Determining Determinacy[*]

Melissa E. O'Neill[†] F. Warren Burton[‡]

June [illegible], [illegible]

Abstract

Parallel programs can avoid determinacy races (race conditions leading to indeterminate program behavior) by avoiding write/write and read/write contention between parallel tasks. Conditions to ensure that these kinds of contention are avoided have been suggested by Bernstein ([illegible]) and generalized by Steele ([illegible]). Enforcing such conditions at compile time is undecidable for general parallel programs. Until now, attempts to enforce the conditions at runtime have had non–constant-factor overheads, sometimes coupled with a serial execution requirement.

We present a runtime determinacy checker for shared-memory parallel programs that avoids some of the drawbacks present in previous approaches to the problem. Our technique has constant-space and constant-time overheads when run on a uniprocessor machine. Our method can also run in parallel on multiprocessor machines (although some loss of parallelism may occur because our determinacy checker must arbitrate access to the data structures it uses to describe task relationships). Our technique can be applied to programs that use nested parallelism, as well as some other classes of parallel programs, including programs that use producer–consumer parallelism.

We examine the performance of our algorithm in practice and show that, like array-bounds checking and other runtime checking techniques, the extent of the overheads depends on the properties of the application.

1 Introduction

How may we ensure that parallel access to shared data is done safely? Techniques for serializing access to a data structure, such as *semaphores* (Dijkstra, [illegible]) and *monitors* (Hoare, [illegible]) are well known, but locking does not address the issue of whether a program's data dependencies are safe. Locking may prevent task A from reading data before task B has finished writing it, but it says nothing about whether task A should depend on data written by task B. A parallel program that is written without regard to intertask data dependencies—even one that uses locks to protect data—can suffer determinacy problems, causing program failures or unexpected variability of output.

Parallel determinacy problems can be extremely subtle. Timing issues and scheduler behavior can mask problems, resulting in code that appears to be free of bugs during testing but fails mysteriously at some later time. Tracking such problems with conventional debugging tools can be difficult, as the race condition may be hard to reproduce reliably.

In this paper we show how *Bernstein's conditions* ([illegible]) can be efficiently enforced at runtime, ensuring that a program behaves deterministically. Data that is accessed in compliance with Bernstein's conditions avoids read/write and write/write contention between tasks and does not need to be protected by locks. On a multiprocessor shared-memory machine with processor caches, obeying Bernstein's conditions can also reduce interprocessor memory contention, eliminating one source of poor performance.

In this section, we will define what we mean by determinacy and describe our model for parallelism. We will also review the ways in which determinacy problems

[*]This work was supported by the National Science and Engineering Research Council of Canada.

[†]Harvey Mudd College, Claremont, CA, U.S.A. E-mail: oneill@acm.org.

[‡]Simon Fraser University, Burnaby, BC, Canada. E-mail: burton@cs.sfu.ca.

Listing 1: Examples of parallel programs exhibiting determinacy and indeterminacy. The first program exhibits indeterminacy, whereas the middle two are clearly deterministic. The fourth program exhibits internal indeterminacy while remaining externally deterministic.

(a)

```
a := 2;
cobegin
   b := a;
   a := a + 1;
coend;
```

(b)

```
a := 1;
b := 2;
cobegin
   c := a + b;
   if a > b then
      d := a;
   else
      d := b;
coend;
cobegin
   a := c / 2;
   d := b + c + d;
coend;
```

(c)

```
a := 1;
b := 2;
cobegin
   begin
      c := a + 3;
      cobegin
         a := (a + b) / c;
         d := b / c;
      coend;
   end;
   e = (b / 2) + 1;
coend;
```

(d)

```
cobegin
   a := -2;
   a := 2;
coend;
a := a * a;
```

can be detected and discuss previous attempts to provide determinacy checking. The remainder of the paper is as follows: In Section 2 we will introduce a naïve method for runtime determinacy checking that is neither time nor space e□cient. Sections 3 and 4 show how this simple determinacy checker can be modified to be e□cient. Section 5 discusses the time and space complexity of the revised algorithm. Sections 6 and 7 examine some of the practical aspects of our algorithm—the former describes some useful optimizations to our technique and the latter examines the performance of an implementation of our algorithm when running realistic benchmarks. Finally, Section 8 presents our conclusions and discusses opportunities for further work.

1.1 Describing Determinacy

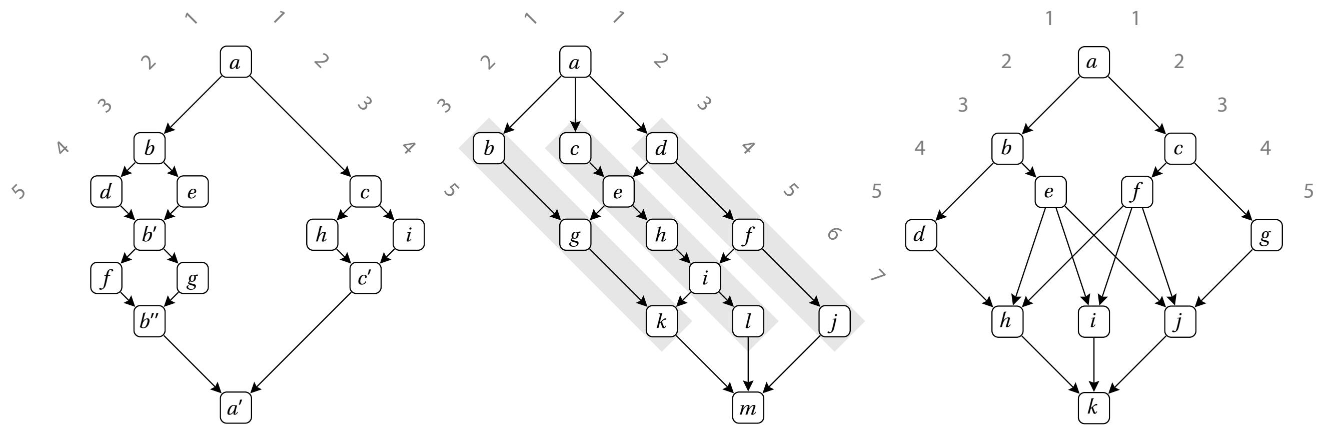
Various terms have been used to describe parallel determinacy problems, including *harmful shared-memory accesses* (Nudler & Rudolph, □□□□), *race conditions* causing *indeterminacy* (Steele, □□□□), *access anomalies* (Dinning & Schonberg, □□□□), *data races* (Mellor-Crummey, □□□□), and *determinacy races* (Netzer & Miller, □□□□).

Feng and Leiserson (□□□□) list most of the above terms and recommend the term "determinacy race". We will adopt this term for our discussion, but also follow Steele's preferences by using "indeterminacy" to describe a problematic lack of determinacy caused by determinacy races and "nondeterminacy" to describe a lack of determinacy intended by the programmer (such as McCarthy's nondeterministic *amb* operator (□□□□)).

Programs do not need to be fully deterministic to generate consistent results from run to run—for example, an operating system may run unrelated tasks in nondeterministic order without causing any problems; or a search algorithm may choose its initial search direction nondeterministically and yet generate unwavering results. Thus programs can be *externally deterministic* even if they are not *internally deterministic* (Emrath & Padua, □□□□) so long as the operations that are performed in nondeterministic order *commute* (Steele, □□□□).

Checking external determinacy is di□cult. For example, suppose $m = 2.0$ and we perform two atomic actions, $m \leftarrow 1/m$ and $m \leftarrow m - 2.5$, in nondeterministic order. For these values, either order results in $m = -2.0$, although addition and division do not usually commute. (Steele (□□□□) discusses *commuting with respect to a memory state* versus commuting in general.) Similarly, far more complex instances of nondeterminacy could eventually cancel out, and the duration of the internal nondeterminacy could be arbitrarily long.

Checking internal determinacy is easier, because we need only be concerned about the local e□ects of program behavior. Internally deterministic programs are easier to reason about than their externally deterministic and nondeterministic cousins, making debugging easier



(a) Tasks that fit the nested-parallelism model (e.g., cobegin and coend constructs), described using a series–parallel DAG.

(b) A pipeline of tasks. Nodes *d*, *f*, and *j* together represent a single producer; nodes *c*, *e*, *h*, *i*, and *l* represent a (UNIX-style) filter; and nodes *b*, *g*, and *k* represent a consumer. The producer, filter, and consumer are each broken into several tasks, representing two occasions when the producer synchronized with the filter and two occasions when the filter synchronized with the consumer.

(c) Tasks synchronized in unusual ways. These tasks use a general, nonplanar DAG to represent their synchronization requirements.

Figure 1: Parallel task models. In all three graphs, a task can begin only after all tasks that have arrows pointing to it have completed.

and formal verification more practical. Even nondeterministic programs can benefit from ensuring that their nondeterminacy is limited to those places where it is intended.

Listing 1 provides some code fragments with and without indeterminacy. Listing 1(a) shows a simple example[1] of indeterminacy—inside the cobegin construct, one task reads the value of a while the other task updates a. Listings 1(b) and 1(c) are both deterministic—how tasks are scheduled has no effect on the final result. Listing 1(d) is a rather ambiguous case—the code is externally deterministic, but during its execution, a has a nondeterministic value.

We will focus on the problem of preventing indeterminacy by ensuring internal determinacy. If the programs in Listing 1 were tested for internal determinacy, only (b) and (c) would pass the test.

1.2 Parallel Model

We focus our discussion on mechanisms to provide determinacy checking for programs that execute on a shared-memory MIMD machine (Flynn, [illegible]). To describe such programs, we will use a model of parallel computation in which parallel programs are composed of *tasks*—sequential sections of code that contain no synchronization operators. All parallelism comes from executing multiple tasks concurrently.

The scheduling dependencies of tasks can be described using a directed acyclic graph (DAG) (Valdes, [illegible]). When a task y cannot begin until another task x has completed, we place an edge in the DAG from x to y and describe

1. All four examples are simple and could be checked for determinacy statically. More complex programs, especially ones involving array accesses with dynamically computed subscripts (e.g., parallel sorting algorithms) can be difficult or impossible to check statically.

x as a *parent* of y and y as a *child* of x. If y and z are both children of x, we say y and z are *siblings*. When a task has multiple children, we call it a *fork node*; when a task has multiple parents, we call it a *join node*. Nothing precludes a node from being both a fork node and a join node. Figure 1 shows three such graphs with increasingly complex synchronization requirements (the graphs also include labels, shown in grey, on their left and right sides; these labels may be ignored for now, but we will return to them in Section 3). The DAG representation of tasks is a *reduced* graph because it contains no transitive edges—transitive edges would be redundant because synchronization is implicitly transitive.

The DAG representation cannot fully represent all kinds of parallelism and synchronization. For example, programs that are inherently nondeterministic, synchronize using locks, or expect lazy task execution will have some of their parallel structure uncaptured by the task DAG. We will restrict our discussion to the kinds of parallelism where the DAG representation is useful.

The scheduling dependencies of a task DAG can also be equivalently expressed using a relation. When task x must complete before task y can begin, we write $x \lhd y$, pronounced "x is a proper ancestor of y". We also define the partial order, $\unlhd$, pronounced "ancestor of", such that

$$(x \unlhd y) \Leftrightarrow \big((x \lhd y) \vee (x = y)\big).$$

These relations exactly mirror the DAG model because $x \unlhd y$ is equivalent to saying that there is a (possibly zero-length) path from x to y in the DAG.

We define two metrics for a task DAG: The *breadth* of the DAG is the maximum number of tasks that could execute concurrently if the program were run using an infinite number of processors. The *size* of the DAG is the total number of tasks it contains.

We use three terms to describe the life cycle of an individual task from a task scheduler's perspective:

- **Nascent tasks** are tasks that are not yet represented in the task scheduler.
- **E ective tasks** are tasks that must be represented by the task scheduler, because they are executing, waiting to execute, or their children are still nascent. By definition, a task cannot be executed by the scheduler until it has become e ective (a scheduler cannot execute a task without representing that task in some way). A task cannot become e ective until one of its parents has completed, and must remain e ective until it has completed its execution and all its children have ceased to be nascent. This specification of when a task may be e ective is deliberately loose—the exact details depend on the scheduler.
- **Ethereal tasks** are tasks that have ceased to be effective, and no longer need to be represented in the task scheduler.

