Enforced Deterministic Program Execution in the Linux Kernel

Chris Cotter
University of Texas

1 Introduction

This thesis describes adding kernel enforced deterministic program execution to Linux. We describe the challenges, design, implementation, and evaluation of a deterministic Linux. In our solution, at a high level, programs enter a *deterministic* mode where the kernel provides a very restricted subset of syscalls designed to enforce determinism.

We are motivated by the difficulties of parallel programming in the presence of nondeterminism. The multicore revolution has encouraged parallel programs over sequential. The inherent nondeterminism in the conventional threading model poses a threat to the quality and correctness of future applications [11]. Bocchino et al. argue that parallel programs must be programmed with a model that is "deterministic by default" [6].

Data races and other nondeterministic inputs force programmers to use difficult to reason about synchronization primitives like semaphores and condition variables. Misuse of these primitives can lead to buggy code and deadlock. Even correct use cannot guarantee deterministic execution: conventional synchronization primitives are not predictable [3]. Determinism is so highly sought, because it overcomes the challenges of nondeterminism. According to Bergan et al., determinism provides benefits in four main areas: debugging, fault tolerance, testing, and security [5].

The research presented in this thesis is based on Determinator [3], a deterministic operating system. We adapt Aviram et al.'s operating system design to make Linux deterministic. The end result is that we are able to write user programs that are indeed deterministic by default: user programs may not execute nondeterministically, even by deliberate design. We choose to adapt Determinator, since it requires no special hardware or specialized programming languages; instead, Determinator enforces determinism through a microkernel approach syscall interface. We are able to write programs in general purpose languages and run them on non-specialized hardware.

This thesis makes the following contributions:

- A presentation of a deterministic Linux kernel heavily based on that of the Determinator kernel. This is the first known adaptation of Determinator's kernel design in a *real* operating system.
- A deterministic high level user library for use by application programmers. This library is motivated by Determinator's user library and the usefulness developing programs using an API similar to that of the standard C library.
- An improvement of Determinator's in-memory file system. Our file system is modeled off of the BSD Fast File System [12], and it provides persistence.
- We evaluate the performance of deterministic Linux against traditional nondeterministic parallel Linux. We also demonstrate a case study of the benefits of determinism.

Determinator contributed a novel programming model building off of existing ideas like transactional memory [9] and distributed shared memory (DSM) [1]. Written from scratch in an academic setting, Determinator has limited uptake in the wider computing community. Linux is a widely deployed open source operating system with a mature, advanced feature set and countless programming libraries and applications; this makes it a very attractive target for providing determinism. If we are lucky, we might be able to influence how future parallel applications are written.

Evaluations show that our deterministic Linux has performance comparable to that of nondeterministic Linux using pthreads. Embarrassingly applications written using deterministic abstractions have little overhead. We are optimistic that the benefits of deterministic execution over the serious drawbacks of programming with nondeterminism, coupled with additional kernel features and improvements in library design will make deterministic Linux a popular choice in the future of parallel application development.

2 Background

Computer architects are under increasing pressure to produce multicore processors: the doubling of transistors every 18 months, or Moore's law, no longer implies increasing processor clock speeds. Rather, processors increasingly have more cores, with clock rates holding steady [4]. The multicore revolution is shifting program development from a sequential to parallel paradigm.

Many parallel applications are written in the conventional nondeterministic threading model. Features like shared memory state are attractive, since they facilitate easy application development. Unfortunately, writing cor-

rect parallel programs in this environment is hard, particularly due to nondeterminism.

Anyone who has written a multithreaded application using the conventional threading model is familiar with the drawbacks. Data races, both low and especially high level, introduce bugs that often must be solved with difficult to reason about lock abstractions. Nondeterministic, or even arbitrarily deterministic, scheduling of threads again introduces bugs and complicates reasoning about the program output.

Lee points out problems with the conventional threading model, and he would like to do away with it [11]. As more applications become parallel, we do not want a degradation in code quality. What we would like, then, is a new programming model that limits or eliminates the possibility of buggy writing programs. In search of a solution, Bocchino et al. make the case that all parallel programs should be written using a programming model that is "deterministic by default" [6].

In general, a program's output is a function of its inputs, both explicit and implicit. We say an input is explicit if it is semantically relevant to the program's output. We consider an input that is irrelevant to the program's intended goal, but nonetheless influences program output, to be implicit. In most cases, implicit inputs are random, arbitrary, and uncontrollable; timing dependencies, quantum size, and cache size are examples [5].

