Monadic Composition for Deterministic, Parallel Batch Processing

RYAN G. SCOTT, Indiana University, United States
OMAR S. NAVARRO LEIJA, University of Pennsylvania, United States
JOSEPH DEVIETTI, University of Pennsylvania, United States
RYAN R. NEWTON, Indiana University, United States

Achieving determinism on real software systems remains difficult. Even a batch-processing job, whose task is to map input bits to output bits, risks nondeterminism from thread scheduling, system calls, CPU instructions, and leakage of environmental information such as date or CPU model. In this work, we present a system for achieving low-overhead deterministic execution of batch-processing programs that read and write the file system—turning them into *pure functions* on files.

We allow multi-process executions where a permissions system prevents races on the file system. Process separation enables different processes to enforce permissions and enforce determinism using *distinct* mechanisms. Our prototype, DetFlow, allows a statically-typed *coordinator* process to use shared-memory parallelism, as well as invoking process-trees of sandboxed legacy binaries. DetFlow currently implements the coordinator as a Haskell program with a restricted I/O type for its main function: a new monad we call `DetIO`. Legacy binaries launched by the coordinator run concurrently, but internally each process schedules threads sequentially, allowing dynamic determinism-enforcement with predictably low overhead.

We evaluate DetFlow by applying it to bioinformatics data pipelines and software build systems. DetFlow enables determinizing these data-processing workflows by porting a small amount of code to become a statically-typed coordinator. This hybrid approach of static and dynamic determinism enforcement permits freedom where possible but restrictions where necessary.

CCS Concepts: • **Software and its engineering** → **Constraints**; **Concurrent programming structures**;

Additional Key Words and Phrases: deterministic parallelism, Haskell, concurrency

ACM Reference Format:
Ryan G. Scott, Omar S. Navarro Leija, Joseph Devietti, and Ryan R. Newton. 2017. Monadic Composition for Deterministic, Parallel Batch Processing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 73 (October 2017), 26 pages. https://doi.org/10.1145/3133897

1 INTRODUCTION

Using today's technology, can we *guarantee* that a script or program will produce the same output when run on two computers, even when identically configured? Alas, we cannot. Deterministic execution is not an abstraction offered by today's operating systems, virtual machines, or containers, and academic work on deterministic execution either does not cover programs' I/O behavior or cannot run on commodity operating systems.

Authors' addresses: Ryan G. Scott, Indiana University, United States, rgscott@indiana.edu; Omar S. Navarro Leija, University of Pennsylvania, United States, omarsa@cis.upenn.edu; Joseph Devietti, University of Pennsylvania, United States, devietti@cis.upenn.edu; Ryan R. Newton, Indiana University, United States, rrnewton@indiana.edu.

2475-1421/2017/10-ART73
https://doi.org/10.1145/3133897

Nondeterminism poses a fundamental challenge to reproducibility, frustrating software development tasks such as debugging, testing, and reproducing defects from deployed software. The reproducibility that determinism provides is instrumental to trustworthy software compilation and providing repeatable scientific results. While the nondeterminism inherent in parallel computation is well-studied, achieving system-level reproducibility (including I/O) for even single-threaded computations is a real-world challenge. For example, efforts like the Nix package manager [Nix 2015] and the Debian Reproducible Builds project [Debian Wiki 2016] share a goal of reproducible software builds, but have made progress only through extensive human effort to manually identify and remove sources of nondeterminism in the build.

Existing deterministic languages [Bocchino et al. 2009], runtime systems [Merrifield et al. 2015; Olszewski et al. 2009], and hardware architectures [Devietti et al. 2009; Hower et al. 2011] are of little help with these real-world challenges as they operate *within* the process abstraction, focusing only on nondeterminism arising from shared-memory interactions. Deterministic operating systems [Aviram et al. 2010; Bergan et al. 2010; Hunt et al. 2013] can make interprocess communication deterministic, but require a custom OS that is often impractical to deploy. We introduce an alternative approach embodied by a system which we call DetFlow. DetFlow's key contribution is to generalize deterministic guarantees beyond shared memory to include filesystem interactions as well, and to do so while running on existing commodity operating systems like Linux. DetFlow provides a principled mechanism for achieving software reproducibility.

Rather than determinizing arbitrary multithreaded, shared-memory programs as previous work has, DetFlow takes a two-level approach: a parallel **coordinator** allows shared-memory parallelism but is strictly statically typed, whereas optional **worker processes** launched by the coordinator run arbitrary, legacy x86 code. Arbitrary code requires dynamic determinism enforcement, but to keep the overhead minimal and predictable, each individual worker runs *sequentially*. Parallelism is thus supported in two ways: 1) within the coordinator (via multithreading) and 2) via multiple workers running in parallel (via multiprocessing) whose side-effects are dynamically checked to ensure non-interference. Parallelism within the coordinator is limited by the expressivity of static typing. Recent research has shown how pipeline [Bocchino and Adve 2011] and some graph-based algorithms [Kuper et al. 2014a,b] are expressible, but algorithms using schedule-dependent lock-free synchronization would be very challenging to check statically for determinism. Sequential worker execution rules out any intra-worker parallelism, but many real-world parallel batch processing tasks, like software builds, use parallelism only at an outer level (e.g., building multiple source files in parallel) and not within each parallel task (e.g., each source file is compiled sequentially).

DetFlow's current implementation of coordinators leverages Haskell's rich type system to provide parallel access to shared memory and filesystem resources in a safe and principled manner. When these Haskell programs make calls to legacy programs, those programs run deterministically via a lightweight runtime system that enforces deterministic serialization. Thus DetFlow draws on techniques from both deterministic statically-typed languages (LVish, Deterministic Parallel Java, etc.) and deterministic runtime systems, and we find that this hybrid approach is well-suited to executing real-world workflows with both good performance and reliable determinism guarantees.

This paper makes the following contributions:

- The first system to use a hybrid approach of static and dynamic determinism enforcement.
- The `DetIO` monad, the first type for a `main` function which guarantees that everything in `main`, including its I/O operations, executes deterministically (Section 3).
- An implementation of the DetFlow system that can run arbitrary legacy code in a deterministic way, and that leverages several Haskell features, such as the compiler's Safe Haskell

mode, and built-in lightweight threads, to obtain good parallel performance and scalability (Section 4).
- Case studies that apply DetFlow to two different application domains—bioinformatics pipelines and parallel build systems—and demonstrate that DetFlow provides strong determinism guarantees with geomean performance overhead of 0.3% over eight applications (Section 7).

2 BACKGROUND: PORTABILITY AND REPRODUCIBILITY ON X86 LINUX

As computer scientists, we naturally study programs as mathematical entities separate from any machine that happens to run them. Yet real programs running on today's hardware have access to the details of the machine they're running on, even if virtualized. While we may strive to make a program repeatable—yielding the same result across a class of machines—we do so only as a soft project goal. We fail to separate machine-specific from portable (deterministic) information using a hard abstraction boundary—unlike, say, memory isolation between processes, or the user/kernel boundary.

One of the simplest ways to make a program inconsistently reproducible is through concurrency. In concurrent programs, one particular detail of the machine—its scheduling decisions—can leak into the program state and program outputs. Consider this Bash script:

```
#!/bin/bash
echo foo &
echo bar &
wait
```

The above program usually prints `foo bar` on Linux (kernel 4.8), but sometimes it prints in the opposite order. Indeed, concurrency is the cause of many Makefile bugs, which are exposed when running in parallel with `make -j`. We will return to the topic of deterministic software builds in detail (Section 6.1).

It is also easy to expose information about the execution environment: the hardware or the time and place of execution. In this paper we focus on primarily on Linux, and here is a Linux-specific example:

```
#!/bin/bash
date
ifconfig
cat /proc/cpuinfo
```

Of course, we may *need* the ability to gather this information. Unfortunately, contemporary systems not only *allow* it, but treat it no differently from other information in our program. As a result, we can't know whether any given byte of output is tainted by machine specifics or not. If influxes of non-portable, non-repeatable information were at least *identified*, then the body of work on information flow control [Myers 1999; Myers and Liskov 1997, 2000] could be brought to bear.

Indeed, *record-and-replay* software [Devecsery et al. 2014; Mozilla 2015; Patil et al. 2010] is precisely focused on identifying and recording the essential information from an execution in order to reproduce it, separating it from deterministic information that can be recomputed. For example, tools like Mozilla's `rr` make it possible to replay entire process trees when debugging. In fact, deterministic execution and record-and-replay are synergistic—even if a program isn't perfectly deterministic, *increasing* determinism can reduce the amount of information logged. Thus nondeterminism is a spectrum, characterized by the rate at which information must be logged for reproduction. In this paper we focus on the extreme case: computations that process and produce data with zero consumption of nondeterministic information.

Goal. We seek to enable practical software to run with the same outcome irrespective of time and place: i.e. the same printed output and bitwise identical output files. Examples include scripts that run data-processing workloads, and software builds that transform sources into binaries. Further, to achieve reasonable performance on today's multicores, parallelism is a mandatory feature. For instance, inside a build configuration, we may compile source files in parallel:

```
gcc -c foo.c &
gcc -c bar.c &
```

Designing new languages or type systems [Bocchino et al. 2009; Burckhardt et al. 2010; Kuper et al. 2014b] can help the goal of deterministic parallel programming. But here we encounter a limitation: we cannot, for instance, *rewrite GCC* in order to get deterministic builds. Thus, at some level, we must also support call-outs to legacy software.

3 APPROACH

Now we define in more detail what it means for a program to run deterministically in DetFlow. We then describe the programming API exposed by DetFlow. This section covers user-visible features, with the implementation deferred to Sections 4 and 5.

