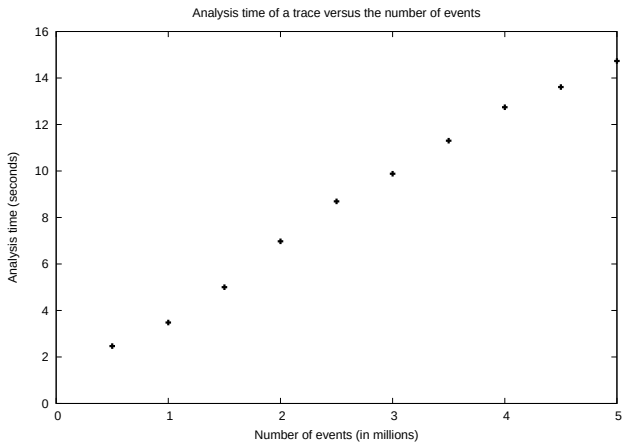

Figure 11: Analysis time versus the number of events in a web server trace.

4.1 Future Work

Our approach could be extended to follow dependencies between processes on different computers communicating through a network. This case is similar to the one of communicating processes on the same computer. For example, when a process is blocked waiting for a packet to arrive from the network, the Blocking Causality report could show what the process that eventually sends that packet was blocked on. Tracing several nodes simultaneously would be necessary. Algorithms to reconcile packet sends and receives would also be required.

Another similar application is to follow blocking chains across the physical/virtual machine boundary. Applying our approach as presented in this paper to virtual machines presents challenges. Indeed, the virtual hardware causing a delay may be software emulated. LTTV is already able to combine traces taken simultaneously on a virtual machine and its host, provided a common time base is used by the tracer.

The addition of a graphical tool to explore the reports would allow to display more information and to cross-reference it with that of other LTTV plugins. Finally, more instrumentation, notably for system call arguments, would permit the production of more detailed reports.

5. ACKNOWLEDGEMENTS

We wish to thank Mathieu Desnoyers for his insightful comments about tracing and performance analysis, and for answering the first author's numerous questions about LTTV and LTTng.

6. REFERENCES

[1] K Desktop Environment (KDE). `http://www.kde.org`. Verified 2009/01/05.

[2] LTTV. `http://ltt.polymtl.ca`. Verified 2009/01/05.

[3] MediaWiki. `http://www.mediawiki.org`. Verified 2009/01/05.

[4] memcached. `http://www.danga.com/memcached/`. Verified 2009/01/05.

[5] QNX Momentics. `http://www.qnx.com`. Verified 2009/01/05.

[6] tbench. `http://samba.org/ftp/tridge/dbench/README`. Verified 2009/01/05.

[7] Tracing Wiki. `http://ltt.polymtl.ca/tracingwiki`. Verified 2009/01/05.

[8] Wind River Workbench. `http://www.windriver.com/products/workbench/`. Verified 2009/01/05.

[9] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Symposium on Operating Systems Design and Implementation*, pages 259–272, 2004.

[10] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. pages 15–28, Boston, MA, USA, 2004.

[11] J. Corbet. Kernel markers. *LWN.net*, Aug. 2007. `http://lwn.net/Articles/245671/`. Verified 2009/01/05.

[12] M. Desnoyers and M. R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Linux Symposium*, Ottawa, Ontario, Canada, June 2006.

[13] M. Desnoyers and M. R. Dagenais. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium*, 2008.

[14] Y. Endo and M. Seltzer. Improving interactive performance using TIPME. *SIGMETRICS Perform. Eval. Rev.*, 28(1):240–251, 2000.

[15] S. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. volume 17, pages 120–6, Boston, MA, USA, 1982.

[16] J. Levon and P. Elie. Oprofile: A system profiler for Linux, 2005.

[17] P. Miller Barton, D. Callaghan Mark, M. Cargille Jonathan, et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[18] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Symposium on Networked Systems Design and Implementation*, pages 115–128, 2006.

[19] Y. Ruan and V. Pai. Making the" box" transparent: system call performance as a first-class result. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association Berkeley, CA, USA, 2004.

[20] S. Sandmann. Sysprof–a system-wide linux profiler. `http://www.daimi.au.dk/~sandmann/sysprof/`. Verified 2009/01/05.