Efficient System-Enforced Deterministic Parallelism

UNPUBLISHED DRAFT

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

Abstract

Deterministic execution offers many benefits for debugging, fault tolerance, and security. Running *parallel* programs deterministically is usually difficult and costly, however—especially if we desire *system-enforced* determinism, ensuring precise repeatability of arbitrarily buggy or malicious software. Determinator is a novel operating system that enforces determinism on both multithreaded and multi-process computations. Determinator's kernel provides only single-threaded, "shared-nothing" address spaces interacting via deterministic synchronization. An untrusted user-level runtime uses distributed computing techniques to emulate familiar abstractions such as Unix processes, file systems, and shared memory multithreading. The system runs parallel applications deterministically both on multicore PCs and across nodes in a cluster. Coarse-grained parallel benchmarks perform and scale comparably to—sometimes better than—conventional systems, though determinism is costly for fine-grained parallel applications.

1 Introduction

It is often useful to run software *deterministically*, ensuring a given program and input always yields exactly the same result. Deterministic execution makes bugs reproducible, and is required for "record-and-replay" debugging [28,40]. Fault tolerance [15,18,49] and accountability mechanisms [33] rely on execution being deterministic and bit-for-bit identical across state replicas. Intrusion analysis [23, 36] and timing channel control [4] can further benefit from *system-enforced determinism*, where the system prevents application code from depending on execution timing or other unintended inputs even if the code is maliciously designed to do so.

Multicore processors and ubiquitous parallelism make programming environments increasingly nondeterministic, however. Nondeterminism makes software harder to develop and debug [43, 44]. Race detectors help [27, 45], but even properly synchronized programs may have higher-level heisenbugs [3]. The cost of logging and replaying the internal nondeterministic events in parallel software [20, 24] can be orders of magnitude higher than that of logging only a computation's external inputs, especially for system-enforced replay [23, 24]. This cost usually precludes logging "normal-case" execution, diminishing the technique's effectiveness. A heisenbug or intrusion that manifests "in the field" with logging disabled may not reappear during subsequent logged attempts to reproduce it—especially with malware *designed* to evade analysis by detecting the timing impact of logging or virtualization [30].

Motivated by its many uses, we would like system-enforced determinism to be available for *normal-case* execution of parallel applications. To test this goal's feasibility, we built *Determinator*, an operating system that not only executes individual processes deterministically, as in deterministic user-level scheduling [8, 9], but can enforce determinism on hierarchies of interacting processes. Rerunning a multi-process Determinator computation with the same inputs yields exactly the same outputs, without internal event logging. Determinator treats all potential nondeterministic inputs to a computation—including all timing information—as "privileged information," which normal applications cannot obtain except via controlled channels. We treat deterministic execution as not just a debugging tool but a security principle: if malware infects an unprivileged Determinator application, it should be unable to evade replay-based analysis.

System-enforced determinism is challenging because current programming environments and APIs are riddled with timing dependencies. Most shared-memory parallel code uses mutual exclusion primitives: even when used correctly, timing determines the application-visible order in which competing threads acquire a mutex. Concurrency makes names allocated from shared namespaces, such as pointers returned by `malloc()` and file descriptors returned by `open()`, timing-dependent. Synchronizing operations like semaphores, message queues, and `wait()` nondeterministically return "the first" event, message, or terminated process available. Even single-threaded processes are nondeterministic when run in parallel, due to their interleaved accesses to shared resources. A parallel '`make -j`' command often presents a chaotic mix of its child tasks' outputs, for example, and missing dependencies can yield "makefile heisenbugs" that manifest only under parallel execution.

Addressing these challenges in Determinator led us to the insight that timing dependencies commonly fall into a few categories: unintended interactions via shared state or namespaces; synchronization abstractions with share-

able endpoints; true dependencies on "real-world" time; and application-level scheduling. Determinator avoids physically shared state by isolating concurrent activities during normal execution, allowing interaction only at explicit synchronization points. The kernel's API uses local, application-chosen names in place of shared, OS-managed namespaces. Synchronization primitives operate "one-to-one," between *specific* threads, preventing threads from "racing" to an operation. Determinator treats access to real-world time as I/O, controlling it as with other devices such as disk or network. Finally, Determinator requires scheduling to be separated from application logic and handled by the system, or else emulated using a deterministic, virtual notion of "time."