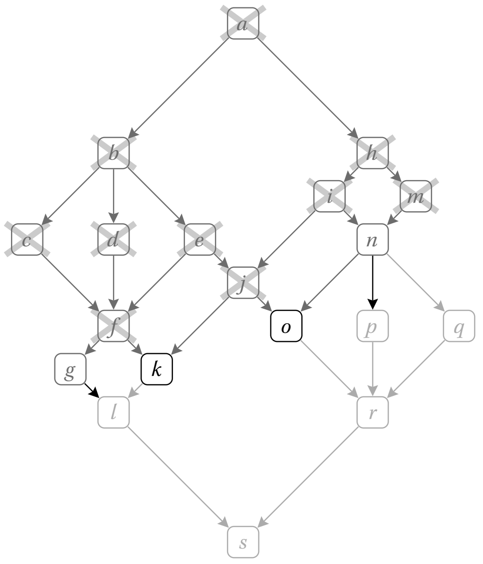


Figure 2: Understanding nascent, e ective, and ethereal tasks. If we consider a moment in time where the light-grey tasks have yet to execute, the black tasks are executing, and the dark-grey tasks have completed, then all the light grey tasks are nascent and the crossed-out tasks are ethereal. The remaining tasks, *g*, *k*, *n*, and *o*, are e ective: k and o are e ective because they are executing, whereas n and g are e ective because they have nascent children. This is not the only possible arrangement of e ective tasks—if l were made e ective, g could become ethereal.

The number of e ective tasks varies as the program executes (see Figure 2), and is a function of both the scheduling-algorithm implementation and the structure of the task DAG. Typically, the peak number of e ective tasks would be higher for a breadth-first task scheduler than a depth-first scheduler. For our discussion, we will assume that the task scheduler requires $\Omega(e)$ memory to execute a program, where e is the peak number of effective tasks during execution under that scheduler.

Note that a programming language's definition of a task, thread, process, or coroutine may be different from a task in this model but nevertheless compatible with it. Some languages, for instance, allow a spawner thread and its subthread to execute in parallel, and later have the spawner wait for its subthread to terminate. As Steele (□□□□) observes, we can represent such a mechanism in our model by representing the spawner thread as three nodes in the DAG: a fork node, a child node, and a join node, where the child node represents that part of the spawner thread's execution that occurs concurrently with its subthread.

The full generality of the DAG model is not required in many cases. Parallel programs that use *nested parallelism* fit the constraints of a series–parallel DAG (MacMahon, □□□□; Riordan & Shannon, □□□□; Duffin, □□□□), a simple and elegant form of planar DAG (see Figure 1(a)). Producer–consumer problems cannot be described using series–parallel graphs, but they can be described using another simple planar DAG form (see Figure 1(b)). In both of these cases, the $\trianglelefteq$ relation describes a planar lattice and the DAG is subsumed by a type of planar DAG known as a planar *st*-graph (which we define in Section 3). Some parallel programs require an arbitrary nonplanar DAG to precisely represent their synchronization requirements. The graph shown in Figure 1(c), for example, cannot be drawn as a planar DAG.[2]

We will devote most of our discussion to the problems faced in checking parallel programs that use nested parallelism, but the LR-tags technique presented in subsequent sections can also be applied to a more general class of parallel program, including parallel programs whose $\trianglelefteq$ relation forms a planar lattice (see Section 3).

1.3 Conditions for Parallel Determinacy

Determining statically whether a general parallel program will avoid parallel indeterminacy problems for all inputs is undecidable (see, for example, O'Neill, □□□□). Sufficient conditions for determinacy exist, however, that are not unduly restrictive, and can either be used as a discipline for programmers or imposed as a condition in language design.

1.3.1 Milner's Confluence Property

Milner (□□□□) points out that even if two systems are deterministic, allowing unconstrained communication between them immediately allows the possibility of nondeterminacy. Milner develops a *confluence* property sufficient to ensure parallel determinacy,[3] but Milner's parallelism follows a CSP (Hoare, □□□□) model, which has its greatest applicability in distributed systems (it can represent shared memory, but the mapping is not ideal). Our interest in this paper is shared-memory parallel computations, so we will now turn our attention to specific attempts to express shared-memory determinacy conditions.

1.3.2 I-Structures

One method for ensuring that shared-memory is accessed deterministically is to instigate a write-once policy and suspend any task that attempts to read a yet-to-be-written memory location until that location is written. This approach is taken by Arvind et al. (□□□□) in their *I-structure* data structure. This condition is simple and easy to enforce, yet fits a number of parallel algorithms. One problem with this approach is that, although indeterminate results are not possible with I-structures, read/write races remain possible and can result in deadlock.

1.3.3 Bernstein's Conditions

Bernstein (□□□□) developed conditions for *noninterference* between parallel tasks. *Bernstein's conditions* are

2. Astute readers may notice that we can add a node to Figure 1(c) to turn it into a planar graph without altering its behavior by adding a node at the point where two lines cross. This transformation is not applicable in general, however, because it may add synchronization behavior and describe data dependencies not required by the original program. Interestingly, the graphs that *can* be trivially transformed into an *st*-graph by inserting dummy nodes can also be directly represented in the efficient graph representation introduced in Section 3 without using any dummy nodes.

3. Hoare (□□□□) also suggests conditions for determinacy for communicating parallel tasks, but his conditions are less interesting and less useful than Milner's.

sufficient conditions to prevent indeterminacy in a pair of parallel tasks. If t and t' are tasks that may run in parallel; W_t and $W_{t'}$ are the set of memory locations written by t and t', respectively; and R_t and $R_{t'}$ are similarly defined to be the set of memory locations read by t and t', respectively; then t and t' are noninterfering iff $(W_t \cap W_{t'} = \varnothing) \wedge (R_t \cap W_{t'} = \varnothing) \wedge (W_t \cap R_{t'} = \varnothing)$.

Bernstein's conditions generalize to a set of tasks executing in parallel, $T = \{t_1, \ldots, t_n\}$. If we use $t_i \not\leftrightarrow t_j$ to mean that there is no interference between t_i and t_j, then the tasks in T will not suffer indeterminacy if $\forall t_i, t_j \in T : t_i \not\leftrightarrow t_j \;\vee\; t_i \trianglelefteq t_j \;\vee\; t_j \trianglelefteq t_i$.

Like Milner's restrictions for confluence, Bernstein's conditions ensure determinacy, but are a sufficient rather than a necessary condition. It is possible for tasks to violate Bernstein's conditions but nevertheless be deterministic—for example, if two tasks write the same value to memory, the action is classed as interference even though it has no negative ramifications for determinacy.

1.3.4 Steele's Conditions

Steele (□□□□) proposes a more generous scheme that subsumes Bernstein's noninterference condition.[4] In Steele's terminology, two tasks, t and t', are *causally related* iff $t \trianglelefteq t' \vee t' \trianglelefteq t$. Steele's condition for avoiding indeterminacy is that any two accesses must either be causally related or *commute*.[5] Intuitively, two operations commute if the order in which they are performed does not matter; thus, multiple reads to a memory location commute with each other, but reads and writes to the same location do not. Similarly, multiple writes to the same location do not commute. Some useful operations that change memory do commute with one another—for example, an atomic increment commutes with an atomic decrement, but not with a read.

Interestingly, algorithms intended to enforce Bernstein's conditions can be generalized in a straightforward manner to enforce Steele's conditions instead, because reads generalize to operations that commute, and writes generalize to operations that do not commute. We will simplify our discussion by focusing on the problem of reads and writes, but all the checking mechanisms we discuss are suited to this generalization.

4. Steele does not explicitly cite Bernstein's paper, perhaps because he was not aware of it—the conditions themselves are fairly straightforward and can be derived from first principles.

5. Steele's conditions are stated differently; we have put them in the same framework as our discussion of Bernstein's conditions to show the parallels between them.

1.3.5 Multiple Conditions

Restrictions for deterministic parallelism in parallel systems using communication channels and restrictions for deterministic access to shared memory do not necessarily conflict with each other. Brinch Hansen's *SuperPascal* (□□□□a; □□□□b) has both of the above features, although the determinacy-enforcement facilities of SuperPascal are not as sophisticated as Milner's confluent parallel systems or Steele's safe asynchronous parallelism.

1.4 Enforcing Determinacy Conditions

Traditionally, parallel languages and runtime systems have done little to ensure determinacy. Programmers have instead had to rely on their intuitions about correct program behavior and depend on established (and frequently low-level) constructs for serializing access to their data structures, such as locks. As we learned earlier, these basic mutual-exclusion mechanisms are not a complete answer—checks for determinacy (based on Bernstein's or Steele's conditions for shared-memory accesses, or Milner's confluence property for channel-based communications) have their place, too.

Static checks at the language level (Taylor, □□□□; Callahan & Subhlok, □□□□; Emrath & Padua, □□□□; Balasundaram & Kennedy, □□□□; Beguelin, □□□□; Beguelin & Nutt, □□□□; Xu & Hwang, □□□□; Bagheri, □□□□; Brinch Hansen, □□□□a), can ensure deterministic execution, but these checks rule out those algorithms that cannot (easily) be statically shown to be deterministic. For example, parallel tasks working on a shared array are likely to decide at runtime which array elements they should access, making static analysis of whether these accesses will lead to nondeterminacy difficult or impossible. Brinch Hansen encounters just this problem in his SuperPascal language, which attempts to enforce determinacy statically. SuperPascal includes an annotation that allows programmers to turn off static determinacy checking when necessary. Brinch Hansen notes that he has used this construct exclusively for dealing with tasks accessing a shared array.

A runtime test for determinacy races allows a wider range of programs to be executed with assured determinacy than similar compile-time tests. Obviously, compile-time tests should be used whenever practical, because they may both provide earlier detection of errors and reduce the amount of runtime testing required, but when compile-time tests cannot pass an algorithm as deterministic, runtime checking can at least assure us that the program is deterministic for a given input. Like other runtime tests, such as null-pointer checks and array-bounds checking, we may either use runtime determinacy checks as a mechanism for catching coding errors during development, turning checking o□ in production code for better performance, or leave checking on permanently, trading some performance for safety.

For some programs, runtime determinacy checking on a few inputs is enough to confirm that the program will always operate correctly. For example, the memory access patterns and parallel task structure of code that performs a parallel matrix multiply may depend only on the sizes of the matrices, not the actual contents of each matrix. Thus, determining that multiplying a particular □ × □□ matrix with a certain □□ × □ matrix generates no indeterminacy also determines that all multiplications of □ × □□ matrices against a □□ × □ matrices will operate without indeterminacy.

There are two classes of dynamic determinacy checks: *on-the-fly checking* and *post-mortem analysis*. In on-the-fly checking (Nudler & Rudolph, □□□□; Emrath & Padua, □□□□; Schonberg, □□□□; Steele, □□□□; Dinning & Schonberg, □□□□, □□□□; Min & Choi, □□□□; Mellor-Crummey, □□□□; Feng & Leiserson, □□□□), accesses to data are checked as they happen, and errors are signaled quickly. In post-mortem analysis (Miller & Choi, □□□□), a log file is created during execution that is checked after the run to discover whether any accesses were invalid. Both methods have problems, however: On-the-fly detection can slow program execution significantly and may not accurately pinpoint the source of indeterminacy.[6] Post-mortem analysis faces the di□culty that logs may be voluminous (and wasteful, because the log becomes an unreliable indicator after its first error) and errors may not be detected in a timely way.

We will focus on using Bernstein's conditions as the basis of an on-the-fly dynamic determinacy checker. As we discussed in Section 1.3.4, it would be a simple matter to use our determinacy checker to enforce Steele's conditions as well.

1.4.1 Enforcing Bernstein's Conditions

Bernstein's conditions provide us with a basis for runtime checking, but although they are algebraically elegant, they do not, by themselves, provide an e□cient determinacy-checking algorithm. Implementations of runtime determinacy checking concentrate on tagging each shared memory cell with enough information for it to be checked, independent of other memory locations.

Previous algorithms for runtime determinacy checking have run into one of two problems: non–constant-factor overheads or a serial-execution requirement. *English-Hebrew Labeling* (Nudler & Rudolph, □□□□), *Task Recycling* (Dinning & Schonberg, □□□□), and *O set-Span Labeling* (Mellor-Crummey, □□□□) su□er from the first problem, whereas the *SP-Bags algorithm* (Feng & Leiserson, □□□□) su□ers from the second problem. Table 1 compares the properties of various determinacy-checking algorithms, including our own: the *LR-Tags method*.

Feng and Leiserson's SP-Bags algorithm, embodied in their *Nondeterminator* determinacy checker for the Cilk language (□□□□), provides the most time- and space-e□cient determinacy-race detector to date for programs using nested parallelism. Their method is inspired by Tarjan's (□□□□) nearly linear-time least-common-ancestors algorithm.

