View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Efficient System-Enforced Deterministic Parallelism

Amittai Aviram, Shu-Chun Weng, Sen Hu, Bryan Ford
Yale University

Abstract

Deterministic execution offers many benefits for debugging, fault tolerance, and security. Current methods of executing *parallel* programs deterministically, however, often incur high costs, allow misbehaved software to defeat repeatability, and transform time-dependent races into input- or path-dependent races without eliminating them. We introduce a new parallel programming model addressing these issues, and use Determinator, a proof-of-concept OS, to demonstrate the model's practicality. Determinator's microkernel API provides only "shared-nothing" address spaces and deterministic interprocess communication primitives to make execution of all unprivileged code—well-behaved or not—precisely repeatable. Atop this microkernel, Determinator's user-level runtime adapts optimistic replication techniques to offer a *private workspace* model for both thread-level and process-level parallel programing. This model avoids the introduction of read/write data races, and converts write/write races into reliably-detected conflicts. Coarse-grained parallel benchmarks perform and scale comparably to nondeterministic systems, on both multicore PCs and across nodes in a distributed cluster.

1 Introduction

We often wish to run software *deterministically*, so that from a given input it always produces the same output. Determinism is the foundation of replay debugging [37, 39, 46, 56], fault tolerance [15, 18, 50], and accountability mechanisms [30, 31]. Methods of intrusion analysis [22, 34] and timing channel control [4] further assume the system can *enforce* determinism even on malicious code designed to evade analysis. Executing parallel software deterministically is challenging, however, because threads sharing an address space—or processes sharing resources such as file systems—are prone to nondeterministic, timing-dependent *races* [3, 40, 42, 43].

User-space techniques for parallel deterministic execution [8, 10, 20, 21, 44] show promise but have limitations. First, by relying on a *deterministic scheduler* residing in the application process, they permit buggy or malicious applications to compromise determinism by interfering with the scheduler. Second, deterministic schedulers emulate conventional APIs by synthesizing a repeatable—but arbitrary—schedule of inter-thread interactions, often using an instruction counter as an artificial time metric. Data races remain, therefore, but their manifestation depends subtly on inputs and code path lengths instead of on "real" time. Third, the user-level instrumentation required to isolate and schedule threads' memory accesses can incur considerable overhead, even on coarse-grained code that synchronizes rarely.

To meet the software development, debugging, and security challenges that ubiquitous parallelism presents, it may be insufficient to shoehorn the standard nondeterministic programming model into a synthetic execution schedule. Instead we propose to rethink the basic model itself. We would like a parallel environment that: (a) is "deterministic by default" [12, 40], except when we inject nondeterminism explicitly via external inputs; (b) introduces no data races, either at the memory access level [25, 43] or at higher semantic levels [3]; (c) can enforce determinism on arbitrary, compromised or malicious code for security reasons; and (d) is efficient enough to use for "normal-case" execution of deployed code, not just for instrumentation during development.

As a step toward such a model, we present *Determinator*, a proof-of-concept OS designed around the above goals. Due to its OS-level approach, Determinator supports existing languages, can enforce deterministic execution not only on a single process but on groups of interacting processes, and can prevent malicious user-level code from subverting the kernel's guarantee of determinism. In order to explore the design space freely, Determinator takes a "clean-slate" approach, making few compromises for backward compatibility with existing kernels or APIs. Determinator's programming model could be implemented in a legacy kernel for backward compatibility, however, as part of a "deterministic sandbox" for example [9]. Determinator's user-level runtime also provides limited emulation of the Unix process, thread, and file APIs, to simplify application porting.

Determinator's kernel enforces determinism by denying user code direct access to hardware resources whose use can yield nondeterministic behavior, including real-time clocks, cycle counters, and writable shared memory. Determinator constrains user code to run within a hierarchy of single-threaded, process-like *spaces*, each having a private virtual address space. The kernel's low-level API provides only three system calls, with which a space can synchronize and communicate with its immediate parent and children. Potentially useful sources of nondeterminism, such as timers, Determinator encapsulates into I/O devices, which unprivileged spaces can access

only via explicit communication with more privileged spaces. A supervisory space can thus mediate all nondeterministic inputs affecting a subtree of unprivileged spaces, logging true nondeterministic events for future replay or synthesizing artificial events, for example.

Atop this minimal kernel API, Determinator's user-level runtime emulates familiar shared-resource programming abstractions. The runtime employs file replication and versioning [47] to offer applications a logically shared file system accessed via the Unix file API, and adapts distributed shared memory [2, 17] to emulate shared memory for multithreaded applications. Since this emulation is implemented in user space, applications can freely customize it, and runtime bugs cannot compromise the kernel's guarantee of determinism.

Rather than strictly emulating a conventional, nondeterministic API and consistency model like deterministic schedulers do [8–10, 21, 44], Determinator explores a novel *private workspace* model. In this model, each thread keeps a private virtual replica of all shared memory and file system state; normal reads and writes access and modify this working copy. Threads reconcile their changes only at program-defined synchronization points, much as developers use version control systems. This model eliminates read/write data races, because reads see only *causally prior* writes in the explicit synchronization graph, and write/write races become *conflicts*, which the runtime reliably detects and reports independently of any (real or synthetic) execution schedule.

Experiments with common parallel benchmarks suggest that Determinator can run coarse-grained parallel applications deterministically with both performance and scalability comparable to nondeterministic environments. Determinism incurs a high cost on fine-grained parallel applications, however, due to Determinator's use of virtual memory to isolate threads. For "embarrassingly parallel" applications requiring little inter-thread communication, Determinator can distribute the computation across nodes in a cluster mostly transparently to the application, maintaining usable performance and scalability. As a proof-of-concept, however, the current prototype has many limitations, such as a restrictive space hierarchy, limited file system size, no persistent storage, and inefficient cross-node communication.

This paper makes four main contributions. First, we present the first OS designed from the ground up to offer system-enforced deterministic execution, for both multithreaded processes and groups of interacting processes. Second, we introduce a *private workspace* model for deterministic parallel programming, which eliminates read/write data races and converts schedule-dependent write/write races into reliably-detected, schedule-independent conflicts. Third, we use this model to emulate shared memory and file system abstractions in Determinator's user-space runtime. Fourth, we demonstrate experimentally that this model is practical and efficient enough for "normal-case" use, at least for coarse-grained parallel applications.

Section 2 outlines the deterministic programming model we seek to create. Section 3 then describes the Determinator kernel's design and API, and Section 4 details its user-space application runtime. Section 5 examines our prototype implementation, and Section 6 evaluates it informally and experimentally. Finally, Section 7 outlines related work, and Section 8 concludes.

2 A Deterministic Programming Model

Determinator's basic goal is to offer a programming model that is *naturally* and *pervasively deterministic*. To be *naturally deterministic*, the model's basic abstractions should avoid introducing data races or other nondeterministic behavior in the first place, and not merely provide ways to control, detect, or reproduce races. To be *pervasively deterministic*, the model should behave deterministically at all levels of abstraction: e.g., for shared memory access, inter-thread synchronization, file system access, inter-process communication, external device or network access, and thread/process scheduling.

Intermediate design points are possible and may yield useful tradeoffs. Enforcing determinism only on synchronization and not on low-level memory access might improve efficiency, for example, as in Kendo [44]. For now, however, we explore whether a "purist" approach to pervasive determinism is feasible and practical.

