FAKEROOT^P: Scaling a Build Environment to Baseline and Beyond

Alexis Champsaur Sanidhya Kashyap

School of Computer Science, Georgia Institute of Technology

Abstract

Fake build environments are widely used among open source organizations and in the industry. Of these, FAKEROOT is the most common. And, with increasing core count in commodity machines, FAKEROOT should be expected to scale. Unfortunately, our experiments with compiling the Linux kernel over FAKEROOT reveal that it does not scale beyond 32 cores.

In this work, we identify some of the inherent design limitations of FAKEROOT, and we design and implement a scalable build environment—FAKEROOT^P. When used with `make`, FAKEROOT^P exhibits a multi-core scalability that is very close to that of `make` alone, and when used to compile the Linux kernel, it is 7× faster than its counterpart on 120 cores.

General Terms Design, experimentation, performance

Keywords Caching, hash-table

1. Introduction

Since the early 2000's, commodity machines with multiple cores have become increasingly common. Over the years, research has shown [1, 2, 4] that with the right implementation and the right changes to system code, multi-core machines can drastically accelerate the execution of many types of services and applications, and this without requiring significant changes to the overall organization of the operating system.

Still, there remain many questions on how to design applications and services on multi-core machines so as to take full advantage of the architecture. At the time that multi-core machines started to become common, significant research existed on distributed systems, where separate machines communicate over a network to provide a unified service to clients. Many of the concepts that apply to distributed systems are transferable to multi-core machines. These include, for example, the paralellization of tasks and several consistency models. The authors of [5], for instance, show that the well-known distributed programming model MapReduce is successfully implementable on a multi-core machine.

However, many of the conclusions reached in distributed systems research are not easily applicable to multi-core machines. Most notably, multi-core machines lend themselves much more easily to programming models based on shared memory. These models can make use of shared address spaces and threads. Failure scenarios, also, are different in multi-core machines. Indeed, a node may fail due to both hardware and software issues in a distributed system, whereas the failure of a single process in a distributed task on a multi-core machine usually only involves problems at the software level. In this environment, a node failure usually leads to the failure of the entire system and application.

One service that could greatly benefit from multi-core support is FAKEROOT, a session-based, single-node service that gives client applications that run over it the illusion that they have root privileges. While the FAKEROOT session lasts, a client application can create and modify files with root permissions. This emulation of file mode and ownership changes is used mainly for creating software packages with correct permissions and ownership.

In order to update the contents of software packages, application developpers and software maintainers need root privileges (a `UID` and `GID` of 0) for various security reasons. In FAKEROOT, this is accomplished by wrapping the program's system calls for file manipulation and emulating the behavior of privilged operations, while only unprivileged operations are actually being carried out, all of this based on the `LD_PRELOAD` mechanism. FAKEROOT can give a `UID` and `GID` of 0 to the programs that require this at the time of building, and it lets developers build distribution-ready packages on machines where they lack root privileges.

Due to its usefulness for building packages, FAKEROOT is most commonly used with `make`. And, with increasing package sizes and the increasing availability of multi-core

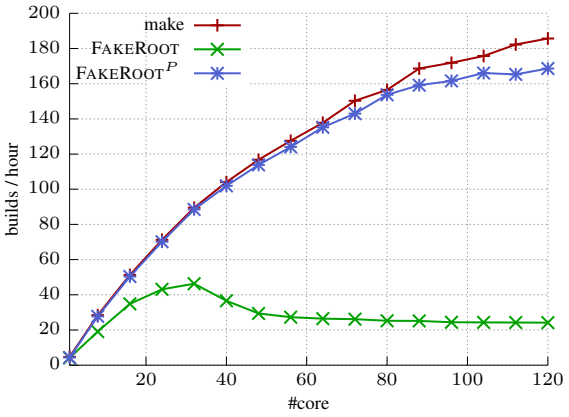


Figure 1: Performance of original FAKEROOT design

machines, it is becoming more important for developers and maintainers to be able to scale build processes on multi-core machines. Unfortunately, our experiments have discovered that `make` over FAKEROOT fails to scale after 32 cores (refer Figure 1) when compiling the Linux kernel. On its own, however, `make` scales very well with increasing core count. Our preliminary results show that `make` over FAKEROOT is $7.67\times$ slower than a simple `make` with 120 cores, and $1.92\times$ slower even with only 32 cores.