Since we wish to derive basic principles for system-enforced determinism, Determinator currently makes no attempt at compatibility with existing operating systems, and provides limited compatibility with existing APIs. The kernel's low-level API offers only one user-visible abstraction, *spaces*, representing execution state and virtual memory, and only three system calls by which spaces synchronize and communicate. The API's minimality facilitates both experimentation and reasoning about its determinism. Despites this simplicity, our untrusted, user-level runtime builds atop the kernel to provide familiar programming abstractions. The runtime uses file replication and versioning [47] to offer applications a logically shared file system via standard APIs; distributed shared memory [2,17] to create multithreaded processes logically sharing an address space; and deterministic scheduling [8, 9, 22] to support pthreads-style synchronization. Since the kernel enforces determinism, bugs or vulnerabilities in this runtime cannot compromise the determinism guarantee.

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. The current prototype is merely a proof-of-concept and has many limitations, such as a restrictive space hierarchy, limited file system size, no persistent storage, and inefficient cross-node communication. Also, our "clean-slate" approach is motivated by research goals; a more realistic approach to deploying system-enforced determinism would be to add a deterministic "sandbox" [19, 32] to a conventional OS.

This paper makes three main contributions. First, we identify five OS design principles for system-enforced determinism, and illustrate their application in a novel kernel API. Second, we demonstrate ways to build familiar abstractions such as file systems and shared memory atop a kernel API restricted to deterministic primitives. Third, we present the first system that can enforce deterministic execution on multi-process computations with performance acceptable for "normal-case" use, at least for some (coarse-grained) parallel applications.

Section 2 describes Determinator's kernel design principles and API, then Section 3 details its user-space application runtime. Section 4 examines our prototype implementation, and Section 5 evaluates it informally and experimentally. Finally, Section 6 outlines related work, and Section 7 concludes.

2 The Determinator Kernel

This section describes Determinator's underlying design principles, then its low-level execution model and kernel API. We do not expect normal applications to use the kernel API directly, but rather the higher-level abstractions the user-level runtime provides, as described in the next section. We make no claim that this API is the "right" design for a determinism-enforcing kernel, but merely use it to explore design challenges and strategies.

2.1 Kernel API Design Principles

We first briefly outline the principles we developed in designing Determinator, which address the common sources of timing dependencies we are aware of. We further discuss the motivations and implications of these principles below as we detail the kernel API. We make no claim that this is a complete or conclusive list, but at least for Determinator these principles prove *sufficient* to offer a deterministic execution guarantee, for which we briefly sketch formal arguments later in Section 2.4.

1. Isolate the working state of concurrent activities between synchronization points. Determinator's kernel API directly provides no shared state abstractions, such as global file systems or writeable shared memory. Concurrent activities operate within private "sandboxes," interacting only at deterministically defined synchronization points, eliminating timing dependencies due to interleaved access to shared state.

2. Use local, application-chosen names instead of global, system-allocated names. APIs that assign names from a shared namespace introduce nondeterminism even when the named objects are unshared: 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. To avoid these

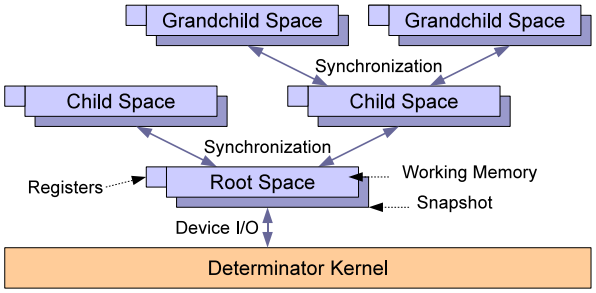

Figure 1: The kernel's hierarchy of *spaces*, each containing private register and virtual memory state.

sources of nondeterminism, Determinator's kernel API uses only *local* names chosen by the application: user-level code 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.

3. User code determines the participants in any synchronization operation, and the point in each participant's execution at which synchronization occurs. The kernel API allows a thread or process to synchronize with a *particular* target, like Unix processes use `waitpid()` to wait for a specific child. The API does *not* support synchronizing with "any" or "the first available" target as in Unix's `wait()`, or interrupting another thread at a timing-dependent point in its execution, as with Unix signals. Nondeterministic synchronization APIs may be emulated deterministically, if needed for compatibility, as described in Section 3.5.

