Olympiads in Informatics, 2010, Vol. 4, 87–94

Performance Analysis of Sandboxes for Reactive Tasks

Bruce MERRY
ARM Ltd
110 Fulbourn Road, Cambridge, CB1 9NJ, United Kingdom
e-mail: bmerry@gmail.com

Abstract. Security mechanisms for programming contests introduce some overhead into time measurements. When large numbers of system calls are made, as is common in reactive tasks with processes communicating over pipes, this may significantly distort timing results. We compared the performance and consistency of two sandboxes based on different security mechanisms. We found that in-kernel security has negligible effect on measured run-times, while ptrace-based security can add overhead of around 75%. We also found that ptrace-based security on a dual-core CPU adds far greater overhead as well as producing highly variable results unless CPU affinity is used.

Key words: sandbox, security, timing.

1. Introduction

In programming contests supporting native languages (such as C), it is common practice to execute the resulting code in some form of secured environment, or *sandbox*. This makes the contest environment robust against malicious or defective solutions that may attempt to interfere with the evaluation process. This is achieved by some form of monitoring of the calls made by the application to the operating system (so-called *system calls*).

The performance impact of such monitoring will of course depend on the number of system calls that are made. Batch tasks are tasks in which the solution inputs data from a file, processes it, and outputs results to another file. Since I/O libraries generally buffer data, this usually results in relatively few system calls, even if there are a large number of individual input and output values[1].

Reactive tasks are tasks where a solution will both send and receive data in an interleaved sequence. Typical reactive tasks are query processing systems (where the solution will receive information, then answer queries about the information, potentially interleaved with updates) and games (where the solution will output moves in the game and receive input about the state of the game). For reasons of security, this is commonly implemented as a communication between processes using a pipe; since each output must

[1]It should be noted, however, that a common mistake of C++ programmers is to end lines using `std::endl`, which also flushes the output and hence produces a system call.

be produced before the response can be received, each communication results in a system call. Thus, reactive tasks can require hundreds of thousands of system calls, making them particularly sensitive to the performance of the sandbox.

2. Experimental Design

2.1. *Sandboxes*

We have used two sandboxes. The first is based on the interception of system calls from userspace (using the `ptrace` system call in Linux), which is used in the USA Computer Olympiad (USACO; Kolstad,2009).

The second sandbox is based on in-kernel security checks and implemented as a Linux Security Module (Merry, 2009). This is the security module used in the South African Computer Olympiad (SACO), but updated to run with Linux 2.6.30. Note that this version of Linux no longer allows such security modules to be loaded at run time, so the "module" is in fact a kernel patch and is part of the binary kernel image.

2.2. *Tasks*

To keep the number of variables manageable, we considered only a single task: *Regions* from the International Olympiad in Informatics 2009 (Fan and Peng, 2009). We also only used test case number 32 (the last and largest). This task is a query processing task, and for this test case there are 200,000 queries. This yields slightly over 400,000 system calls (one to receive each query, one for each reply, and a small number during startup and shutdown) — this was confirmed using strace (`http://linux.die.net/man/1/strace`).

The source code for the actual reactive grader was not available, so we used a dummy grader, which passes on queries from a file and accepts responses, but does not discards the responses rather than checking them.

We also ran each test in a "batch" mode. This used the same solution code (in particular, including a flush after each write), but directed input and output from and to files rather than an inter-process pipe. This has half the number of system calls (since output is flushed but input is buffered), but allows us to observe the effects of a large number of system calls in the absence of a reactive grader.

2.3. *System Setup*

Tests were run on a MacBook Pro with a 2.16 GHz processor and 1GB of RAM, running Linux 2.6.30. Some steps were taken to reduce the effect of random factors on the runtimes so that small differences between sandboxes would not be lost in the noise:

- All the important files (sandbox userspace, program to test, reactive grader, input and output files) were held on an in-memory filesystem (tmpfs), to eliminate the effects of disk caching and disk access time.

- To make the above more effective, all programs were statically linked, so that library code was also stored in RAM.
- Since the machine is a laptop, the CPUs are normally run at lower frequency when idle. For these tests, the frequency was forced to the maximum.

The same kernel was used for all tests, which included the patches necessary for the SACO sandbox. However, the boot-time kernel argument to enable this sandbox was only used where it was actively being tested. When the kernel argument is not passed, the patches may slightly increase the memory footprint of the kernel, but its security hooks are never called and so it should not degrade performance.

The USACO sandbox is written to deal with input and output files in a particular way. For our testing, it was modified to allow the sandboxed program to communicate directly with standard input and standard output. However, the security aspects of the sandbox were not modified.

The USACO sandbox also relies on the system calling conventions in the i386 architecture (i.e., 32-bit code). Although the kernel and operating system are 64-bit, the solution was compiled as a 32-bit binary (using the `-m32` option to GCC).