The SP-Bags algorithm requires $\mathrm{O}(T\,\alpha(e+v, e+v))$ time to run a program that runs in T time on one processor with checking turned o□ and uses v shared-memory locations, where α is Tarjan's (□□□□) functional inverse of Ackermann's function and e is the maximum number of e□ective tasks in the Cilk program. For any practical situation, α has a constant upper bound of 4, so we may regard Feng and Leiserson's algorithm as an "as good as

6. If two accesses are made to a location, one correct and one erroneous, it may be that the error is not detected at the point the erroneous access is made, but is instead detected when the entirely correct access is made. The extent of detection and error reporting depends largely on the technique; typically we will know what task last accessed the datum, but not when, where, or why it performed that access. Choi & Min (□□□□) propose a software debugging assistant that can be helpful for those methods where this poor reporting would be a problem.

Algorithm	Parallel Checking	Space Required	Time for Fork/Join	Time for Data Access
English-Hebrew Labeling	Yes	$O(vb + \min(ns, nbv))$	$O(n)$	$O(nb)$
Task Recycling	Yes	$O(vb + pb)$	$O(b)$	$O(b)$
Offset-Span Labeling	Yes	$O(e + v + \min(ns, nv))$	$O(n)$	$O(n)$
SP-Bags	No	$O(e + v)$	$O(\alpha(e + v, e + v))$	$O(\alpha(e + v, e + v))$
LR-Tags	Yes	$O(e + v)$	$O(1)$	$O(1)$

where			
	b	$=$	breadth of the task DAG
	e	$=$	maximum number of effective tasks during execution
	n	$=$	maximum depth of nested parallelism
	p	$=$	maximum number of running tasks during execution
	s	$=$	size of the task DAG (number of tasks)
	v	$=$	number of shared locations being monitored
	α	$\equiv$	Tarjan's ([illegible]) functional inverse of Ackermann's function

Note: The time complexities given are for execution on a single processor; multiprocessor implementations may incur additional loss of parallelism due to synchronization. Also, the time complexity of the SP-Bags method is an amortized bound. The space complexities include the need to represent effective tasks, any of which may be the subject of thread operations (usually $e \ll v$, so e is frequently omitted when space complexities are described in the literature).

Table 1: Properties of runtime determinacy checkers.

constant" amortized-time algorithm. The SP-Bags algorithm is also space efficient, needing $O(v + e)$ space to execute. Execution of the program under CILK without determinacy checking also requires $\Omega(v + e)$ space; thus the algorithm has constant-factor space overheads.

Feng and Leiserson's method is a serial method, however. It can find the internal indeterminacies that would arise if the program were executed in parallel, but does so by running the program serially. Although this restriction may not be as much of a disadvantage it first appears to be (as developers often develop and debug their code on uniprocessor systems before running it on a more expensive parallel machine), it is nevertheless an unfortunate limitation. The SP-Bags method is also restricted to programs that use nested parallelism and so cannot enforce Bernstein's conditions for programs using producer–consumer parallelism or other, more esoteric, forms of parallelism.

2 A Simple Determinacy Checker

This section introduces a simple determinacy checker that is neither time nor space efficient. This checker will act as a stepping stone to the final LR-tags determinacy checker. Sections 3 and 4 explain how the inefficiencies of this simple determinacy checker can be eliminated.

To enforce internal determinacy at runtime, we need to check two conditions. We must ensure that

1. Each read is valid, given previous writes

2. Each write is valid, given previous reads and writes

We need to consider prior reads when checking writes because reads might be scheduled in a different order on other runs—even when tasks are scheduled in a simple serial fashion, error detection should not be influenced by the order of task execution (see Listing 1(a)).

We saw in the previous section that Bernstein's noninterference conditions provide us with a simple rule to ensure deterministic execution; we will now express those conditions in a slightly different way. We will associate each datum d with both the last writer for that datum, $w(d)$, and the set of tasks that have accessed that datum since it was written, $R(d)$. We may think of $R(d)$ as the "reader set" for d, but $R(d)$ also includes the task that last wrote d. (Sometimes, when d is obvious from context or irrelevant to the discussion, we will write R and w rather than $R(d)$ and $w(d)$. We will also apply this convention to $r_{\mathrm{y}}(d)$ and $r_{\mathrm{x}}(d)$, which we define later.)

- *Reads* — A read is valid if the task that last modified the data item is an ancestor of the task performing the read. Expressing this restriction algebraically, a task t may read a datum d if

  $$w(d) \trianglelefteq t \qquad \text{(Bern-1)}$$

  where $w(d)$ is the task that wrote the value in d. When a read is valid, we update $R(d)$ as follows:

  $$R(d) := R(d) \cup \{t\}$$

- *Writes* — Writes are similar to reads, in that the task that last modified the data item must be an ancestor of the task performing the write, but writes also require that all reads done since the last write are also ancestors of the task performing the write. Thus, a task t may write a datum d if

  $$\forall r \in R(d) : r \trianglelefteq t \qquad \text{(Bern-2)}$$

  where $R(d)$ is the set of tasks that have accessed d since it was last written (including the task that performed that write). If the write is valid, we update $R(d)$ and $w(d)$ as follows:

  $$\begin{aligned} w(d) &:= t \\ R(d) &:= \{t\} \end{aligned}$$

Even this simple method provokes some interesting implementation questions, such as "How do we provide $R(d)$ and $w(d)$ for a datum d?". One way to store $R(d)$ and $w(d)$ is to add a writer field and a readers field to each determinacy-checked object. Feng and Leiserson ([illegible]) provide an alternate way to address this question with their *shadow-spaces* instrumentation technique. In their approach, determinacy-checking information is stored in additional memory (shadow space) that mirrors the memory where the determinacy-checked objects are stored. Calculating where $R(d)$ and $w(d)$ are stored in this scheme is a simple function of the memory address of d. The shadow-spaces approach has the advantage that the way program data is laid out in memory need not change when determinacy checking is being used. Unfortunately, the shadow-spaces approach can waste memory because some objects take up more space than others (e.g., extended-precision complex numbers may take twenty-four times more space than eight-bit integers), leading to unused gaps in the shadow spaces.

But considering implementation questions is perhaps a little premature, as this naïve determinacy checker has poor asymptotic time and space requirements. For example, $R(d)$ can contain an arbitrary number of tasks, potentially causing tests involving $R(d)$ to be slow. There are also performance issues for ancestor queries. Traditional representations such as adjacency lists or adjacency matrices do not offer good computational complexity, especially given that a static data structure cannot represent the $\trianglelefteq$ relation because the parallel structure of the program may be determined by its input data. A linked data structure that mirrors the structure of the task DAG yields similarly poor performance.

Many parallel algorithms can enjoy better determinacy-checking performance, however, because they do not need the full power of an arbitrary DAG to describe how their tasks relate. In the next section, we will introduce restrictions to the task DAG that allow it to be represented efficiently.

3 Defining a Graph with Two Relations

As we have seen, task DAGs form a reduced graph that can be adequately represented by a partial order ($\trianglelefteq$). In this section, we will discuss how adding structure and restricting the class of graphs that we can represent enables us to use a compact representation for tasks and sets of tasks; this representation facilitates fast ancestor queries, which may either compare one task with another or determine whether all members of a set of tasks are ancestors of a particular task.

Definition 3.1 *An* LR-graph G *is a reduced* DAG *in which each vertex* v *in the* DAG *is associated with two* *labels, $x(v) \in G_\mathrm{x}$ and $y(v) \in G_\mathrm{y}$, where G_x and G_y are totally ordered sets.*[7] *Several vertices may share the same label, but each vertex is given a unique* pair *of left and right labels. From these two orderings, we define the partial order $\trianglelefteq$ as*

$$p \trianglelefteq q \Leftrightarrow (x(p) \preccurlyeq x(q)) \land (y(p) \preccurlyeq y(q)) \qquad (□)$$

and define the DAG *such that it reflects the $\trianglelefteq$ relation: there is a (possibly zero-length) path from p to q in the* DAG *iff $p \trianglelefteq q$.*

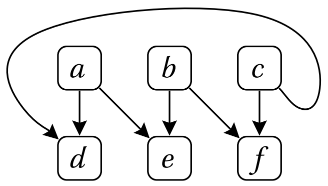


Figure 3: A graph that cannot be drawn as a dominance drawing.

The definition above is essentially identical to that of a dominance drawing, where the x- and y-coordinates of each vertex v in the drawing are equivalent to $x(v)$ and $y(v)$. The only di□erence is that coordinates in a dominance drawing are typically taken to be integers, whereas our labels are from an ordered set (whose members need not be numbers). This close equivalence means that LR-graphs embody exactly those graphs that can be drawn as dominance drawings.

The class of graphs that can be expressed as dominance drawings is a large one. All the graphs shown in Figure 1 are LR-graphs, in fact the grey labels along the sides of the graph can be used to provide the x- and y-labelings for the nodes shown in the graph (the labels increasing from left to right are the x-labels, and the ones increasing from right to left are the y labels). Some graphs (such as the one shown in Figure 3) cannot be drawn as dominance drawings and are thus not LR-graphs.

LR-graphs include a class of graphs known as *planar* st*-graphs*, which subsume both series–parallel graphs and producer–consumer graphs. A planar *st*-graph is a planar DAG with one *source node* and one *sink* (or *terminal*) *node*, both of which lie on the external face of the graph. A source node, conventionally labeled *s*, is a node with no inbound edges; a sink node, conventionally labeled *t*, is a node with no outbound edges. Di Battista et al. (□□□□) shows how any *st*-graph can be drawn as a dominance drawing, and thus shows how the x- and y-labelings can be defined for such a graph.

Any task graph might have several possible x- and y-labelings. We adopt the convention that the order of the x-labels is consistent with the order in which the tasks in the task graph would be executed by a left-to-right depth-first serial scheduler, and the y-labels match the order tasks would be executed by a right-to-left depth-first serial scheduler (hence the name, LR-graphs). We use the term "is consistent with" rather than "is identical to" because, as with the graphs shown in Figure 1, the labeling can be compacted by allocating leftmost children the same x-label as the leftmost parent, and rightmost children the same y-label as their rightmost parent.[8] A complete labeling algorithm, including compaction is given by Di Battista et al (□□□□, pages □□□–□□□).

Although the fundamental concepts underlying LR-graphs and dominance drawings are not new (Birkho□, □□□□; Kelly & Rival, □□□□; Kameda, □□□□), LR-graphs avoid the planarity requirement present in earlier treatments of this subject. These earlier works show that it is always possible to define the x- and y-labelings (or a left-to-right node ordering that can be used to define these labelings) for planar lattices (or planar *st*-graphs). Such a proof is tautological for LR-graphs because LR-graphs are *defined* using these labelings. Thus, whereas all planar lattices are LR-graphs (Birkho□, □□□□, page □□, exercise □(c)), there are potentially other (nonplanar) graphs that are also LR-graphs (e.g., Figure 1(c)). Nevertheless, LR-graphs have similar expressive power to planar *st*-graphs and planar

7. The only important quality of the labels is their relative order. For the time being, we might wish to suppose that G_x and G_y consist of suitably chosen rational numbers. Alternatively, we can see the totally ordered set of labels as an abstract data type with four operations. We can create a new totally ordered set containing a single label. We can create a new label that falls immediately after a specified label. We can compare two labels from the same set to determine which comes first. Finally, we can remove a label from the set.

8. Children that are the only child of a single parent cannot be compacted in this way, but such children serve no useful purpose in a task graph.

lattices.[9]

Representing task graphs as LR-graphs allows us to easily determine whether one task is an ancestor of another, but there is another, more significant, benefit we can reap from using this representation. Recall that the naïve determinacy checker we discussed in Section 2 needed to maintain a set R of all the tasks that had read a particular datum, and check to make sure that the current task t was a descendant of all those tasks. For LR-graphs, we do not need to store the entire set R—instead we can use a the greatest left and right labels from the members of R, r_{x}, and r_{y}, and use them as surrogates for the entire set.

Definition 3.2 *A non-empty readset R, containing nodes from an* LR-graph *G, can be represented by a left reader* r_{x} *and right reader* r_{y} *defined as*

$$r_{\mathrm{x}} = \max\{x(r) \mid r \in R\} \qquad (2a)$$

$$r_{\mathrm{y}} = \max\{y(r) \mid r \in R\} \qquad (2b)$$

(Note that because the determinacy checkers we have discussed include the task that last wrote (or created) the object in its readset, readsets are never empty.)