3.1 The Determinism Guarantee

We say that a deterministic program computes a *pure function* from a set of read-only directories on disk, to a set of writable output directories (plus stdout). To simplify the exposition, our examples use a single input and output directory, with the output directory initially empty. Here's an example of how to invoke a portable, deterministic program:

```
$ detflow -i in/ -o out/ Hello.hs
Reading in/hello.txt and writing out/hello.txt.
$ cat out/hello.txt
Hello world
```

The static contents of the program sources—either a single-file script, like `Hello.hs`, or a whole project directory—uniquely identify the pure function in question. This means the program sources must be self-descriptive with respect to exact versions of compiler toolchains and library dependencies. Fortunately, this is a solved problem; we can rely on existing "hermetic build" technologies[1].

The guarantee we provide is that on any x86 Linux machine where we run the above command successfully[2], the same input directory yields the same output directory contents and stdout. Indeed, this notion of observable equivalence can easily be validated using a simple utility such as `hashdeep` or `diff`. We guarantee determinism up to the *bitwise contents of files*, and treat directories with *set semantics*—inodes and the ordering of directory contents are abstracted away in this view. Consequently, programs run with `detflow` must *not* be able to observe true system inode numbers or directory orderings. For example, we sort directory contents by name before returning listings to the user. We likewise hide file timestamps, replacing them with constants, which prevents the program from inferring time information from files it has written.

To an external observer, the above source-only DetFlow program has only *one* valid execution, which is the one defined by the restricted view of the system provided by DetFlow. If the example program called out to legacy binaries (as in Figure 1), then the determinism guarantee holds but

[1]Such as the `stack` tool for Haskell (stackage.org) or the Nix package manager [Nix 2015].

[2]Note that the program may fail to run, for example with an out-of-memory error, which may differ between machines.

```bash
#!/bin/bash

function compile() {
  gcc -c $1
  echo "Done compiling $1"
}

compile foo.c &
compile bar.c
wait
```

```haskell
import Control.Monad.DetIO

compile x =
  do system   ("gcc -c " ++ x)
     putStrLn ("Done compiling " ++ x)
main =
  do a ← forkWPerms [R "foo.c", RW "foo.o"]
           (compile "foo.c")
     compile "bar.c"
     joinThread a
```

Fig. 1. An example: compiling two files in parallel, in Bash (left) and using `DetIO` (right) to achieve statically-checked deterministic parallelism. DetFlow determinizes the program on the left by sequentializing it, whereas the program on the right can run in parallel, with (unobservable) nondeterministic OS scheduling. A configuration file in the same directory would specify the version of the library ecosystem to use, e.g. "lts-8.6".

grows a bit more complicated. First, the program identity depends on read-only system-directories as well as the script source code (system directories which are best controlled with containerization technology). Second, the legacy binaries were originally created with the intent of nondeterministic semantics, based on all the unspecified and under-specified behaviors in the language, the hardware, and POSIX. In this case, DetFlow consistently selects *the same* valid execution among the set of valid executions, on every invocation. For example, because directory order in POSIX file systems is undefined, always choosing an alphabetically sorted order is valid.

3.1.1 Avoiding Overly Fine Equivalence Classes. If a program's exact identity depends on the full state of the disk containing it, then we've erred too far on the side of finely sliced equivalence classes. Programs become unique snowflakes. Instead, our portable programs must *abstract* over aspects of the machine and *ignore* aspects of their environment—that is the essence of their portability.

We will cover implementation details in Section 4, but in broad strokes we use existing containerization approaches (Docker, Nix), or cluster management software, to provide the system image that provides the backdrop for a portable computation. A deterministic starting image requires fixing the contents of system directories, like `/bin` and `/usr`, to which DetFlow grants legacy binaries read-only access. At the same time as we control access to system software, we also *trust* a small set of abstractions to remain backwards compatible. Specifically:

- **The instruction set architecture**. Rather than require an exact match on processor model, we assume ISA backward compatibility and that applications do not query architecture details via `cpuid` instructions.
- **The Linux kernel**. Requiring an exact match on kernel version would decrease usability, so we assume different minor versions of Linux behave the same.
- **The Docker CLI**. A DetFlow program names a precise image as a starting point, but we don't require exactly the same version of the harness software that instantiates the container.
- **The detflow tool itself**. Much like the Docker CLI, upgrading `detflow` itself shouldn't invalidate previous workflow outputs.
- **A build tool**. In our case, we trust a Haskell build tool, `stack`, to maintain backwards compatibility, but we don't trust the compiler—we always build a workflow with a fixed compiler version.

```haskell
duplicate :: String → String
duplicate x = x ++ x

main :: IO ()
main = do x ← getLine
          let y = duplicate x
          putStrLn y

getLine   :: IO String
putStrLn :: String → IO ()
```

(a)

```haskell
import Control.Monad.DetIO

main :: DetIO ()
main = do x ← getLine
          let y = duplicate x
          putStrLn y

getLine   :: DetIO String
putStrLn :: String → DetIO ()
```

(b)

Fig. 2. Simple Haskell programs that use the IO (a) and DetIO (b) monads

DetFlow must ensure the program cannot explicitly *query* these versions and thus create conditional behavior. But DetFlow leaves it up to these trusted abstractions to guarantee equivalent behavior for equivalent executions (equivalence defined variously by instructions retired, system calls, and Haskell source code). Each such choice is discretionary. DetFlow *could* guarantee equivalent outcomes only narrowly, requiring exact matches on the above dimensions. Or future work could emphasize system emulation for greater portability. But we find the current compromise practical—any deviation from our determinism guarantee is a bug in the implementation of one of the above abstractions[3].

3.2 The `DetIO` Monad and Its Operations

We use Haskell to build a restricted programming API that allows parallelism while retaining determinism. This API is showcased in the complete example in Figure 1, and described piece-by-piece in the subsections that follow. The design does not depend deeply on the choice of Haskell. We could create a deterministic version of any language (as discussed in Section 9.1), and expose the same API, but we would need compiler support to aggressively limit the subset of the language permitted. In a purely functional language, we can instead provide a library.

Haskell already uses its type system to strictly partition pure functions and side-effecting ones. A regular Haskell program has an imperative `main` function that serves as the entrypoint for execution. For example, Figure 2a shows a Haskell program which reads a result from standard input, duplicates it, and prints it to standard output. This illustrates several key properties of Haskell programs. One is that side effects are expressed using monadic actions, and moreover, external effects that can affect the state of the system (e.g. the filesystem) are required to live in the `IO` monad. The type of `main`, `IO ()`, indicates that it does not produce a useful result, so a programmer would only ever be interested in running `main` for the sole purpose of causing side effects. In contrast, the `duplicate` function is pure, so its type signature need not involve a monad such as `IO`.

There is a serious flaw, however, with using `IO` as the entrypoint for a deterministic program. `IO` allows unrestricted access to side effects that jeopardize reproducibility, just as in Java or any other language. These effects include concurrency—with no guarantees of avoiding race conditions—and reading from filesystem locations with system-specific contents. For this reason, Haskell's IO type is sometimes nicknamed the "sin bin", and especially in the context of deterministic programming, uncontrolled access to IO is a cardinal sin indeed.

[3]Or it is running up against one of the unsupported features of our runtime encapsulation, described in Section 5.1.

Fortunately, I/O can be reigned in and tamed using techniques from the Haskell programming tradition. Instead of using naked I/O, we define a bespoke `DetIO` type. A simplistic model of `DetIO` is a simple wrapper around `IO`:

```
newtype DetIO a = MkDetIO (IO a)
```

`DetIO`, like `IO`, is also a monad, so `DetIO` computations can be composed just like IO actions can. Moreover, the `DetIO` API functions are designed to look quite similar to their counterparts in `IO`, so in quite a few cases, `DetIO` can serve as a more-or-less drop-in replacement for IO. We take this similarity to its logical conclusion by requiring that the `main` entrypoint for a DetFlow program must live in `DetIO` instead of `IO`, as shown in Figure 2b.

Notice that the code in this `DetIO main` function is almost identical to the one that lived in IO—the only difference is we had to use `DetIO`-capable versions of the `getLine` and `print` functions. The differences between `DetIO` and `IO` are more apparent in the implementations of these functions, which we will cover in some detail in Section 4.

3.3 Basic Capabilities: Files, Printing, and Threads

A `DetIO` computation works its side effects on the world by operating on stdin/stdout and by reading and writing files. The routines for doing this mirror the standard Haskell IO interface :

```
readFile  :: FilePath → DetIO String
writeFile :: FilePath → String → DetIO ()
```

Further, since our goal is to write parallel programs in `DetIO`, `DetIO` also includes the ability to fork threads, where `forkIO x` creates a new thread that executes the monadic action `x`.

```
forkIO     :: DetIO a → DetIO (Thread a)
joinThread :: Thread a → DetIO a
```

Yet as soon as we allow threads, we also allow racing calls to `putStrLn` or `writeFile`! For example, consider the program shown below.

```
main :: DetIO ()
main = do forkIO (do foo ← readFile "foo.txt"
                     if foo == "Hello, World"
                        then putStrLn "A"
                        else putStrLn "B")
          writeFile "foo.txt" "Hello, World"
```

If the file `foo.txt` is empty at the start of the program, then depending on the order in which the main thread and the spawned thread complete, this program will either output `A` or `B`! Following other work on deterministic thread scheduling [Olszewski et al. 2009], we could define a deterministic logical clock that orders these racing operations. But for our current `DetIO` design, we can also do something simpler. First, we define a deterministic ordering for print statements, simply by *lazily* issuing the print statements from a thread only when it is *joined* into the main thread, which issues all prints to stdout. Second, for disk access, we simply focus on *non-interfering* parallelism, where parallel computations are race-free by virtue of operating on disjoint subsets of files. To enforce this disjoint-access dynamically, we need a concept of *per-thread permissions*. Permissions are granted from the parent thread to a child thread that it forks. We currently opt for *explicit* transfer of these permissions, with `forkWPerms :: [PathPerm] → DetIO a → DetIO (Thread a)`:

```
do tid ← forkWPerms perms action -- Specify permissions to grant to child thread.
   joinThread tid                -- We get back the permissions when we join.
  -- Note, forkIO is a special case of forkWPerms, where (forkIO == forkWPerms [])
```

3.4 A Simple Model of File Permissions

How do we specify “perms”, above? We need a model of permissions. Previous work [Ridge et al. 2015] has built a detailed formal characterization of POSIX filesystem specifications and implementations. For DetIO, however, it is more appropriate to build an abstract model of permissions that is simpler than, but consistent with, permissions on POSIX file systems. Namely, we 1) ignore ownership and group, 2) ignore (assume) execute permissions, and 3) model only read ($\mathcal{R}$) or read/write ($\mathcal{RW}$) permissions for each path.