2.4. *CPU Affinity*

As well as comparing sandbox methods, we have investigated the effects of the number and configuration of CPU cores in the system. This was necessarily limited to two CPU cores on a single CPU package, since that was what was available in the test system. Four configurations were tested:

nosmp The `nosmp` command-line argument was passed to the kernel during boot, which limits the system to a single CPU, and also affects some code that needs to take additional steps in multi-processor systems.

none No special steps are taken. Processes run on both CPU cores, and are free to migrate.

separate Both the solution and the reactive grader (the process on the other end of the pipes) use CPU affinity masks to restrict them to a single core. The solution is locked to one core, the reactive grader to the other.

same Similar to the above, but both processes are locked to the same core.

3. Results and Analysis

Fig. 1 shows box plots for all combinations of sandbox and CPU affinity[2]. Times reported are the time measured for the solution process alone, computed as user plus system time. A number of observations can be made:

[2]The thick horizontal black bar indicates the mean, the boxes show the range covered by the central 50% of observations, and circles show outliers.

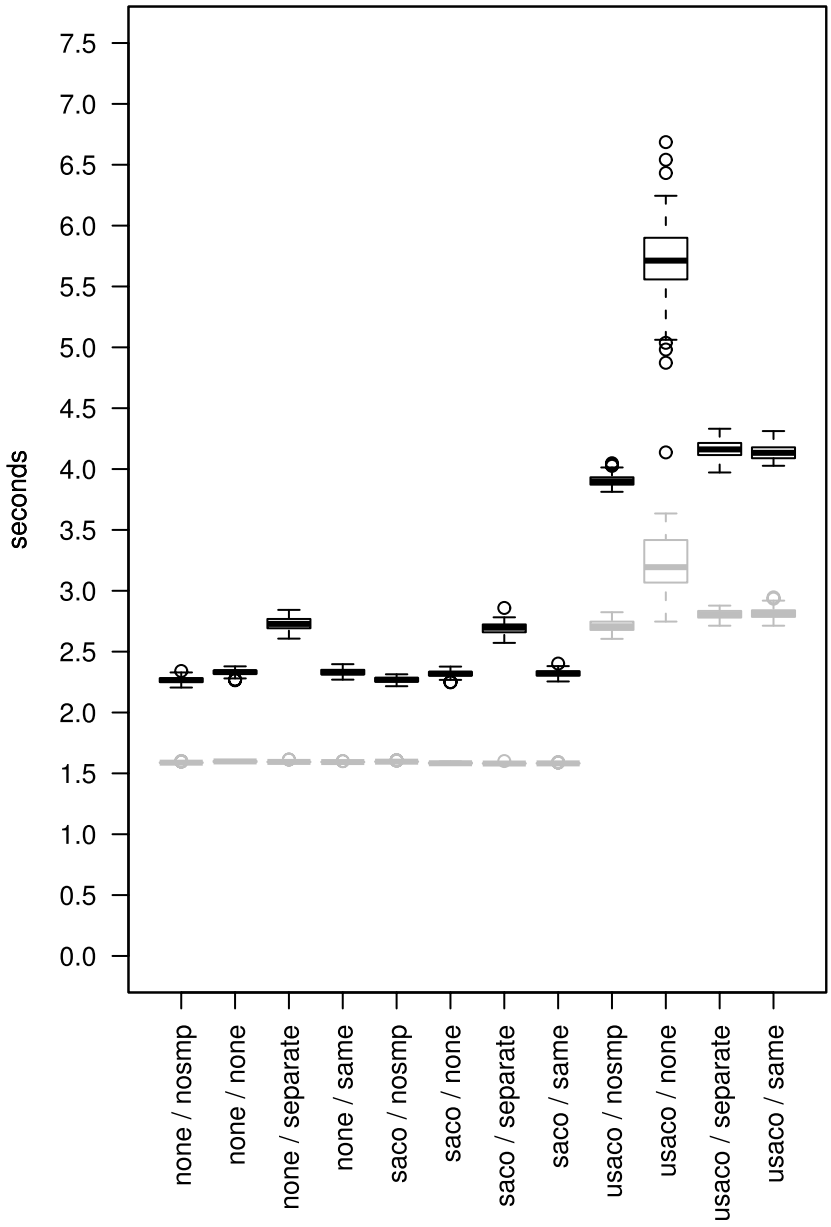

Fig. 1. Observed CPU times over 100 runs. Labels are of the form *sandbox* / *CPU affinity*. Black boxes show reactive grading, gray boxes show batch grading.