To achieve this goal, we must address timing dependencies in at least four aspects of current systems: in way applications obtain semantically-relevant nondeterministic inputs they require for operation; in shared state such as memory and file systems; in the synchronization APIs threads and processes use to coordinate; and in the namespaces with which applications use and manage system resources. We make no claim that these are the only areas in which current operating systems introduce nondeterminism, but they are the aspects we found essential to address in order to build a working, pervasively deterministic OS. We discuss each area in turn.

2.1 Explicit Nondeterministic Inputs

Many applications use nondeterministic *inputs*, such as incoming messages for a web server, timers for an interactive or real-time application, and random numbers for a cryptographic algorithm. We seek not to eliminate application-relevant nondeterministic inputs, but to make such inputs explicit and controllable.

Mechanisms for parallel debugging [39, 46, 56], fault tolerance [15, 18, 50], accountability [30, 31], and intrusion analysis [22, 34] all rely on the ability to replay a computation instruction-for-instruction, in order to repli-

cate, verify, or analyze a program's execution history. Replay can be efficient when only I/O need be logged, as for a uniprocessor virtual machine [22], but becomes much more costly if *internal* sources of nondeterminism due to parallelism must also be replayed [19, 23].

Determinator therefore transforms useful sources of nondeterminism into explicit I/O, which applications may obtain via controllable channels, and eliminates only internal nondeterminism resulting from parallelism. If an application calls `gettimeofday()`, for example, then a supervising process can intercept this I/O to log, replay, or synthesize these explicit time inputs.

2.2 A Race-Free Model for Shared State

Conventional systems give threads direct, concurrent access to many forms of shared state, such as shared memory and file systems, yielding data races and heisenbugs if the threads fail to synchronize properly [25, 40, 43]. While replay debuggers [37,39,46,56] and deterministic schedulers [8,10,20,21,44] make data races reproducible once they manifest, they do not change the inherently race-prone model in which developers write applications.

Determinator replaces the standard concurrent access model with a *private workspace* model, in which data races do not arise in the first place. This model gives each thread a complete, private virtual replica of all logically shared state a thread may access, including shared memory and file system state. A thread's normal reads and writes affect only its private working state, and do not interact directly with other threads. Instead, Determinator accumulates each threads's changes to shared state, then *reconciles* these changes among threads only at program-defined synchronization points. This model is related to and inspired by early parallel Fortran systems [7, 51], replicated file systems [47], transactional memory [33, 52] and operating systems [48], and distributed version control systems [29], but to our knowledge Determinator is the first OS to introduce a model for pervasive thread- and process-level determinism.

If one thread executes the assignment '$x = y$' while another concurrently executes '$y = x$', for example, these assignments race in the conventional model, but are race-free under Determinator and always swap x with y. Each thread's read of x or y always sees the "old" version of that variable, saved in the thread's private workspace at the last explicit synchronization point.

Figure 1 illustrates a more realistic example of a game or simulator, which uses an array of "actors" (players, particles, etc.) to represent some logical "universe," and updates all of the actors in parallel at each time step. To update the actors, the main thread forks a child thread to process each actor, then synchronizes by joining all these child threads. The child thread code to update each actor is shown "inline" within the **main()** function, which

```
struct actor_state actor[nactors];

main()
  initialize all elements of actor[] array
  for (time = 0; ; time++)
      for (i = 0; i < nactors; i++)
          if (thread_fork(i) == IN_CHILD)
            // child thread to process actor[i]
            examine state of nearby actors
            update state of actor[i] accordingly
            thread_exit();
      for (i = 0; i < nactors; i++)
          thread_join(i);
```

Figure 1: C pseudocode for lock-step time simulation, which contains a data race in standard concurrency models but is bug-free under Determinator.

under Unix works only with process-level `fork()`; Determinator offers this convenience for shared memory threads as well, as discussed later in Section 4.4.

In this example, each child thread reads the "prior" state of any or all actors in the array, then updates the state of its assigned actor "in-place," without any explicit copying or additional synchronization. With standard threads this code has a read/write race: each child thread may see an arbitrary mix of "old" and "new" states as it examines other actors in the array. Under Determinator, however, this code is correct and race-free. Each child thread reads only its private working copy of the actors array, which is untouched (except by the child thread itself) since the main thread forked that child. As the main thread rejoins all its child threads, Determinator merges each child's actor array updates back into the main thread's working copy, for use in the next time step.

While read/write races disappear in Determinator's model, traditional write/write races become *conflicts*. If two child threads concurrently write to the same actor array element, for example, Determinator detects this conflict and signals a runtime exception when the main thread attempts to join the second conflicting child. In the conventional model, by contrast, the threads' execution schedules might cause either of the two writes to "win" and silently propagate its likely erroneous value throughout the computation. Running this code under a conventional deterministic scheduler causes the "winner" to be decided based on a synthetic, reproducible time metric (e.g., instruction count) rather than real time, but the race remains and may still manifest or vanish due to slight changes in inputs or instruction path lengths.

2.3 A Race-Free Synchronization API

Conventional threads can still behave nondeterministically even in a correctly locked program with no low-

level data races. Two threads might acquire a lock in any order, for example, leading to high-level data races [3]. This source of nondeterminism is inherent in the lock abstraction: we can record and replay or synthesize a lock acquisition schedule [44], but such a schedule is still arbitrary and effectively unpredictable to the developer.

Fortunately, many other synchronization abstractions are naturally deterministic, such as fork/join, barriers, and futures [32]. Deterministic abstractions have the key property that when threads synchronize, *program logic alone* determines at what points in the threads' execution paths the synchronization occurs, and which threads are involved. In fork/join synchronization, for example, the parent's **thread_join**(t) operation and the child's **thread_exit()** determine the respective synchronization points, and the parent indicates explicitly the thread t to join. Locks fail this test because one thread's **unlock()** passes the lock to an arbitrary successor thread's **lock()**. Queue abstractions such as semaphores and pipes are deterministic if only one thread can access each end of the queue [24, 36], but nondeterministic if several threads can race to insert or remove elements at either end. A related draft elaborates on these considerations [5].

Since the multicore revolution is young and most application code is yet to be parallelized, we may still have a choice of what synchronization abstractions to use. Determinator therefore supports only race-free synchronization primitives natively, although it can emulate nondeterministic primitives via deterministic scheduling for compatibility, as described later in Section 4.5.

2.4 Race-Free System Namespaces

Current operating system APIs often introduce nondeterminism unintentionally by exposing shared namespaces implicitly synchronized by locks. Execution timing affects the pointers returned by `malloc()` or `mmap()` or the file numbers returned by `open()` in multithreaded Unix processes, and the process IDs returned by `fork()` or the file names returned by `mktemp()` in single-threaded processes. Even if only one thread actually uses a given memory block, file, process ID, or temporary file, the assignment of these names from a shared namespace is inherently nondeterministic.

Determinator's API therefore avoids creating shared namespaces with system-chosen names, instead favoring thread-private namespaces with application-chosen names. Application code, not the system, decides where to allocate memory and what process IDs to assign children. This principle ensures that naming a resource reveals no shared state information other than what the application itself provided. Since implicitly shared namespaces often cause multiprocessor contention, designing system APIs to avoid this implicit sharing may be synergistic with recent multicore scalability work [14].