Since implicit inputs by definition are irrelevant, we would like programs to be functions of only explicit inputs. We call such programs *deterministic*. Running a deterministic program on the same input will always generate the same output, regardless of implicit inputs. Programs whose output depends on implicit inputs are considered nondeterministic.

Olszewski et al. characterize determinism into two categories: strong and weak [14]. *Strong determinism* guarantees a deterministic order of memory operations for a fixed program input and thus always provides deterministic execution. *Weak determinism* only guarantees a deterministic order of lock acquisitions. Weak determinism provides deterministic execution only when a program is written free of data races. In the presence of racy data unprotected by a lock, weakly deterministic systems fail to execute deterministically.

2.1 Motivation

Determinism provides many benefits to application developers [5, 6, 14]. Bergan et al. suggest there are four main benefits in the following areas: debugging, fault tolerance, testing, and security.

Debugging Debugging multithreaded programs can be difficult, because often bugs are not easily reproducible, and tools such as `gdb` are not always useful for tracking down heisenbugs [13]. Finding a bug's root cause becomes easier when a program's execution can be replayed over and over. Deterministic execution naturally provides replay debugging as a benefit.

Fault tolerance Fault tolerance through replication obviously relies on the assumption that running a program multiple times will always return the same output. Determinism again provides this benefit naturally.

Testing The difficulties in testing multithreaded applications are compounded by racy nondeterministic scheduling. Developers and automated test systems must consider the exponential blow up of possible scheduling sequences. Determinism helps alleviate this problem by guaranteeing a one-to-one correspondence between input and output. For each input, there is exactly one possible logical scheduling sequence of threads. This observation eliminates the need to consider what scheduling interactions can occur, and ultimately helps developers design test strategies [5].

In addition to explicit inputs, schedule sequences are affected by implicit inputs like quantum size, data races, and memory ownership granularity. Deterministic systems like Kendo [14] and DMP [7] are especially dependent on quantum size [5]. In other words, these systems provide deterministic execution, but the scheduling sequence is still a function of implicit inputs.

On the other hand, Determinator's programming model provides something stronger than determinism: *predictability*. Whereas systems like DMP seek to "control, detect, or reproduce" data races, Determinator's programming model is *naturally deterministic*: it avoids "introducing data races or other nondeterministic behavior in the first place" [3]. With predictability, programmers can reason about every relevant aspect of a program, including scheduling. Determinism, and the stronger property of predictability, thus make testing applications easier.

Security Processes sharing a CPU or other hardware should be conscious about leaking sensitive data. A malicious thread can exploit covert timing channels to extract sensitive data from other, perhaps privileged, threads [2]. Determinism eliminates covert timing channels, since a program is purely a function of explicit inputs and cannot possible rely subtly on the timings of hardware operations.

To further motivate determinism, we consider systems that solve that above problems. So called "point solutions" solve in problems in single areas at once. Record and replay debuggers, like Leblanc et al.'s `Instant Replay` system, aid in debugging parallel programs by logging scheduling sequences and other relevant interactions in order to replay an execution sequence exactly. However, these debuggers are costly in terms of storage and perfor-

mance. In general, these "point solutions...do not compose well with one another" [5]. On the other hand, determinism provides benefits in all four areas at once with a single mechanism.

2.2 Determinator

Aviram et al. set out to provide

> a parallel environment that: (a) is "deterministic by default," except when we inject nondeterminism explicitly via external inputs; (b) introduces no data races, either at the memory access level or at higher semantic levels; (c) can enforce determinism on arbitrary, compromised or malicious code for security reasons; and (d) is efcient enough to use for "normal-case" execution of deployed code, not just for instrumentation during development. [3]

To this end, they presented Determinator, a novel OS written from the ground up. For most of the remainder of this section, we will recapitulate Aviram et al.'s work and contributions; first we will discuss aspects that influenced Determinator's design. Then, we will look at the actual kernel design itself and the accompanying user library.

The primary cause of nondeterminism is data races introduced by timing dependencies. Each source of implicit nondeterminism must be accounted for in designing a deterministic programming model. We discuss them here, and describe how Determinator handles them.

Explicit Nondeterminism Often, programs rely on nondeterministic inputs such as network packets, user input, or clock time. These inputs are essential to a program being useful; therefore, a deterministic programming model must incorporate these inputs while still enforcing determinism. Determinator addresses these "semantically relevant" inputs by turning them into explicit I/O [3]. Applications have complete control over these input sources and can even log the inputs for reply debugging.

Shared program state Traditional multithread programming models provide shared state: threads using the pthreads API share the entire memory state, and Linux's file system is shared by all running programs. Data races and incorrect synchronization lead to nondeterministic execution traces and often introduce unpredictable bugs.