Further, we use a concept of *fractional permissions* [Boyland 2003], where permissions are weighted by rational numbers and may be split into fragments, sent to different threads, and then recombined. The end user never sees fractional permissions but asks only for whole permissions: `forkWPerms [ R "/a"]` or `forkWPerms [ RW "/a"]`. Fractional permissions are part of the internal model used by the thread scheduler. A fractional permission of weight one, written $\mathcal{R}$ `1.0`, represents “full” permission on the path. Reading a file takes a $\mathcal{R}$ or $\mathcal{RW}$ permission of weight greater than zero, but a full $\mathcal{RW}$ `1` permission is necessary to write or delete it.

3.4.1 Abstract Model. A *permission set* is a total function from paths to per-path, *atomic* permission values, each describing the permission on a single file or directory:

$$\textsc{Perms} : \mathit{Path} \longrightarrow \{\mathcal{R}\ q \mid q \in [0,1]\} \cup \{\mathcal{RW}\ q \mid q \in [0,1]\}$$

In order to accumulate the initial permission set for a process, we define a join operation $p_1 \vee p_2$ on permissions which takes the *maximum* permission for each path in p_1 and p_2: $(p_1 \vee p_2)(x) = p_1(x) \vee p_2(x)$. And join on atomic permissions is given by:

$$\begin{aligned}\mathcal{R}\ n \vee \mathcal{R}\ m &= \mathcal{R}\ \max(n,m)\\ \mathcal{R}\ n \vee \mathcal{RW}\ m &= \mathcal{RW}\ \max(n,m)\\ \mathcal{RW}\ n \vee \mathcal{R}\ m &= \mathcal{RW}\ \max(n,m)\\ \mathcal{RW}\ n \vee \mathcal{RW}\ m &= \mathcal{RW}\ \max(n,m)\end{aligned}$$

This semilattice models *stronger* vs *weaker* permissions where a higher point in the lattice allows all executions possible under lower permissions. The lattice is bounded by $\top = (\lambda x. \mathcal{RW}\ 1)$, and $\bot = (\lambda x. \mathcal{R}\ 0)$. (In fact, we could form a full bounded lattice, but we have no need of a meet operation for our purposes.) When the `detflow` harness launches a job with an initial list of permissions, the $\vee$ operation is used to combine those permissions into a single $p \in \textsc{Perms}$, which becomes the initial permission of the main thread in the application.

3.4.2 Permission Transfers at Runtime. The permissions held by a thread change dynamically. Whenever a child thread “checks out” a read permission, we *halve* our permission on that path and all its descendants.

```
do -- Assume we start here with (R /a) at weight 1
   th1 ← forkWPerms [ R "/a" ] comp1 -- Now we have 1/2 weight left on /a
   th2 ← forkWPerms [ R "/a" ] comp2 -- Now we have 1/4 weight left on /a
```

While the user requests qualitative, unweighted permissions, the exact amount of permission they receive at runtime depends on the quantity held by the parent thread. Further, we can never give away more permission than we start with. In fact, the total quantity of permissions available at the start of an application is globally conserved: permissions are passed around but neither created nor destroyed. This can be seen in the below definitions, which return a tuple of child and parent weighting functions.

```
checkoutR(path,f) = ( λp. if under(p,path) then ½ (f p) else 0
                                , λp. if under(p,path) then ½ (f p) else f p )
```

We trivially lift this function on weights to operate on atomic permission values $\mathcal{R}\ q$ or $\mathcal{RW}\ q$, always returning the same kind of permission already held. Likewise, as well as halving permissions, we combine them with addition. In this case, lifting addition to operate on *matching* pairs of $\mathcal{R}/\mathcal{R}$ or $\mathcal{RW}/\mathcal{RW}$ permissions, and lifting it over functions to define $p_1 + p_2$ for $p_1, p_2 \in$ Perms, which forms a monoid.

Gaining permissions at runtime only occurs via an explicit synchronization with another other thread. When one thread *joinThread*s another it waits for the target's termination and then inherits its permissions. But we don't require a strict fork/join structure. Rather, threads are futures, synchronized in an arbitrary directed acyclic graph.

To extend the example above, we can synchronize with the two forked threads.

```
do th1 ← forkWPerms [ R "/a" ] comp1 –– Parent has 1/2 weight on /a
   th2 ← forkWPerms [ R "/a" ] comp2 –– Parent has 1/4 weight on /a
   joinThread th1 –– Parent back up to 3/4, regaining th1's 1/2 permission on /a.
   joinThread th2 –– Back up to weight 1, where we started before forking.
```

A forked thread may only be joined once, because joining linearly passes permissions from the joinee to the joiner waiting for its completion. Two attempts to join the same thread causes the second attempt to raise an exception (deterministically, on every run). An alternate function, `waitThread t`, simply waits for `t` to terminate. Unlike `joinThread`, `waitThread` is idempotent and doesn't affect permissions.

3.4.3 Checking Out Write Permissions. Above, we saw how checking out a read permission takes half the weight under that path. Checking out a write permission gives *exclusive* write access that cannot be shared between threads, thus it takes 100% of the parent's permission share. The core operation on weights becomes

```
checkoutRW(path,f) = ( λp. if under(p,path) then f p else 0
                     , λp. if under(p,path) then 0   else f p )
```

which is lifted to operate on $\mathcal{RW}$ values.

Note that if the parent has already lent out a read permission, it may only have, say 1/2 a permission on the path in question. In this case, the child still takes all of the weight from the parent, but it is insufficient to actually issue write operations on the path.

3.4.4 Creating Files and Adding Them to the Parent Directory. `writeFile "/a/b"` requires $\mathcal{RW}$ `1` on "/a/b", but what if the file doesn't already exist? Then we create the file, which implicitly requires modifying the *containing directory*. In the underlying UNIX permissions system we indeed need write access to the directory for this action, but we can also write multiple files in the same directory *concurrently*. (Recall that we're not concerned with physical ordering or inode numbers.) Thus in our abstract permissions model, we require permission $\mathcal{RW}\ v$, where $v > 0$, in order to create a file in the directory `d`. To conveniently enable file creation we *implicitly* take a half-permission on the enclosing directory when we checkout write access on a file:

```
do –– Assume this thread starts with permission on /d of weight 1
   t1 ← forkWPerms [ RW "/d/a" ] (writeFile "/d/a" str1) –– 1/2 perm on /d
   t2 ← forkWPerms [ RW "/d/b" ] (writeFile "/d/b" str2) –– 1/4 perm on /d
   joinThread t1; joinThread t2 –– Regain full permission.
```

This enables us to conveniently write files in parallel. It also means that if we have zero permission on a directory, we can only *overwrite* existing files in it, not create new ones. This is consistent with

POSIX file systems, since they allow modifying file objects in a directory without any permissions on the containing directory itself. Deleting a directory, on the other hand, requires a full 1 permission on it and every path under it.

3.5 DetIO Complements Existing Haskell Parallel Programming Libraries

A computation in `DetIO` doesn't need to use `forkWPerms` as its sole means creating parallel work. First, it's always possible to call *any* purely functional code from the monadic code, including internal task parallelism [Marlow et al. 2010, 2009]. Second, existing deterministic parallel programming libraries in Haskell [Kuper et al. 2014b; Leijen et al. 2011; Marlow et al. 2011], can, by virtue of their determinism, expose pure interfaces. For example a library exposing a "`Par`" monad, also exposes `runPar :: Par a → a`, making that parallelism library usable anywhere, including inside `DetIO` computations.

3.6 system: Shell Calls

While `DetIO` and Haskell provide the ability to write fully deterministic programs, there are many useful pieces of software which are not written in Haskell that would be too much effort to rewrite. We wish to preserve the ability to use legacy software in while still retaining a determinism guarantee. Indeed, `DetIO` exposes exactly the same interface for calling external processes as the standard Haskell `IO` monad, which, at its simplest, is:

```
main :: DetIO a
main = do system "ls inputs/"   -- Haskell has the usual mix of routines for shell subprocesses,
          system "date"         -- optionally specifying stdin, retrieving stdout, stderr, etc.
          system "echo hello > output/file.txt"
```

The shell calls *inherit* the file access permissions of the calling thread, thus above we require read access to `./inputs` and write access to `file.txt`. In addition to output files, subprocesses produce deterministic output on stderr and stdout. For example, `date` will always print "Wed Dec 31 19:00:00 EST 1969". Retaining the determinism guarantee here is the challenge faced by our determinizing runtime, whose implementation is described in Section 5. In short, our current prototype uses `LD_PRELOAD` to load the `libdet` library into the address space of the subprocess(es). `libdet` restricts application behavior, which means some applications will not work in this mode. However, if they fail, they fail every time. Conversely, compatible applications will produce the same output each time they run to completion.

3.7 Advanced DetIO Capabilities

Two additional capabilities that are useful for the kinds of deterministic scripts we run are: (1) blocking data-flow communication between `DetIO` threads, and (2) network access to fetch source code, data, etc. Both of these are deterministic, or can be made so.