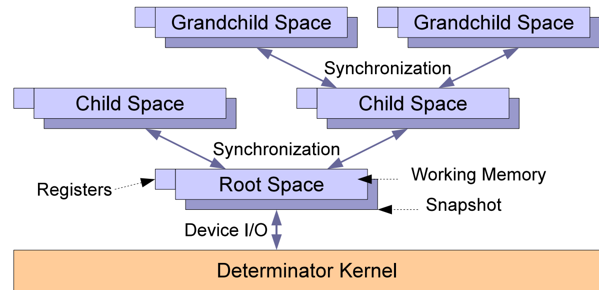

Figure 2: The kernel's hierarchy of *spaces*, each containing private register and virtual memory state.

3 The Determinator Kernel

Having outlined the principles underlying Determinator's programming model, we now describe its kernel design. Normal applications do not use the kernel API directly, but rather the higher-level abstractions provided by the user-level runtime, described in the next section. We make no claim that our kernel design or API is the "right" design for a determinism-enforcing kernel, but merely that it illustrates one way to implement a pervasively deterministic application environment.

3.1 Spaces

Determinator executes application code within an arbitrarily deep hierarchy of *spaces*, illustrated in Figure 2. Each space consists of CPU register state for a single control flow, and private virtual memory containing code and data directly accessible within that space. A Determinator space is analogous to a single-threaded Unix process, with important differences; we use the term "space" to highlight these differences and avoid confusion with the "process" and "thread" abstractions Determinator emulates at user level, described in Section 4.

As in a nested process model [27], a Determinator space cannot outlive its parent, and a space can directly interact *only* with its immediate parent and children via three system calls described below. The kernel provides no file systems, writable shared memory, or other abstractions that imply globally shared state.

Only the distinguished *root space* has direct access to nondeterministic inputs via I/O devices, such as console input or clocks. Other spaces can access I/O devices only indirectly via parent/child interactions, or via I/O privileges delegated by the root space. A parent space can thus control all nondeterministic inputs into any unprivileged space subtree, e.g., logging inputs for future replay. This space hierarchy also creates a performance bottleneck for I/O-bound applications, a limitation of the current design we intend to address in future work.

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

Table 1: System calls comprising Determinator's kernel API.

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

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

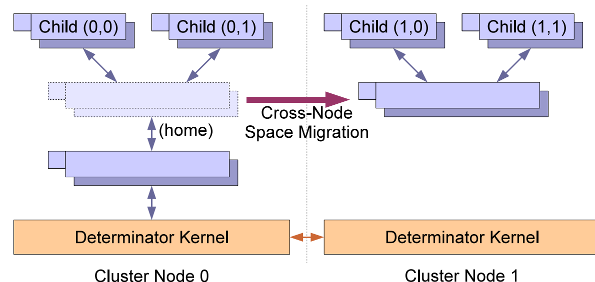

Figure 3: A spaces migrating among two nodes and starting child spaces on each node.

3.2 System Call API

Determinator spaces interact only as a result of processor traps and the kernel's three system calls—Put, Get, and Ret, summarized in Table 1. Put and Get take several optional arguments, summarized in Table 2. Most options can be combined: e.g., in one Put call a space can initialize a child's registers, copy a range of the parent's virtual memory into the child, set page permissions on the destination range, save a complete snapshot of the child's address space, and start the child executing.

Each space has a private namespace of child spaces, which user-level code manages. A space specifies a child number to Get or Put, and the kernel creates that child if it doesn't already exist, before performing the requested operations. If the specified child did exist and was still executing at the time of the Put/Get call, the kernel blocks the parent's execution until the child stops due to a Ret system call or a processor trap. These "rendezvous" semantics ensure that spaces synchronize only at well-defined points in both spaces' execution.

The Copy option logically copies a range of virtual memory between the invoking space and the specified child. The kernel uses copy-on-write to optimize large copies and avoid physically copying read-only pages.

Merge is available only on Get calls. A Merge is like a Copy, except the kernel copies only bytes that *differ* between the child's current and reference snapshots into the parent space, leaving other bytes in the parent untouched. The kernel also detects conflicts: if a byte changed in *both* the child's and parent's spaces since the snapshot, the kernel generates an exception, treating a conflict as a programming error like an illegal memory access or divide-by-zero. Determinator's user-level runtime uses Merge to give multithreaded processes the illusion of shared memory, as described later in Section 4.4. In principle, user-level code could implement Merge itself, but the kernel's direct access to page tables makes it easy for the kernel to implement Merge efficiently.

Finally, the Ret system call stops the calling space, returning control to the space's parent. Exceptions such as divide-by-zero also cause a Ret, providing the parent a status code indicating why the child stopped.

To facilitate debugging and prevent untrusted children from looping forever, a parent can start a child with an *instruction limit*, forcing control back to the parent after the child and its descendants collectively execute this many instructions. Counting instructions instead of "real time" preserves determinism, while enabling spaces to "quantize" a child's execution to implement scheduling schemes deterministically at user level [8, 21].

Barring kernel or processor bugs, unprivileged spaces constrained to use the above kernel API alone cannot behave nondeterministically even by deliberate design. While a formal proof is out of scope, one straightforward argument is that the above Get/Put/Ret primitives reduce to blocking, one-to-one message channels, making the space hierarchy a deterministic Kahn network [36].

3.3 Distribution via Space Migration

The kernel allows space hierarchies to span not only multiple CPUs in a multiprocessor/multicore system, but also multiple nodes in a homogeneous cluster, mostly transparently to application code. While distribution is semantically transparent to applications, an application may have to be designed with distribution in mind to perform well. As with other aspects of the kernel's design, we make no pretense that this is the "right" approach to cross-node distribution, but merely one way to extend a deterministic execution model across a cluster.

Distribution adds no new system calls or options to the API above. Instead, the Determinator kernel inter-

prets the higher-order bits in each process's child number namespace as a "node number" field. When a space invokes Put or Get, the kernel first logically migrates the calling space's state and control flow to the node whose number the user specifies as part of its child number argument, before creating and/or interacting with some child on that node, as specified in the remaining child number bits. Figure 3 illustrates a space migrating between two nodes and managing child spaces on each.

Once created, a space has a *home node*, to which the space migrates when interacting with its parent on a Ret or trap. Nodes are numbered so that "node zero" in any space's child namespace always refers to the space's home node. If a space uses only the low bits in its child numbers and leaves the node number field zero, the space's children all have the same home as the parent.

When the kernel migrates a space, it first transfers to the receiving kernel only the space's register state and address space summary information. Next, the receiving kernel requests the space's memory pages on demand as the space accesses them on the new node. Each node's kernel avoids redundant cross-node page copying in the common case when a space repeatedly migrates among several nodes—e.g., when a space starts children on each of several nodes, then returns later to collect their results. For pages that the migrating space only reads and never writes, such as program code, each kernel reuses cached copies of these pages whenever the space returns to that node. The kernel currently performs no prefetching or other adaptive optimizations. Its rudimentary messaging protocol runs directly atop Ethernet, and does not support TCP/IP for Internet-wide distribution.

4 Emulating High-Level Abstractions

Determinator's kernel API eliminates many convenient and familiar abstractions; can we reproduce them under strict determinism? We find that many familiar abstractions remain feasible, though with important tradeoffs. This section describes how Determinator's user-level runtime infrastructure emulates traditional Unix processes, file systems, threads, and synchronization.

