Abstract

Deterministic OpenMP

Amittai F. Aviram

2012

Researchers widely agree that determinism in parallel programs is desirable. Although experimental parallel programming languages have long featured deterministic semantics, in mainstream parallel environments, developers still build on non-deterministic constructs such as mutexes, leading to time- or schedule-dependent heisenbugs. To make deterministic programming more accessible, we introduce DOMP, a deterministic extension to OpenMP, preserving the familiarity of traditional languages such as C and Fortran, and maintaining source-compatibility with much of the existing OpenMP standard. Our analysis of parallel idioms used in 35 popular benchmarks suggests that the idioms used most often (89% of instances in the analyzed code) are either already expressible in OpenMP's deterministic subset (74%), or merely lack more general reduction (12%) or pipeline (3%) constructs. DOMP broadens OpenMP's deterministic subset with generalized reductions, and implements an efficient deterministic runtime that acts as a drop-in replacement for Gnu's widely used conventional OpenMP support library GOMP, on mainstream Unix platforms. We evaluate DOMP with several existing OpenMP benchmarks, each requiring under 50 source line changes and a majority requiring none, as well as several benchmarks we ported to OpenMP/DOMP. We find DOMP's efficiency and scalability comparable to GOMP in 7 of 11 benchmarks, suggesting that a deterministic model for mainstream parallel programming may be well within reach.

Deterministic OpenMP

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Amittai F. Aviram

Dissertation Director: Bryan A. Ford

December 2012

Contents

List of Figures

List of Tables

Acknowledgments

Above all, I wish to thank my advisor, Bryan A. Ford, for his guidance, inspiration, help, and collaboration throughout this project. I also thank the other members of my dissertation committee for their helpful comments and advice: Emery Berger, Ramakrishna Gummadi, and Zhong Shao. At an earlier stage of the project, James Aspnes, Paul Hudak, Drew McDermott, and Yang Richard Yang provided helpful guidance. I was privileged to work alongside brilliant classmates and colleagues, including Antonios Stampoulis, Andreas Voellmy, Alex Vaynberg, Bandan Das, Yu Zhang, Liang Gu, and all other members of the Yale Decentralized and Distributed Systems (DeDiS) lab group. Finally, I thank my family and friends for their unyielding support and encouragement: Octavio Zaya, Blake Gilson, Rachel Aviram, Mariva Aviram, Amalia Kessler, Adam Talcott, and Alex Demac.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1017206.

Chapter 1

Introduction

The proliferation and continual widening of commodity multicore hardware have increased the importance of parallel computing [8, 52, 80]. A major challenge that this development introduces, particularly for shared memory parallelism, is how to ensure that execution will be *deterministic*. Nondeterministic execution of a buggy program can make certain bugs manifest themselves only unpredictably on some runs; such hard-to-reproduce "heisenbugs" hamper debugging and development [71, 74]. By contrast, determinism goes beyond eliminating heisenbugs: it implies the guaranteed ability to replicate execution sequences exactly, which, in turn, lies at the core of techniques for fault tolerance [27] and accountability [46]. Determinism may even provide a basis for timing channel control [12].

To address this challenge, researchers have developed new programming languages, record-and-replay systems, and thread management schemes.

Experimental languages such as Cilk [20], SHIM [36], Parallel Haskell [28], and Deterministic Parallel Java [22] have explored the appeal and benefits of "deterministic by default" programming models [68, 21]. However, new deterministic languages require developers to adopt unfamiliar coding styles, type systems, and/or synchro-

nization constructs, however, and to rewrite or extensively modify existing code.

Deterministic record-and-replay systems such as ReVirt [35], LReplay [29], and Karma [15] are helpful for finding heisenbugs, but software-based systems are generally too slow for continuous deployment, while sufficiently fast systems require custom extensions to hardware.

In the area of thread management schemes, deterministic schedulers [33, 77, 18, 17, 70] can execute existing software reproducibly by imposing artificial thread interleaving schedules, but their *programming model* remains nondeterministic: any program, input, compiler, or scheduler change may yield a different schedule and reveal heisenbugs. The experimental operating system Determinator [13] offers a deterministic, race-free programming model compatible with existing languages, but supports only hierarchical fork/join synchronization and requires adoption of a new OS.

Many common parallel programming idioms yield naturally deterministic synchronization behavior [11, 88]: e.g., fork/join, barriers, tasks, pipelines, futures [49], and write-once structures [7]. Conventional models such as `pthreads`, however, leave to the developer the burden of implementing these deterministic idioms correctly atop nondeterministic low-level primitives, such as mutexes and condition variables. Even frameworks such as OpenMP, offering higher-level block-structured synchronization, also include nondeterministic primitives that developers often mix into programs: e.g., *atomic* and *critical* sections, and *flush* memory barriers used in ad-hoc synchronization [101].