First, *IVars* are write-once variables, with blocking read semantics, that are well suited to deterministic parallel programming [Arvind et al. 1989]. They can be used to construct arbitrary dependence graphs between threads:

```
do v ← new                   -- A fresh, empty IVar.
   forkWPerms [] (put v 3)   -- Asynchronously fill the IVar.
   get v                     -- Block until the child thread writes the IVar.
```

We will use IVars, for instance, to implement the dependencies between build targets in a deterministic version of the `make` program (Section 6.1). One complication of IVars is that they extend the *happens before relation* and thus affect how concurrent print statements are serialized (Section 4.1).

Second, while DetIO does not currently support networked, multi-machine applications, we do support fetching immutable resources from elsewhere on the network. For example, to fetch source code to build:

```
do wget "https://example.com/file-v1.2.3.tar.gz" (SHA256 "123··· def")
```

Here, a hash is provided so that if the external resource changes, this code throws an error. Thus, when successful, the returned result is deterministic, but the application has a new failure mode. In the future, supporting content retrieval from a content-addressable network (like the Interplanetary File System [Benet 2014]), would increase the robustness of such a workflow in the face of an unstable, dynamic World-Wide Web[4]. Of course, for a workload we want to *archive*, prefetching all the bits it needs at runtime is the best solution.

4 IMPLEMENTATION: HASKELL LIBRARY AND DETFLOW HARNESS

The full DetIO monad is implemented as a state monad that stores all the per-thread state. This state monad is a transformer on top of the IO monad. Finally DetIO is wrapped as a newtype, with its constructor hidden, so its contents are private, inaccessible by the end user:

```
newtype DetIO a = DetIO (StateT ThreadState IO a)
```

ThreadState includes a number of per-thread variables, tracking each thread's permissions and pedigree (ancestry).

Any threads that are not joined when the main thread finishes are implicitly waited upon before the application exits. This is necessary for determinism, because terminating without computing *all* the computations may give false confidence that the run succeeded, when in fact an outstanding thread was about to throw an error.

4.1 Pedigree and Deterministic Concurrency

To track the happens-before relation at runtime, we first track the *pedigree* of each computation. This is the index in the *fork tree* for the current thread, i.e., a series of bits indicating whether it was on the child or parent side of each enclosing fork. Disregarding IVars and thread joins, we could define a simple sequential program order based on this, where child threads always complete before their parents continue (Cilk's "serial elision" [Frigo et al. 1998]).

Second, arbitrary synchronizations through thread joins and IVar reads add additional synchronization edges between threads. As in dynamic race detectors [Mellor-Crummey 1991; Raman et al. 2012], we log these synchronizations. Our implementation combines into one log (1) the deferred print statements, and (2) synchronization edges from other threads.

The main thread issues *all* print statements to stdout. Print statements executed by the main thread are issued immediately. When the main thread joins in another thread, it does a backwards traversal through that thread's log, which forms a DAG (directed acyclic graph) of print statements, induced by join edges and IVar write-read pairs. This backwards traversal can stop whenever it hits part of the log that has already been printed. The traversal returns a collection of all print statements that exist logically-before the join in question. All such print statements are ordered in a deterministic topological sort. Aside from respecting edges in the DAG, pedigree is used as a tie breaker to create a unique topological sort.

[4]Breaking upstream package repositories is also a problem faced by Nix. See https://github.com/NixOS/nix/issues/859 for an example.

4.2 Finite, Printable Representations of Permission Sets

While treating $p \in$ Perms permission sets as functions makes the semantics clear, in practice we want to print out a thread's permissions in error messages, and a more transparent representation is preferred. In our DetFlow prototype, we take advantage of the fact that only a finite number of atomic permissions form the basis for a permission set in practice. We represent a permission set as a trie, indexed by each component of a path. We interpret a path in this trie as representing infinite sets of deeper paths containing it as a prefix. For instance, a permission of weight $1/2$ on path `/a/b`, applies to `/a/b/c`, `/a/b/d`, and so on. This representation makes it efficient to check "all paths under p"—as when checking permissions to delete a directory—without specifically querying a permission set function for every extant file under that directory.

4.3 `detflow`: A Harness for Launching Deterministic Programs

As we saw above, we use `detflow` to launch a Haskell program whose `main` function has type `DetIO`. We cannot use the plain Haskell toolchain (i.e. `runghc`), because it of course expects a main function of type `IO`, and we don't want to trust any user code of that type. In order to guarantee the entire application is deterministic, the `detflow` harness generates a short, trusted `IO` wrapper function that sets up and launches the `DetIO` main function. In this generated wrapper, `detflow` initializes the permission of the main thread, based on which directories are marked for reading and writing in `detflow`'s command line arguments.

Further, as described in the next section, the `detflow` harness has the final say as to what Glasgow Haskell Compiler (GHC) options are used, in order to prevent use of unsafe Haskell features anywhere in the application. Finally, as our single entrypoint to running deterministic workflows, `detflow` implements various convenience features, like the ability to launch shell commands directly ("`detflow --runshell cmd`"), launch the workflow inside a given Docker or Nix image (a capability inherited from the "`stack`" tool for Haskell), or run the same source code in a raw, nondeterministic mode without any determinism-related overheads.

4.4 Safe Haskell, Trusted Code Boundaries

In order that `DetIO` be guaranteed deterministic, it is crucial that users cannot promote arbitrary IO effects to `DetIO`, or else `DetIO` becomes just as unreliable as IO. Towards this end, we made `DetIO` an abstract type (not exporting the `MkDetIO` constructor). But adversarial users can circumvent this by using unsafe features of Haskell. For instance, lurking within the standard libraries is the notorious `unsafePerformIO`, which allows the use of arbitrary side effects within "pure" code, ostensibly reserved for those who know what they are doing.

Thankfully, we can safeguard against this using the "Safe Haskell" language extension [Terei et al. 2012]. Safe Haskell was designed for execution of untrusted code received over the network, and permits only a subset of the language in which you can truly "trust the types". Safe Haskell allows users to annotate modules as `Unsafe`, `Trustworthy`, or `Safe`. `Safe` code encapsulates most code in the Haskell ecosystem: machine-checkable as safe by construction. The `Unsafe` label is intended for code that can be used to circumvent type safety or module abstraction, while `Trustworthy` is intended for code written by trusted experts which may *internally* use unsafe features, but exposes only a safe interface. In order to be marked as `Trustworthy`, a module's package must be on a whitelist of trusted packages by trusted authors. In order for a module to be marked as `Safe`, every module that it imports, transitively, must either be `Trustworthy` or `Safe`.

By default, `detflow` allows *only* `Safe` code in the deterministic applications it launches. This prevents ordinary DetFlow users from compromising the determinism of their programs, while still allowing advanced library authors to extend DetFlow in a `Trustworthy` fashion.

5 IMPLEMENTATION, LIBDET: A DETERMINIZING RUNTIME

To determinize x86 software, we must intercept nondeterministic system calls, nondeterministic instructions, and data races on shared memory. We aim to support realistic existing software, but it is not our goal to handle *adversarial* software. To do so would require heavy-weight solutions such as a new hypervisor or OS modifications; unfortunately, in user-space, even today's state-of-the-art, high-overhead binary instrumentation systems such as Intel Pin [Hazelwood et al. 2009] and Valgrind [Nethercote and Seward 2007] cannot keep adversarial programs from "breaking out of jail" [Sun et al. 2016].

For a lightweight, user-space solution on Linux, we use LD_PRELOAD to inject our own wrapper library, `libdet`. We intercept and determinize a variety of nondeterministic functions in the C standard library libc. We group functions into the following categories:

- **Time queries**: Calls to gather information about current time, elapsed clock cycles, etc. This also determinizes pseudorandom number generation by determinizing its source of seeds.
- **Filesystem access**: The DetFlow Haskell process sets environment variables containing the list of paths the subprocess is allowed to read from and write to. When the user tries to open a file for reading or writing, we verify the user has permissions to access the file. This catches nondeterminism from directories like `/dev/urandom` and `/proc/`. (Of course, we also intercept attempts to change environment variables, to prevent a backdoor where the user changes their own permissions.)
- **Concurrency**: We impose an ordering on threads and processes by intercepting calls like `fork` and `pthread_create`. We force the parent to wait until the child exits. This is our main limitation on the applications we can currently support—they must run with sequential semantics.
- **Unique IDs**: Linux uses nondeterministic unique identifiers like `pids` and `inodes` for book-keeping. `libdet` replaces these unique IDs with deterministic virtual IDs. The client program is given virtual IDs, and `libdet` translates virtual IDs to real ones when interacting with the OS.
- **File metadata**: The filesystem provides a wealth of nondeterministic metadata for each file: time of last access, number of hardlinks, etc. We force the `stat` system call to return dummy constant values instead.

Finally, we also disable address space randomization (ASLR) for the child processes of `DetIO` programs (but not the Haskell `DetIO` programs themselves). We currently rely on deterministic Linux `mmap` behavior for sequential programs, rather than intercepting `mmap` and enforcing our own strategy. We do not determinize the full contents of process memory, because the `libdet` library itself stores nondeterministic state inside the process (i.e. the true inodes and PIDs being virtualized), but, again, in the current prototype we assume the determinized process is not adversarially trying to circumvent `libdet`.

5.1 Limitations of libdet Runtime Determinization

While `libdet` provides a straightforward and lightweight approach to runtime determinism enforcement, it has several limitations which we discuss here. First, `libdet`'s interception of system calls is incomplete, and it cannot track calls made through inline assembly, or by statically linking against libc. Second, there are nondeterministic CPU instructions we fail to catch, such as RDRAND which returns hardware-generated random numbers. In future, we could scan the instruction stream and catch all these instructions dynamically, or statically rewrite the binary to avoid them. Finally, we use a simple method to handle concurrency. Since threads in `libdet` run sequentially, some programs will deadlock if they use spin-waiting. We plan to use a more general deterministic runtime system [Kai Lu et al. 2014; Liu et al. 2011; Merrifield et al. 2015; Merrifield and Eriksson 2013] to enable true concurrency in future work.