4.1 Processes and fork/exec/wait

We make no attempt to replicate Unix process semantics exactly, but would like to emulate traditional `fork`/`exec`/`wait` APIs enough to support common uses in scriptable shells, build tools, and multi-process "batch processing" applications such as compilers.

Fork: Implementing a basic Unix `fork()` requires only one Put system call, to copy the parent's entire memory state into a child space, set up the child's registers, and start the child. The difficulty arises from Unix's global process ID (PID) namespace, a source of nondeterminism as discussed in Section 2.4. Since most applications use PIDs returned by `fork()` merely as an opaque argument to a subsequent `waitpid()`, our runtime makes PIDs local to each process: one process's PIDs are unrelated to, and may numerically conflict with, PIDs in other processes. This change breaks Unix applications that pass PIDs among processes, and means that commands like '`ps`' must be built into shells for the same reason that '`cd`' already is. This simple approach works for compute-oriented applications following the typical fork/wait pattern, however.

Since `fork()` returns a PID chosen by the system, while our kernel API requires user code to manage child numbers, our user-level runtime maintains a "free list" of child spaces and reserves one during each `fork()`. To emulate Unix process semantics more closely, a central space such as the root space could manage a global PID namespace, at the cost of requiring inter-space communication during operations such as `fork()`.

Exec: A user-level implementation of Unix `exec()` must construct the new program's memory image, intended to replace the old program, while still executing the old program's runtime library code. Our runtime loads the new program into a "reserved" child space never used by `fork()`, then calls Get to copy that child's entire memory atop that of the (running) parent: this Get thus "returns" into the new program. To ensure that the instruction address following the old program's Get is a valid place to start the new program, the runtime places this Get in a small "trampoline" code fragment mapped at the same location in the old and new programs. The runtime also carries over some Unix process state, such as the the PID namespace and file system state described later, from the old to the new program.

Wait: When an application calls `waitpid()` to wait for a specific child, the runtime calls Get to synchronize with the child's Ret and obtain the child's exit status. The child may return to the parent before terminating, in order to make I/O requests as described below; in this case, the parent's runtime services the I/O request and resumes the `waitpid()` transparently to the application.

Unix's `wait()` is more challenging, as it waits for *any* (i.e., "the first") child to terminate, violating the constraints of deterministic synchronization discussed in Section 2.3. Our kernel's API provides no system call to "wait for any child," and can't (for unprivileged spaces) without compromising determinism. Instead, our runtime waits for the child that was forked earliest whose status was not yet collected.

This behavior does not affect applications that fork one or more children and then wait for all of them to complete, but affects two common uses of `wait()`. First, interactive Unix shells use `wait()` to report when back-

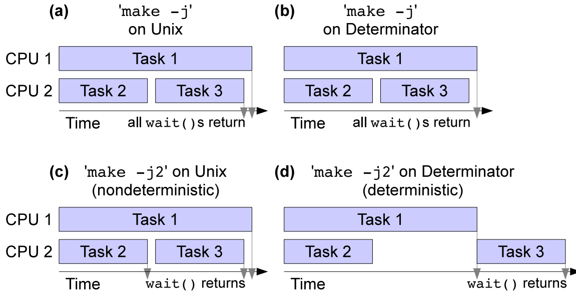

Figure 4: Example parallel make scheduling scenarios under Unix versus Determinator: (a) and (b) with unlimited parallelism (no user-level scheduling); (c) and (d) with a "2-worker" quota imposed at user level.

ground processes complete; thus, an interactive shell running under Determinator requires special "nondeterministic I/O privileges" to provide this functionality (and related functions such as interactive job control). Second, our runtime's behavior may adversely affect the performance of programs that use `wait()` to implement dynamic scheduling or load balancing in user space.

Consider a parallel `make` run with or without limiting the number of concurrent children. A plain '`make -j`', allowing unlimited children, leaves scheduling decisions to the system. Under Unix or Determinator, the kernel's scheduler dynamically assigns tasks to available CPUs, as illustrated in Figure 4 (a) and (b). If the user runs '`make -j2`', however, then `make` initially starts only tasks 1 and 2, then waits for one of them to complete before starting task 3. Under Unix, `wait()` returns when the short task 2 completes, enabling `make` to start task 3 immediately as in (c). On Determinator, however, the `wait()` returns only when (deterministically chosen) task 1 completes, resulting in a non-optimal schedule (d): determinism prevents the runtime from learning which of tasks 1 and 2 completed first. The unavailability of timing information with which to make good application-level scheduling decisions thus suggests a practice of leaving scheduling to the system in a deterministic environment (e.g., '`make -j`' instead of '`-j2`').

4.2 A Shared File System

Unix's globally shared file system provides a convenient namespace and repository for staging program inputs, storing outputs, and holding intermediate results such as temporary files. Since our kernel permits no physical state sharing, user-level code must emulate shared state abstractions. Determinator's "shared-nothing" space hierarchy is similar to a distributed system consisting only of uniprocessor machines, so our user-level runtime borrows distributed file system principles to offer applications a shared file system abstraction.

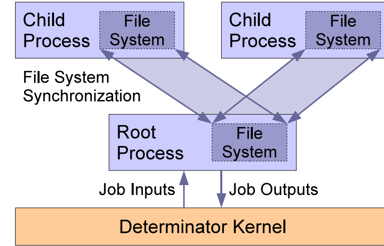

Figure 5: Each user-level runtime maintains a private replica of a logically shared file system, using file versioning to reconcile replicas at synchronization points.

Since our current focus is on emulating familiar abstractions and not on developing storage systems, Determinator's file system currently provides no persistence: it effectively serves only as a temporary file system.

While many distributed file system designs may be applicable, our runtime uses replication with weak consistency [53, 55]. Our runtime maintains a complete file system replica in the address space of each process it manages, as shown in Figure 5. When a process creates a child via `fork()`, the child inherits a copy of the parent's file system in addition to the parent's open file descriptors. Individual `open`/`close`/`read`/`write` operations in a process use only that process's file system replica, so different processes' replicas may diverge as they modify files concurrently. When a child terminates and its parent collects its state via `wait()`, the parent's runtime copies the child's file system image into a scratch area in the parent space and uses file versioning [47] to propagate the child's changes into the parent.

If a shell or parallel `make` forks several compiler processes in parallel, for example, each child writes its output `.o` file to its own file system replica, then the parent's runtime merges the resulting `.o` files into the parent's file system as the parent collects each child's exit status. This copying and reconciliation is not as inefficient as it may appear, due to the kernel's copy-on-write optimizations. Replicating a file system image among many spaces copies no physical pages until user-level code modifies them, so all processes' copies of identical files consume only one set of pages.

As in any weakly-consistent file system, processes may cause *conflicts* if they perform unsynchronized, concurrent writes to the same file. When our runtime detects a conflict, it simply discards one copy and sets a conflict flag on the file; subsequent attempts to `open()` the file result in errors. This behavior is intended for batch compute applications for which conflicts indicate an application or build system bug, whose appropriate solution is to fix the bug and re-run the job. Interactive use would demand a conflict handling policy that avoids losing data. The user-level runtime could alternatively use

pessimistic locking to implement stronger consistency and avoid unsynchronized concurrent writes, at the cost of more inter-space communication.