We therefore ask, how indispensable are these nondeterministic primitives in parallel software? Could most mainstream parallel software rely mostly or exclusively on deterministic constructs, given the right language and system capabilities? And could a purely deterministic programming model accommodate legacy parallel pro-

grams, originally written without determinism in mind?

As a step toward answering these questions and making deterministic programming models mainstream, we analyzed three parallel benchmark suites (SPLASH, NPB-OMP, and PARSEC), hand-counting the instances of synchronization constructs and classifying each according to its use in the program. We find that 66% of synchronization instances already use deterministic constructs such as fork/join and barrier, and a further 25% of instances used nondeterministic primitives such as mutexes as building blocks to express higher-level *idioms* that are, in fact, naturally deterministic. These latter cases reflect situations in which the programming language apparently lacked the means to express the deterministic parallel idiom the developer intended. The largest such class was *reductions* (12% of instances), in languages that had no such construct, or, in OpenMP, where language restrictions failed to support the desired reductions, such as vector addition. In a few cases, the programmer used nondeterministic primitives to build pipelines (3%) or task queues (4%), which are also expressible deterministically with appropriate language support. Only 9% of instances contributed to algorithms we found to be genuinely nondeterministic, such as user-level scheduling or load balancing. Although these results cover only a limited benchmark code-base and may not fully represent larger parallel applications, they nevertheless suggest that a vast majority of parallel idioms in realistic software should be expressible given appropriate deterministic constructs.

To test this hypothesis, we developed Deterministic OpenMP (DOMP), as first proposed in [9], an adaptation of the popular (but nondeterministic) OpenMP environment [78], to offer a practical deterministic parallel programming model, while retaining familiarity and code compatibility. DOMP is implemented as a user-level library that serves as a drop-in replacement for Gnu OpenMP (GOMP) [45], and supports most of OpenMP's familiar block-structured parallel constructs in both C

and Fortran, with the added guarantees of deterministic execution and freedom from data races. DOMP follows the Working-Copies Determinism (WCD) programming model [11], which enforces determinism by providing each thread an isolated logical copy of shared state at fork events, and deterministically merging changes at joins or barriers. As a side-effect, DOMP consistently and deterministically detects and reports conflicting writes by parallel threads, which, in a traditional OpenMP environment, would yield nondeterministic races. In addition, DOMP includes an *extended reduction* mechanism for arbitrary types and operations alongside OpenMP's existing value-type reductions. Both the standard and extended reductions maintain determinism, not only in the final outcome, but also in the intermediate evaluation steps, making it possible to reduce correctly over operations that are associative but not necessarily commutative.

Experiments indicate that, on many standard benchmarks, DOMP's deterministic machinery incurs relatively modest overheads compared with nondeterministic execution using GOMP: under 12% overhead in 5 of 11 benchmarks on 16 CPUs, and under 65% overhead in 9 of 11 benchmarks. Scaling patterns largely follow those of the reference implementation. It is not DOMP's aim to improve on the performance or scalability of the many prior approaches to deterministic parallelism [33, 77, 18, 17, 13, 70]. Nevertheless, our results suggest that it may be reasonable to expect future versions of familiar, mainstream parallel environments such as OpenMP to include an expressive deterministic parallel programming model as a subset, and to offer a deterministic, race-free, and conflict-detecting execution mode of the type DOMP implements, which developers can enable in production or only for debugging, as performance considerations permit.

Our contributions include, to our knowledge, the first analysis of the uses of nondeterministic synchronization primitives in standard benchmarks; a new, determin-

istic version of OpenMP; a deterministic extended reduction construct for OpenMP; and a demonstration that programs currently written using nondeterministic synchronization mechanisms could, in principle, be converted so as to conform to a thoroughly deterministic programming model with fairly small changes and only modest additional overhead in many cases.

The rest of this paper proceeds as follows: Chapter 2 gives a broader background for the DOMP project, including discussion of related work; Chapter 3 reports on our analysis of the uses of synchronization in parallel programs; Chapter 4 presents the design and semantics of DOMP; Chapter 5 describes DOMP's extended reduction feature in greater detail, with examples of replacing nondeterministic code with deterministic equivalents using this feature; Chapter 6 provides details on our DOMP implementation; Chapter 7 evaluates DOMP's performance; and Chapter 8 concludes.