6 CASE STUDIES

6.1 Case Study 1: Parallel Software Builds with detmake

To exercise DetFlow's capabilities, we developed a clone of the GNU make build tool using DetFlow, which we have named detmake. Like make, detmake is capable of reading a Makefile which specifies one or more targets to be run, and runs the target of the user's choice. Targets are allowed to declare dependencies, which indicate that a file must exist (if declaring a dependency on a file) or that a target must have completed (if declaring a dependency on another target) before running the dependent target.

detmake is an ideal use case for DetFlow for several reasons. Makefile dependence graphs expose natural parallelism, as non-dependent targets can be run in parallel with relative ease. Parallel make is not trivial, however, as Makefile dependencies are not always well specified. As a simple example, consider this hypothetical Makefile with two targets:

```
all: create-bindir install-exec-local
install-exec-local:
        cd $(DESTDIR)/$(bindir) && cmd
create-bindir:
        mkdir -p $(DESTDIR)/$(bindir)
```

Upon a quick glance, it appears that this install-exec-local target has no dependencies. In practice, however, this is not true! This target's command will only succeed if the directory located at $(DESTDIR)/$(bindir) exists. But notice that the create-bindir target itself creates $(DESTDIR)/$(bindir), so install-exec-local can succeed or fail depending on whether create-bindir was run before it! This is an implicit race condition, and one that is not too uncommon in Makefiles, leading to apprehension about activating make -j with unfamiliar codebases.

Happily, reimplementing make with DetFlow forces one to fix these race conditions. Due to DetFlow's path permissions, one cannot simply read or write from files with wild abandon, as DetFlow forces you to request permissions for files up front. Fixing a Makefile to declare its dependencies properly means that running the targets in parallel will not race. We designed detmake to be parallel by default, although the number of threads can be controlled in the usual way with -j1,-j2, etc to manage memory usage and processor oversubscription. Irrespective of -j setting, however, the terminal and disk output from the parallel build is always identical.

We tested detmake on various Makefiles, and measured its performance (Section 7.3). One Makefile in particular, for RAxML [Stamatakis 2014]—a bioinformatics tool for maximum-likelihood based phylogenetic inference—stood out, as RAxML had a concurrency-unsafe Makefile! Running make in parallel (using GNU make -j) resulted in errors, whereas running sequentially it worked without issue. The problem was undeclared dependencies which *happened* to be satisfied in the sequential build. Running under detmake forced these errors to the forefront, as we cannot build successfully without fixing them.

6.2 Case Study 2: Data Processing, Bioinformatics

DetFlow targets batch software that processes data on disk. Scientific data-processing workloads are a paradigmatic example. In biology, for instance, researchers frequently build scripts or workflows that invoke one or more programs to process data in simple textual formats: genes and genomes, raw sequencing reads, phylogenetic trees, and so on. To conduct this case study, we interviewed a microbiologist to inquire what programs they use frequently, arriving at the following sample list: **RAxML** [Stamatakis 2014]: maximum-likelihood based phylogenetic inference (deriving likely ancestry trees); **Clustal** [Chenna et al. 2003]: multiple sequence alignment; **mothur** [Schloss et al. 2009]: various tools for the microbial ecology community; **BWA** [Li and Durbin 2010]: Burrows-Wheeler aligner, aligning sequences against a reference genome; and **HMMER** [Eddy 1998]: searching for sequence homologs and making sequence alignments (using hidden Markov models).

All of these programs are written in C/C++. For each of these programs, we run them on a sample dataset: both natively and then under determinism enforcement. We write a `DetIO` script to drive the execution of the software. Recall that in DetFlow, foreign subprocesses are sequentialized, with the parallelism only at the script level. Fortunately, bioinformatics applications frequently work on a large directory containing many individual input files. Thus we write a small, statically typed `DetIO` script that invokes the program in parallel on each input file. While the particulars vary—in directory layout and setup—this script always contains roughly the following logic:

```
...
thrds ← forM inputFiles (λ inFile → do
  let outFile = changeExtension inFile
  forkWPerms [ R inFile, W outFile ] (do
    putStrLn ("Processing file "++show inFile)
    system ("./bin/prog "++inFile++" "++outFile)))
...
mapM joinThread thrds
```

Above, we have a fork-join parallel region executing an external program for each input, producing disjoint outputs. The basic proposition of `DetIO` is to ensure that the above scripts produce deterministic printed output and disk output, in spite of running in a nondeterministic internal scheduling order. However, because of the restrictions DetFlow puts on legacy programs, not all legacy software will work. (Yet at least the program will fail reliably if it is incompatible!) In this section, we document our experiences with running the five programs above via the `detflow` wrapper. Four out of the five programs run under DetFlow at the time of this writing. Further, we find that we learn a lot about unexpected program behaviors by *attempting* to run them deterministically.

RAxML and Clustal. We built RAxML 8.2.10 with AVX support, and Clustal 2.1 (in `-ALIGN` or `-BOOTSTRAP` mode), which all ran without incident under DetFlow.

mothur. We ran the `make.contigs` mode of mothur 1.39.5. Our first attempt to run under DetFlow revealed that multiple mothur invocations in the same working directory have an I/O race. They both attempt to write the same log file. In fact, even if the log file name is specified by the user, mothur writes it at an uncontrollable location in the current directory, and then *moves* the log into place after. When running natively, this results in no errors, but in garbled logs that mix output from multiple runs. Running under `DetIO` instead results in a permissions error when the program attempts to write a log outside of its designated output directory.

Attempting to `forkWPerms` and pass permission to the logfile would not work—the permissions system prohibits shared write-access between threads. Instead, we apply the easy fix of running different mothur invocations in separate working directories within the output space: a trivial fix, but one we wouldn't have known to make if not for determinism enforcement.

Finally, mothur implements *internal parallelism* with each call to the mothur binary. However, mothur's parallelism works with DetFlow's sequential semantics when passed the `processors=1` option.

BWA. When we compiled the latest version of BWA (0.7.15), we found two ways that it bumps up against the restrictions enforced by DetFlow. First, BWA caches files in memory using `/dev/shm`, to be shared *between* different BWA processes. Second, BWA uses a hand-coded thread pool (rather than Cilk, OpenMP, etc). Unfortunately, even when run with a 1-thread setting, the thread-pool requires concurrency (condition variables are used to communicate between threads) and deadlocks if run with sequential semantics.

By modifying the source code, it is trivial to remove the `/dev/shm` optimization, and somewhat less trivial to remove the thread-pool implementation to implement a sequential version of the program. We found that both these features were recent additions. Thus, rather than patch the software ourselves, we rolled back to an earlier minor version, `0.7.10` which ran under DetFlow without incident.

HMMER. HMMER was another program with an internal thread-pool implementation. We found that this thread-pool library, even with one worker thread, deadlocks (deterministically) under DetFlow. The source code uses pthreads in several places, and we have not yet made a patched version of HMMER which executes with sequential semantics. Thus we include HMMER only as a negative result.

7 PERFORMANCE EVALUATION

In this section we evaluate the performance of DetFlow on the parallel builds and bioinformatics workloads of the previous section. We measure the overhead of determinization. For this purpose, we introduce a `detflow --nondet` mode, which executes the same code, but without determinism enforcement, i.e. it simply does not buffer prints, and does not load `libdet.so`. As executing deterministically under unmodified Linux is a novel capability, we lack a direct point of comparison to previous works. (For example, running under DThreads [Liu et al. 2011] would not determinize I/O, and would add little to our legacy shell-outs, which are already sequentialized.)

Record-and-replay frameworks, however, are much more mature and widely deployed than their a-priori-deterministic execution counterparts. In debugging, record-and-replay and determinism offer similar benefits. Thus we compare DetFlow's performance against Mozilla's Record and Replay Framework (rr) [Mozilla 2015], which uses a similar approach of intercepting nondeterministic constructs in user space.

While our system supports running against a fixed software image with a `--docker` or `--nix` flag, all experiments in this section are run without containerization on a cluster of 16 identically configured Ubuntu 14.04 machines (Linux 13.19.0-28), with two Xeon E5-2670 CPUs each, at 2.6GHz with hyperthreading disabled. In this case, cluster management software ensures the identity of `/usr` on these machines, rather than containerization. In all places we compile C code, we use GCC 4.8.4.

7.1 Microbenchmarks

Before looking at full application benchmarks, we first characterize the basic overheads of DetFlow.

7.1.1 Deterministic Printing. Because DetFlow defers print statements to achieve deterministic output, there is some extra overhead per print statement. An application that produces high-rate output on stdout would run into these overheads. In Figure 3, we compare the throughput of printing in deterministic and nondeterministic modes, varying numbers of threads. All threads print to stdout in a tight loop. We use the standard `criterion` library for benchmarking, which, rather than running a fixed iteration count or benchmark duration, varies the number of print statements (on all threads), and computes a linear regression over many executions, relating time and number of prints, but discounting constant overhead from, e.g., forking and joining threads.

The cost of an individual print call, on an individual thread, varies from 585 *ns* (R^2 = 97.5%) at one thread, to 730 μs (R^2 = 95%) at 16 threads, for the native Haskell `Data.Text.putStrLn` function. Concurrently printing to the screen is a fairly noisy process to measure, which results in some outliers, and a somewhat reduced R^2 goodness-of-fit in the linear regression. Deterministic printing from non-main threads is slower when uncontended, by almost a factor of 5: 2.66 μs (R^2 = 97.5%) at one thread, and then 30.6 μs (R^2 = 95%) at 16 threads. In Figure 3, we always print from a child worker thread created by `forkIO`/`forkWPerms`. Thus the output must always be buffered. Conversely, if we printed from the main `DetIO` thread only, there is **no overhead**, and we would degenerate to the native `putStrLn` performance. Even with deferred printing, however, there is an advantage at higher contention levels. When all threads in the system are trying to print, not only do they produce garbled, interleaved output, but they also slow down through contention on handles[5]. Here deferring prints to the main thread is an incidental optimization (essentially, the standard elimination-trees trick from concurrent data structure literature [Shavit and Touitou 1995]).