The current design's placement of each process's file system replica in the process's own address space has two drawbacks. First, it limits total file system size to less than the size of an address space; this is a serious limitation in our 32-bit prototype, though it may be less of an issue on a 64-bit architecture. Second, wild pointer writes in a buggy process may corrupt the file system more easily than in Unix, where a buggy process must actually call `write()` to corrupt a file. The runtime could address the second issue by write-protecting the file system area between calls to `write()`, or it could address both issues by storing file system data in child spaces not used for executing child processes.

4.3 Input/Output and Logging

Since unprivileged spaces can access external I/O devices only indirectly via parent/child interaction within the space hierarchy, our user-level runtime treats I/O as a special case of file system synchronization. In addition to regular files, a process's file system image can contain special *I/O files*, such as a console input file and a console output file. Unlike Unix device special files, Determinator's I/O files actually hold data in the process's file system image: for example, a process's console input file accumulates all the characters the process has received from the console, and its console output file contains all the characters it has written to the console. In the current prototype this means that console or log files can eventually "fill up" and become unusable, though a suitable garbage-collection mechanism could address this flaw.

When a process does a `read()` from the console, the C library first returns unread data already in the process's local console input file. When no more data is available, instead of returning an end-of-file condition, the process calls Ret to synchronize with its parent and wait for more console input (or in principle any other form of new input) to become available. When the parent does a `wait()` or otherwise synchronizes with the child, it propagates any new input it already has to the child. When the parent has no new input for any waiting children, it forwards all their input requests to its parent, and ultimately to the kernel via the root process.

When a process does a console `write()`, the runtime appends the new data to its internal console output file as it would append to a regular file. The next time the process synchronizes with its parent, file system reconciliation propagates these writes toward the root process, which forwards them to the kernel's I/O devices. A process can request immediate synchronization and output propagation by explicitly calling `fsync()`.

The reconciliation mechanism handles "append-only"

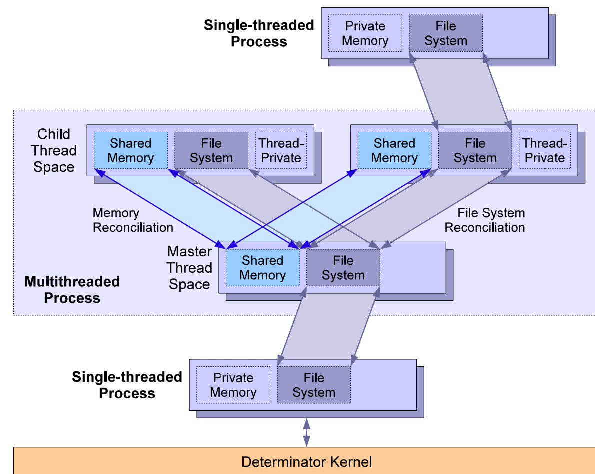

Figure 6: A multithreaded process built from one space per thread, with a master space managing synchronization and memory reconciliation.

writes differently from other file changes, enabling concurrent writes to console or log files without conflict. During reconciliation, if both the parent and child have made append-only writes to the same file, reconciliation appends the child's latest writes to the parent's copy of the file, and vice versa. Each process's replica thus accumulates all processes' concurrent writes, though different processes may observe these writes in a different order. Unlike Unix, rerunning a parallel computation from the same inputs with and without output redirection yields byte-for-byte identical console and log file output.

4.4 Shared Memory Multithreading

Shared memory multithreading is popular despite the nondeterminism it introduces into processes, in part because parallel code need not pack and unpack messages: threads simply compute "in-place" on shared variables and structures. Since Determinator gives user spaces no physically shared memory other than read-only sharing via copy-on-write, emulating shared memory involves distributed shared memory (DSM) techniques. Adapting the private workspace model discussed in Section 2.2 to thread-level shared memory involves reusing ideas explored in early parallel Fortran machines [7, 51] and in release-consistent DSM systems [2, 17], although none of this prior work attempted to provide determinism.

Our runtime uses the kernel's Snap and Merge operations (Section 3.2) to emulate shared memory in the private workspace model, using fork/join synchronization. To fork a child, the parent thread calls Put with the Copy, Snap, Regs, and Start options to copy the shared part of its memory into a child space, save a snapshot of that memory state in the child, and start the child running, as illustrated in Figure 6. The master thread may fork mul-

tiple children this way. To synchronize with a child and collect its results, the parent calls Get with the Merge option, which merges all changes the child made to shared memory, since its snapshot was taken, back into the parent. If both parent and child—or the child and other children whose changes the parent has collected—have concurrently modified the same byte since the snapshot, the kernel detects and reports this conflict.

Our runtime also supports barriers, the foundation of data-parallel programming models like OpenMP [45]. When each thread in a group arrives at a barrier, it calls Ret to stop and wait for the parent thread managing the group. The parent calls Get with Merge to collect each child's changes before the barrier, then calls Put with Copy and Snap to resume each child with a new shared memory snapshot containing all threads' prior results. While our private workspace model conceptually extends to non-hierarchical synchronization [5], our prototype's strict space hierarchy currently limits synchronization flexibility, an issue we intend to address in the future. *Any* synchronization abstraction may be emulated at some cost as described in the next section, however.

An application can choose which parts of its address space to share and which to keep thread-private. By placing thread stacks outside the shared region, all threads can reuse the same stack area, and the kernel wastes no effort merging stack data. Thread-private stacks also offer the convenience of allowing a child thread to inherit its parent's stack, and run "inline" in the same C/C++ function as its parent, as in Figure 1. If threads wish to pass pointers to stack-allocated structures, however, then they may locate their stacks in disjoint shared regions. Similarly, if the file system area is shared, then the threads share a common file descriptor namespace as in Unix. Excluding the file system area from shared space and using normal file system reconciliation (Section 4.2) to synchronize it yields thread-private file tables.

4.5 Emulating Legacy Thread APIs

As discussed in Section 2.3, we hope much existing sequential code can readily be parallelized using naturally deterministic synchronization abstractions, like data-parallel models such as OpenMP [45] and SHIM [24] already offer. For code already parallelized using nondeterministic synchronization, however, Determinator's runtime can emulate the standard pthreads API via deterministic scheduling [8, 10, 21], at certain costs.

In a process that uses nondeterministic synchronization, the process's initial *master space* never runs application code directly, but instead acts as a deterministic scheduler. This scheduler creates one child space to run each application thread. The scheduler runs the threads under an artificial execution schedule, emulating a schedule by which a true shared-memory multiprocessor might in principle run them, but using a deterministic, virtual notion of time—namely, number of instructions executed—to schedule all inter-thread interactions.

Like DMP [8, 21], our deterministic scheduler *quantizes* each thread's execution by preempting it after executing a fixed number of instructions. Whereas DMP implements preemption by instrumenting user-level code, our scheduler uses the kernel's instruction limit feature (Section 3.2). The scheduler "donates" execution quanta to threads round-robin, allowing each thread to run concurrently with other threads for one quantum, before collecting the thread's shared memory changes via Merge and restarting it for another quantum.

A thread's shared memory writes propagate to other threads only at the end of each quantum, violating sequential consistency [38]. Like DMP-B [8], our scheduler implements a weak consistency model [28], totally ordering only synchronization operations. To enforce this total order, each synchronization operation could simply spin for a a full quantum. To avoid wasteful spinning, however, our synchronization primitives interact with the deterministic scheduler directly.