Determinator eliminates data races caused by shared program state by eliminating shared state altogether. Threads operate using a private workspace model and synchronize program state at explicitly defined program points. When two or more threads begin executing, each has identical private virtual memory images. Writes to memory are not visible to other threads until the threads synchronize.

Nondeterministic scheduling abstractions Traditional multithreaded synchronization abstractions are often neither deterministic nor predictable. Random hardware races determine the next thread to acquire a mutex lock, and as mentioned before this has debugging and testing implications. Even though we can record lock acquisition sequences to replay program execution or use some arbitrary device to choose a deterministic sequence, the order of acquisition is not predictable. Determinator restricts itself to only allow naturally deterministic and predictable synchronization abstractions, such as fork-join.

Globally shared namespaces Operating systems introduce nondeterminism by using namespaces that are shared by the entire system. Process IDs returned by `fork()` and files created by `mktemp()` are examples. Since these identifiers are nondeterministic, and only the resource itself, not the identifier, is semantically relevant to the application, Determinator disallows the system from choosing resource identifiers from globally shared namespaces. Instead, applications themselves choose identifiers deterministically.

2.3 The Determinator Kernel

Determinator organizes processes in a nested process model [8]. Processes cannot outlive their parents and can only communicate with their parents and children. In line with the earlier discussion of nondeterminism, the kernel "provides no file systems, writable shared memory, or other abstractions that imply globally shared state". Only "the distinguished root [process] has direct access to nondeterministic inputs" [3]. It is through this root process that explicitly nondeterministic inputs can be controlled. All other processes must communicate directly or indirectly with the root process to access I/O devices.

Kernel Interface Processes communicate with the kernel via three syscalls, `Put`, `Get`, `Ret`. Table 1 and Table 2, reproduced from [3], summarize how the syscalls work and the options available to `Put` and `Get`.

Determinator enforces a deterministic schedule by requiring programs to explicitly define synchronization points; this mechanism is described here. Since the kernel does not manage any global namespaces, user programs specify a child process ID parameter to `Put` and `Get`. The first `Put` syscall with a previously unused child ID creates a new child process. `Put` calls can start a child's execution, and the child will continue to execute until it invokes `Ret` or generates a processor exception (e.g. divide by zero). `Put` and `Get` calls block until the child process stops.

Process state is composed of register state and its entire virtual memory. The Regs option copies register state

Call	Interacts with	Description
Put	Child	Copy register state and/or virtual memory range into child, and optionally start child executing.
Get	Child	Copy register state, virtual memory range, and/or changes since the last snapshot out of a child.
Ret	Parent	Stop and wait for parent to issue a Get or Put. Processor traps also cause implicit Ret.

Table 1—System calls comprising Determinators kernel API.

Put	Get	Option	Description
X	X	Regs	PUT/GET childs register state.
X	X	Copy	Copy memory to/from child.
X	X	Zero	Zero-fill virtual memory range.
X		Snap	Snapshot childs virtual memory.
X		Start	Start child space executing.
	X	Merge	Merge childs changes into parent.
X	X	Perm	Set memory access permissions.
X	X	Tree	Copy (grand)child subtree.

Table 2—Options/arguments to the Put and Get calls.

from a parent to child or vice versa. Determinator provides more sophisticated virtual memory options: the Zero option zeros a virtual memory region in a child; the Copy option copies virtual memory regions between a parent and its child; the snapshot-merge mechanism is similar to Copy, but more complicated.

Snap copies the calling process's entire virtual memory state into the specified child. Invoking `Get` with Merge copies bytes from the child that have changed since the previous Snap invocation into the parent. Bytes that changed in the parent but not the child are not copied. Bytes that changed in both the parent and child generate a *merge conflict*. The kernel implements Merge efficiently by examining page table entires.

Aviram et al. conclude their discussion of Determinator's kernel by mentioning the three syscall "primitives reduce to blocking, one-to-one message channels, making the space hierarchy a deterministic Kahn network" [10].

2.4 Deterministic Linux

With Determinator presented, we can now motivate a deterministic Linux. Determinator was written from scratch in an academic environment with determinism as the main OS design goal. In some sense, Determinator is not a *real* operating system. The potential uptake outside the academic world is minimal. On the other hand, Linux is a mature and widely deployed nondeterministic operating system. Linux is installed on millions of systems including desktop computers, embedded systems, and mobile devices. In other words, Linux is a real operating system used in the real world, and by adding determinism to Linux, we are able to take advantage of its widespread use and adoption; the potential userbase for a deterministic Linux is much greater than that of Determinator.