4. Treat access to explicit time sources as I/O. User code has no direct access to clocks counting either real time, as in `gettimeofday()`, or nondeterministic "virtual time" measures, as in `getrusage()`. Determinator treats such timing sources as I/O devices that user code may access only via controlled channels, as with other devices such as network, disk, and display.

5. Separate application logic from scheduling. Deterministic applications cannot make timing-dependent internal scheduling or load-balancing decisions, as today's applications often do using thread pools or work queues. Applications may *expose* arbitrary parallelism and provide scheduling hints—in principle they could even download extensions into the kernel to customize scheduling [10]—provided the kernel prevents custom scheduling policies from affecting computed results.

2.2 Spaces

Determinator executes application code within a hierarchy of *spaces*, illustrated in Figure 1. 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 several 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, as described later in Section 3.

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.

As in a nested process model [29], 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. Following principle 1 above, the kernel provides no file systems, writable shared memory, or other shared state abstractions.

Following principle 4, only the distinguished *root space* has direct access to nondeterministic I/O devices including 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.)

2.3 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.

As per principle 2 above, 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'

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

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

execution, as required by principle 3.

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 3.4. 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, 22].

2.4 Reasoning about Determinism

Can we be certain the kernel API above indeed guarantees that space subtrees execute deterministically despite parallelism? While a detailed proof is out of scope, we briefly sketch two formal arguments for this guarantee.

The first argument leverages an existing formal parallel computing model: a Kahn process network [38] is a network of single-threaded processes, which run sequential code deterministically and interact only via blocking, one-to-one message channels. Under these restrictions, a Kahn network behaves deterministically. Determinator's Get, Put, and Ret calls are implementable in terms of messages on one-to-one channels, making Determinator's space hierarchy formally equivalent to a Kahn process network, thereby ensuring its determinism.

For a more "first-principles" argument, consider a graph of possible execution traces of a space hierarchy. Each node represents a synchronization point in a possible execution history of one space, vertical edges represent local computation sequences in one space between synchronization points, and horizontal edges represent pairwise interactions where a parent space's Get or Put synchronizes with a child's Ret. From this graph we construct a "happens-before" partial order over all synchronization points in all possible executions. At each synchronization point, assuming all prior (on the partial order) computation sequences and synchronization interactions yield only one possible result for a given set of inputs, then the same is true after that synchronization point: each synchronization point in a parent space interacts with only one corresponding point in a specific child, and vice versa, and synchronization effects such as memory changes depend only on the two spaces' states prior to synchronization. By induction on the partial order, the entire execution history is therefore deterministic.

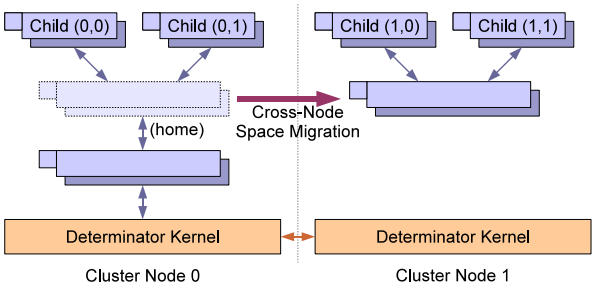

Figure 2: A spaces migrating among two nodes and starting child spaces on each node.

2.5 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 cluster, mostly transparently to application code. While distribution is semantically transparent to applications, we say "mostly transparently" because an application may have to be designed with distribution in mind to achieve acceptable performance. 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 support adds no new system calls or options to the API above. Instead, the Determinator kernel interprets 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 a child on that node specified in the remaining child number bits. Figure 2 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.

3 Emulating High-Level Abstractions

The kernel API described above eliminates many conveniences to which developers and users are accustomed. Can we reproduce them under the constraint of strict determinism? We find that many familiar abstractions remain feasible, although some semantically nondeterministic abstractions may be costly to emulate precisely. This section details the user-level runtime infrastructure we developed to emulate traditional Unix processes, file systems, threads, and synchronization under Determinator.