Each mutex, for example, is always "owned" by some thread, whether or not the mutex is locked. The mutex's owner can lock and unlock the mutex without scheduler interactions, but any other thread needing the mutex must first invoke the scheduler to obtain ownership. At the current owner's next quantum, the scheduler "steals" the mutex from its current owner if the mutex is unlocked, and otherwise places the locking thread on the mutex's queue to be awoken once the mutex becomes available.

Since the scheduler can preempt threads at any point, a challenge common to any preemptive scenario is making synchronization functions such as `pthread_mutex_lock()` atomic. The kernel does not allow threads to disable or extend their own instruction limits, since we wish to use instruction limits at process level as well, e.g., to enforce deterministic "time" quotas on untrusted processes, or to improve user-level process scheduling (see Section 4.1) by quantizing process execution. After synchronizing with a child thread, therefore, the master space checks whether the instruction limit preempted a synchronization function, and if so, resumes the preempted code in the master space. Before returning to the application, these functions check whether they have been "promoted" to the master space, and if so migrate their register state back to the child thread and restart the scheduler in the master space.

While deterministic scheduling provides compatibility with existing parallel code, it has drawbacks. The master space, required to enforce a total order on synchronization operations, may be a scaling bottleneck unless execution quanta are large. Since threads can interact only at quanta boundaries, however, large quanta increase the

time one thread may waste waiting to interact with another, to steal an unlocked mutex for example.

Further, since the deterministic scheduler may preempt a thread and propagate shared memory changes at any point in application code, the *programming model* remains nondeterministic. In contrast with our private workspace model, if one thread runs '$x = y$' while another runs '$y = x$' under the deterministic scheduler, the result may be repeatable but is no more predictable to the programmer than on traditional systems. While rerunning a program with *exactly* identical inputs will yield identical results, if the input is perturbed to change the length of any instruction sequence, these changes may cascade into a different execution schedule and trigger schedule-dependent if not timing-dependent bugs.

5 Prototype Implementation

Determinator is written in C with small assembly fragments, currently runs on the 32-bit x86 architecture, and implements the kernel API and user-level runtime facilities described above. Source releases are available at '`http://dedis.cs.yale.edu/`'.

Since our focus is on parallel compute-bound applications, Determinator's I/O capabilities are currently limited. The system provides text-based console I/O and a Unix-style shell supporting redirection and both scripted and interactive use. The shell offers no interactive job control, which would require currently unimplemented "nondeterministic privileges" (Section 4.1). The system has no demand paging or persistent disk storage: the user-level runtime's logically shared file system abstraction currently operates in physical memory only.

The kernel supports application-transparent space migration among up to 32 machines in a cluster, as described in Section 3.3. Migration uses a synchronous messaging protocol with only two request/response types and implements almost no optimizations such as page prefetching. The protocol runs directly atop Ethernet, and is not intended for Internet-wide distribution.

The prototype has other limitations already mentioned. The kernel's strict space hierarchy could bottleneck I/O-intensive applications (Section 3.1), and does not easily support non-hierarchical synchronization such as queues or futures (Section 4.4). The file system's size is constrained to a process's address space (Section 4.2), and special I/O files can fill up (Section 4.3). None of these limitations are fundamental to Determinator's programming model. At some cost in complexity, the model could support non-hierarchical synchronization [5]. The runtime could store files in child spaces or on external I/O devices, and could garbage-collect I/O streams.

Implementing instruction limits (Section 3.2) requires the kernel to recover control after a precise number of instructions execute in user mode. While the PA-RISC architecture provided this feature [1], the x86 does not, so we borrowed ReVirt's technique [22]. We first set an *imprecise* hardware performance counter, which unpredictably overshoots its target a small amount, to interrupt the CPU before the desired number of instructions, then run the remaining instructions under debug tracing.

6 Evaluation

This section evaluates the Determinator prototype, first informally, then examining single-node and distributed parallel processing performance, and finally code size.

6.1 Experience Using the System

We find that a deterministic programming model simplifies debugging of both applications and user-level runtime code, since user-space bugs are always reproducible. Conversely, when we do observe nondeterministic behavior, it can result only from a kernel (or hardware) bug, immediately limiting the search space.

Because Determinator's file system holds a process's output until the next synchronization event (often the process's termination), each process's output appears as a unit even if the process executes in parallel with other output-generating processes. Further, different processes' outputs appear in a consistent order across runs, as if run sequentially. (The kernel provides a system call for debugging that outputs a line to the "real" console immediately, reflecting true execution order, but chaotically interleaving output as in conventional systems.)

While race detection tools exist [25, 43], we found it convenient that Determinator always detects conflicts under "normal-case" execution, without requiring the user to run a special tool. Since the kernel detects shared memory conflicts and the user-level runtime detects file system conflicts at every synchronization event, Determinator's model makes conflict detection as standard as detecting division by zero or illegal memory accesses.

A subset of Determinator doubles as *PIOS*, "Parallel Instructional Operating System," which we used in Yale's operating system course this spring. While the OS course's objectives did not include determinism, they included introducing students to parallel, multicore, and distributed operating system concepts. For this purpose, we found Determinator/PIOS to be a useful instructional tool due to its simple design, minimal kernel API, and adoption of distributed systems techniques within and across physical machines. PIOS is partly derived from MIT's JOS [35], and includes a similar instructional framework where students fill in missing pieces of a "skeleton." The twelve students who took the course, working in groups of two or three, all successfully reimplemented Determinator's core features: multiprocessor scheduling with Get/Put/Ret coordination, virtual memory with copy-on-write and Snap/Merge, user-level

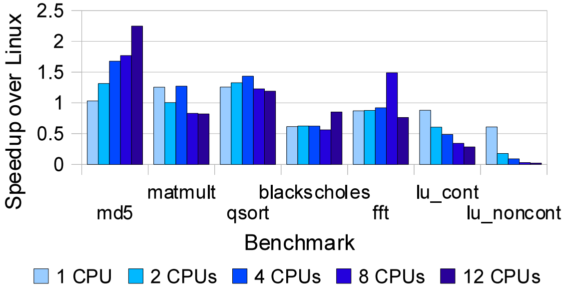

Figure 7: Determinator performance relative to pthreads under Ubuntu Linux on various parallel benchmarks.

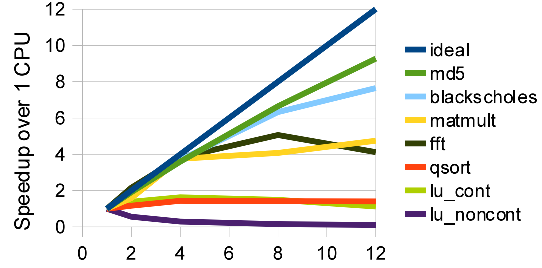

Figure 8: Determinator parallel speedup over its own single-CPU performance on various benchmarks.

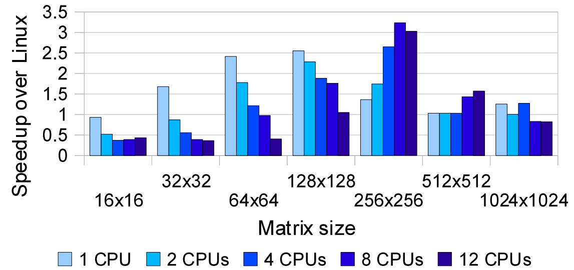

Figure 9: Matrix multiply with varying matrix size.

threads with fork/join synchronization (but not deterministic scheduling), the user-space file system with versioning and reconciliation, and application-transparent cross-node distribution via space migration. In their final projects they extended the OS with features such as graphics, pipes, and a remote shell. While instructional use by no means indicates a system's real-world utility, we find the success of the students in understanding and building on Determinator's architecture promising.