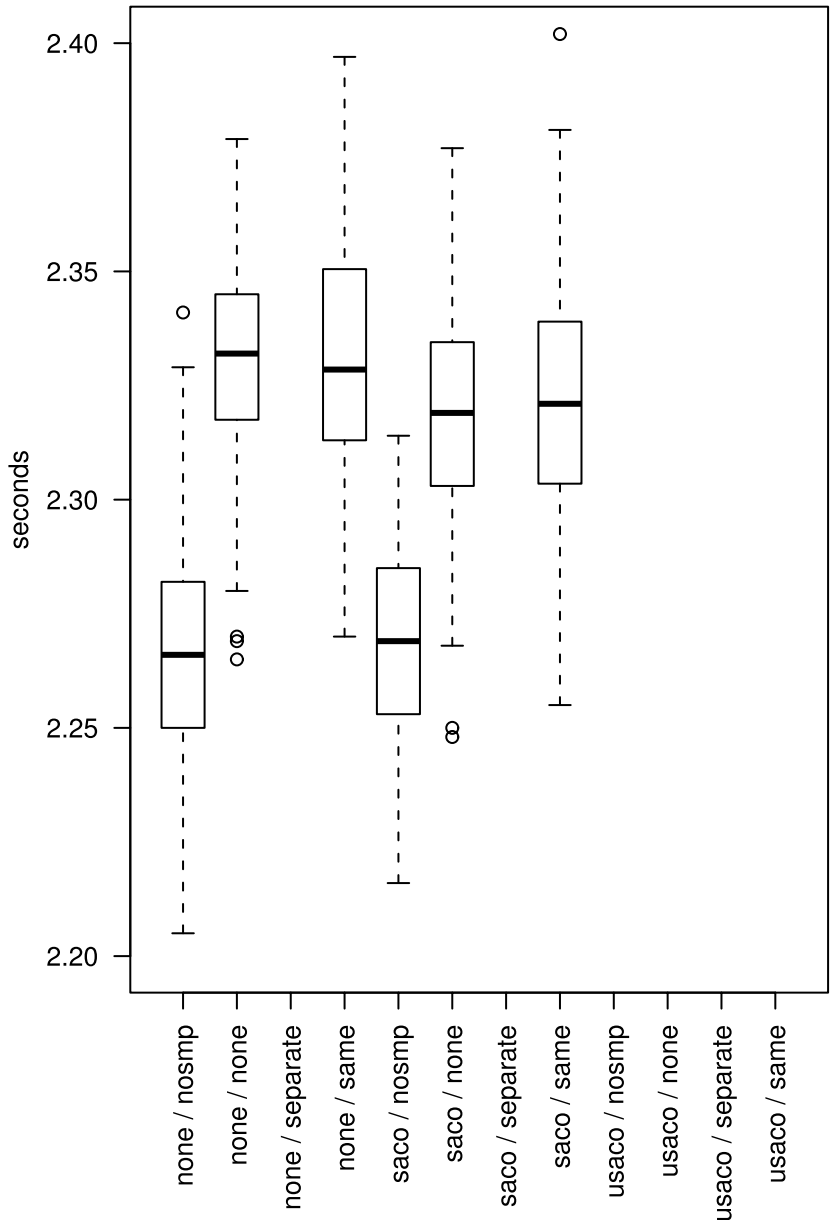

Fig. 2. Observed CPU times over 100 runs, close-up view. Refer to Fig. 1 for details.