For this project, we re-designed much of FAKEROOT so as to allow it to scale on multi-core machines. After our improvements, `make` over FAKEROOT is only $1.1\times$ slower than `make` on its own with 120 cores. Our improvements were based first on making better use of some of the mechanisms that are key to taking advantage of multi-core architectures, such as shared memory, threads, and Unix message queues, and second on optimizations we made based on our profiling of the `make` over FAKEROOT process for the Linux kernel code. While our optimizations are based on profiling the Linux kernel build process, we expect them to apply to the build process of other codes as well.

2. Background

2.1 Original Fakeroot Design

The original design of FAKEROOT implies clear limitations in its scalability. As illustrated in Figure 2, launching a FAKEROOT session starts `faked`, the FAKEROOT daemon. Client processes may be created and terminated during a FAKEROOT session. Any number of concurrent client processes communicate with the daemon through a single message queue. Client requests have the form of regular system calls, and the daemon handles them by keeping track of the files accessed by the clients in such a way that for the duration of the FAKEROOT session, the clients have the illusion of root permissions on these files.

After inspecting a message received on the message queue, `faked` dispatches the associated request (eg. `stat`, `chmod`) to one of several handler functions. Depending on the type of

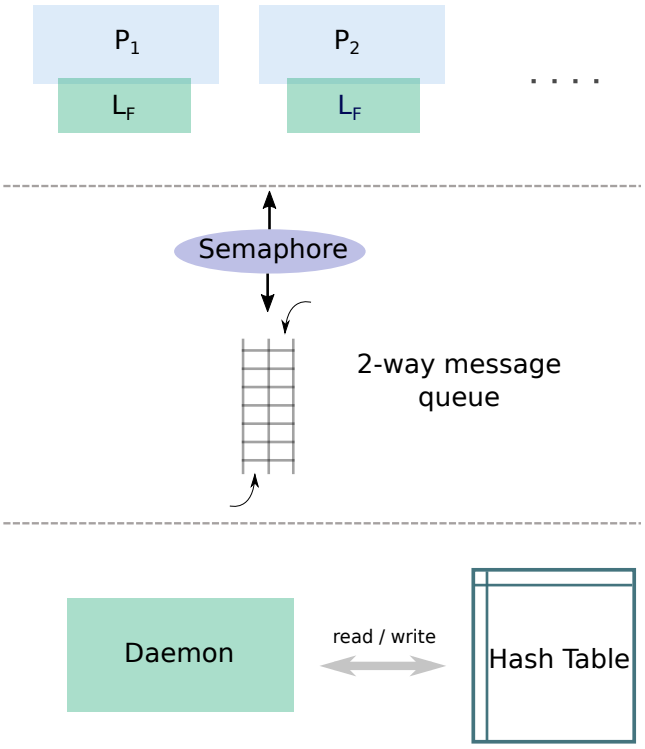


Figure 2: Original FAKEROOT Design

request, this function may or may not need to change the state of the FAKEROOT session's files. After the request is handled, the daemon sends the response back to the client process through the message queue directed the other way.

`faked` keeps track of the state of files created and accessed by the clients in a chaining hash table, to which it has both read and write access. Concurrent accesses to the hash table are controlled not by `faked`, but by the client processes themselves. Indeed, any process that runs over FAKEROOT links in `libFakeroot`, which gives it access to the message queues and to a semaphore. The clients acquire the semaphore before sending a request message to the daemon, and release it after they receive a response. This ensures that a response a client receives from the daemon corresponds to the request that it sent.