6.2 Single-node Multicore Performance

Since Determinator runs user-level code "natively" on the hardware instead of rewriting user code [8, 21], we expect it to perform comparably to conventional systems when executing single-threaded, compute-bound code. Since thread interactions require system calls, context switches, and virtual memory operations, however, we expect determinism to incur a performance cost in proportion to the frequency of thread interaction.

Figure 7 shows the performance of several shared-memory parallel benchmarks we ported to Determinator, relative to the same benchmarks using conventional pthreads on 32-bit Ubuntu Linux 9.10. The *md5* benchmark searches for an ASCII string yielding a particular MD5 hash, as in a brute-force password cracker; *matmult* multiplies two 1024×1024 integer matrices; *qsort* is a recursive parallel quicksort on an integer array; *blackscholes* is a financial benchmark from the PARSEC suite [11]; and *fft*, *lu_cont*, and *lu_noncont* are Fast Fourier Transform and LU-decomposition benchmarks from SPLASH-2 [57]. We tested all benchmarks on a 2 socket $\times$ 6 core, 2.2GHz AMD Opteron PC.

Coarse-grained benchmarks like *md5*, *matmult*, *qsort*, *blackscholes*, and *fft* show performance comparable with that of nondeterministic multithreaded execution under Linux. The *md5* benchmark shows better scaling on Determinator than on Linux, achieving a $2.25\times$ speedup over Linux on 12 cores. We have not identified the precise cause of this speedup over Linux but suspect scaling bottlenecks in Linux's thread system [54].

Porting the *blackscholes* benchmark to Determinator required no changes as it uses deterministically scheduled pthreads (Section 4.5). The deterministic scheduler's quantization, however, incurs a fixed performance cost of about 35% for the chosen quantum of 10 million instructions. We could reduce this overhead by increasing the quantum, or eliminate it by porting the benchmark to Determinator's "native" parallel API.

The fine-grained *lu* benchmarks show a higher performance cost, indicating that Determinator's virtual memory-based approach to enforcing determinism is not well-suited to fine-grained parallel applications. Future hardware enhancements might make determinism practical for fine-grained parallel applications, however [21].

Figure 8 shows each benchmark's speedup relative to single-threaded execution on Determinator. The "embarrassingly parallel" *md5* and *blackscholes* scale well, *matmult* and *fft* level off after four processors (but still perform comparably to Linux as Figure 7 shows), and the remaining benchmarks scale poorly.

To quantify further the effect of parallel interaction granularity on deterministic execution performance, Figures 9 and 10 show Linux-relative performance of *matmult* and *qsort*, respectively, for varying problem sizes. With both benchmarks, deterministic execution incurs a high performance cost on small problem sizes requiring frequent interaction, but on large problems Determinator is competitive with and sometimes faster than Linux.

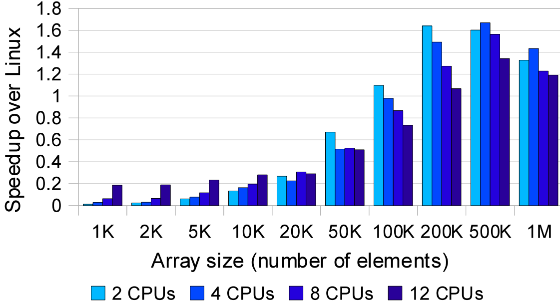

Figure 10: Parallel quicksort with varying array size.

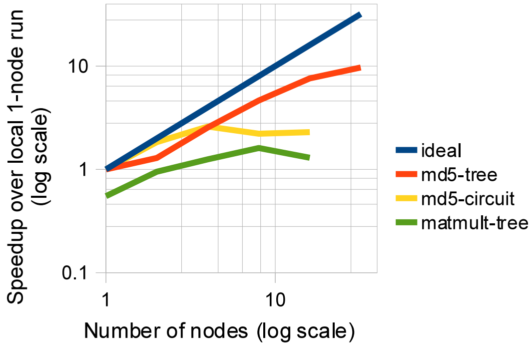

Figure 11: Speedup of deterministic shared memory benchmarks on varying-size distributed clusters.

6.3 Distributed Computing Performance

While Determinator's rudimentary space migration (Section 3.3) is far from providing a full cluster computing architecture, we would like to test whether such a mechanism can extend a deterministic computing model across nodes with usable performance at least for some applications. We therefore changed the *md5* and *matmult* benchmarks to distribute workloads across a cluster of up to 32 uniprocessor nodes via space migration. Both benchmarks still run in a (logical) shared memory model via Snap/Merge. Since we did not have a cluster on which we could run Determinator natively, we ran it under QEMU [6], on a cluster of 2 socket $\times$ 2 core, 2.4GHz Intel Xeon machines running SuSE Linux 11.1.

Figure 11 shows parallel speedup under Determinator relative to local single-node execution in the same environment, on a log-log scale. In *md5-circuit*, the master space acts like a traveling salesman, migrating serially to each "worker" node to fork child processes, then retracing the same circuit to collect their results. The *md5-tree* variation forks workers recursively in a binary tree: the master space forks children on two nodes, those children each fork two children on two nodes, etc. The *matmult-tree* benchmark implements matrix multiply with recursive work distribution as in *md5-tree*.

The "embarrassingly parallel" *md5-tree* performs and scales well, but only with recursive work distribution. Matrix multiply levels off at two nodes, due to the amount of matrix data the kernel transfers across nodes via its simplistic page copying protocol, which currently performs no data streaming, prefetching, or delta compression. The slowdown for 1-node distributed execution in *matmult-tree* reflects the cost of transferring the matrix to a (single) remote machine for processing.

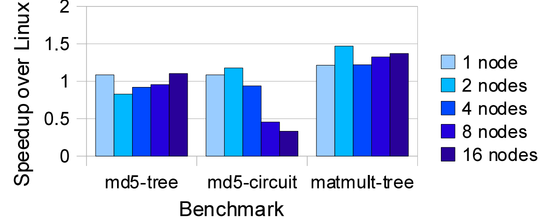

Figure 12: Deterministic shared memory benchmarks versus distributed-memory equivalents for Linux.

Component	Determinator Semicolons	PIOS Semicolons
Kernel core	2044	1847
Hardware/device drivers	751	647
User-level runtime	2952	1079
Generic C library code	6948	394
User-level programs	1797	1418
Total	14,492	5385

Table 3: Implementation code size of the Determinator OS and of PIOS, its instructional subset.

As Figure 12 shows, the shared memory *md5-tree* and *matmult-tree* benchmarks, running on Determinator, perform comparably to nondeterministic, distributed-memory equivalents running on Puppy Linux 4.3.1, in the same QEMU environment. Determinator's clustering protocol does not use TCP as the Linux-based benchmarks do, so we explored the benchmarks' sensitivity to this factor by implementing TCP-like round-trip timing and retransmission behavior in Determinator. These changes resulted in less than a 2% performance impact.