Adding an inherently deterministic and predictable programming model to Linux is a huge step in attempting to influencing how developers write the parallel applications of the future.

3 Overview

We begin the discussion of adding determinism to Linux by discussing overall design goals of the project. Next we look at the challenges of making nondeterministic Linux deterministic. For the rest of this thesis, we distinguish between *legacy* Linux (an unmodified Linux kernel) and *deterministic* Linux.

3.1 Design Goals

We wish to make 64-bit Linux deterministic, and in doing we will present an interface similar to that of Determinator. We also would like to run legacy Linux applications alongside deterministic applications, for this is one of the motivating factors applying Determinator's design to Linux.

Whereas Determinator forces all but one process to operate in *deterministic* mode, we wish to be able to run legacy Linux programs without modification; however, we won't make any attempt to force legacy programs to run deterministically, so legacy applications will run in legacy nondeterministic mode. In order to take advantage of determinism in Linux, legacy programs must be rewritten using the new operating system abstractions.

We would also like to write a user level C library with familiar abstractions such as fork-join and an in-memory file system based on those of Determinator. In some cases, we improve upon Determinator's user library, especially the limitations of the in-memory file system [2, 3].

We do not wish to apply all of Determinator's features to a deterministic Linux. Determinator supports deterministic compute clusters by extending its nested process model to a cluster of nodes. Determinator also supports a "tree" copy operation for `Put` and `Get`. Lastly, Determinator allows threads to place an instruction limit on children threads. We have no intention of supporting these features, but we note this limitation does not detract from the goal of a deterministic Linux.

Snce the primary goal is to make Linux deterministic, we may decide to limit or ignore features of the Linux kernel internals. For example, Linux supports huge pages of memory alongside "normal" 4-KB pages. Since this is

an internal optimization that is hidden from user applications, for reasons of implementation complexity, we may not allow deterministic applications to take advantage of certain kernel features.

3.2 Challenges

We have already discussed the four sources of nondeterminism identified by Aviram et al; these obvservations are general enough that they apply in making Linux deterministic. In applying Determinator's design to Linux, however, we must address the following issues.

Inherent nondeterminisim in Linux In order to run legacy Linux applications, we cannot enforce that all but a single root process operate in deterministic mode; this design aspect must be reexamined to allow legacy and deterministic applications to run side-by-side. Furthermore, Linux's process model allows reparenting and for children to outlive parents, directly opposed to Determinator's nested process model.

Linux's threading model is inherently nondeterministic and provides many additional sources of nondeterminism than those already addressed by the above discusion: Linux supports signals and System V IPC. To address some sources of nondeterminism, Determinator's designers simply did not add support for these features, since Determinator was written from scratch. On the other hand, Linux already provides extensive support for nondeterministic features (e.g. the `gettimeofday()` syscall).

Memory subsystem Linux supports a wide range of virtual memory features, including memory mapped files, huge pages, and swapping to disk; all of these features are layered on top of an abstraction for supporting memory management units of a wide range of processor types. Compared to Determinator, Linux's memory subsystem uses much more complicated abstractions to support these features. Understanding and overcoming this complicated system is essential to implementing determinism in Linux.

Standard C Library Many applications written in C on Linux use the standard C library. This library in large part functions as a wrapper around legacy Linux syscalls, and thus is highly nondeterministic. Whereas some functions, such as `strlen()` might not use nondeterminstic syscalls, many other functions do use nondeterministic syscalls (e.g. `printf()`). Thus, deterministic programs may be forced to use a completely different library. Moreover, the libraries in Linux are often linked dynamically with shared libraries, but Determinator does not provide any native kernel support for dynamic linking. We may lose the ability to dynamically link shared libraries.

3.3 High level approach

To address concerns about Linux's more flexible process model, we present a *hybrid process model*. A Linux process invokes a syscall to become a *root* process, akin to Determinator's single root process. This root process has full access to the legacy Linux kernel API, with some minor restrictions noted below. Root processes then create *deterministic* children. Within the process hierarchy, processes abide by Determinator's nesting rules (e.g. children cannot outlive parents). A deterministic process's death automatically triggers reaping that process's subtree.

Legacy Linux applications run alongside deterministic applications with absolutely no kernel restrictions. In some sense, each deterministic application resembles an entire Determinator "virtual machine" of sorts.

We then add three new syscalls: `dput()`, `dget()`, and `dret()`; we restrict deterministic processes to only using these three syscalls. These syscalls function exactly as their Determinator counterparts, excepting cluster support, the copy (grand)child subtree option, and instruction count limits. By restricting deterministic processes to these three syscalls, we can nearly remove all sources of nondeterminisim; we only have to modifiy the kernel to ignore all signals sent to deterministic processes, and thus we have effectively blocked all sources of nondeterminism.