In this design, the hash table holds the states of the files accessed by all client processes in a single session. It is the only resource that could potentially be corrupted by concurrent client requests. In using a single message queue with a semaphore at the client level, FAKEROOT essentially serializes all requests to the daemon. This allows the daemon to handle requests from any number of concurrently running client processes, and it protects the hash table from concurrent accesses. However, it also results in the unscalable nature of this design.

Indeed, however many cores are allocated to the client's multiple processes, requests can only be handled by the single daemon process one at a time. This explains our initial

results showing that with large numbers of cores allocated to `make` over FAKEROOT, there is little or no performance improvement compared to when smaller numbers of cores are provided. In addition, many requests do not modify state, putting into question both the number of messages sent to the daemon and the need for using the semaphore with every request at the client level. Before coming up with our own design, however, we profiled both `faked` and `libFakeroot` during a FAKEROOT session used for building the Linux kernel.

2.2 Profiling

2.2.1 Messages and Accesses

In a single build task using 16 cores, we found that `faked` received and processed 1,806,287 messages on the message queue, resulting in as many hash table accesses. Of these accesses, 483 were writes.

2.2.2 Common Case

We also noticed that 1,798,038 of the messages (99.54%) received on the message queue corresponded to `stat` requests from clients for files whose information was not stored in the hash table. More specifically, this common case results in the following sequence of events in the daemon:

1. After discovering that the message corresponds to a `stat` request, the daemon dispatches the `process_stat` function to handle the request.
2. `process_stat` searches the hash table for an entry corresponding to the file on which the `stat` request was made.
3. In a vast majority of the cases, the file is not found, and the function simply set the file's `UID` and `GID` to 0 and returns the modified file `stat` structure to the client.

While it would have been short-sighted to entirely base our own design of FAKEROOT^P on the profiling of this single build, we expect different builds to use similar access patterns, and our profiling did inform our design to some extent.

3. Design

Two elements of FAKEROOT's original design stand out as clear performance limitations in a multi-core environment. First, the single two-way message queue protected by a single, global semaphore and used by all clients during a build task clearly hinders scalability. The daemon is a single process and can only handle one message at a time. Therefore when using fakeroot, the only speedup that can be obtained from using a distribed `make` is the parallel execution of operations that don't involve system calls. However, `make` involves many file accesses and other types of system calls.

Secondly, even if the daemon were distributed, either as multiple processes or as a single multi-threaded process, the hash table is a shared resource whose consistency must be maintained if concurrent accesses are used. While in the original design, the semaphore serializes both message exchanges and hash table accesses, the hash table exists in the daemon's space, and not all messages result in hash table accesses. Therefore, the semaphore is too coarse a granularity of protection for hash table accesses. It preferable for the daemon, instead of the client-side library, to control access to the table.

The "common case" discovered in the profiling (Section 2.2.2) allowed us to make an important design choice. Indeed, the fact that 99.5% of messages do not result in hash table writes, and instead lead to a well-known modification of the data structure holding the file information says that (a) while there is a hash table lookup performed in this case, there is no need to protect the hash table from data corruption due to concurrent writes and (b) if the hash table lookup can be done by the client-side FAKEROOT library, there is no need for a client-daemon message exchange. Indeed, if the client-side library could somehow look up the file information in the hash table and perform the modification when the entry does not exit, a significant performance benefit could be expected, since it would obviate the need for sending 99.5% of messages.

Our design (called FAKEROOT^P) is illustrated in Figure 3. It is the result and combination of two separate approaches, which we call FAKEROOT-SB and FAKEROOT-MT. To address the bottleneck imposed by the single daemon process and single message queue, we use a multi-threaded daemon, with a two-way message queue on each thread. This multi-threaded daemon approach is what we refer to as FAKEROOT-MT. Because the client-side library needs to know which message queue to use, and in order to balance the numbers of messages processed by the queues, the number of threads used by the daemon is equal to the number of cores available on the host machine, with each daemon thread pinned to a single core for its lifetime. Immediately before sending a message, the client-side library linked into the client process checks its CPU number, and sends its message to the corresponding daemon thread.