3.1 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 violating our design principle 2 (Section 2.1). 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 it wishes to terminate, 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 violates principle 3 by waiting for *any* (i.e., "the first") child to terminate. Our kernel's API provides no system call to "wait for any child," and can't (for unprivileged spaces) without violating its determinism guarantee. 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 com-

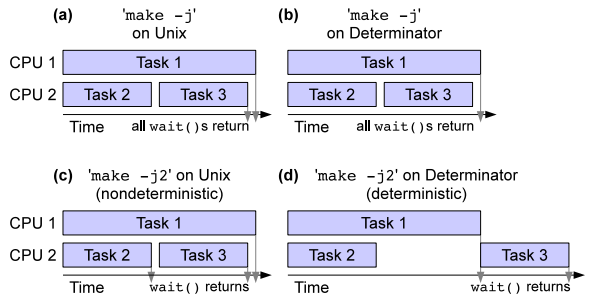

Figure 3: Example parallel `make` scheduling scenarios under Unix versus Determinator: (a) and (b) with unlimited parallelism (no user-level scheduling); (c) and (d) with a "2-worker" quota imposed at user level.

mon uses of `wait()`. First, interactive Unix shells use `wait()` to report when background processes complete; thus, an interactive shell running under Determinator requires special "nondeterminism 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, which violates principle 5.

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 3 (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. This example illustrates the importance of separating scheduling from application logic, as per principle 5.

3.2 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.

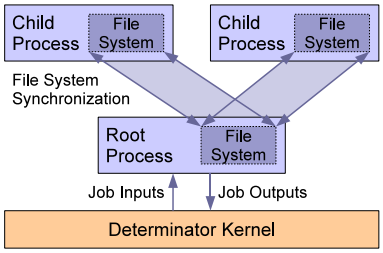

Figure 4: Each process's user-level runtime maintains an individual replica of a logically shared file system, using file versioning to reconcile replicas at synchronization points.

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 4. 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 los-

ing 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.

3.3 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.

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 file system reconciliation mechanism handles "append-only" writes differently from other file changes, enabling processes to write concurrently to the console or to log files without conflict. During reconciliation, if both the parent and child process 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 appends the parent's latest writes to the child's copy. Each process's output file 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.

3.4 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.

As with file systems, there are many approaches to DSM, but ours builds on release-consistent DSM [2, 17], which balances efficiency with programming convenience. Although release consistency normally makes memory access behavior even *less* deterministic by relaxing the rules of sequential consistency, we have adapted it into a memory model we call *deterministic consistency* (DC), which we detail elsewhere [5]. DC's roots lie in early parallel Fortran systems [7,50], in which all processors make private copies of shared data at the beginning of a parallel "for" loop, then read and modify only their private "workspaces" within the loop, and merge their results once all processors complete.

DC propagates memory changes between threads predictably, only at program-defined synchronization points. If one thread executes the assignment '$x = y$' while another concurrently executes '$y = x$', for example, this code yields a nondeterministic data race in standard memory models, but in DC it is race-free and always swaps x with y. DC's semantics might simplify simulations in which threads running in lock-step read and update large arrays in-place, for example. The absence of read/write conflicts in DC also simplifies implementation, eliminating the need to execute parallel sequences speculatively and risk aborting and wasting effort if a dependency is detected, as when deterministically emulating sequential consistency [8, 9, 22].

Our runtime uses the kernel's Snap and Merge operations (Section 2.3) to emulate shared memory with deterministic consistency and "fork/join" thread 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

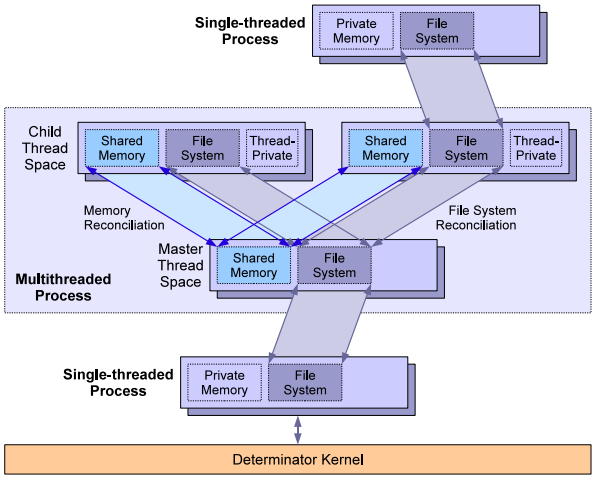

Figure 5: A multithreaded process built from one space per thread, with a master space managing synchronization and memory reconciliation.

that memory state in the child, and start the child running, as illustrated in Figure 5. The master thread may fork multiple children in parallel 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 its shared address space, since the child's snapshot was taken, back into the parent's space. If both the parent and child—or the child and other children whose changes the parent has collected—have concurrently modified the same shared memory byte since the snapshot, the kernel detects and reports this write/write conflict (which is DC's only form of data race).