Chapter 2

Background and Related Work

The DOMP project contributes both a user-level scheme for ensuring parallel determinism and, within that scheme, a new extended reduction feature. In this chapter, we review both the research context for the overall project of deterministic parallelism and that of the more specific reduction construct. Following this, we review areas of research closely connected to the DOMP project, including other solutions to the problem of nondeterministic parallel computing. These research areas fall into several categories:

- Programming languages
- Record and replay systems
- Debugging tools
- Transactional memory systems

First, however, we place DOMP in general, and extended reductions in particular, in a conceptual context.

2.1 Deterministic Parallel Computing

A parallel program is *deterministic* if the input alone determines the output, regardless of extrinsic events such as the OS's thread scheduling. The contrary of this condition is *nondeterminism*. Note that nondeterminism is distinct from underspecification. A programming language or library, for instance, may leave some details unspecified, and yet the system may execute the program deterministically, in the sense that the same input always results in the same output and behavior. In such a case, some other elements of the runtime system, "downstream" from the program, are sufficiently specified so as to fill in the semantic gaps that the program has left indeterminate. We call an execution *nondeterministic* if the output or behavior depend in some way on events, such as thread scheduling or hardware timing, over which the program has no control, and that no programmer would be able to predict based on the program and the input. Underspecification in a system allows such nondeterminism to occur.

If any two threads have access to the same location in shared memory, and at least one such access is a write, the definite ordering of such accesses is essential. Such a condition is a *conflict* in accesses, which may be either a read-write conflict (one thread reads from the same location that another thread updates) or a write-write conflict (two threads write different values to the same location). A lack of order on conflicting accesses, where the sequence of accesses and data visibility changes differs from one run to the next, is a prime manifestation of nondeterminism in the system. We call this condition a *data race*. In a data race, the reading thread may get either the old or the updated value, depending on timing or scheduling factors not prescribed in the program. Thus any data race is likely to cause a heisenbug, since it will likely result in different outcomes on different runs, at least one of which

is unwanted and erroneous, but, by the same token, not reliably reproducible.

By itself, the above definition of determinism allows for a range of behaviors, depending on the *synchronization* and *memory consistency models*.

Synchronization is *naturally* deterministic if program logic alone determines how and at what points different threads interact, depending only on computation state and not on timing. A *fork* deterministically creates a child thread at a program-defined point in the parent's execution, for example. Similarly, a *join* deterministically combines the parent's and child's flows at program-defined points in *both* threads. Other common constructs, such as mutex locks, condition variables, semaphores, monitors, and OpenMP's *atomic*, *critical*, and *flush*, are semantically nondeterministic: they allow a thread to signal or wake up an *unspecified*, nondeterministic recipient—e.g., the next holder of a lock—or to wait for an event from a nondeterministic source—e.g., any of several threads that might signal a condition variable. The choice of the recipient or source thread in such cases depends on the runtime scheduler or some other agent, not on what the program itself specifies.

In a similar way, a memory consistency model is deterministic if program logic alone determines the order in which threads update shared memory and in which each thread sees such updates. A naturally deterministic memory consistency model precludes data races, since it by definition orders all memory accesses to shared memory in which any done thread can affect another.

Classic memory consistency models, including sequential consistency [65, 89] and relaxed models [43, 86], introduce nondeterminism, by leaving memory access interleavings underspecified. That is, even if a program uses only deterministic synchronization abstractions (e.g., fork/join) and runs on sequentially consistent hardware, data races and execution timing can make the program exhibit any one of an exponentially large variety of sequentially consistent memory access interleavings.

A sequentially consistent program, for instance, meets the following requirements [65]:

- "The result of any execution is the same as if the [memory] operations of all the processors [or threads] were executed in some sequential order," implying that every thread in the program sees the same updates to shared memory in the same order (but not at all necessarily at the same time)
- The order in which every thread sees these shared memory updates corresponds to the order in which the program specifies them.

Thus if the program has Thread A update x twice, first to the value 1 and to the value 42, Thread B reads x twice, but its first read operation returns 42, the system violates sequential consistency. Now suppose that the system is indeed sequentially consistent (so that Thread B reads 1 first). After these operations, both Thread A and Thread B increment x (whose current value is 42). Suppose that, in their respective non-atomic increment operations, both threads read x before either one writes to x. Each thread sees the same sequence of updates, i.e., from 42 to 43 (erroneously), and thus they do not violate sequential consistency. The same applies if Thread A writes 43 back to x before Thread B reads x: in this case, both threads see the same sequence of two updates to x. Thus the classic example of a data race can easily occur under in a sequentially consistent system.

Weaker memory consistency models impose more constraints on the *synchronization* operations—here, the fork before and the join after any of the memory accesses mentioned—but even fewer constraints on other operations. Thus the same data race could occur *a fortiori*.