Theorem 3.3

$$(\forall r \in R : r \trianglelefteq t) \Leftrightarrow (r_{\mathrm{x}} \preccurlyeq x(t)) \wedge (r_{\mathrm{y}} \preccurlyeq y(t)) \qquad (3)$$

Proof

The forward implication is trivial to prove:

$$\begin{aligned} &\forall r \in R : r \trianglelefteq t \\ &\quad \Rightarrow \forall r \in R : (x(r) \preccurlyeq x(t)) \wedge (y(r) \preccurlyeq y(t)) && \text{[from 1]} \\ &\quad \Rightarrow (r_{\mathrm{x}} \preccurlyeq x(t)) \wedge (r_{\mathrm{y}} \preccurlyeq y(t)). && \text{[from 2a and 2b]} \end{aligned}$$

Proving in the other direction is almost as easy:

$$\begin{aligned} &(r_{\mathrm{x}} \preccurlyeq x(t)) \wedge (r_{\mathrm{y}} \preccurlyeq y(t)) \\ &\quad \Rightarrow \big((\forall r \in R : x(r) \preccurlyeq r_{\mathrm{x}}) \wedge (r_{\mathrm{x}} \preccurlyeq x(t))\big) && \text{[from 2a]} \\ &\qquad\qquad \wedge \big((\forall r \in R : y(r) \preccurlyeq r_{\mathrm{y}}) \wedge (r_{\mathrm{y}} \preccurlyeq y(t))\big) && \text{[and 2b]} \\ &\quad \Rightarrow (\forall r \in R : x(r) \preccurlyeq x(t)) \wedge (\forall r \in R : y(r) \preccurlyeq y(t)) && \text{[transitivity]} \\ &\quad \Rightarrow \forall r \in R : r \trianglelefteq t. && \text{[from 1]} \end{aligned}$$

□

Given these properties, the following rules are sufficient for checking that reads and writes performed by a task t on a datum d are deterministic:

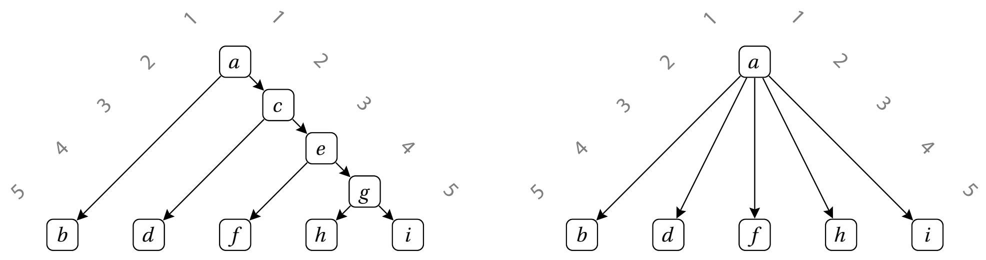
- *Reads* — A read is valid if $(x(w(d)) \preccurlyeq x(t)) \wedge (y(w(d)) \preccurlyeq x(t))$—that is, if it satisfies condition Bern-1. If the read is valid, $r_{\mathrm{x}}(d)$ and $r_{\mathrm{y}}(d)$ may need to be updated:
  - If $r_{\mathrm{x}}(d) \preccurlyeq x(t)$ then $r_{\mathrm{x}}(d) := x(t)$
  - If $r_{\mathrm{y}}(d) \preccurlyeq y(t)$ then $r_{\mathrm{y}}(d) := y(t)$
- *Writes* — A write is valid if $(r_{\mathrm{x}}(d) \preccurlyeq x(t)) \wedge (r_{\mathrm{y}}(d) \preccurlyeq y(t))$—that is, it satisfies condition Bern-2. If the write is valid, $w(d) := t$, $r_{\mathrm{x}}(d) := x(t)$, and $r_{\mathrm{y}}(d) := y(t)$.

The next section shows how it is possible to provide an efficient implementation of the G_{x} and G_{y} orderings and $\preccurlyeq$ relation for a dynamically created task DAG, and thereby provide an efficient determinacy checker.

4 Dynamic LR-graphs

In the previous section, we saw that LR-graphs make it easy to perform ancestor queries between individual nodes and sets of nodes. But we have not yet discussed how to obtain suitable x- and y-labels for nodes. For a static graph, it may be trivial to perform two traversals to generate the left and right node labelings, but parallel programs are usually dynamic and the final form of the task DAG is often unknown until the run has completed. Thus we require an algorithm that can maintain labelings

9. It is fairly straightforward to transform a planar *st*-graph into an LR-graph or vice versa by adding dummy nodes to the graph. When transforming a planar *st*-graph to an LR-graph, dummy nodes may need to be added because an LR-graph is a reduced graph (see Di Battista *et al.*, [illegible], page [illegible], for an algorithm), whereas nonplanar LR-graphs (or LR-graphs without a single source or sink node) may require the addition of dummy nodes to form a planar *st*-graph (see Di Battista *et al.*, [illegible], page [illegible], for a review of common planarization algorithms).



(a) With intermediate nodes, five x-labels and five-y labels are required to represent five children.

(b) If intermediate nodes are not used, we require the same number of x- and y-labels.

Figure 4: Handling n-ary forks.

for an *incomplete* graph and efficiently update these orderings as nodes are added to the bottom of the graph, eventually completing it.

In this paper, we will give an algorithm that can extend LR-graphs following the growth patterns that occur in programs with nested parallelism and simple task pipelines (producer–consumer parallelism).

4.1 Nested Parallelism

The algorithm for constructing an LR-graph dynamically for *fork*/*join* (or cobegin/coend) parallelism is remarkably straightforward. First let us consider the algorithm for simple binary forks and joins.

- *fork* node: p spawns two children, c_1 and c_2
  - Insert a new label l_x into G_x, immediately after $x(p)$
  - $x(c_1) := x(p)$
  - $x(c_2) := l_\mathrm{x}$
  - Insert a new label l_y into G_y, immediately after $y(p)$
  - $y(c_1) := l_\mathrm{y}$
  - $y(c_2) := y(p)$
- *join* node: p_1 and p_2 spawns one child, c
  - $x(c) := \max\{x(p) \mid p \in \{p_1, p_2\}\}$
  - $y(c) := \max\{y(p) \mid p \in \{p_1, p_2\}\}$

We assume here that these fork and join procedures will be used to form series–parallel graphs. In particular, join would not be correct if asked to join arbitrary nodes to form a graph that was not an LR-graph.[10]

Both of these procedures can be trivially extended to support forks and joins of more than two tasks at a time, but (as we shall see) there is actually no need for to do so. As most readers will be aware, it is easy to emulate n-ary forks and joins using only binary forks and joins (see Figure 4), yet interestingly the binary DAG version is as efficient in its use of labels as a direct n-ary version. As can be seen clearly in Figure 4, creating an LR-graph using binary forks and joins uses the same number of labels in G_x and G_y as creating a graph with n-ary forks and joins directly. Moreover, creating the graph using intermediary nodes allows all nascent children to be represented by a single (intermediary) node, rather than being created all at once.

4.2 Pipeline Parallelism

LR-graphs can also represent the kinds of parallelism seen in simple producer–consumer problems.[11]

As we mentioned in Section 1.2, sometimes a single concurrency unit, such as a process, may map to several tasks in the task DAG during its execution. As we

10. We will use fork and join in forming LR-graphs that are not series–parallel graphs in Section 4.2.

11. Complex pipelines of data between tasks are more likely to fit the CSP task model and be better checked with a determinacy checker that enforces Milner's conditions.

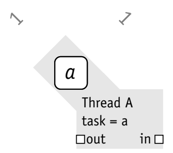
saw in Figure 1(b), pipelines are an instance of this phenomenon, being represented by several tasks. Thus, we will introduce the notion of a *thread*, where a thread is a sequence of related tasks that form a producer, filter, or consumer. An active thread T will have an associated *current task* T.task, but may have been represented by other tasks at earlier points in its execution. In addition, whenever a (potential) pipeline P $\rightarrow$ Q exists between two threads P and Q, there will be an associated *control queue* associated with the threads into which task labels can be queued and removed, such that P.out refers to the back of the queue and Q.in refers to the front of the queue.

Threads and thread pipelines are controlled by the following five operations (see Figure 5 for an example of how these primitives can used in practice to form a task graph that involves threads communicating via pipeline parallelism):

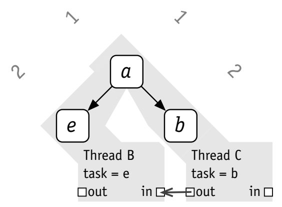
- *start*: given a task s, create a initial thread S
  - S.task := s
  - S.in := $\perp$ (an invalid control queue)
  - S.out := $\perp$ (an invalid control queue)
- *split*: thread P spawns two child threads C and D such that D $\rightarrow$ C (and P is destroyed)
  - Let c_1 and c_2 be the result of *fork*(P.task), where c_1 is the left child and c_2 is the right child
  - Let Q be a newly created empty control queue
  - C.task := c_1
  - C.in := Q
  - C.out := P.out
  - D.task := c_2
  - D.in := Q
  - D.out := P.in
- *merge*: threads G and H, where H $\rightarrow$ G, join to form a single child thread J (and G and H are destroyed)
  - If G and H are both ready to join, H.out and G.in will both refer to the same (empty) queue, which can be destroyed
  - Let j be the result of *join*(G.task, H.task)
  - J.task := j
  - J.in := H.in
  - J.out := G.out
- *send*: thread S sends a message
  - Enqueue x(S.task) into S.out
  - Insert a new label s_x into G_x, immediately after x(S.task)
  - Create a new task s such that $y(s)$:= y(S.task) and $x(s) := s_\mathrm{x}$
  - S.task := s
- *receive*: thread T receives a message
  - Dequeue a label, t_x from, T.in (waiting if necessary)
  - Create a new task t such that $y(t)$:= y(T.task) and $x(t) := t_\mathrm{x}$
  - S.task := t

Thread pipelines are created using the *split* and *merge* primitives given above, which are analogous to *fork* and *join* described in the previous section. Pipeline communication events between threads are modeled by *send* and *receive*. Notice that the destination in a *send* and the source in a *receive* are implicit because a task cannot have more than one input pipeline or more than one output pipeline in this scheme. Communication cannot take place between two arbitrary threads.

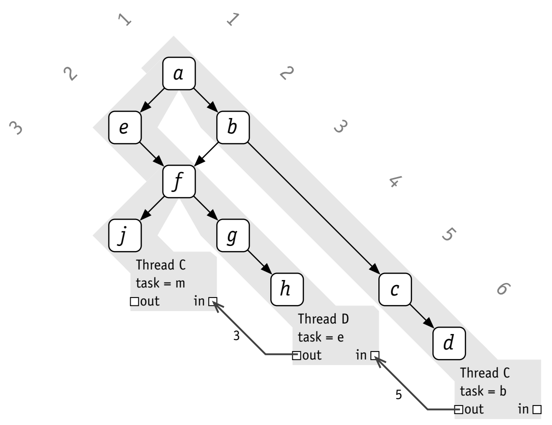
The *send* and *receive* primitives do not themselves send or receive program data. As with nested parallelism, all determinacy checking is performed on data in shared memory. These primitives simply indicate a synchronization point after which one task (the receiver) may access the data of another task (the sender). In particular, the primitives do not require the sending thread to wait for the receiving thread. Any synchronization required to ensure that a producer task does not overwrite data before a consumer has read it must be implemented in addition to these primitives. Moreover, if this additional synchronization is incorrectly implemented and causes invalid memory accesses (i.e., violations of Bernstein's conditions), determinacy checking will detect the error.



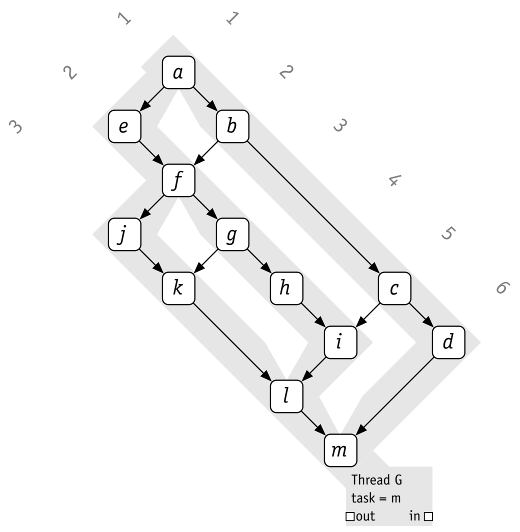
(a) When pipeline parallelism begins (with *start*) there is a single thread A consisting of one task, *p*.