This way, the complex scheduling of client processes is left to the operating system, but for any core that a client process happens to be on when it is sending a message, there is a corresponding message queue and daemon-side thread to handle the message. We also tried pinning client processes to their cores for the entire duration of their existence. The advantage of this approach would be a NUMA-aware design: if we were to use a distributed hash table, with a portion of the table on each of the threads, pinning client processes for their entire lifetime would allow their corresponding daemon thread to access nearby data most of the time. However, we found that this results in a tremendous slowdown in execution, telling us that the distibuted hash table is not a good approach.

We removed the single semaphore previously used by all client processes in the original design. However, with the possibility of the OS scheduler swapping client processes on the same CPU in between a request being sent and its

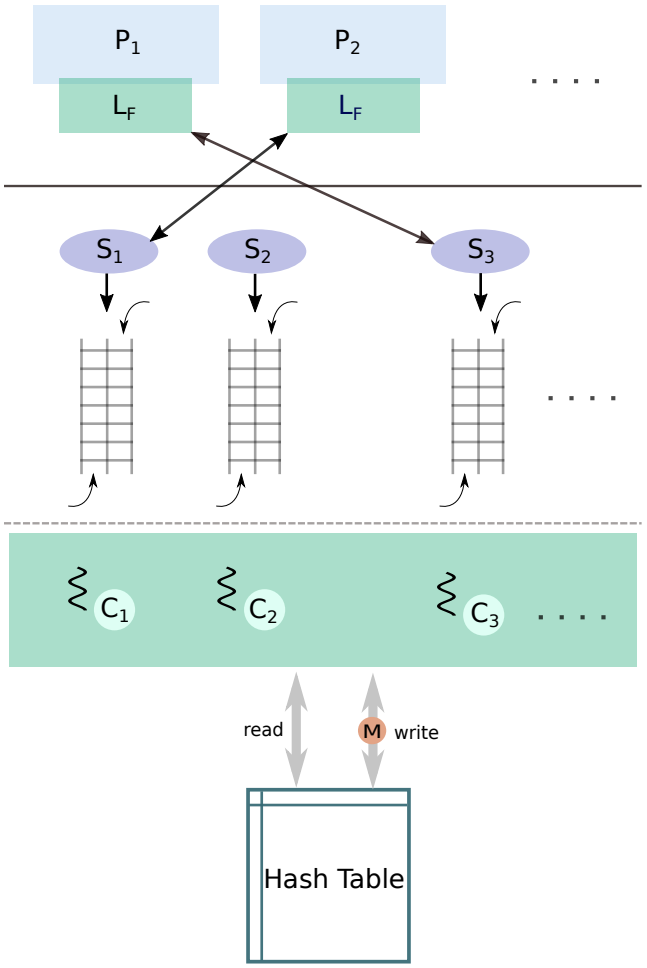


Figure 3: FAKEROOT^P Design. The server is multi-threaded, with one thread pinned to each core in the machine, and one two-way message queue per thread. Client processes send messages to the queue corresponding to the CPU they happen to be running on right before sending the message.

corresponding reply being received, there is the possibility of client processes receiving the wrong messages. To prevent this, FAKEROOT^P uses a per-client semaphore that prevents client processes on any CPU from sending messages until previous clients on the same CPU have received their reponse. Of course, this only works if the client waits for a reply from the daemon thread to which it sent a request. Therefore, clients are temporarily pinned to their CPU immediately before sending the request, and unpinned after the corresponding reply.