Transactional memory [52], whether implemented in software [34, 53] or in hardware [54], may assume that the underlying memory architecture implements either

sequential consistency or a more relaxed memory consistency model. Transactional memory ensures the *atomicity* of groups of memory accesses within a transaction, and therefore prevents data races within the transaction unit. It does not, however, specify the order of transactions among threads, but leaves this specification to the programmer, much as do low-level synchronization primitives such as mutex locks and condition variables. By itself, then, transactional memory effectively allows higher-level data races [5] at the larger scale of transactions, though it prevents them at the level of particular read or write accesses.

By contrast, workspace consistency [11] requires that every write to shared memory be associated with the ensuing read or reads according to a prescribed pattern that the programmer could therefore deduce by examining the program, with knowledge of the ordering rules of the system. We examine workspace consistency further in 4.1.

Several research efforts have focused on running a nondeterministic program, perhaps containing data races, deterministically. Deterministic schedulers such as DMP [33] and CoreDet [17] execute a semantically nondeterministic program repeatably, by artificially synthesizing *one particular* (arbitrary) interleaving of the program's synchronization and memory access events. Deterministic scheduling can reproduce races once detected, but it neither eliminates races nor guarantees that they *will* be detected. A program's behavior may still depend on the (deterministic) execution schedule in subtle ways not explicit in program logic, as in this example:

```
// Thread A:
{
  if (input_is_typical)
    do_a_lot();
  x++;
}
```

```
// Thread B:
{
  do_a_little();
  x++;
}
```

Under "typical" program inputs, abstracted here via '`input_is_typical`', a deterministic scheduler may always cause thread B to reach its increment of x while thread A is executing its long-running and non-conflicting `do_a_lot()`. But some particular "rare" input, which unit tests may not have covered, may cause the threads' increments to line up in the deterministic execution schedule, resulting in a classic data race and an "input-dependent heisenbug."

Another scheduling approach, Kendo [77], enforces deterministic synchronization by ordering lock acquisitions and releases, but does not constrain memory accesses, so that programs containing data races (through a failure to lock) will run nondeterministically.

Grace [18] is a deterministic scheduler that emulates sequential consistency by means of speculative execution and transactional memory techniques. Determinator [13] and Revisions [26] avoid the complexity of speculative execution by straying from sequential consistency. These projects achieve acceptable overhead for some workloads, but constrain programs to a minimal set of deterministic synchronization primitives such as *fork/join* and *barrier*. Revisions, moreover, a C# library, does not directly support legacy code written in C-like languages.

Another deterministic scheduling system that avoids both the input sensitivity described above for quantum-based systems and the overhead of speculative execution and rollback is Tern [31], which *memoizes* execution schedules for reuse. This solution, like all deterministic schedulers, solves a related but distinct problem from

the one that both Determinator and DOMP solve: the former allow racy programs to run reproducibly, while the latter eliminate data races by enforcing a deterministic programming model.

Aside from deterministic scheduling, record and replay systems, such as Recap [79], Instant Replay [67], DejaVu [30], and ReVirt [35], as well as many others, at least enable the user to reproduce any bug on demand. These systems, however, generally impose too much overhead to work feasibly on deployment systems, or else require special hardware.

Deterministic parallel *functional* programming languages have long been available [56, 85], including dataflow languages [1, 39] and parallel Haskell [2, 28, 62, 96]. Following in the dataflow tradition is the recent Concurrency Collections (CnC) language [23], which specifies the parallel execution of code "kernels" (routines), which may, in turn, be written in any of a wide range of languages, including conventional imperative ones such as C++ or Java. All of these languages derive their determinism from the "single assignment rule": a variable is undefined until a thread defines it with a value, at which point it is also immutable. In this way, data races are impossible. Programmers have been slow to adopt the functional language paradigm, however, and CnC, which accommodates conventional programming for the (serial) "kernels", presupposes a division of labor between the "domain experts" who develop the "kernels" and the "tuning experts" who parallelize it using CnC's relatively unfamiliar notation and concepts.

Some recent languages support deterministic parallelism while allowing for a conventional imperative style. The SHIM language [36, 37, 94] implements a deterministic message passing model, avoiding the challenges of making shared memory deterministic, but also foregoing the programming convenience of the shared memory abstraction and requiring programmers to marshal data into explicit messages. De-

terministic Parallel Java [22] offers shared memory, but requires the programmer to adopt a new Java type system, and to "prove" statically via typing rules that parallel code is race-free. Array Building Blocks [44] promise deterministic parallelism, but their proprietary nature hampers detailed inspection.