(b) When thread A *splits*, it creates two threads: C $\longrightarrow$ B. Thus, there is a control queue between C and B that allows C to send the x-label of its current task (currently *b*, which has an x-label of 2) to thread B.



(c) At this point, thread C has used *send* twice, and thread B has performed a *receive* and then *split* to form two threads E $\longrightarrow$ D. Prior to the split, C $\longrightarrow$ B, now C $\longrightarrow$ E. Thread E has also used *send* to provide data to thread D. Threads D and E both have x-labels waiting in their control queues that will be retrieved when they perform a *receive*.



(d) Threads D and E have performed *receive* operations and then *merge*d to form a new thread F such that C $\longrightarrow$ F, after which threads C and F *merge*d to produce thread G.

Figure 5: How pipelines are formed using threads of tasks.

4.3 E□cient Ordered Lists

In the previous two subsections we saw how we can dynamically maintain two total orders, G_x and G_y, representing a task DAG, but we have not yet shown that these total orders can be maintained and queried e□ciently. Thankfully, this problem has e□cient solutions (Dietz & Sleator, □□□□; Tsakalidis, □□□□) that can perform insertions, deletions, and order queries in constant time and require space proportional to the number of items in the list.

We prefer the first of Dietz and Sleator's two ordered-list algorithms. Dietz and Sleator's second, more complex, algorithm has better theoretical performance, because it can perform all three operations in constant worst-case time, whereas their simpler algorithm can only perform ordered-list deletions and order queries in constant worst-case time, requiring constant amortized time for ordered-list insertions. In practice, Dietz and Sleator's first algorithm is less complex, easier to implement, and better suited to parallelization than its completely real-time counterpart.

4.3.1 Parallel Access to Ordered Lists

For determinacy checking to run in parallel, it is necessary to allow parallel access to the ordered-list structure. Sometimes, however, these accesses must be serialized to ensure that the ordered list is not inadvertently corrupted.

To describe the locking strategy required, we need to expose a little of the underlying details of Dietz and Sleator's ordered-list data structure (complete details are provided in the appendix).

Each item in the ordered list is tagged with two integers: an *upper tag*, which provides a coarse-grained ordering for tasks; and a *lower tag*, which orders nodes that have the same upper tag. When an item is inserted into the list, it is given the same upper tag as its predecessor and a suitable new lower tag. Sometimes the ordered list "runs out" of lower tags, at which point all the ordered-list entries with that upper tag are renumbered and given new upper and lower tags. Let us call this process a *local reorganization*. Sometimes the algorithm also runs out of room in the upper-tag space, and so must renumber the upper tags so that they are more evenly distributed. Let us call this process a *global reorganization*. Both kinds of reorganization can only be triggered by ordered-list insertions and are fairly rare in practice (e.g., for the benchmarks in Section 7, the average is about one reorganization per thousand ordered-list insertions).

The most common operation for the ordered lists in the determinacy checker will be comparisons between items in the list. Other list operations, such as insertions and deletions are less common. To reduce the chances of the ordered list becoming a bottleneck, we desire a locking strategy that maximizes the amount of parallelism while nonetheless remaining correct.

In our parallelization of the ordered-list structure (described in more detail elsewhere (O'Neill, □□□□)), order queries are lock free, although they may have to be retried if the ordered list undergoes a local or global reorganization while the order query is being performed.

Ordered-list insertions are performed while holding a shared lock, allowing multiple insertions to take place concurrently. This lock is upgraded to a local or global exclusive lock if a local or global reorganization is required. Deletions from the ordered list are performed while holding an exclusive lock.

We do not claim that this parallelization scheme is the most e□cient one possible, but it does appear to work well in practice, especially when the optimizations detailed in Section 6.3.3 are implemented.

5 Space and Time Complexity

So far, we have presented the foundations of the LR-tags method for determinacy checking. In this section, we will discuss its time and space complexity.

5.1 Space Complexity

The space overheads of determinacy checking come from two sources: the space needed to maintain w, r_x, and r_y for each determinacy-checked shared object, and the space needed for the ordered lists that represent task relationships. Clearly, the first overhead is constant factor, so our discussion will focus on showing that the space required to represent tasks in the ordered lists is also constant factor.

Task relationships are entirely expressed by their labels in the G_x and G_y orderings. The ordered lists that represent these orderings require space proportional to the length of the list, so the space required to represent task relationships is proportional to the size of G_x and G_y. A loose bound for the size of G_x and G_y is $O(s)$, where s is the final size of the task DAG they represent, but some applications have very large task DAGs without requiring much memory to execute, so we will seek a tighter bound.

If we remove those elements of G_x and G_y that cannot be referenced by the determinacy checker, we can obtain a tighter bound on the size of these ordered lists. By definition, nascent tasks do not need to be included in G_x and G_y because they cannot perform any actions—tasks only need to be added when they become effective. Similarly, a simple reference-counting strategy is sufficient to ensure that ethereal tasks that are no longer referenced as readers or writers of any data item are removed from G_y and G_x. (Reference counting adds a constant-space overhead to each ordered-list item.)

Thus, if v is the maximum number of determinacy-checked objects in existence at any one time during the program's execution, those objects could refer to at most $3v$ tasks. The only other references into G_x and G_y come from effective tasks, which, by definition, are either currently running or are represented in the task scheduler for other reasons (see Section 1.2).

The space overheads of determinacy checking are thus $O(v + e)$, where v is the maximum number of determinacy-checked objects in existence at any one time and e is the maximum number of effective tasks that existed during the program's run.

Usually, $e \ll v$, in which case the space overheads simplify to $O(v)$, but we should nevertheless consider that e could be greater than v.[12] As we discussed in Section 1.2, the upper bound for e depends on both the task scheduler and the program being executed. Because the effective tasks of a program are defined to be exactly those that the task scheduler must represent, it is reasonable to assume that a general-purpose task scheduler will store information about the effective tasks of the program, requiring $\Omega(e)$ space to do so. Thus the $O(e)$ space required to represent tasks in G_y and G_x is mirrored by the $\Omega(e)$ space used in task scheduler. (Highly specialized task schedulers may be able to avoid storing information about the relationships between tasks they schedule, but such schedulers may also admit highly specialized implementations of G_x and G_y; these specialized systems are beyond the scope of this paper.)

Thus the space overheads of determinacy checking both general nested-parallel and producer–consumer programs mirror the overheads of the programs themselves, resulting in constant-factor space overhead for determinacy checking.

12. Perhaps this common case explains why most published literature on the subject fails to take account of the space required to represent the effective tasks of the program when assessing the space requirements of determinacy checking.

5.2 Serial Time Complexity

For serial execution of a parallel program, the time overheads are governed by the time complexity of the ordered-list algorithm. The best currently known algorithm (Dietz & Sleator, [illegible]) is a constant real-time algorithm. Thus, the LR-tags algorithm can run in constant time for a parallel program executed using a serial scheduler.

As we mentioned in Section 4.3, the actual algorithm we use in our implementation of the LR-tags algorithm uses constant amortized time for list insertions. In normal use, comparisons (which require constant worst-case time) dominate insertions by a significant margin (see Table 2), so the impact of the amortization in the simpler of Dietz and Sleator's algorithms is slight.

5.3 Parallel Time Complexity

The time complexity of the LR-tags method is more difficult to assess when it runs in parallel. When running on a multiprocessor machine, multiple processors may attempt to access the same ordered list at the same time—these accesses must be arbitrated and sometimes serialized, as we discussed in Section 4.3. Serialization can introduce delays in accessing the ordered list and prevent accesses from taking constant-time.

As we show in Section 7, the actual overheads of serialization depend on the program—for many programs, serialization does not cause significant overhead.

6 Optimizations to the LR-tags Technique

In this section we will examine some useful optimizations that can be applied to the LR-tags technique. Although these optimizations do not improve the theoretical complexity of our approach, they can be useful in reducing the overheads involved in performing determinacy checking in practice.

6.1 Speeding Serial Performance

If execution is performed serially, following a left-to-right depth-first strategy that respects the commencement restrictions of the DAG, the order of tasks in G_{x} matches the order in which tasks are executed (as mentioned in the discussion of Figure 1 in Section 3). Thus, in this case, an ordered-list data structure is not required to represent G_{x}.

In fact, G_{x} is completely irrelevant in the serial case, because every "$r_{\mathrm{x}}(d) \preccurlyeq x(t)$" and "$x(w(d)) \preccurlyeq x(t)$" comparison (described in Section 3) will evaluate to true. The only way these comparisons could evaluate to false would be for a task from later in the serial execution sequence to have already read or written the datum, and that is clearly impossible.

Thus, for serial execution, there is little point in storing $r_{\mathrm{x}}(d)$ for each datum d or maintaining the G_{x} ordering. The following rules are sufficient for checking that reads and writes performed by a task t on a datum d are deterministic when the program is executed following a left-to-right depth-first strategy:

- *Reads* — A read is valid if $y(w(d)) \preccurlyeq x(t)$—that is, if it satisfies condition Bern-1. If the read is valid, $r_{\mathrm{y}}(d)$ may need to be updated: If $r_{\mathrm{y}}(d) \preccurlyeq y(t)$ then $r_{\mathrm{y}}(d) := y(t)$

- *Writes* — A write is valid if $r_{\mathrm{y}}(d) \preccurlyeq y(t)$—that is, if it satisfies condition Bern-2. If the write is valid, $w(d)$ and $r_{\mathrm{y}}(d)$ are updated to be t

6.2 Write-Restricted Data

In some cases, Bernstein's conditions are needlessly general and tighter determinacy restrictions can be used. For example, it is common for a program to read some input data representing the parameters of the parallel algorithm and then perform a parallel computation using those parameters. If the parallel component of the program only *reads* these parameters, it seems wasteful to use full-blown determinacy checking to ensure that these parameters are never corrupted by parallel code. In this section we will describe an obvious specialization for handling such cases.

A *write-restricted* object is a shared object that is subject to a stronger form of determinacy checking than we have described in previous sections. A write-restricted object may only be written to by a task that has no siblings or cousins. Both the start points and end points of the DAG are such points, but it is also possible for other tasks to satisfy this condition (such tasks occupy positions where the task DAG narrows to being a single task wide).

It is not necessary to perform *any* read checking for write-restricted data because any task can read the data without risking interference. It is similarly unnecessary for write-restricted data to store r_{x}, r_{y}, and w, because the only tasks that could have written the data are other tasks that have no cousins, and such tasks must be ancestors of the task performing the write.

Thus, the rules for a task t accessing a write-restricted object are:

- *Reads* — Reads are always deterministic; no checking is required.

- *Writes* — A write is allowed (and is deterministic) if $\forall s \in S : s \lhd t$, where S is the set of all tasks so far created by the program.

(Notice that write-restricted data does not require that any housekeeping information be stored with the data, in contrast to the normal LR-tags method, which requires that we store $r_{\mathrm{x}}(d)$, $r_{\mathrm{y}}(d)$, and $w(d)$ for each datum, d.)

Enforcing the above conditions for write-restricted data is orthogonal to checking determinacy using the LR-tags method. The write condition for write-restricted data can be enforced by keeping track of the width of the task DAG, or by some scheduler-dependent means. Thus, if all shared data is write-restricted, we can avoid maintaining G_{x} and G_{y} at all.

Write-restriction is not the only restriction we could impose to gain more e□cient determinacy checking. We mention it in part because it has proven useful in practice, but also because it illustrates that determinacy checking may be implemented more specifically and more e□ciently when the full generality of Bernstein's conditions is not required. Other useful possibilities, such as data that can only be written at object-creation time (and thus would not need to store r_x and r_y), or data that does not allow multiple concurrent readers, can also be developed within the LR-tags determinacy-checking framework. These variations can be useful not only because they can speed determinacy checking, but also because they can ensure that the use of shared objects matches the programmer's intent.

6.3 Storage Management

In Section 5.1, we mentioned that we may use reference counts to reclaim the entries in G_x and G_y that are no longer required, and thereby ensure a good space bound for determinacy checking. In this section we will look more closely at storage-management issues and discuss alternatives to reference counting.

When managing storage, there are four common approaches to deallocating storage that is no longer required: *explicitly programmed deallocation*, *no deallocation*, *reference-counted deallocation*, and *tracing garbage collection*. Explicitly programmed deallocation is not a viable strategy for managing G_y and G_x because we cannot statically determine when entries are no longer required,[13] but the three remaining approaches are all workable solutions, each with its own advantages and disadvantages.