7.1.2 Subprocess Creation. The other place `DetIO` adds overhead is subprocess creation. In this benchmark, each subprocess reads from a string for `stdin`, and returns its `stdout` in another string (Haskell's `System.Process.readProcess`). When running in its default, deterministic mode, DetFlow spends extra time setting up environment variables and even checking the disk[6]. We also wrap each subprocess in a call to `setarch` to disable ASLR, which means we double the number of subprocesses to add this indirection[7]. Finally, when the subprocess starts, it spends extra time initializing and then tearing down the `libdet` library, even if it does no other work.

The end result, pictured in Figure 4, is that determinism increases the cost of calling a subprocess that immediately returns by about a factor of four. At one thread, the native subprocess calls take 929 μs (R^2 = 99%), whereas the determinized calls take 4.05 *ms* (R^2 = 99.9%).

Finally, in the right half of Figure 4 we hit scalability limits; at higher thread counts we are essentially process-bombing the system. Linux scales well but not perfectly: a simple C program forking these processes would achieve a 10.7× increase in processes forked when using all 16 cores of our test platform. Our Haskell programs, on the other hand, have a more severe scalability problem, where at higher thread counts we spend more and more time in Haskell's GC (spending 40% of time in the GC at 16 threads). It turns out that Haskell performs substantial heap allocation on subprocess creation, but this only becomes an issue if we're forking thousands of subprocesses, as in this microbenchmark. However, for most applications with coarser-grained subprocesses, both per-subprocess overheads and scalability limits do not significantly affect application performance.

7.2 Bioinformatics Applications

We benchmark the four bioinformatics applications described in Section 6.2. Each application runs 9 times, and we report the median end-to-end execution time for the batch job. Because `DetIO` scripts are Haskell programs, they can be run interpreted or compiled. We precompile all scripts before beginning our benchmarking trials.

For RAxML and Clustal we use a sample input dataset consisting of 905 genes (7.4MB FASTA format) drawn from a set of orthologous Wolbachia proteins. For BWA, we use 102MB sequencing data in 32 paired files: Drosophila melanogaster genes sequenced from flies infected with Wolbachia pipientis. For Mothur, we use the SOP sample dataset provided on the software's homepage,

[5]We find the noisy timings under contention are exacerbated on multi-socket NUMA systems, like our evaluation platform, and less of a problem on one-socket Linux machines.

[6]For a pregenerated randomness substitute on the disk, which we use in place of `/dev/urandom`.

[7]This could be optimized by disabling ASLR once for the entire Haskell process and its descendants. But it is important to us to that the Haskell process *cannot* leak randomness from its address space.

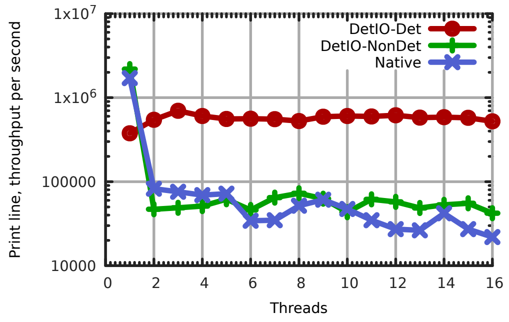

Fig. 3. Per-second throughput in printing “Hello, World!”. Standard output is redirected to `/dev/null`. Using nondeterministic methods causes throughput to start higher, but fall due to contention.

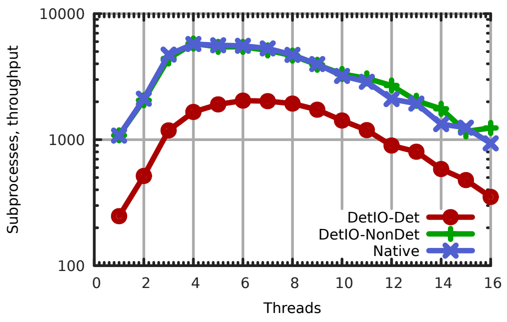

Fig. 4. Per-second throughput in creating (almost) empty processes. Each subprocess is a call to an 8K ELF binary produced with `gcc -O3` and a `main()` function that immediately returns.

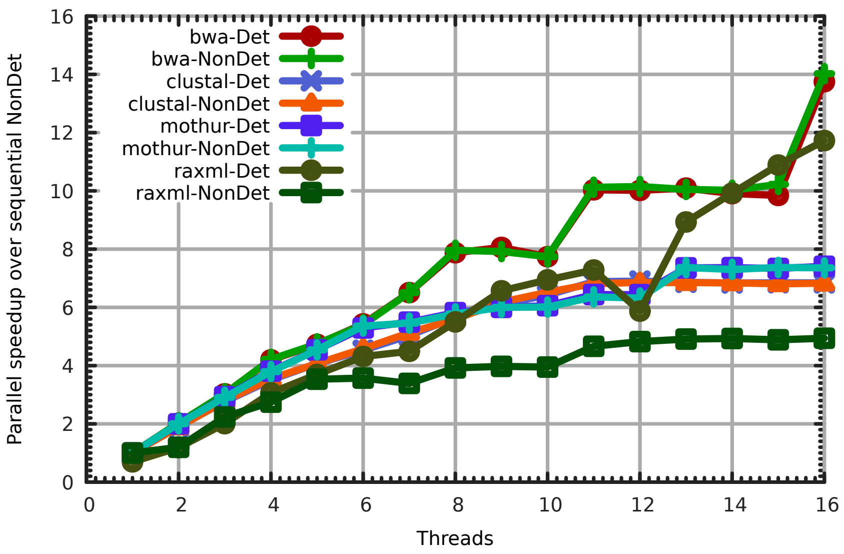

Fig. 5. Parallel speedup of bioinformatics applications. All run via a Haskell `DetIO` script, with or without determinism enforcement. All points are normalized against the non-deterministic, single-thread execution time.

containing a 163MB directory with 20 separate input files. Several of these datasets provide a relatively small numbers of input files, limiting the amount of data parallelism available for load balancing. (In mothur, for example, there is no good way to execute 20 jobs on 16 processors.) If running these applications on much larger data sets, then we would expect more abundant parallelism and better load balancing.

We vary the number of threads used from 1 to 16. Figure 5 and Table 1 show the results. First, the applications we start with demonstrate various degrees of parallel scalability, and we can see that scalability is undiminished by determinism enforcement. Overall slowdown for determinism

Table 1. The overhead incurred by running bioinformatics applications through DetFlow. The baseline is DetFlow's `--nondet` mode. Also reported is the maximum self-speedup attained from running in parallel, and overhead incurred by running through rr. rr results are normalized to running through `detmake` in `--nondet` mode on one thread.

Name	Sequential time (sec)		Geomean overhead	Self-speedup		rr overhead	
	NonDet	Det	(all threads)	NonDet	Det	record	replay
`bwa`	68.855	69.163	0.587%	14.160×	14.223×	1.313×	1.790×
`clustal`	64.128	64.525	0.254%	6.903×	6.945×	1.143×	2.021×
`mothur`	140.672	140.385	-0.370%	7.410×	7.395×	1.565×	1.945×
`raxml`	32.693	47.012	-27.545%	7.320×	16.866×	2.368×	2.638×

enforcement is less than 1% per application, with the geomean taken across thread settings. Nevertheless, raxml is a mixed case, as it slows down significantly (44%) when determinized at one thread, but runs *faster* at higher thread counts. The cause is that `raxml` produces an excessive amount of stdout (without a flag to turn it off); thus it runs up against the same issue as in Figure 3—poor scalability of excessive pipe communication. As a result, the determinized version of `raxml` achieves a better parallel speedup than the nondeterministic version, which spends much of its time on this IO and achieves the lowest scalability.

Table 1 shows the baseline execution times for these applications, which run long enough to compensate for common system-level sources of noise. The right two columns of Table 1 show overhead when running under Mozilla `rr` to achieve reproducibility, as an alternative to `libdet`. We show the application slowdown both when recording nondeterminism and replaying it. Both modes are sequential in `rr`, and here we run the entire script compiled by DetFlow, under a single `rr` recording (rather than a separate recording per shell call). In contrast to DetFlow's overheads, `rr` incurs overheads of up to 2.64×.

7.2.1 Opportunity Cost of Sacrificing Nested Parallelism. The hybrid determinism strategy doesn't *require* that we sequentialize shell calls to legacy programs, but it simplifies the implementation and makes overheads more predictable (than in multithreaded determinization). One downside of sequentialization, however, is that we miss opportunities to employ *nested parallelism*, where both the coordinator and its external shell calls are parallel (outer and inner loops). This can be substantial for some applications, especially because our applications spends substantial time reading from disk. Simply running one job per core does not always saturate the machine, as we can see in the table to the right. In this table we run Mothur with exactly 16 input files, and we run it in `detflow --nondet` mode so as to enable Mothur-internal parallelism. Here, 1600% CPU would represent full utilization by a CPU-bound workload, but running 16 DetFlow threads only runs at 648% CPU utilization over the whole execution. We see that parallelizing exclusively the inner loop in Mothur is more effective, and parallelizing at *both* levels (oversubscribing the machine) is most effective. However, simply increasing the threads used by the outer loop (we ran up to 32, for 2× oversubscription), did not increase throughput. This provides motivation for future work to either (1) better refactor applications to be parallelized externally, or

Outer Threads	Inner Threads	Seconds	% CPU
1	16	15.73	859%
2	8	10.79	1192%
4	4	11.28	1133%
8	2	13.35	954%
16	1	19.26	648%
16	4	10.37	1244%
16	16	9.71	1376%

(2) add deterministic multithreading support, and deal with the consequences of less predictable overhead.