DOMP has many architectural similarities to Dthreads [70]. Like Grace and DOMP, the Dthreads scheme supports programs in C-like languages, written using a standard parallel API. Like Determinator and DOMP, Dthreads do not require speculative execution and rollback, and therefore achieve good performance. Dthreads' efficiency, however, depends on their imposition of an arbitrary shared memory commit order—a form of deterministic scheduling—which allows racy programs to execute deterministically. By contrast, DOMP offers a way to allow conventional programs to conform, as much as possible, to a purely deterministic programming model, which treats data races as errors.

DOMP's approach has most in common with that of Determinator [13] and Revisions [26], operating in the workspace consistency model [11], while supporting a wider range of naturally deterministic synchronization abstractions to increase compatibility with legacy code.

2.2 The Reduction Construct

In addition to supporting OpenMP's core features, DOMP offers a generalized reduction construct. A reduction (or *right fold* in functional programming) is a higher-order function that applies a binary *combining operation* first to an initial value and the first element of a list, and then iteratively to the results of the previous iteration and the next element of the list, until every element has been consumed. For example, if the list contains integers and the combining operation is addition, a reduction over

the list will give the sum of the integers. In parallel programming, each "element" of the "list" is a single thread's instance of a shared variable; reduction aggregates the values of this variable across threads according to the combining operation.

The standard OpenMP reduction construct takes the form of a clause modifying the OpenMP directive that stands at the head of a structured block and specifies that the ensuing block be executed in parallel. Within the block, the variable (or variables) listed in the clause will be *updated* using conventional updating syntax as in sequential programming, as in the following example:

```
int x = 0;
#pragma omp parallel reduction(+:x)
  {
    x += 42;
  }   // x == 42 * num_threads
```

This reduction has the special semantic feature that it produces the same results even if the program disregards the OpenMP directives, e.g., if the programmer compiles without OpenMP support. (With GCC, this means compiling without the `-fopenmp` flag.) Unfortunately, OpenMP's reduction construct only supports a handful of arithmetic, bitwise, and logical operations, and only scalar, value types. This means that such simple operations as vector addition and matrix multiplication, as well as more complex and algorithm-specific ones, are unsupported. Furthermore, the OpenMP specification does not stipulate an evaluation order. In general, implementations such as the one in GOMP evaluate the reduction in a nondeterministic order, using low- or machine-level mutex locks.

Before and outside of OpenMP, reductions have a long history in programming language theory [59], and are a key concept in a theoretical understanding of parallel

programming in a functional programming language [91]. Thus DOMP's extended reduction is not a new concept, but its conformity to OpenMP reduction semantics, and its use in bridging the gap between conventional parallel programming and a purely deterministic programming model, are among its contributions.

A wide range of modern programming languages, both functional and imperative, support reductions in sequential programs in one way or another, including C#, C++, D, Haskell, JavaScript, Lisp, ML, Perl, PHP, and Python. The C++ `std::accumulate` library routine, for instance, allows the programmer to reduce over an iterable sequence, using an initial value and an arbitrary combining operation—or sum by default, if the programmer supplies no operation (see pp. 682 – 3 in [93]):

```
template <class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op) {
  while (first != last) init = op(*first++);
}
```

This amounts to mere syntactic sugar for iterating over the sequence and repeatedly applying the operation, since it implies no parallel execution and cannot be readily adapted to parallel programming.

The MapReduce algorithm's second, *reduce*, phase applies a reduction across large lists of data entries in the nodes of a distributed system[32]. First, the *map* phase, running in parallel on many nodes, takes the input and returns a set of key-value pairs. In the *shuffle* phase, the system redistributes the key-value pairs among the *reducer* nodes, so that each node has a subset, or list, of such key-value pairs to reduce. Like the mappers, the reducer nodes work independently and in parallel, although the MapReduce algorithm does not require parallelism within the reducer node itself. Moreover, the algorithm is not embedded within a larger

scheme of parallel execution as is the OpenMP reduction. The Phoenix project [82] ports MapReduce to the multicore platform, but, likewise, launches and terminates parallel execution for the special purpose of the MapReduce problem alone. Neither the original MapReduce nor Phoenix defines a deterministic order for evaluating the reduction.

CnC has a proposed new reduction construct [24], which, like the rest of CnC, is provably deterministic [23]. This reduction works with arbitrary combining operations but requires the same unfamiliar notation and programming paradigm as the rest of CnC.