Our runtime also supports barriers, the foundation of data-parallel programming models like OpenMP [12]. 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 DC conceptually extends to non-hierarchical synchronization patterns as well [5], such as Lisp-style futures [34], our kernel's current strict space hierarchy naturally supports only hierarchical synchronization, a limitation 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. If threads wish to pass pointers to stack-allocated structures, however, then they may locate their stacks in disjoint shared regions. Similarly,

```
md5search(unsigned char *hash, int len, int nthreads)
  char buf[len+1], output[len+1];
  int done = 0, found = 0, i;
  first_string(&buf, len);
  while (!done && !found)
      for (i = 0; i < nthreads; i++)
          next_string(&buf, len, &done);
          if (thread_fork(i) == IN_CHILD)
            check_md5(&buf, hash, &output, &found);
            thread_exit();
      for (i = 0; i < nthreads; i++)
          thread_join(i);
```

Figure 6: Pseudocode for parallel "MD5 cracker."

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 3.2) to synchronize it yields thread-private file tables.

The C pseudocode in Figure 6, a simplified fragment of a brute-force "MD5 cracking" benchmark we use later in Section 5, illustrates two convenient properties of deterministic consistency. First, since threads can have private stacks in overlapping address ranges, `thread_fork()` acts like Unix's process-level `fork()`, cloning the parent's stack into the child, so the program need not separate the child thread's code into a separate function as pthreads requires. Second, the parent thread's `next_string()` call updates *buf* in-place before forking each child, whose "work function" `check_md5()` refers to this buffer. In a nondeterministic thread model, this code contains a data race: the parent may update *buf* for the next child before the previous child has finished reading it. Under Determinator, however, this code is race-free: each child's view of *buf* remains as it was when that child was forked, until the child explicitly calls `thread_exit()`.

3.5 Legacy Synchronization APIs

Although some synchronization abstractions naturally fit a deterministic model, others do not. Mutex locks are semantically nondeterministic: that they guarantee that only one thread may own a lock at once, but allow competing threads to acquire the lock in any order. Condition variables, semaphores, and message queues allow multiple threads to race to signal, post, or send, respectively, and these events wake up any of several waiting or reading threads, violating our principle 3.

For existing sequential code not yet parallelized, we hope this code might be parallelized using naturally deterministic synchronization abstractions like data-parallel programming models such as OpenMP [12] and SHIM [26] provide. For code already parallelized

using nondeterministic synchronization, however, Determinator's runtime can emulate the standard pthreads API via deterministic scheduling [8, 9, 22], at certain costs.

In a process that uses nondeterministic synchronization, the process's initial *master space* never runs application code directly, but instead runs 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"—e.g., number of instructions executed—to schedule thread interactions.

Like DMP [8, 22], 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 2.3). 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 [41]. Like DMP-B [8], our deterministic scheduler implements release consistency [31], 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 is 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 3.1) 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 for 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. If one thread runs '$x = y$' while another runs '$y = x$', the result may be *repeatable* but is no more *predictable* to the programmer than on traditional systems—in contrast with the previous section's multithreading model. 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 heisenbugs.

4 Prototype Implementation

Determinator is implemented in C with small assembly fragments, runs on the 32-bit x86 architecture, and implements the kernel API and user-level runtime facilities described above. Source code is available on request.

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 3.1). 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 2.5. 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.

Implementing instruction limits (Section 2.3) 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 [23]. We first set an *imprecise* hardware performance counter, which unpre-

dictably overshoots its target a small amount, to interrupt the CPU before the desired number of instructions, then run the remaining instructions under debug tracing.

5 Evaluation

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

5.1 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 like standard systems.)

While race detection tools exist [27, 45], we found it convenient that Determinator detects races all the time 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 race 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 [37], 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 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 is 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.