7.3 Deterministic Make

We benchmark `detmake` by using it to compile and link software. Like all DetFlow programs, `detmake` is written in Haskell using `DetIO`. It doesn't achieve perfect compatibility with GNU make, but it does support the most commonly used Makefile features: variable expansion, wildcard patterns, etc. As benchmarks, we select programs drawn from the SPLASH2 collection, but recall that here we time *compiling* the software rather than running it. In Figure 6, we can see that the modest size and degree of parallelism in these builds makes parallel speedup less than perfectly smooth. However, `detmake` is able to keep up with GNU make in performance, and, moreover, the additional overhead in determinization is negligible. For `barnes`, the overhead is 0.384%; for `ocean_cp`, 2.576%; for `raytrace`, 7.514%; and for `fft`, 2.33%. The overheads look larger than they actually are, as the sub-second execution times when running on many threads amplify even the smallest differences in runtime when determinized.

8 RELATED WORK

The most closely-related work to DetFlow are deterministic operating systems like Determinator [Aviram et al. 2010] and dOS [Bergan et al. 2010] that provide determinism for a process, or a set of processes, along with the filesystem and IPC mechanisms like pipes. DDOS [Hunt et al. 2013] extends the dOS system to a local network of machines. While these systems provide strong deterministic guarantees, they require a custom operating system which is a highly invasive change.

Many other deterministic systems focus solely on making shared memory interactions deterministic. We divide this work into two camps: programming languages that enforce deterministic parallelism, and runtime systems that do so. Many of the deterministic languages are extensions to or libraries for Haskell, such as Data Parallel Haskell [Chakravarty et al. 2007], Evaluation Strategies [Marlow et al. 2010], monad-par [Marlow et al. 2011], LVish [Kuper et al. 2014b], and Concurrent Revisions [Leijen et al. 2011]. Concurrent Revisions, like Determinator, takes the view that each thread or task logically copies the entire heap, with changes reconciled at control-flow join points. Outside of Haskell, several approaches to deterministic parallelism have also been proposed, including the NESL data-parallel functional language [Blelloch 1992], stream-based programming models [Thies et al. 2002], type-and-effect systems for imperative languages like Java [Bocchino et al. 2009], and a deterministic version of the Galois system for task-based parallelism [Nguyen et al. 2014].

There are also many runtime systems that focus on making programs deterministic in a language-agnostic way. Some focus on programs without data races [Olszewski et al. 2009], while others focus on making even racy programs deterministic via hardware support [Devietti et al. 2009, 2011; Hower et al. 2011] or purely in software [Kai Lu et al. 2014; Liu et al. 2011; Merrifield et al. 2015; Merrifield and Eriksson 2013].

9 DISCUSSION AND FUTURE WORK

9.1 Implementation in Other Languages

Implementing a framework like DetFlow does not require a purely functional language like Haskell as the coordinator language. On the other hand, the more a language can reduce its “nondeterministic surface area”, exposed by its language features and libraries, the easier it becomes to support DetFlow's capabilities. Implementing a deterministic mode for a given programming language can be a matter of subsetting the allowed language features, or in some cases it is sufficient

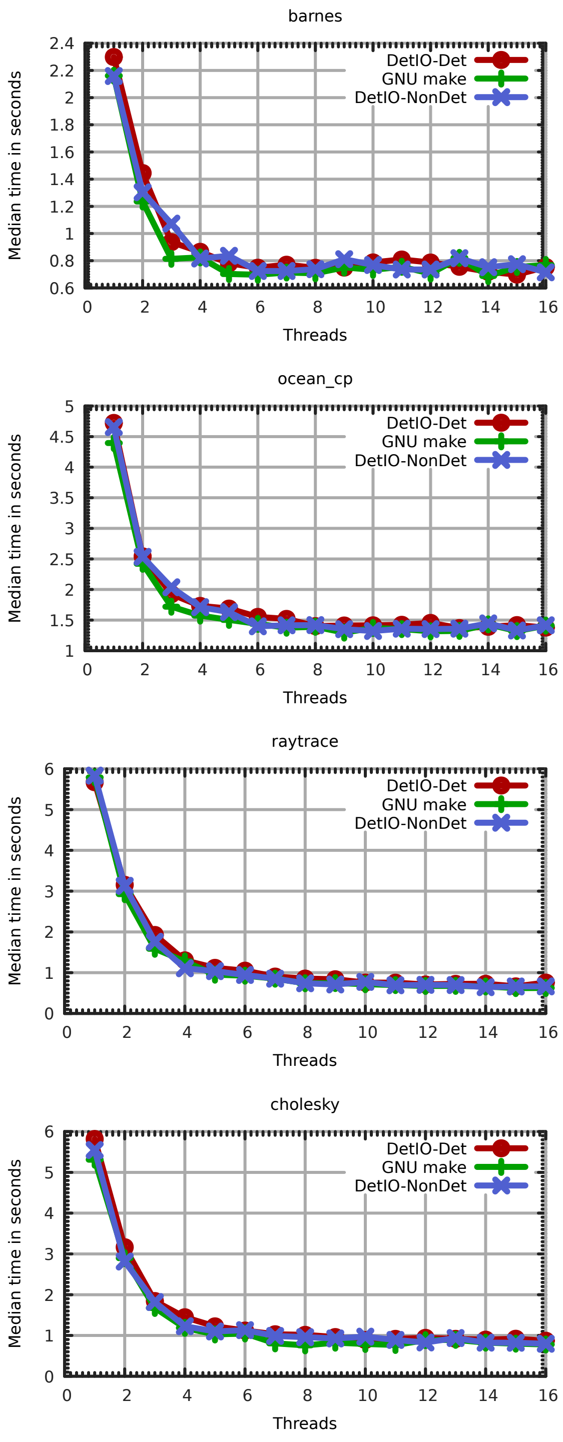

Fig. 6. Median times in seconds to complete software builds with Makefiles.

to limit the libraries the programmer may access. The key goal is to limit the combination of side effects and threads. For example, in Standard ML, impure features are merely a matter of libraries, so without access to mutable data structures (and I/O) the language becomes purely functional, and parallelism (as in MultiMLton, Manticore, and other implementations) doesn't threaten determinism. Conversely, in imperative languages it is easier to achieve determinism by disabling threads, but leaving imperative features alone. Consider, for example, a subset of JavaScript—which lacks shared memory concurrency anyway. In this scenario, if we want to enable deterministic parallel batch processing we need a way of expressing parallel compositions of shell calls *without* using threads inside the coordinator language.

Finally, whatever restrictions determinize the coordinator language, to give them teeth there must be a strict, static means of enforcing these restrictions, such as a special compiler mode. Ideal would be a fine-grained notion of safety and trust such as Safe Haskell provides. However, few compilers track safety in this way. The research language Modula3 is perhaps the closest fit, as it supports marking modules as safe and unsafe, with safe being the default [Cardelli et al. 1989]. Some languages have already moved in the direction of safe subsets, with examples including the Secure EcmaScript [Taly et al. 2011] and ADsafe [Crockford 2008] subsets of JavaScript.

9.2 Quasi-determinism and Uncatchable Exceptions

Ultimately, a machine running a deterministic computation may fail: out of memory, hardware errors, etc. Thus it is best in practice to think of a `detflow` invocation as *quasideterministic* [Kuper et al. 2014b]: there is only a unique (successful) return value, but the job may fail and not return any result. Fortunately, it can be retried, or run on another machine, etc, until it does produce a result. Given the reality of quasideterministic behavior, we have elected to also allow certain nondeterminism in the exact output of *failed* runs. First, our host language for DetFlow, Haskell, already has imprecise exception semantics [Peyton Jones et al. 1999]—if a program is definitely to throw an exception, *which* exception it throws may be indeterminate, for example, if two parallel computations *both* throw an uncaught exception. Indeed, for this reason Haskell exceptions can only be caught in the `IO` monad, where nondeterminism is permitted. As a corollary, `DetIO` may *not* catch Haskell exceptions. Thus our philosophy is to "let it fail". Likewise, there is a variety of error behavior (failed subprocesses, signals), that we do not attempt to determinize currently, so `libdet` turns these exceptional circumstances into failures of the whole `detflow` job. In the future, we could aim to determinize a higher percentage of subprocess failures, but if the subprocess fails due to out-of-memory, or cosmic rays, then we *must not* be able to recover from that failure inside a deterministic job, because that recovery itself would be nondeterministic (at least without a strong proof of observational equivalence between the program and its recovery strategy).

9.3 Variable-Strictness Determinism Enforcement

In this paper, we have introduced a user-space method for determinism enforcement that leaves open some holes that would be exploitable by adversarial programs. Indeed, there are a spectrum of implementation options creating a trade-off between overhead and the robustness of determinization. We could go further, and use a ptrace-based approach similar to Mozilla rr, which would shrink our attack area but not yield a 100% secure guarantee. One avenue of future work is to design hypervisors that can close all the remaining holes through which an adversary can sneak. In fact, there are several implementation technologies, each with different uses along the overhead/strictness spectrum:

- **Ptrace**: Ptrace is a system call for observing and controlling the execution of another process. It is primarily used for breakpoint debugging and system call tracing [Linux 2015]. Ptrace can enforce runtime determinism by intercepting system calls made by the tracee. This method incurs higher overheads as all system calls made by the tracee will send a `SIGTRAP` signal to the ptrace tracer.