Once this kernel work is done, we begin work on a C user library. We won't use the standard C library with deterministic processes, since many library calls invoke disallowed legacy syscalls. The common use case of multithreaded applications is to `fork()` a child with a copy of the parent's virtual memory image, thus giving the child access to the same library API as the parent. This is undesirable for our system, since this would automatically let deterministic children use the standard C library. To avoid this and namespace problems (we want deterministic processes to use familiar function names like `printf()`), we require root and deterministic processes must use our new deterministic library, even though many functions must be rewritten (e.g. `sprintf()`, `strlen()`).

To increase the usefulness of the system, we provide an in-memory file system just as Determinator does. Whereas Determinator's file system used fixed file size [2], our file system design is similar to that of the BSD Fast File System [–cite–]. The file system is divided into 4096-byte *blocks*. The first block is a *superblock* containing metadata about the file system. A region of fixed size following the superblock is reserved for *inodes* and a bitmap for managing free blocks. The rest of the blocks are data blocks. In addition to direct block pointers, inodes support a singly and doubly indirect block of pointers. Directories are files containing a list of files within the directory.

We also note that since root processes have access to the system file system, our user library can save the in-memory file system to permanent storage if so desired. Thus, our file system improves upon that of Determinator by supporting hard linking, supporting larger file sizes, and being more flexible in managing underlying resources (inodes, blocks).

4 Kernel Implementation

With a high level design in mind, we now discuss the final version of the implementation. We started by forking Linux from a `git` repository, and worked on x86_64 Linux 3.0. We developed and tested incrementally on an 8-core machine with 8 gigabyets of RAM running Arch Linux.

4.1 Process synchronization

The first step was adding the three new syscalls: `dput()`, `dget()`, and `dret()`; slowly we added the various features to the syscall implementations. Our initial focus was adding process creation functionality; `dput()` relies heavily on existing `fork()` kernel code to create new processes. We also used existing `do_exit()` code to delete processes, and enforce that a deterministic processes death implies the death of its process subtree. We block all external signals generated by user applications, but allow signals generated by the kernel itself; it is through this mechanism that exceptions, such as divide-by-zero faults that generate a `SIGFPE`, cause an implicit `dret()`.

We augment Linux's `task_struct` process structure with a *deterministic PID* and synchronization primitives. `dput()` and `dget()` use these synchronization primitives to correctly synchronize deterministic process communication within the hybrid process model.

In Determinator, *all* processes that issue a `dput()` or `dget()` block until the child in question issues a `dret()`. This is a limitation for applications that benefit from concurrency, such as running `make -j2` [3]. As noted by Aviram et al., Determinator might miss opportunities to start a parallel job, because a deterministic `make` in Determinator schedules itself and might be blocked waiting for a thread to finish when other children are runnable. In our implementation we allow the root process to perform a special non-blocking `dget()` to determine whether or not a child is still running. This allows more optimal scheduling, since the root can find a runnable thread and give it work. Determinism is achieved by recording the scheduling sequence for replay, if desired. This feature is desirable for inherently nondeterministic applications like web servers that may wish to exploit as much concurrency as possible.

Root processes can use `dput()` to specify a set of signals to block while in a blocking `dput()` or `dget()`. Blocking versions of these syscalls ignore signals specified in the block set sent to the root until the child issues a `dret()`. This can be useful when a console operation wishes to kill an application with a `SIGINT`, but does not want other signals to interrupt the root process.

The last feature relating strictly to process organization is register state copying. The initial `dput()` call that creates a child automatically copies register state, since we effectively delegate work to `fork()`. Subsequent calls to `dput()` and `dget()` pass general purpose register state structure pointer to set or get a child's register set.

4.2 Memory operations

5 Conclusion

Remind reader about the contributions of the proposed work, and what the proposed work will actually look like.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[2] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *CCSW*, 2010.

[3] A. Aviram, S. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2012.

[4] C. BANDWIDTH and T. USABLE. Multicore cpus for the masses. 2005.

[5] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *WoDet*, 2011.

[6] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX workshop on hot topics in parallelism (HotPar)*, 2009.

[7] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[8] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *OSDI*, 1996.

[9] M. Herlihy and J. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[10] G. Kahn. The semantics of a simple language for parallel programming. 1974.

[11] E. Lee et al. The problem with threads. *Computer*, 39(5):33–42, 2006.

[12] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[13] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[14] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ACM Sigplan Notices*, volume 44, pages 97–108. ACM, 2009.