While the multi-threaded daemon addresses the single message queue problem, there remains the issue of controlling access to the shared resource, namely the hash table. As stated previously, we found that there would be no advantage in using a distributed hash table. FAKEROOT^P still uses a single daemon-side hash table accessed by all of the daemon threads. However, whereas in FAKEROOT, a single client-side semaphore serializes both message queue usage and hash table access, FAKEROOT^P lets the daemon control access to the table. This is advantageous because the hash table exists in the daemon's address space, and the daemon can better control access to it. However, with the existence of multiple threads, the daemon in FAKEROOT^P has the added responsibility of controlling concurrent accesses.

In our design, `faked` uses a single mutex to control access to the hash table. Reads are performed without locking the mutex. Writes, on the other hand, are controlled by the mutex. Since during a single build, 99.97% of accesses are reads, a significant performance improvement can be expected from this design choice. This loose control of the shared hash table, however, can lead to the well-known read-write problem [3]. In our case, this would correspond to a reader looking up an entry in the hash table and finding that it is present, and a writer modifying or deleting the entry's data before the reader accesses it. Due the limited number of writes compared to reads, and to the file access pattern by the kernel building task, we found that this is not a problem in our test case, and that the kernel always builds without errors when using our design. However, in **??**, we discuss a tighter consistency model that we can use to prevent false reads in a workload that has a higher proportion of writes.

To optimize FAKEROOT^P for the "common case," we expose the hash table to the client-side library using a bitmap. This way, clients that are about to make the common `stat` request to the server can check before sending a message for the presence of an entry in the hash table. If no such entry exists, the clients can perform the well-known modifications (setting `UID` and `GID` to 0) to the file information structure without ever sending a message. This optimization is what we refer to as FAKEROOT-SB.

4. Evaluation

We evaluate FAKEROOT^P by addressing the following questions:

- Does FAKEROOT^P improve the scalability?
- How do FAKEROOT-MT and FAKEROOT-SB individually contribute to the performance improvement?
- Is the design of FAKEROOT^P generic across other multi-core machines?

Experimental setup. The 120-core experiments were run on a 120-core Intel 8870v2 machine with 768 GB of memory. It has 10 sockets, each with 15 cores. The 16-core experiments were run on a 16-core Intel E5-2630 machine. Both machines ran Linux kernel 4.2, and the benchmark builds the Linux kernel version 3.18.

4.1 Performance Analysis

Figure 4 illustrates the performance of the final system, FAKEROOT^P, which is a combination of the two approaches described in §3, FAKEROOT-MT and FAKEROOT-SB. The figure also shows the performance of `make` with the original FAKEROOT implementation, and the performance of `make` on its own.

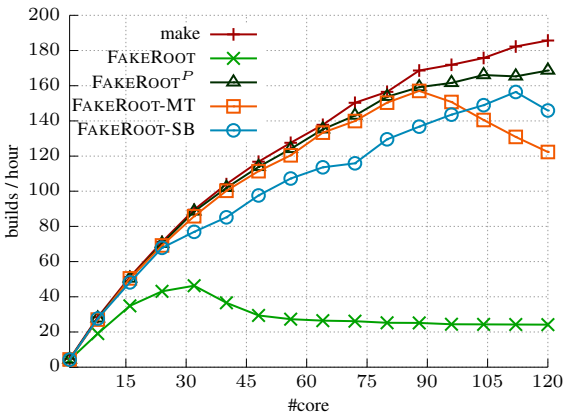


Figure 4: The performance impact of each optimization for the Linux kernel compile. FAKEROOT-MT and FAKEROOT-SB are the two optimizations of FAKEROOT^P.

We can see that on their own, each of FAKEROOT-MT and FAKEROOT-SB offer a signicant improvement to the native design. With its multi-threaded daemon, FAKEROOT-MT allows messages to be simultaneously sent to and processed by as many message queues and daemon threads as there are cores used in the experiment. This offers a key improvement in scalability, compared to the single message queue design that effectively serialized all messages. In FAKEROOT-MT, the functions that handle messages received can execute concurrently on the various threads. The only part of the daemon's operation that is serialized is the writes, which are protected by a daemon-side mutex that is common to all threads. However, out of 1,806,287 hash table accesses, only 483 are also writes. This explains the very good scalability of FAKEROOT-MT. We do not yet understand the decreasing performance of FAKEROOT-MT with more than 88 cores, but we are exploring the hash table access patterns to understand its cause.