6.3.1 Reference Counting

Reference counting has good asymptotic complexity, but can be costly in practice due to its bookkeeping overheads. Reference counting requires that objects store reference-count information and keep that reference-count information up to date, increasing both the size of objects and the time it takes to store or release a reference to an object by a constant factor. In practice, reference counts can a□ect the cache behavior of a program by increasing the number of writes it must perform.

Reference-counting techniques can also have di□culty with circularly linked structures, but this problem is not relevant to the LR-tags method because the references to entries in G_y and G_x are unidirectional.[14]

6.3.2 Tracing Garbage Collection

Tracing garbage collection can also reclaim unused entries in G_y and G_x. The exact asymptotic behavior of garbage collection depends on the collector, but, in practice, we can expect time overheads to decrease compared to reference counting, and storage requirements to increase slightly. As we are concerned with parallel code, it is worth noting that parallel garbage collectors exist with performance that scales well on multiprocessor machines (Endo *et al.*, □□□□; Blelloch & Cheng, □□□□). Our implementation of the LR-tags algorithm includes support for the Boehm garbage collector (□□□□) to show that such collection is feasible.

6.3.3 No Deallocation

For many applications, the overheads of deallocating entries from G_y and G_x outweigh the storage saved. For example, an application that spawns □□ threads over the course of its execution will use only a few kilobytes of memory for G_y and G_x. If we must link the program against □□ kilobytes of garbage-collector code (or increase the code size by □□% by using reference counting), we may reasonably wonder if the e□ort to reclaim such a small amount of storage is worthwhile.

More formally, the wasted space from performing no deallocation of entries from G_y and G_x is proportional to the size s of the complete task DAG for the program's execution. If $s \in \mathrm{O}(e + v)$, where e is the peak number of e□ective tasks that may exist during a run of the program and v is the number of determinacy-checked data items, this "leaky" approach falls within the space bound of the technique with reference counting employed.

In our experience, it tends to be quite straightforward to analyze an algorithm and determine when this ap-

13. In our C++ implementation of the LR-tags algorithm, we use explicitly programmed deallocation whenever possible; it is only G_y and G_x that require a more sophisticated storage-management strategy.

14. The internal representation of ordered lists G_y and G_x may involve a circularly linked list, but these internal pointers should never be included in the reference count for the entry.

proach is acceptable. See Table 2 and the accompanying discussion in the next section.

7 Real-World Performance

We have already discussed the theoretical complexity of the LR-tags method, but we also need to examine how it performs in practice. If the constant factors are too high, the technique is merely a theoretical curiosity, rather than a useful tool for software developers.

We will examine the performance of several parallel algorithms, both with and without determinacy checking. Given the complex interactions found on a modern computer system, with processor caches, virtual memory, and so forth, we are not trying to find an exact constant for the overheads, but rather to discover the extent of the performance penalty in broad terms. Also, we should regard our implementation of the LR-tags algorithm as only being a prototype to show the merits of the technique—a production version would no doubt be more carefully optimized and tied more closely to the particular development platform.

7.1 Benchmarks and Platform

The tests for our performance analysis are based on the benchmark programs provided with the Cilk parallel programming environment and used by Feng and Leiserson in benchmarking their Nondeterminator determinacy checker (□□□□). Cilk is a parallel extension of C that uses a work-stealing scheduler to achieve good time and space performance (Frigo *et al.*, □□□□). For our tests, we developed Cotton/Pthieves, an unremarkable parallel-programming environment that can run many Cilk programs after a few superficial source-code changes (O'Neill, □□□□).[15] Cotton uses the C preprocessor to perform a translation analogous to that performed by Cilk's cilk2c translator, whereas Pthieves is a work-stealing scheduler built on top of Posix threads.

We ported the following programs from the Cilk benchmark suite:

- mmult — A matrix-multiplication test that performs a matrix multiply without using a temporary array to hold intermediate results
- lu — An LU-decomposition test, written by Robert Blumofe
- fft — A fast Fourier transform, based on the machine-generated fftw code of Matteo Frigo and Steven G. Johnson (□□□□)
- heat — Simulates heat di□usion using a Jacobi-type iteration, written by Volker Strumpen
- knapsack — Solves the □-□ knapsack problem using a branch-and-bound technique

All of these benchmarks spend the bulk of their time accessing determinacy-checked data. Table 2 shows some of their properties.

7.2 Variations

We will compare several versions of the test algorithms:

- Serial execution of the algorithm with no determinacy checking
- Serial execution using Feng and Leiserson's SP-Bags determinacy-checking method
- Serial execution using the LR-tags determinacy-checking method
- Parallel execution of the algorithm running under the Cotton/Pthieves environment with no determinacy checking
- Parallel execution of the algorithm running under the Cilk environment with no determinacy checking
- Parallel execution using the LR-tags determinacy-checking method running under the Cotton/Pthieves environment

A test revealing the parallel performance of algorithms under the Cilk parallel system is included solely to show that the Cotton/Pthieves environment provides a reasonable test bed on which to run the LR-tags determinacy checker.

15. Initially we hoped to extend Cilk and substitute our LR-tags determinacy checker for Cilk's SP-Bags algorithm, but it turned out to be easier to implement a replacement for Cilk.

Benchmark	Checked Objects	Object Size	Number of Reads	Number of Writes	Total Dag Nodes	Dag Breadth	Total G_y Insertions	Peak G_y Entries
mmult	49,152	2048	4,194,304	2,146,304	2,843,791	49,152	947,931	65,097
lu	16,384	2048	1,398,016	723,776	1,765,174	4096	588,392	24,011
fft	12,582,913	8	65,273,856	62,914,563	932,251	262,144	310,751	28,755
heat	1,048,592	8	1,057,743,823	109,051,979	159,589	1024	12,956	263
knapsack	34	8	18,956,584	216	9,477,871	3,159,291	3,159,291	59

Table 2: Benchmark properties.

In the interest of fairness, the comparisons against Feng and Leiserson's SP-Bags determinacy-checking method use our implementation of their algorithm rather than their freely-available code. Had we used Feng and Leiserson's code, the results would be unfairly skewed against the SP-Bags method—in tests, using Cilk's standard settings, we have found the performance of Cilk's Nondeterminator to be disappointing, about one-tenth the speed of our implementation of their algorithm.[16] Also, recall that the SP-Bags algorithm cannot run in parallel, thus there is no test of the SP-Bags algorithm running in parallel.[17]

7.3 Results and Commentary

Figure 6 shows the performance of the five benchmarks, running both serially and in parallel on a □□-processor Sun Enterprise □□□□□, with □□ GB of memory, running *Solaris 7* (the benchmarks only use up to □□ of the □□ processors because we did not have exclusive access to the machine).[18]

7.3.1 mmult and lu

Figure 6(a) shows the performance of the mmult benchmark. This Cilk benchmark uses a divide-and-conquer technique to multiply two large (□□□□ × □□□□) matrices. Internally, the mmult benchmark represents the large matrix as a □□□ × □□□ matrix of □□ × □□ element blocks. The parallel matrix-multiplication algorithm decomposes the multiplication of the matrices into multiplications of these blocks, which are themselves multiplied using a fast serial algorithm.

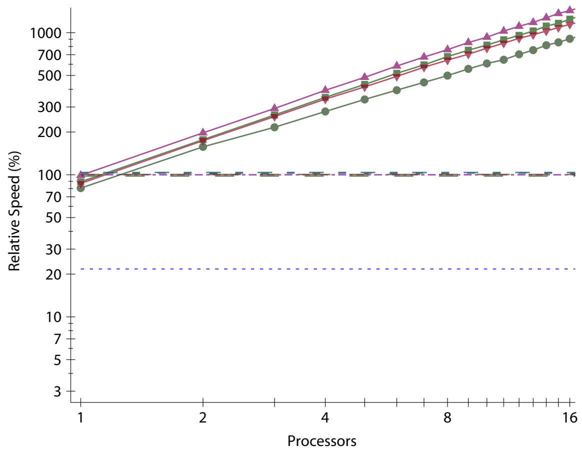
This test shows that for the right program, determinacy checking can exert a very low impact. For this program, the parallel implementation of the LR-tags algorithm determinacy checker exerts an overhead of less than □%, and both serial determinacy checkers exert so little overhead that it is virtually undetectable (in fact, it is sometimes *negative*, a matter we will discuss shortly). For serial execution, the SP-Bags and LR-tags methods turn in equivalent performance. Even on only two processors, parallel execution of this benchmark with determinacy checking enabled is faster than serial execution with no determinacy checking. Given these results, leaving determinacy checking permanently turned on may be perfectly acceptable for this application, if doing so allows us to feel more confident about the results of parallel execution.

The LU-decomposition benchmark, lu, is similar to the mmult benchmark both in terms of its algorithmic structure and its parallel structure. Both algorithms operate on matrices of □□ × □□ element blocks, and use $O(n \log n)$ tasks to decompose a matrix of n elements. As with the mmult benchmark, the overheads of determinacy checking are very low. In fact some of the test results show a *negative* overhead—for serial determinacy checking, the program runs □.□% faster with determinacy checking turned on. This anomalous result appears to be due to the cache behavior of the test platform. We investigated this quirk by adding two words of padding to

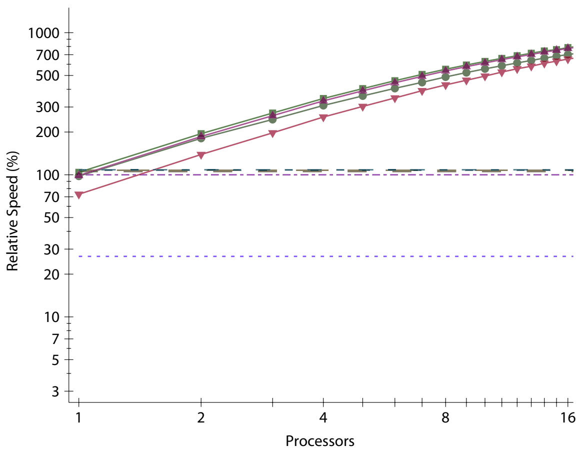
16. These observations are not unexpected—in the conclusion of their paper on the Nondeterminator, Feng and Leiserson remark that the released version of the Nondeterminator lacks some of the performance optimizations present in the benchmarking version they describe.

17. Actually, the SP-Bags algorithm *can* be generalized to run in parallel, but the generalization is nonobvious (O'Neill, □□□□).

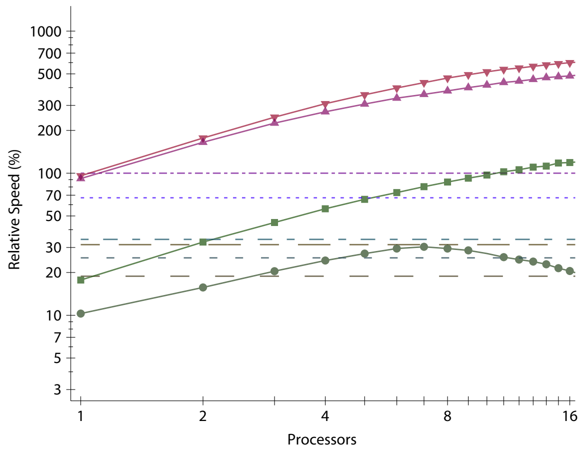
18. Our thanks to Emerich Winkler of Sun Microsystems Canada, who provided us with access to this powerful and expensive machine.