- **Hypervisor**: A process running inside a lightweight hypervisor can be containerized in a deterministic environment and have all its system calls determinized. While providing strong determinism guarantees, we expect significant virtualization overhead. Moreover, retrofitting a hypervisor to enforce determinism would require significant effort.
- **Kernel Modules**: Modern OSs allow extensions to the kernel through modules [Salzman 2009]. A kernel module can intercept all system calls to enforce determinism with little overhead. This approach involves subtle kernel programming where errors can crash the system or create security vulnerabilities. Furthermore, the user needs superuser access to add kernel modules and must trust this code to be error free.
- **Compiler Pass**: Whenever the source code is available, a compilation pass can scan the low-level target language (e.g. assembly, LLVM IR) for nondeterministic instructions and system calls and rewrite them to be deterministic. Then, the binary is linked against a deterministic runtime similar to `libdet`. This is the approach take by [Mashtizadeh et al. 2017] for their record/replay system.

Once we have a *range* of implementations offering different trade-offs, that diversity itself opens up new possibilities. For instance, in the Debian Reproducible Builds scenario, one can deliberately run at a *weaker* enforcement level in order to reproduce a build with a known output-hash. If the job reproduces the outcome—even with weak or missing determinism enforcement—then we are done! In the (hopefully rare) cases where it does not, it is possible to simply rerun with *stricter* enforcement before registering a true reproduction failure. In this scenario, weaker enforcement strategies serve as the optimistic, fast-path execution.

9.4 Arbitrary Programs as *Pure* Functions

What is a purely functional program? It is subject to debate [Sabry 1998]. Purely functional languages like Haskell still require a runtime, system calls, and other machinery to actually compute pure functions. Purity is simple to reason about since the pure code must not access any state outside of its process. The moment that a function oversteps the process boundary, e.g., by printing to stdout or by performing a computation that depends on the disk state, it is labeled as impure (required to live in the monadic `IO` type) to avoid labeling operations which could potentially break referential transparency as "pure".

One interesting consequence of this work is that we can expand the scope of what is considered pure. We can run arbitrary x86 executables under `libdet` and get the same results each time (assuming we track I/O appropriately). There is no risk of running programs and breaking referential transparency. Legacy programs become pure functions, and we can expand the scope of pure code to include calls to legacy software without requiring a monadic type. Pure functions can even access scratch space on disk, as `libdet` will confine writes to this scratch space and wipe it clean when the function terminates, enforcing referential transparency.

One way to integrate existing software into pure Haskell code would be through a foreign function interface (FFI). Currently, Haskell's FFI for C can import both pure and impure functions into Haskell. However, the burden is on the programmer to ensure that impure code is given an `IO` return type. With `libdet`, this burden could be shifted to the language, as any x86 program that reads from stdin and prints to stdout could be imported as a foreign function of type `String → String`. This would be a form of IO-free FFI based on runtime encapsulation. Indeed, given Haskell's semantics and its compiler's assumptions about referential transparency and code reordering, what is important is determinism, not a stylistic notion of functional versus imperative code.

REFERENCES

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* 11 (October 1989), 598–632. Issue 4. DOI: http://dx.doi.org/10.1145/69558.69562

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.*

Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). http://arxiv.org/abs/1407.3561

Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven Gribble. 2010. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.*

Guy Blelloch. 1992. *NESL: A Nested Data-Parallel Language.* Technical Report CMU-CS-92-103. Carnegie Mellon University, Pittsburgh, PA.

Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09.* Orlando, Florida, USA, 97. DOI: http://dx.doi.org/10.1145/1640089.1640097

Robert L. Bocchino and Vikram S. Adve. 2011. Types, Regions, and Effects for Safe Programming with Object-oriented Parallel Frameworks. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11).* Springer-Verlag, Berlin, Heidelberg, 306–332. http://dl.acm.org/citation.cfm?id=2032497.2032519

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03).* Springer-Verlag, Berlin, Heidelberg, 55–72. http://dl.acm.org/citation.cfm?id=1760267.1760273

Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10).* ACM, New York, NY, USA, 691–707. DOI: http://dx.doi.org/10.1145/1869459.1869515

Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. 1989. *Modula-3 report (revised).* Vol. 52. Digital Systems Research Center.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07).* ACM, New York, NY, USA, 10–18. DOI: http://dx.doi.org/10.1145/1248648.1248652

Ramu Chenna, Hideaki Sugawara, Tadashi Koike, Rodrigo Lopez, Toby J. Gibson, Desmond G. Higgins, and Julie D. Thompson. 2003. Multiple sequence alignment with the Clustal series of programs. *Nucleic Acids Research* 31, 13 (2003), 3497. DOI: http://dx.doi.org/10.1093/nar/gkg500

Douglas Crockford. 2008. ADsafe: Making JavaScript Safe for Advertising. http://www.adsafe.org/. (2008).

Debian Wiki. 2016. ReproducibleBuilds. (2016). https://wiki.debian.org/ReproducibleBuilds?action=recall&rev=339 [Online; accessed 14-April-2017].

David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14).* USENIX Association, Berkeley, CA, USA, 525–540. http://dl.acm.org/citation.cfm?id=2685048.2685090

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09).* Washington, DC, USA, 85. DOI: http://dx.doi.org/10.1145/1508244.1508255

Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A Relaxed Consistency Deterministic Computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems.*

Sean R. Eddy. 1998. Profile hidden Markov models. *Bioinformatics* 14, 9 (1998), 755–763.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98).* ACM, New York, NY, USA, 212–223. DOI: http://dx.doi.org/10.1145/277650.277725

Kim Hazelwood, Greg Lueck, and Robert Cohn. 2009. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09).* ACM, New York, NY, USA, 20–29. DOI: http://dx.doi.org/10.1145/1542431.1542435

Derek R. Hower, Polina Dudnik, David A. Wood, and Mark D. Hill. 2011. Calvin: Deterministic or Not? Free Will to Choose. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA).*

Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. 2013. DDOS: taming nondeterminism in distributed systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Vol. 48. 499–508. DOI: http://dx.doi.org/10.1145/2451116.2451170

Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. 2014. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R Newton. 2014a. Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2.

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014b. Freeze after writing: quasi-deterministic parallel programming with LVars. In *POPL*. 257–270.

Daan Leijen, Manuel Fahndrich, and Sebastian Burckhardt. 2011. Prettier Concurrency: Purely Functional Concurrent Revisions. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, 83–94.

Heng Li and Richard Durbin. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26, 5 (2010), 589. DOI: http://dx.doi.org/10.1093/bioinformatics/btp698

Linux. 2015. *ptrace(2) Linux User's Manual*.

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. New York, NY, USA, 327–336. DOI: http://dx.doi.org/10.1145/2043556.2043587

Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, and Phil Trinder. 2010. Seq no more: better strategies for parallel Haskell. In *ACM Sigplan Notices*, Vol. 45. ACM, 91–102.

Simon Marlow, Ryan R. Newton, and Simon Peyton Jones. 2011. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell (Haskell '11)*. ACM, 71–82.

Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 65–78. DOI: http://dx.doi.org/10.1145/1596550.1596563

Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 693–708. DOI: http://dx.doi.org/10.1145/3037697.3037751

John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*. Albuquerque, New Mexico, United States, 24–33. DOI: http://dx.doi.org/10.1145/125826.125861

Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-performance Determinism with Total Store Order Consistency. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 31, 13 pages. DOI: http://dx.doi.org/10.1145/2741948.2741960

Timothy Merrifield and Jakob Eriksson. 2013. Conversion: multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 127–139. DOI: http://dx.doi.org/10.1145/2465351.2465365

Mozilla. 2015. rr: lightweight recording & deterministic debugging. (2015). http://rr-project.org/ [Online; accessed 16-April-2017].

Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99)*. New York, NY, USA, 228–241. DOI: http://dx.doi.org/10.1145/292540.292561

Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP '97)*. ACM, New York, NY, USA, 129–142. DOI: http://dx.doi.org/10.1145/268998.266669

Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology* 9, 4 (Oct. 2000), 410–442. DOI: http://dx.doi.org/10.1145/363516.363526

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28thACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, CA, USA, June 10-13, 2007*, Vol. 42. ACM, 89–100.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, Portable and Parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 499–512. DOI: http://dx.doi.org/10.1145/2541940.2541964

Nix. 2015. Nix: The Purely Functional Package Manager. (2015). https://nixos.org/nix/ [Online; accessed 16-April-2017].

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems -*

ASPLOS '09. Washington, DC, USA, 97. DOI: http://dx.doi.org/10.1145/1508244.1508256

Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2–11.

Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. 1999. A semantics for imprecise exceptions. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 25–36.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 531–542.

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 38–53. DOI: http://dx.doi.org/10.1145/2815400.2815411

Amr Sabry. 1998. What is a purely functional language? *Journal of Functional Programming* 8, 01 (1998), 1–22.

Peter Jay Salzman. 2009. *The Linux Kernel Module Programming Guide*. CreateSpace, Paramount, CA.

Patrick D Schloss, Sarah L Westcott, Thomas Ryabin, Justine R Hall, Martin Hartmann, Emily B Hollister, Ryan A Lesniewski, Brian B Oakley, Donovan H Parks, Courtney J Robinson, and others. 2009. Introducing mothur: open-source, platform-independent, community-supported software for describing and comparing microbial communities. *Applied and environmental microbiology* 75, 23 (2009), 7537–7541.

Nir Shavit and Dan Touitou. 1995. Elimination Trees and the Construction of Pools and Stacks: Preliminary Version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*. ACM, New York, NY, USA, 54–63. DOI: http://dx.doi.org/10.1145/215399.215419

Alexandros Stamatakis. 2014. RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics* 30, 9 (2014), 1312. DOI: http://dx.doi.org/10.1093/bioinformatics/btu033

Ke Sun, Xiaoning Li, and Ya Ou. 2016. Break Out of the Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. In *Black Hat Asia (Black Hat Asia '16)*. https://www.blackhat.com/docs/asia-16/materials/asia-16-Sun-Break-Out-Of-The-Truman-Show-Active-Detection-And-Escape-Of-Dynamic-Binary-Instrumentation.pdf

Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 363–378. DOI: http://dx.doi.org/10.1109/SP.2011.39

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 137–148. DOI: http://dx.doi.org/10.1145/2364506.2364524

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*.