Intel's Threading Building Blocks C++ library (TBB) [58] defines a `parallel_reduce` template function, bearing some similarities to C++ `std::accumulate`, but providing for parallel execution of the reduction. Like the latter and unlike OpenMP's reduction clause, `parallel_reduce` allows for arbitrary types and operations. Its data argument is an object of the `Range` template class, which is more general than the iterable object supplied to `std::accumulate`; having instead a `split` function, to split the data into two parts, recursively. Like Phoenix and unlike OpenMP's reduction clause, this reduction spawns and terminates its own parallel team of threads rather than working within a larger parallel block, although TBB does support nested parallelism. TBB's `parallel_reduce` recursively splits the data as far as possible and then applies the combining operation to pairs of data entries along a binary tree in the process of joining threads—a pattern similar to the workings of DOMP's extended reduction, as further explained in 5.1 and 6.4. However, in contrast to DOMP, TBB's order of splitting the data and thus the shape of the resulting evaluation tree are explicitly nondeterministic.

* * *

The design of DOMP should be seen within the broader context of various relevant areas of research. We turn to these next.

2.3 Programming Languages

The functional programming language research community has been interested in parallelism at least since the 1980s [81]. Parallelism and functional programming make a good fit because functional programming languages can express computations guaranteed to be devoid of side effects. (Languages such as Haskell and ML do provide special features to express side effects separately from pure functions, e.g., *monads* in Haskell and *references* in ML.) A program written in a *purely functional* language, devoid of side effects, or in the purely functional subset of a functional language, is deterministic by nature, since it does not allow any interactions among concurrent processes: the result of a parallel program will always be the same for a given input, and also the same as that of its sequential version [50]. Moreover, functional languages do not generally include an *assignment* operation, since such an operation is defined as exerting a side effect upon the value of the left-hand operand. For this reason, functional languages are also called "single-assignment" languages: the definitional expression `let x = 42` means that x will forever have the value 42 in the given scope and no other value. This "single-assignment rule" is another aspect of what makes functional languages deterministic when parallelized: the concept of *updating* a variable has no place here, so data races are impossible.

(The functional programming language ML has a parallel version, CML [83]. However, CML seems to have been developed specifically to make certain kinds of nondeterminism possible in interactive systems.)

Early research focused primarily on *implicit parallelism*, in which the compiler

could easily infer from the function what elements of it could be computed independently and therefore in parallel, as in Id [76], pH (a parallel dialect of Haskell) [2], and pHluid [40]. This focus continues with Manticore [41]. At the same time, Paul Hudak developed a system of *annotations* whereby a programmer could nudge the compiler to parallelize particular regions of code and in what configuration [56], which made better performance possible for a wider range of programs, while at the same time separating off parallelization from core algorithmic concerns [95].

Dataflow languages are functional programming languages developed specifically for dataflow architectures [99], which are machine architectures designed to exploit parallelism by circumventing the bottlenecks believed inherent in the von Neumann architecture [6, 61]. Moreover, their determinism is evident in that they follow the model of the Kahn process network in the transfer of data among (physical or virtual) processors [39]. Dataflow languages are compiled into *dataflow graphs*, which the system uses to control the flow of data. Languages such as Cajole [51], Lucid [98], and LUSTRE [47] are textually based and treat the data flow graph as an internal representation. But other dataflow languages have visual external representations for the benefit of clarity and ease in software engineering. These include the commercial products LabVIEW [57] and, more recently, Simulink[72], which is widely used today for the design of control systems. These languages can run on any modern platform, but are specialized in their purposes. As a whole, dataflow languages are clearly both deterministic and useful, but limited with respect to use cases.

A number of more conventionally imperative-style languages have emerged in recent years that offer deterministic parallelism. SHIM [36, 94, 37] is a deterministic parallel programming language with C-like syntax and in the imperative style, whose parallelism follows the message-passing paradigm. It was developed particularly for the design of embedded systems. Deterministic Parallel Java (DPJ) [22] uses shared-

memory parallelism, but requires the programmer to annotate data according to a type and effect system that determines which data are visible and otherwise accessible to which threads at various points in the program. Guava [14], presented as a "dialect of Java without data races," uses a comparable annotation system, and restricts programs so as to allow concurrent threads to have access to data only if such accesses are properly synchronized. Although this principle might at first appear to eliminate data races, it seems to us to leave open the possibility of the "higher-level data races" [5] mentioned in **??** with regard to transactional memory.

Like DPJ and Guava, Jade [84] involves the annotation of data to make access permissions explicit, but uses this information to parallelize serial code automatically, working with an original source in a conventional language such as C. Jade's promise is to preserve sequential semantics in parallel execution, something to which standard OpenMP aspires in principle, but without the latter's excluding nondeterminism. Another programming language based on C and providing for parallelism through annotations is Cilk [20, 69, 42], which also strives to provide the same semantics under sequential and parallel execution, but which, like standard OpenMP, does not actually prevent nondeterminism.