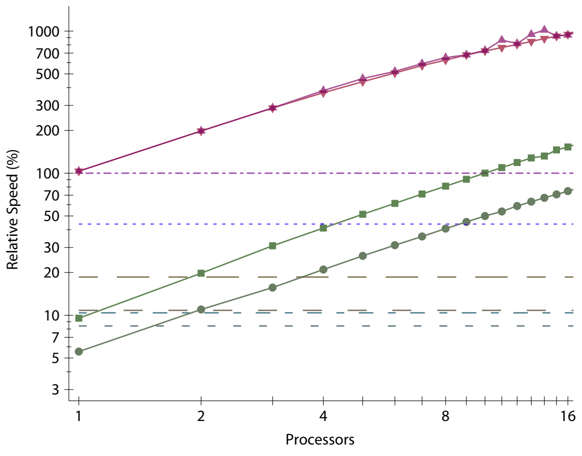
(a) mmult



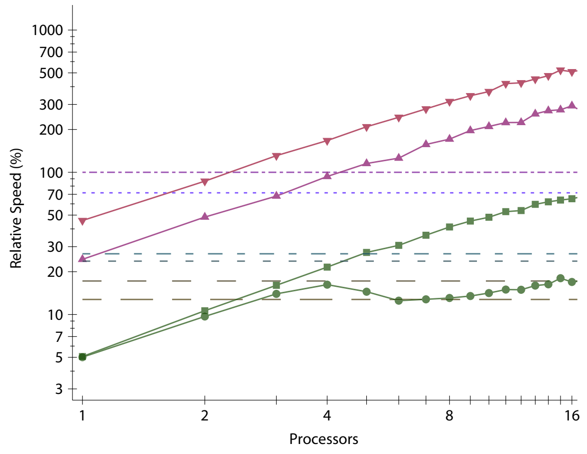
(b) lu



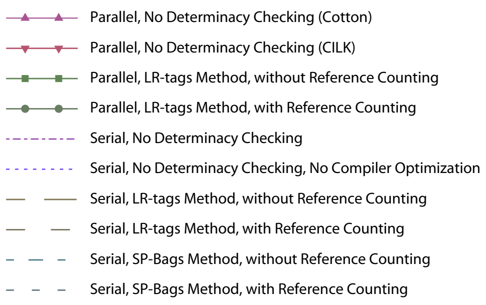
(c) fft



(d) heat



(e) knapsack


Parallel, No Determinacy Checking (Cotton)
Parallel, No Determinacy Checking (CILK)
Parallel, LR-tags Method, without Reference Counting
Parallel, LR-tags Method, with Reference Counting
Serial, No Determinacy Checking
Serial, No Determinacy Checking, No Compiler Optimization
Serial, LR-tags Method, without Reference Counting
Serial, LR-tags Method, with Reference Counting
Serial, SP-Bags Method, without Reference Counting
Serial, SP-Bags Method, with Reference Counting

key

Figure 6: Benchmark results.

the front of each [illegible] × [illegible] element block, mirroring the two words used to store determinacy-checking information, and found that the program ran almost [illegible]% faster. Thus, although determinacy checking does have a cost, in this particular case (an UltraSparc machine performing LU decomposition of a [illegible] × [illegible] matrix) that cost is offset by altered cache behavior.

The way in which determinacy checking was applied to these two algorithms played a significant role in their good performance. We employed determinacy checking at the level of the [illegible] × [illegible] blocks rather than at the level of individual elements. This choice was a natural one because the parallel parts of the both programs work at the level of these blocks. The alternative would have been to apply determinacy checking at the level of individual matrix elements, but such an approach would incur substantially worse time and space overheads, as we will discuss below.

To use determinacy checking on large objects, such as the matrix blocks used in both mmult and lu, we need to be confident that code that claims to be accessing the interior of a determinacy-checked object does not step beyond the boundaries of that object. *Bounds checking* provides one way to ensure this kind of correctness. For mmult and lu, we examined the code (which was, fortunately, trivial to reason about) and also double-checked that examination using runtime bounds checking. Runtime bounds checking is not used in the test runs evaluated here because we can be confident about the memory-access behavior of the serial algorithms. Moreover, adding bounds checking to all versions of the algorithm would inflate the running time of the algorithm, making the additional cost of determinacy checking appear smaller.

Using coarse-grained determinacy checking improves both the time and space performance of the algorithms by a significant constant factor. In our implementation, each determinacy-checked object must store r_x, r_y, and w, a space overhead of less than [illegible]% per [illegible] × [illegible] block. Had we been checking each matrix element individually, the space overhead would be an additional [illegible]% per object. Time overheads are also reduced by this block-based strategy. The matrix multiply performed by mmult performs three determinacy checks on the blocks it passes to the serial block-multiplication algorithm (a read check for each of the two const operands, and a write check for the result area). The serial algorithm then performs [illegible] floating-point operations to multiply the blocks. If we were checking at the level of individual numbers rather than blocks, we would have to perform [illegible] determinacy checks for each block multiply.

Finally, these two benchmarks are also interesting because they show some of the tradeoffs between time and space when it comes to the management of G_x and G_y. If v is the total number of elements in the matrices, both benchmarks create $O(v \log v)$ tasks in the course of their execution. With reference counting, the number of entries in G_x and G_y is bounded at $O(v)$. Without reference counting, the ordered lists exceed the $O(v)$ bound, resulting in a non-constant-factor space overhead for determinacy checking; but, as we can see from the performance graphs, avoiding reference counting does achieve a small but noticeable speed improvement.

7.3.2 fft, heat, and knapsack

In the remaining benchmarks, the overheads of determinacy checking are higher then they were for mmult and lu. As we saw in Table 2, these benchmarks perform determinacy checking on a large number of very small objects. In the fft benchmark, the objects are single-precision complex numbers; in the heat benchmark, the bulk of the determinacy-checked objects are double-precision floating-point numbers (but three integer objects are also checked). In the knapsack benchmark, the checked objects describe items that may be placed into the knapsack: Each object is a weight/value pair.

The heat benchmark performs more than a billion determinacy-checked reads and more than [illegible] million determinacy-checked writes, making it the benchmark that performs the most determinacy checking. Just under a third of those reads are performed on data that represent parameters of the test, defined at the time the benchmark begins. Because these parameters are never modified by parallel code, we declared them to be write-restricted (see Section 6.2), allowing them to be read with minimal overhead.

The knapsack benchmark is perhaps the cruelest benchmark of the set. It is the only benchmark where the number of threads grossly outnumbers the number of determinacy-checked objects. Whereas the other benchmarks consume a few megabytes of additional memory if reference counting is disabled, this benchmark requires a couple of hundred megabytes of memory to run if reference counting is disabled, which seems far less likely to be acceptable in a production environment.

Although the knapsack benchmark has the highest determinacy-checking overhead of the group, we can take comfort in the fact that it does not actually need the full generality of our runtime determinacy checker at all. All the determinacy-checked data in the program is written during the initialization by the initial thread. If the only determinacy-checked data is write-restricted data, the bulk of our general determinacy-checking mechanisms, and their costs, are unnecessary. If we strip out the general determinacy checker and only support determinacy checking for write-restricted data, the performance of this benchmark with determinacy checking becomes virtually indistinguishable from its performance without determinacy checking.

7.4 Performance Summary

Parallel determinacy checking can be practical, both as a debugging aid and in production code. The tests show the impact of runtime determinacy checking in both a good and a not-so-good light, revealing that the overheads of using runtime determinacy checking depend on the particular application, varying from virtually undetectable to quite noticeable. The number of shared objects, the size of those objects, and the frequency with which they are accessed influence the overheads of runtime checking. Similarly, the number of threads and the frequency with which threads are created also affects the cost that determinacy checking exacts.

8 Conclusion

We have developed a straightforward runtime test which can verify that a parallel program does not violate Bernstein's conditions for determinacy. Much like other runtime tests, such as runtime array-bounds checking, runtime checking only assures us that a particular kind of error did not occur for the given input, but such tests are nevertheless useful. As a debugging aid, these test results can be invaluable, and even in production code it may be better to suffer the overheads of determinacy checking, which as we have seen can be small for some algorithms, than generate an incorrect result.

Our techniques are orthogonal to other schemes that can enforce determinacy, such as deterministic communication channels or *I-structures* (Arvind *et al.*, [illegible]). Our methods are compatible with static checking, since we can apply runtime checking only to those algorithms and data structures that cannot be statically shown to be deterministic. Similarly, it is possible to mix runtime determinacy checks of shared data structures with use of I-structures and intertask communication through deterministic communication channels in one program. If some of the program uses locks to access some shared data, it is possible to debug that component using a lock-discipline checker such as Eraser (Savage *et al.*, [illegible]), or serial determinacy checker that supports locks such as All-Sets or Brelly (Cheng *et al.*, [illegible]). Finally, as with other determinacy-checking techniques that enforce Bernstein's conditions, our method can form the core of a system that enforces Steele's determinacy conditions.

Some questions remain open. For example, is it possible to further reduce the amount of contention and serialization required to access the ordered lists representing task relationships when running on a multiprocessor machine? Also, can our technique be extended to efficiently represent programs whose task dag is not a planar *st*-graph, or whose dag is a planar *st*-graph but its planar embedding is non-obvious and must be calculated on the fly?

Even if the above questions remain unaddressed, our technique has proven to be efficient in theory and applicable in practice.

Appendix: The List-Order Problem

In this appendix we present an algorithm that addresses the list-order problem.[19] This algorithm has been presented previously by Dietz and Sleator ([illegible]), but we explain it here both because maintaining an ordered list is fundamental to our method, and to show that this tech-

19. This appendix is a minor revision of one that appeared in a prior paper by the present authors (O'Neill & Burton, [illegible]).

nique can be implemented relatively easily.

We have chosen to present the simplest practical solution to the list-order problem that requires constant amortized time and space for insertion, deletion, and comparison. Other, more complex, solutions to the list-order problem exist, including a constant real-time algorithm (Dietz & Sleator, [illegible]; Tsakalidis, [illegible]).

Although the algorithm has been presented before, our presentation of it may be of some interest to those who might encounter it elsewhere, because we present it from a slightly different perspective, and reveal some properties that may not have been obvious in the original presentation.[20] Note, however, that for brevity we omit proofs of the complexity of this algorithm, referring the interested reader to the original paper (Dietz & Sleator, [illegible]) for such matters.

We will start by presenting an $\mathrm{O}(\log n)$ amortized-time solution to the list-order problem, which we will then refine to give the desired $\mathrm{O}(1)$ amortized-time solution.

An $\mathrm{O}(\log n)$ Solution to the List-Order Problem

The algorithm maintains the ordered list as a circularly linked list. Each node in the list has an integer label, which is occasionally revised. For any run of the algorithm, we need to know N, the maximum number of versions that might be created. This upper limit could be decided for each run of the algorithm, or, more typically, be fixed by an implementation. Selection of a value for N should be influenced by the fact that the larger the value of N, the *faster* the algorithm runs (because it operates using an interval subdivision technique), but that real-world considerations[21] will likely preclude the choice of an extremely large value for N. In cases where N is fixed by an implementation, it would probably be the largest value that can fit within a machine word, or perhaps a machine half-word (see below).

20. In particular, we show that it is only necessary to refer to the "base" when performing comparisons, not insertions. Also, some of the formulas given by Dietz and Sleator would, if implemented as presented, cause problems with overflow (in effect causing "mod M" to be prematurely applied) if M is chosen, as suggested, to exactly fit the machine word size.

21. Such as the fact that arithmetic on arbitrarily huge integers cannot be done in constant time. In fact, if we *could* do arithmetic on arbitrary-sized rationals in constant time, we would not need this algorithm, because we could then use a labeling scheme based on rationals.

The integers used to label the nodes range from 0 to $M-1$, where $M > N^2$. In practice, this means that if we wished N to be $2^{32}-1$, we would need to set M to 2^{64}. If it is known that a large value for N is not required, it may be useful for an implementation to fix M to be 2^w, where w is the machine word size, because much of the arithmetic needs to be performed modulo M, and this choice allows the integer-arithmetic overflow behavior of the processor to accomplish this modulo arithmetic with minimal overhead.

In the discussion that follows, we shall use $l(e)$ to denote the label of an element e, and $s(e)$ to denote its successor in the list. We shall also use the term $s^n(e)$ to refer to the nth successor of e; for example, $s^3(e)$ refers to $s(s(s(e)))$. Finally, we define two "gap" calculation functions, $g(e,f)$ and $g^*(e,f)$, that find the gap between the labels of two elements:

$$g(e,f) = (l(f) - l(e)) \mod M$$

$$g^*(e,f) = \begin{cases} g(e,f) & \text{if } e \neq f \\ M & \text{if } e = f. \end{cases}$$

To compare two elements of the list, x and y, for order, we perform a simple integer comparison of $g(\mathit{base}, x)$ with $g(\mathit{base}, y)$, where *base* is the first element in the list.

Deletion is also a simple matter—we just remove the element from the list. The only remaining issue is that of insertion. Suppose that we wish to place a new element i such that it comes directly after some element e. For most insertions, we can select a new label that lies between $l(e)$ and $l(s(e))$. The label for this new node can be derived as follows:

$$l(i) = \left(l(e) + \left\lfloor \frac{g^*(e, s(e))}{2} \right\rfloor \right) \mod M.$$

This approach is only successful, however, if the gap between the labels of e and its successor is greater than 1 (i.e., $g(e, s(e)) > 1$), as there must be room for the new label. If this is not the case, we must relabel some of the elements in the list to make room. Thus we relabel a stretch of j nodes, starting at e, where j is chosen to be the least integer such that $g(e, s^j(e)) > j^2$. (The appropriate value of j can be found by simply stepping through the list until this condition is met). In fact, the label for e is left as is, and so only the $j-1$ nodes that succeed e must have their labels updated. The new labels for the nodes $s^1(e), \ldots, s^{j-1}(e)$ are assigned using the formula below:

$$l(s^k(e)) = \left(l(e) + \left\lfloor \frac{k \times g^*(e, s^j(e))}{j} \right\rfloor\right) \mod M.$$

Having relabeled the nodes to create a su□ciently wide gap, we can then insert a new node using the procedure we outlined earlier.