Illustrating the simplicity benefits of Determinator's shared memory thread API, the Determinator version of *md5* is 63% the size of the Linux version (62 lines containing semicolons versus 99), which uses remote shells to coordinate workers. The Determinator version of *matmult* is 34% the size of its Linux equivalent (90 lines versus 263), which passes data explicitly via TCP.

6.4 Implementation Complexity

To provide a feel for implementation complexity, Table 3 shows source code line counts for Determinator, as well as its PIOS instructional subset, counting only lines con-

taining semicolons. The entire system is less than 15,000 lines, about half of which is generic C and math library code needed mainly for porting Unix applications easily.

7 Related Work

Recognizing the benefits of determinism [12, 40], parallel languages such as SHIM [24] and DPJ [12, 13] enforce determinism at language level, but require rewriting, rather than just parallelizing, existing serial code. Race detectors [25, 43] detect low-level heisenbugs in nondeterministic parallel programs, but may miss higher-level heisenbugs [3]. Language extensions can dynamically check determinism assertions [16, 49], but heisenbugs may persist if the programmer omits an assertion.

Early parallel Fortran systems [7, 51], release consistent DSM [2, 17], transactional memory [33, 52] and OS APIs [48], replicated file systems [53, 55], and distributed version control [29] all foreshadow Determinator's private workspace programming model. None of these precedents create a deterministic application programming model, however, as is Determinator's goal.

Deterministic schedulers such as DMP [8, 21] and Grace [10] instrument an application to schedule inter-thread interactions on a repeatable, artificial time schedule. DMP isolates threads via code rewriting, while Grace uses virtual memory as in Determinator. Developed simultaneously with Determinator, dOS [9] incorporates a deterministic scheduler into the Linux kernel, preserving Linux's existing programming model and API. This approach provides greater backward compatibility than Determinator's clean-slate design, but makes the Linux programming model no more *semantically* deterministic than before. Determinator offers new thread and process models redesigned to eliminate conventional data races, while supporting deterministic scheduling in user space for backward compatibility.

Many techniques are available to log and replay nondeterministic events in parallel applications [39, 46, 56]. SMP-ReVirt can log and replay a multiprocessor virtual machine [23], supporting uses such as system-wide intrusion analysis [22, 34] and replay debugging [37]. Logging a parallel system's nondeterministic events is costly in performance and storage space, however, and usually infeasible for "normal-case" execution. Determinator demonstrates the feasibility of providing system-enforced determinism for normal-case execution, without internal event logging, while maintaining performance comparable with current systems at least for coarse-grained parallel applications.

Determinator's kernel design owes much to microkernels such as L3 [41]. An interesting contrast is with the Exokernel approach [26], which is incompatible with Determinator's. System-enforced determinism requires hiding nondeterministic kernel state from applications, such as the physical addresses of virtual memory pages, whereas exokernels deliberately expose this state.

8 Conclusion

While Determinator is only a proof-of-concept, it shows that operating systems can offer a pervasively and naturally deterministic application environment, avoiding the introduction of data races in shared memory and file system access, thread and process synchronization, and throughout the API. Our experiments suggest that such an environment can efficiently run coarse-grained parallel applications, both on a single multicore machine and across a cluster, though supporting fine-grained parallelism efficiently may require hardware evolution.

Acknowledgments

We thank Zhong Shao, Ramakrishna Gummadi, Frans Kaashoek, Nickolai Zeldovich, Sam King, and the OSDI reviewers for their valuable feedback. We also thank NSF for their support under grant CNS-1017206.

References

[1] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard, Feb. 1994.

[2] C. Amza et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] C. Artho, K. Havelund, and A. Biere. High-level data races. In *VVEIS*, pages 82–93, Apr. 2003.

[4] A. Aviram et al. Determinating timing channels in compute clouds. In *CCSW*, Oct. 2010.

[5] A. Aviram and B. Ford. Deterministic consistency, Feb. 2010. `http://arxiv.org/abs/0912.0926`.

[6] F. Bellard. QEMU, a fast and portable dynamic translator, Apr. 2005.

[7] M. Beltrametti, K. Bobey, and J. R. Zorbas. The control mechanism for the Myrias parallel computer system. *Computer Architecture News*, 16(4):21–30, Sept. 1988.

[8] T. Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, Mar. 2010.

[9] T. Bergan et al. Deterministic process groups in dOS. In *9th OSDI*, Oct. 2010.

[10] E. D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, Oct. 2009.

[11] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *17th PACT*, October 2008.

[12] R. L. Bocchino et al. Parallel programming must be deterministic by default. In *HotPar*. Mar. 2009.

[13] R. L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, Oct. 2009.

[14] S. Boyd-Wickizer et al. Corey: An operating system for many cores. In *8th OSDI*, Dec. 2008.

[15] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *TOCS*, 14(1):80–107, Feb. 1996.

[16] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, Aug. 2009.

[17] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *13th SOSP*, Oct. 1991.

[18] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd OSDI*, pages 173–186, Feb. 1999.

[19] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59. 1998.

[20] H. Cui, J. Wu, and J. Yang. Stable deterministic multithreading through schedule memoization. In *9th OSDI*, Oct. 2010.

[21] J. Devietti et al. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, Mar. 2009.

[22] G. W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th OSDI*, Dec. 2002.

[23] G. W. Dunlap et al. Execution replay for multiprocessor virtual machines. In *VEE*, Mar. 2008.

[24] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, Mar. 2008.

[25] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *19th SOSP*, Oct. 2003.

[26] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th SOSP*, Dec. 1995.

[27] B. Ford et al. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[28] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th ISCA*, pages 15–26, May 1990.

[29] git: the fast version control system. `http://git-scm.com/`.

[30] A. Haeberlen et al. Accountable virtual machines. In *9th OSDI*, Oct. 2010.

[31] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *21st SOSP*, Oct. 2007.

[32] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 7(4):501–538, Oct. 1985.

[33] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th ISCA*, pages 289–300, May 1993.

[34] A. Joshi et al. Detecting past and present intrusions through vulnerability-specific predicates. In *20th SOSP*, pages 91–104. 2005.

[35] F. Kaashoek et al. 6.828: Operating system engineering. `http://pdos.csail.mit.edu/6.828/`.

[36] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475. 1974.

[37] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, pages 1–15, Apr. 2005.

[38] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.

[39] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

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

[41] J. Liedtke. On micro-kernel construction. In *15th SOSP*, 1995.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th ASPLOS*, pages 329–339, Mar. 2008.

[43] M. Musuvathi et al. Finding and reproducing Heisenbugs in concurrent programs. In *8th OSDI*. 2008.

[44] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *14th ASPLOS*, Mar. 2009.

[45] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[46] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *PADD '88*, pages 124–129. 1988.

[47] D. S. Parker, Jr. et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[48] D. E. Porter et al. Operating system transactions. In *22nd SOSP*, Oct. 2009.

[49] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *18th ESOP*, Mar. 2009.

[50] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, Dec. 1990.

[51] J. T. Schwartz. The burroughs FMP machine, Jan. 1980. Ultracomputer Note #5.

[52] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb. 1997.

[53] D. B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th SOSP*, 1995.

[54] R. von Behren et al. Capriccio: Scalable threads for internet services. In *SOSP'03*.

[55] B. Walker et al. The LOCUS distributed operating system. *OSR*, 17(5), Oct. 1983.

[56] L. Wittie. The Bugnet distributed debugging system. In *Making Distributed Systems Work*, Sept. 1986.

[57] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, June 1995.