2.4 Deterministic Scheduling

The approach of imposing a deterministic schedule on a possibly nondeterministic and even racy or buggy program has interested many research groups. DMP [33] and CoreDet [17] divide program "time" into uniform units, or *quanta*, consisting of a fixed number of instructions, with each quantum divided between an initial parallel phase and a sequential phase. Through static analysis, the compiler detects and marks potentially conflicting accesses to shared memory. The runtime system shifts

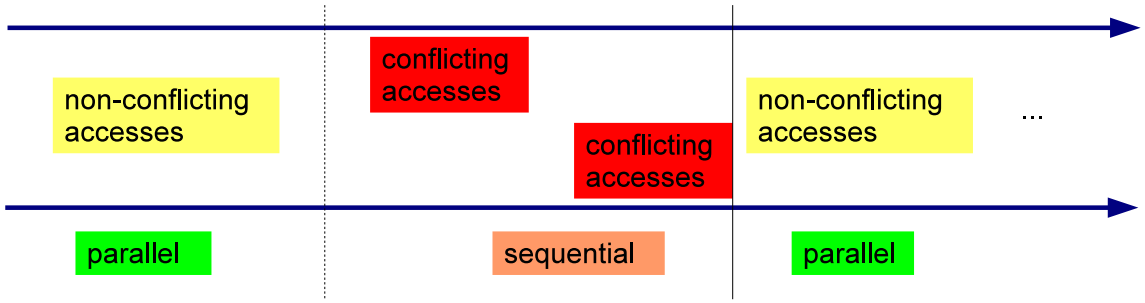


Figure 2.1: Execution quantum scheme in deterministic schedulers such as DMP and CoreDet.

from parallel to sequential execution as soon as it encounters the first such conflicting instruction within a quantum. Figure 2.1 illustrates this scheme. In the sequential portion, of course, the runtime imposes an arbitrary but consistent order on threads' accesses to shared memory, which ensures the deterministic outcome of the program.

Grace [18] does not follow the same quantizing model as DMP and CoreDet, and, indeed, it shares many elements with DOMP: it implements threads as separate processes, each working within its own address space; it tracks writes (and also reads) using access protection and a signal handler; it eliminates mutex locks, converting those that occur in source code into no-ops. Grace threads write their updates to shared memory to local copies, as in DOMP, and the runtime *commits* those updates as transactions at synchronization points, which are any points where threads fork or join. A major difference from DOMP, however, is that Grace threads update shared memory *speculatively*: at synchronization points, they attempt to commit, but if a thread's attempt has the wrong version number, kept in a global data structure, execution aborts and restarts the commit sequence. The resulting sequence of changes to shared state is always the same, and therefore deterministic, but aborted and restarted transactions could waste resources. Dthreads [70] combines ideas such as threads-as-processes for thread isolation and protection and trapping to track

writes from Grace with the quanta divided into parallel and sequential phases and a deterministic "write token" that orders writes to shared memory in the sequential phase from DMP and CoreDet. Dthreads also achieves remarkable efficiency by means of tricks such as minimizing writes by "diffing" them against the original state, a technique borrowed from TreadMarks [4] and Munin [16]. It also minimizes false sharing, resulting in performances sometimes better than those of the reference `pthreads` implementation. And unlike both Grace and DOMP, Dthreads supports parallel programs that do not conform to the simple fork/join model.

Kendo, like DMP, CoreDet, and Grace, uses performance counters to compute a deterministic schedule; but unlike these solutions, it concentrates exclusively on the deterministic ordering of lock acquisitions. It therefore does not have to quantize execution and is both simpler and more efficient than DMP or CoreDet. However, a program that has a data race—arising from a failure to use mutual exclusion locks where necessary—will remain racy and nondeterministic under Kendo. Thus Kendo can only guarantee that a race-free program will always execute with the same schedule, a guarantee of some usefulness, especially in Byzantine fault tolerance and other systems that require exact replication of computations.

Revisions [26] follows a model very close to that of DOMP: threads are processes, each one working in its own, isolated copy of shared state. However, Revisions resolves memory access conflicts according to the *isolation type* of each conflicting object, rather than, as in DOMP, by signaling a race condition error. This approach can only work in a strongly typed language that can record additional type information—and, of course, it does not assume a deterministic programming model, but rather the deterministic execution of a strongly typed programming model.