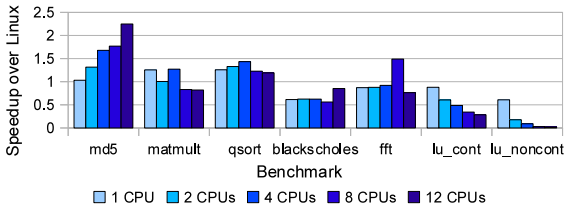

Figure 7: Determinator performance relative to Linux on various parallel benchmarks.

5.2 Single-node Multicore Performance

Since Determinator runs user-level code "natively" on the hardware instead of rewriting user code [8, 22], we expect it to perform comparably to conventional systems when executing single-threaded, compute-bound code. Since space interactions require system calls, context switches, and virtual memory operations, however, we expect determinism to incur a performance cost in proportion to the amount of interaction between spaces.

Figure 7 shows the performance of several shared-memory parallel benchmarks we ported, relative to the same benchmarks running on the 32-bit version of 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* performs 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 [56]. 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 3.5). 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 increas-

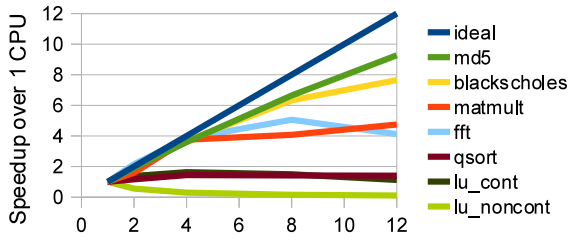

Figure 8: Determinator parallel speedup over single-CPU performance on various benchmarks.

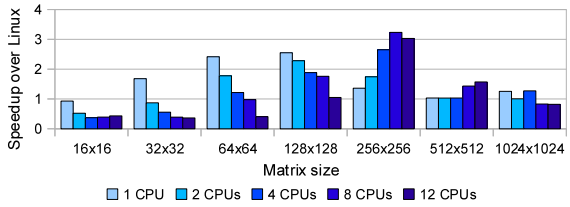

Figure 9: Matrix multiply with varying matrix size.

ing 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 [22].

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.

5.3 Distributed Computing Performance

While Determinator's rudimentary space migration (Section 2.5) 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.

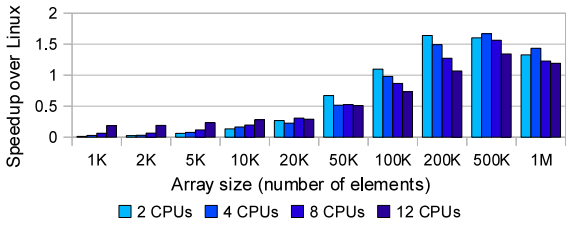

Figure 10: Parallel quicksort with varying array size.

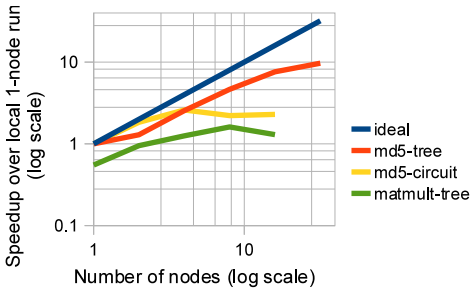

Figure 11: MD5 benchmark on varying-size clusters.

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 shows that 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. The Determinator version of *md5* is 63% the size of the Linux version (62 lines con-

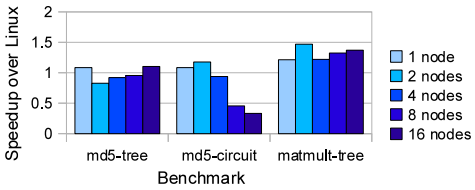

Figure 12: Deterministic, shared-memory MD5 benchmark compared with a nondeterministic, distributed-memory Linux implementation.

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.

taining 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 via TCP.

5.4 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 containing 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.

6 Related Work

The benefits of deterministic programming models are well-known [13, 43]. Recognizing these benefits, parallel languages such as SHIM [25, 26, 52] and DPJ [13, 14] enforce determinism at language level, but cannot run legacy or multi-process parallel code. Race detectors [27, 45] can detect heisenbugs in nondeterministic parallel programs, but may miss heisenbugs resulting from higher-level order dependencies [3]. Language extensions can dynamically check determinism assertions in parallel code [16, 48], but heisenbugs may persist if the programmer omits an assertion. Only a deterministic environment prevents heisenbugs in the first place.