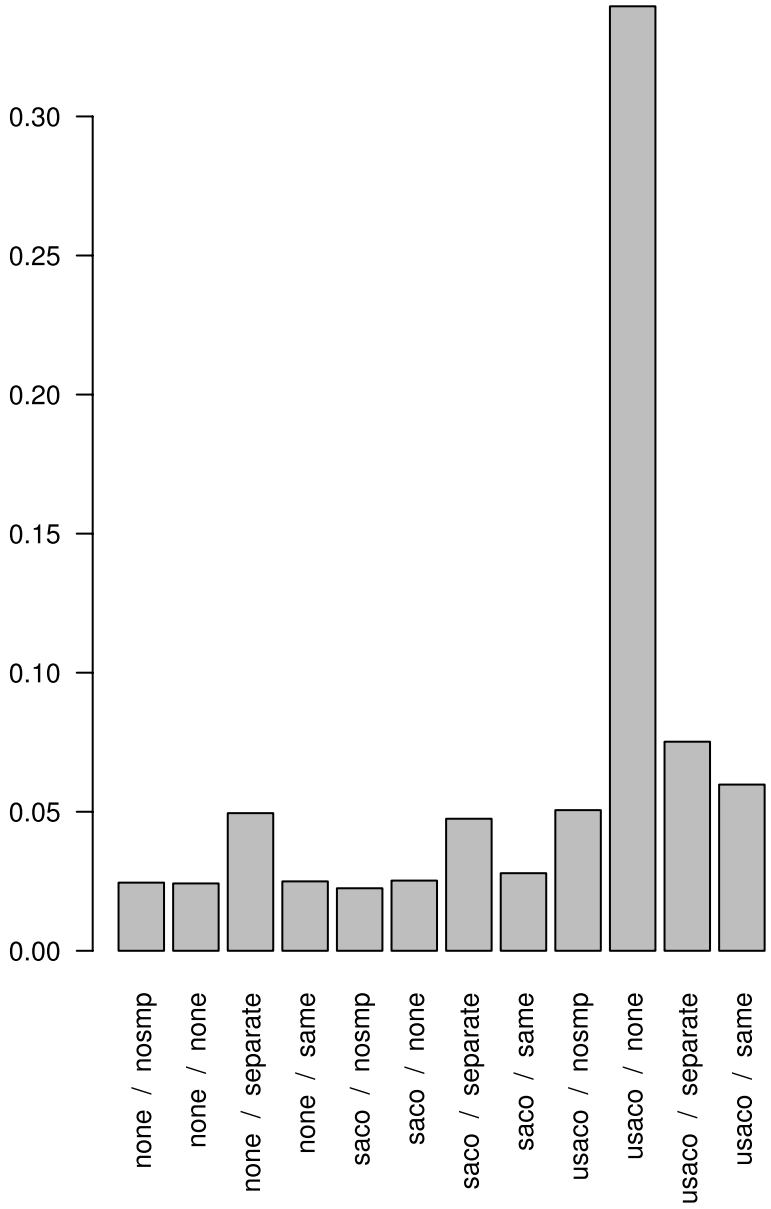

Fig. 3. Standard deviations of the test runs, for reactive mode only.

- The most striking result is that the USACO sandbox suffers both extremely high overhead and high variability when run on two CPUs without affinity masks, even in batch mode. While we have not performed a detailed investigation, a possible explanation is that the sandbox runs as a user-space process, and must examine memory in the solution process to determine whether the system call is safe. If the processes happen to be running on separate processors, then the memory may not be in the cache of the processor which needs to access it.
- For all choices of affinity and mode, the USACO sandbox introduces significant overhead.
- In reactive mode, forcing the processes to run on separate cores generally reduces performance. This may be due to the overheads of inter-processor communication, which may trigger cache writebacks or invalidations to ensure that the processors have a coherent view of the data.
- The batch times are generally lower, due to having half the number of system calls. Excluding the USACO grader, the differences between groups are statistically significant (other than between **same** and **separate**, which are equivalent without a reactive grader), but are clearly very small.

Fig. 2 shows a close-up view of some of the box plots, for reactive mode only. It is clear that when running with the SACO sandbox or without a sandbox, a single CPU yields lower times than two. These graphs also show that the SACO sandbox adds no statistically significant overhead (confirmed by t-test).

Fig. 3 shows the standard deviation associated with each test. It confirms that the USACO sandbox without affinity control suffers from much greater variability, while the SACO sandbox has no obvious effect. It also indicates that forcing the solution and the reactive grader to run on separate cores increases variation.

4. Conclusions and Future Work

It should be noted that the results show the measured run-times of the solution process. Lower run-times do not necessarily equate to faster evaluation, and indeed the **nosmp** tests may take longer to complete due to only a single core being available. From the contestants' point of view, timing should satisfy two requirements:

1. Times should have low variability, to ensure that evaluation is fair. If two contestants have functionally equivalent solutions, it should not be the case that one scores higher than the other due to random fluctuations.
2. The security system should not significantly alter times. While contestants ought to test their solutions on the evaluation system (this facility being commonly available in contests with on-line submission), it is nevertheless more convenient to be able to test locally with some expectation that similar timing will be seen during evaluation.

It is clear that ptrace-based security does not satisfy the second requirement where large numbers of system calls are involved. The first requirement can be met to some

extent with ptrace-based security, but only if CPU affinity masks are correctly configured on multi-core systems.

It should also be noted that this is a limited experiment, involving only a single CPU design, task, solution and test case, and so conclusions might not generalise. CPU cache and memory architecture in particular may significantly influence the effects of CPU affinity; and the pattern of timing in system calls may also have an effect.

Acknowledgements

Rob Kolstad kindly provided access to the USACO sandbox, and also suggested investigating CPU affinity.

References

Strace – Trace System Calls and Signals. `http://linux.die.net/man/1/strace`.
Fan, L., Peng, R. (2009). Task 2.3 Regions. In *21st International Olympiad in Informatics*, pp. 28–31.
Kolstad, R. (2009). Infrastructure for contest task development. *Olympiads in Informatics*, 3, 38–59.
Merry, B. (2009). Using a Linux security module for contest security. *Olympiads in Informatics*, 3, 67–73.

B. Merry took part in the IOI from 1996 to 2001, winning two gold medals. Since then he has been involved in numerous programming contests, as well as South Africa's IOI training program. He obtained his PhD in computer science from the University of Cape Town and is now a senior software engineer at ARM.