On its own, FAKEROOT-SB also performs very well. It allows the client-side library to intercept and process 99.5% of requests on its own (without communicating with the daemon). However, we have found that even with this optimization, around 45,000 messages are received by the daemon. Because FAKEROOT-SB does not use the multi-threaded daemon, all of these messages share the same message queue, and are effectively serialized. This explains the limited scalability we observe in FAKEROOT-SB compared to FAKEROOT-MT on most of its range.

At any given core number, FAKEROOT^P performs better than both FAKEROOT-MT and FAKEROOT-SB. In FAKEROOT^P, only 45,000 requests are sent to the daemon, thanks to the FAKEROOT-SB optimization, and these messages are processed on as many message queues as there are cores on the platform. Because FAKEROOT^P still needs to wrap system calls that are not wrapped by `make` alone, we still expect FAKEROOT^P or any other implementation of the FAKEROOT functionality to perform worst than `make`. However, on 120 cores, the test case of FAKEROOT^P runs only $1.1\times$ slower than `make` on its own, and $7\times$ faster than the native FAKEROOT implementation.

4.2 Machine Independence

The scalability trend of FAKEROOT^P is independent from the machine. Figures 5a and 5b show the performance of the benchmark on a 16-core, Intel E5-2630v3 @ 2.40 GHz machine and a 32-core, Intel E7-4820 @ 2.00 Ghz machine, respectively. The performance on both machines is very close.

References

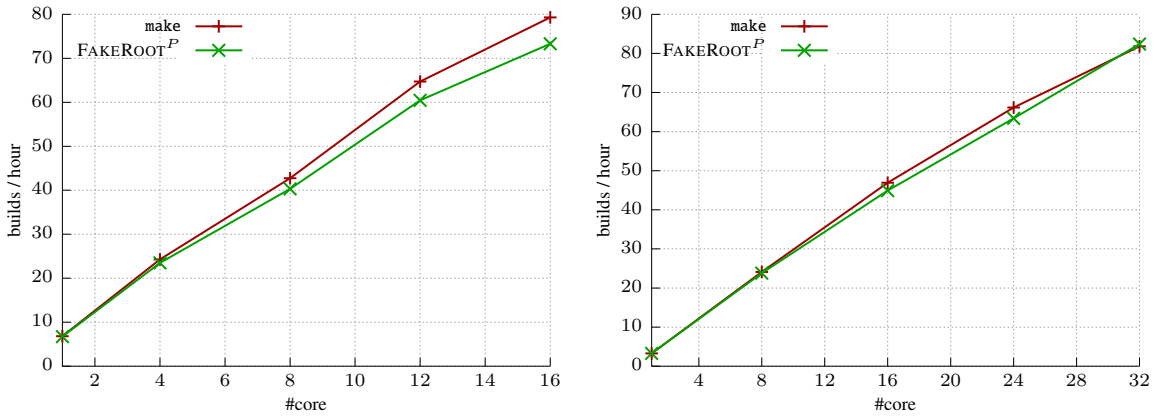
[1] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, M. F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008).

[2] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[3] COURTOIS, P.-J., HEYMANS, F., AND PARNAS, D. L. Concurrent control with âĂIJreadersâĂİ and âĂIJwritersâĂİ. *Communications of the ACM 14*, 10 (1971), 667–668.

[4] GRUENWALD III, C., SIRONI, F., KAASHOEK, M. F., AND ZELDOVICH, N. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the ACM EuroSys Conference* (Bordeaux, France, Apr. 2015).

[5] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), Ieee, pp. 13–24.



(a) 16-core

(b) 32-core

Figure 5: A comparison of the performance of FAKEROOT^P and `make` on different machines.