Application-level deterministic schedulers such as DMP [22], Grace [9], and CoreDet [8] instrument an application process to isolate threads' memory accesses, and run the threads on an artificial, deterministic execution schedule. DMP and CoreDet isolate threads via code rewriting, while Grace uses virtual memory techniques as in Determinator. Since these schedulers run in the same process as the application itself, bugs or malicious code can violate determinism by corrupting the scheduler, as the authors acknowledge. Determinator's kernel-enforced model ensures repeatability of arbitrary code in both multithreaded and multi-process computations. Determinator's user-level runtime also develops deterministic versions of OS abstractions such as shared file systems, which lie outside the domain of application-level deterministic schedulers.

DMP and Grace emulate sequential consistency [41] by running parallel tasks speculatively, detecting read/write dependencies between tasks, and re-executing tasks serially on detecting a dependency. DMP-B [8] relaxes memory consistency to optimize parallel execution, but still emulates a nondeterministic programming model where writes propagate between threads at arbitrary points unpredictable to the developer. Determinator combines ideas from early parallel Fortran systems [7, 50] with release consistency [2, 17, 31, 39] to develop a "naturally deterministic" programming model [5]. In this model, read/write conflicts do not exist (only write/write conflicts), and shared memory or file changes propagate among concurrent threads or processes *only* at explicit synchronization points. While focusing on this deterministic programming model, Determinator's runtime can emulate nondeterministic models via deterministic scheduling to run legacy parallel code.

Many techniques are available for logging and replaying nondeterministic events in parallel applications [21, 28, 42, 46]. SMP-ReVirt can log and replay a multiprocessor virtual machine [24], supporting uses such as system-wide intrusion analysis [23, 36] and replay debugging [40]. 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.

Transactional memory (TM) [35, 51] isolate threads' writes from each other between transaction start and commit/abort. TM offers no deterministic ordering between transactions, however: like mutex locks, transactions guarantee only atomicity, not determinism.

7 Conclusion

Determinator is only a first step towards making deterministic execution readily available and broadly usable for normal-case execution of parallel applications. Nevertheless, our experiments suggest that, with appropriate kernel and user-level runtime designs, it is possible to provide system-enforced deterministic execution effi-

ciently at least for coarse-grained parallel applications, both on a single multicore machine and across a cluster.

References

[1] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard, third edition, 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 and B. Ford. Determinating timing channels in statistically multiplexed clouds, Mar. 2010. `http://arxiv.org/abs/1003.5303`.

[5] A. Aviram and B. Ford. Deterministic consistency: A programming model for shared memory parallelism, 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, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, Mar. 2010.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, Oct. 2009.

[10] B. N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, 1995.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[12] O. A. R. Board. OpenMP application program interface version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[13] R. L. Bocchino Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *1st HotPar*. Mar. 2009.

[14] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. Oct. 2009. `http://dpj.cs.uiuc.edu/DPJ/Publications_files/`

[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] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th SOSP*, pages 140–153, Dec. 1999.

[20] 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.

[21] R. S. Curtis and L. D. Wittie. BugNet: A debugging system for parallel programming environments. In *3rd ICDCS*, pages 394–400, Oct. 1982.

[22] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, Mar. 2009.

[23] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th OSDI*, Dec. 2002.

[24] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay for multiprocessor virtual machines. In *VEE*, Mar. 2008.

[25] S. A. Edwards and O. Tardieu. Shim: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, Aug. 2006.

[26] 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.

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

[28] S. I. Feldman and C. B. Brown. IGOR: A system for program debugging via reversible execution. In *Workshop on Parallel & Distributed Debugging*, pages 112–123, May 1988.

[29] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[30] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HotOS-XI*, May 2007.

[31] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th ISCA*, pages 15–26, May 1990.

[32] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium*, 1996.

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

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

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

[36] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104. 2005.

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

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

[39] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA*, pages 13–21, May 1992.

[40] 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.

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

[42] 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.

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

[44] 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.

[45] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, pages 267–280. 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] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *18th ESOP*, Mar. 2009.

[49] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. Technical Report 86-800, Cornell University, Jan. 1990.

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

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

[52] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *EMSOFT*, pages 142–151, Oct. 2006.

[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, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP'03*.

[55] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. *SIGOPS Operating Systems Review*, 17(5), Oct. 1983.

[56] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, June 1995.