Refining the Algorithm to $\mathbf{O(1)}$ Performance

The algorithm, as presented so far, takes $\mathrm{O}(\log n)$ amortized time to perform an insertion (Dietz & Sleator, □□□□). However, there is a simple extension of the algorithm which allows it to take $\mathrm{O}(1)$ amortized time per insertion (Tsakalidis, □□□□; Dietz & Sleator, □□□□), by using a two-level hierarchy: an ordered list of ordered sublists.

The top level of the hierarchy is represented using the techniques outlined earlier, but each node in the list contains an ordered sublist which forms the lower part of the hierarchy. An ordered-list element e is now represented by a node in the lower (child) list, $c(e)$, and a node in the upper (parent) list, $p(e)$. Nodes that belong to the same sublist will share the same node in the upper list; thus,

$$p(e) = p(f), \ \forall e, f \text{ s.t. } c(e) = s_c(c(f))$$

where $s_c(e_c)$ is the successor of sublist element e_c. We also define $s_p(e_p)$, $l_c(e_c)$, and $l_p(e_p)$ analogously.

The ordered sublists are maintained using a simpler algorithm. Each sublist initially contains $\lceil \log n_0 \rceil$ elements, where n_0 is the total number of items in the ordered list we are representing. That means that the parent ordered list contains $n_0 / \log n_0$ entries.

Each sublist element receives an integer label, such that the labels of the elements are, initially, $k, 2k, \ldots, \lceil \log n_0 \rceil k$, where $k = 2^{\lceil \log n_0 \rceil}$. When a new element n_c is inserted into a sublist after some element e_c, we choose a label in between e_c and $s_c(e_c)$. More formally,

$$l_c(n_c) = \left\lceil \frac{l_c(e_c) + l_c(s_c(e_c))}{2} \right\rceil.$$

Under this algorithm, a sublist can receive at least $\lceil \log n_0 \rceil$ insertions before there is any risk of there not being an integer label available that lies between e_c and $s_c(e_c)$.

To insert an element i after e in the overall ordered list, if the sublist that contains $c(e)$ has su□cient space, all that needs to be done is to insert a new sublist element i_c after $c(e)$, and perform the assignments $c(i) \leftarrow n_c$ and $p(i) \leftarrow p(e)$. However, if the sublist contains $2\lceil \log n_0 \rceil$ elements, it may not be possible to make insertions after some of its elements. In that case, we must split the sublist into two sublists of equal length, relabeling both sets of $\lceil \log n_0 \rceil$ nodes following the initial labeling scheme. The nodes of the first sublist are left with the same parent e_p but nodes of the second sublist are given a new parent i_p which is inserted in the upper ordered list immediately after e_p.

These techniques are used for insertions until the number of nodes n in the overall ordered list is greater than $2^{\lceil \log n_0 \rceil}$, because at that point $\lceil \log n \rceil > \lceil \log n_0 \rceil$. When this happens (every time n doubles), we must reorganize the list so that we now have $n / \lceil \log n \rceil$ sublists each containing $\lceil \log n \rceil$ nodes, rather than having $n / \lceil \log n_0 \rceil$ sublists of $\lceil \log n_0 \rceil$ nodes.

Since this new scheme only creates $n / \lceil \log n \rceil$ entries in the upper ordered list, M can be slightly lower. Recall that previously we imposed the condition $M > N^2$. Now we have a slightly smaller M, since it need only satisfy the condition

$$M > (N / \lceil \log N \rceil)^2$$

In practice, this condition would mean that if we required up to 2^{32} list entries, we would need an arena size of 2^{54} (instead of 2^{64}). Similarly, if we wished all labels to fit within a machine word, and so wished M to be 2^{32}, we would be able to have a little over 2^{20} items in an ordered list at one time (instead of 2^{16} items).

Following this scheme, we can implement e□cient ordered lists and by simple derivation, a quick and e□ective scheme for representing relationships between tasks in a task DAG.

References

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. □□□□. I-Structures: Data Structures for Parallel Computing.

ACM Transactions on Programming Languages and Systems, □□(□), □□□–□□□.

Babak Bagheri. □□□□. *Parallel Programming With Guarded Objects*. Ph.D. thesis, The Pennsylvania State University.

Vasanth Balasundaram and Ken Kennedy. □□□□ (June). Compile-Time Detection of Race Conditions in a Parallel Program. *Pages – of: Proceedings of Fourth International Conference on Supercomputing*, Crete, Greece.

Adam Beguelin. □□□□. *Deterministic Parallel Programming in Phred*. PhD Dissertation, University of Colorado at Boulder.

Adam Beguelin and Gary Nutt. □□□□. Visual Parallel Programming and Determinacy: A Language Specification, an Analysis Technique, and a Programming Tool. *Journal of Parallel and Distributed Computing*, □□(□), □□□–□□□.

Arthur J. Bernstein. □□□□. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, □□(□), □□□–□□□.

Garrett Birkho□. □□□□. *Lattice Theory*. □rd edition. Colloquium Publications, vol. □□. American Mathematical Society, New York.

Guy E. Blelloch and Perry Cheng. □□□□. On Bounding Time and Space for Multiprocessor Garbage Collection. *Pages – of: Proceedings of the ACM SIGPLAN ' Conference on Programming Language Design and Implementation*, Atlanta. ACM SIGPLAN Notices, vol. □□, no. □. ACM Press.

Hans-Juergen Boehm and Mark Weiser. □□□□. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, □□(□), □□□–□□□.

Per Brinch Hansen. □□□□a. Interference control in SuperPascal — a block-structured parallel language. *The Computer Journal*, □□(□), □□□–□□□.

Per Brinch Hansen. □□□□b. SuperPascal—A Publication Language for Parallel Scientific Computing. *Concurrency: Practice and Experience*, □(□), □□□–□□.

David Callahan and Jaspal Subhlok. □□□□. Static Analysis of Low-Level Synchronization. *Pages – of: Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, New York, NY, vol. □□. ACM Press.

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. □□□□. Detecting Data Races in Cilk Programs that Use Locks. *Pages – of: SPAA ' . Proceedings of the Tenth Annual ACM Symposium On Parallel Algorithms and Architectures*, Puerto Vallarta Mexico. ACM Press.

Jong-Deok Choi and Sang Lyul Min. □□□□. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. *Pages – of: Proceedings of the rd ACM Symposium on Principles & Practice of Parallel Programming (PPoPP)*, Williamsburg, VA, vol. □□. ACM Press.

Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. □□□□. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall. Chapter □.

Paul F. Dietz and Daniel D. K. Sleator. □□□□. *Two Algorithms for Maintaining Order in a List. To appear.* A preliminary version appeared in *Proceedings of the th Annual ACM Symposium on Theory of Computing*, New York, May □□–□□, □□□□.

Edsger Wybe Dijkstra. □□□□. Co-operating Sequential Processes. *Pages – of:* F. Genuys (ed), *Programming Languages*. Academic Press.

Anne Dinning and Edith Schonberg. □□□□. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. *Pages – of: Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*. ACM Press.

Anne Dinning and Edith Schonberg. □□□□. Detecting Access Anomalies in Programs with Critical Sections. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, □□(□□), □□–□□.

Richard J. Du□n. □□□□. Topology of Series-Parallel Networks. *Journal of Mathematical Analysis and Applications*, □□, □□□–□□□.

Perry A. Emrath and Davis A. Padua. □□□□ (May). Automatic detection of nondeterminacy in parallel programs. *Pages – of: Proceedings of the Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin.

Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. □□□□. A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. *In:* ACM (ed), *SC' : High Performance Networking and Computing: Proceedings of the ACM/IEEE SC Conference*, San Jose, California, USA. ACM Press and IEEE Computer Society Press.

Mingdong Feng and Charles E. Leiserson. □□□□. E□cient Detection of Determinacy Races in CIlk Programs. *Pages – of: SPAA ' . Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, Newport, RI, USA. ACM Press.

Michael J. Flynn. □□□□. Some Computer Organizations and Their E□ectiveness. *IEEE Transactions on Computers*, **C**-□□(□), □□□–□□□.

Matteo Frigo and Steven G. Johnson. □□□□ (Sept.). *The Fastest Fourier Transform in the West*. Technical Report MIT/LCS/TR-□□□. Massachusetts Institute of Technology.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. □□□□ (May). The Implementation of the Cilk-□ Multithreaded Language. *Pages – of: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI')*, vol. □□.

Charles Antony Richard Hoare. □□□□. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, □□(□□), □□□–□□□.

Charles Antony Richard Hoare. □□□□. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice Hall International, Englewood Cli□s, N.J., USA.

Tiko Kameda. □□□□. On the Vector Representation of the Reachability in Planar Directed Graphs. *Information Processing Letters*, □(□), □□–□□.

Gregory Maxwell Kelly and Ivan Rival. □□□□. Planar Lattices. *Canadian Journal of Mathematics*, □□(□), □□□–□□□.

Percy Alexander MacMahon. □□□□. The Combination of Resistances. *The Electrician*, Apr.

John McCarthy. □□□□. A Basis for a Mathematical Theory of Computation. *Pages – of:* P. Bra□ort and D. Hirschberg (eds), *Computer Programming and Formal Systems*. North Holland, Amsterdam, The Netherlands.

John Mellor-Crummey. □□□□. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. *Pages – of: Proceedings of Supercomputing ' .* IEEE Computer Society Press.

Barton P. Miller and Jong-Deok Choi. □□□□. A Mechanism for E□cient Debugging of Parallel Programs. *Pages – of: Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, New York, NY, vol. □□. ACM Press.

Robin Milner. □□□□. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science. Prentice Hall International, Englewood Cli□s, N.J., USA.

Sang Lyul Min and Jong-Deok Choi. □□□□ (Apr.). An E□cient Cache-Based Access Anomaly Detection Scheme. *Pages – of: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Palo Alto, California.

Robert H. B. Netzer and Barton P. Miller. □□□□. What Are Race Conditions? *ACM Letters on Programming Languages and Systems*, □(□), □□–□□.

Itzhak Nudler and Larry Rudolph. □□□□ (May). Tools for the E□cient Development of E□cient Parallel Programs. *In: Proceedings of the First Israeli Conference on Computer Systems Engineering*.

Melissa E. O'Neill. □□□□ (□□ Nov.). *Version Stamps for Functional Arrays and Determinacy Checking: Two Applications of Ordered Lists for Advanced Programming Languages*. Ph.D. thesis, School of Computing Science, Simon Fraser University.

Melissa E. O'Neill and F. Warren Burton. □□□□. A New Method for Functional Arrays. *Journal of Functional Programming*, □(□), □□□–□□□.

John Riordan and Claude Elwood Shannon. □□□□. The Number of Two-Terminal Series-Parallel Networks. *Journal of Mathematics and Physics*, □□, □□–□□.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. □□□□. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, □□(□), □□□–□□□.

Edith Schonberg. □□□□ (□□–□□ June). On-the-Fly Detection of Access Anomalies. *Pages – of: Proceedings of the ACM SIGPLAN ' Conference on Programming Language Design and Implementation*, Portland, OR, USA, vol. □□.

Guy L. Steele, Jr. □□□□. Making Asynchronous Parallelism Safe for the World. *Pages – of: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press.

Robert Endre Tarjan. □□□□. E□ciency of a Good but Not Linear Set Union Algorithm. *Journal of the ACM*, □□(□), □□□–□□□.

Robert Endre Tarjan. □□□□. Applications of Path Compression on Balanced Trees. *Journal of the ACM*, □□(□), □□□–□□□.

Richard N. Taylor. □□□□. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, □□(□), □□□–□□□.

Athanasios K. Tsakalidis. □□□□. Maintaining Order in a Generalized Linked List. *Acta Informatica*, □□(□), □□□–□□□.

Jacobo Valdes. □□□□ (Dec.). *Parsing Flowcharts and Series-Parallel Graphs*. Ph.D. thesis, Stanford University.

Zhiwei Xu and Kai Hwang. □□□□. Language Constructs for Structured Parallel Programming. *Pages – of:* V.K. Prasanna and L.H. Canter (eds), *Proceedings of the Sixth International Parallel Processing Symposium*. IEEE Computer Society Press.