2.5 Record and Replay Systems

An approach to deterministic parallelism entirely different from programming languages or deterministic schedulers is to have the system monitor the execution of a parallel program, log every event important to parallel execution, and then make it possible to replay the execution, using the log, with the exact same interleaving, resulting in the same output and behavior. This approach is called record and replay, and it can be of great use in debugging parallel code that may contain heisenbugs [3, 64, 92], as well as in intrusion detection [35]. Record and replay systems present a difficult trade-off between cost and performance. Software record and replay systems, such as DejaVu [30], ReVirt [35], and Doubleplay [97] are inexpensive, but tend to slow application performance down to the point where it would not be feasible to run them continually on high-use deployment systems such as Web servers. By contrast, systems including special hardware, such as Karma [15] and FDR [102] are very efficient but expensive. Research continues in this area to develop a record and replay system that is at once fast and affordable.

2.6 Debugging Tools

A number of debugging tools and systems are available to help programmers to cope with nondeterminism in their parallel programs. Cilk has its own special debugging tool to help catch data races [69]. For more general purposes, RacerX [38] can find concurrency bugs, both data races and deadlocks, through static analysis. The Valgrind project's concurrency bug detector Helgrind also works through static analysis—in particular, by building a graph of "happens-before" relations [75]. In our own, admittedly slight experience with Helgrind, however, we found it to produce so many false positive results as to be of limited usefulness. Eraser [87] detects

concurrency bugs dynamically, by monitoring all shared memory references, making it somewhat resemble record and replay systems.

2.7 Transactional Memory

As mentioned in 2.1, transactional memory ensures atomicity of updates to shared memory but does not, in and of itself, prevent data races in the program. However, transactional memory shares some techniques with DOMP, and provides a partial model for addressing some of DOMP's central issues: thread isolation and the minimization of synchronization overhead. Herlihy and Moss's hardware transactional memory [55], for example, uses a slight modification of the standard "snoopy" cache coherence protocol to enable the cache to detect transaction conflicts. In so doing, it minimizes synchronization overhead, essentially to setting or clearing bits in the cache directory. This design takes care to allow normal memory accesses to proceed without interference, as if isolated from the transactions. Moreover, both this system and Shavit and Touitou's software transactional memory (STM) [90], whose design is based on the former, avoid locks and blocking entirely. These two and other STM systems generally assume that multiple *processes* will be sharing the same transactional memory but not the same address space, i.e., that they work essentially in isolation except when interacting through a transaction [34, 53, 52]. This model, in which the memory serves as a lock-free synchronization manager, resembles the role of the runtime library in DOMP. However, DOMP does not resolve or retry conflicting memory accesses, but rather signals an error in the program in such cases.

* * *

Among deterministic parallel programming solutions that support C-style languages, Determinator and DOMP are unusual in requiring that the program conform

to a deterministic programming model that does not allow data races and that precludes the use of such low-level synchronization primitives as muteness and condition variables, which assume an underlying nondeterministic model. This approach raises a question: how important or necessary are such nondeterministic language elements to real-world programming? In order to approximate an answer to this question, we pursued the analysis that follows in Chapter 3.

Chapter 3

Analysis of Synchronization

How crucial are nondeterministic synchronization abstractions to the logic of parallel programs? To find out, we manually counted invocations of synchronization abstractions in the programs of three parallel benchmark suites—SPLASH [100], NPB-OMP [60], and PARSEC [19], choosing OpenMP versions of programs whenever possible and pthreads versions otherwise. This analysis extended and refined earlier work by [9].

Our working hypothesis that the practices and patterns in these benchmarks reflect those found in the larger software world remains, admittedly, unproven. Ideally, we would have analyzed larger, real-world OpenMP programs, but hand analysis of these was impractical.

We counted some matching pairs of events as single instances:

- A *fork* and its corresponding *join*
- A lock acquisition and its corresponding release
- A condition variable "wait" and "broadcast" statements, along with associated lock acquisition

In this way, we attempted to overcome some of the heterogeneity involved in comparing OpenMP with pthreads programs.

We counted the locations where code invoked the naturally deterministic abstractions *fork/join* and *barrier*, and recorded each count directly, along with *work sharing* and *reduction* constructs in the case of OpenMP code. Invocations of nondeterministic abstractions—locks, condition variables, and OpenMP's *atomic*, *critical*, and *flush*—we grouped by the *idioms* in which they were used, for which we identified five classes:

- *Work sharing*: pthreads only, since OpenMP has work sharing constructs
- *Reduction*: pthreads, and OpenMP with unsupported types or operations
- *Pipeline*: pthreads and OpenMP
- *Task Queue*: pthreads and OpenMP—assigning tasks to threads as the former appear and the latter become available
- *Legacy*: pthreads, in SPLASH only; these are utilities to make I/O and heap allocation thread safe, from a time before thread safety became standard in the C library, and are now obsolete
- *Nondeterministic* idioms: pthreads and OpenMP, for any genuinely nondeterministic algorithm, such as simulated user interactions, user-level scheduling, or load balancing.

Figures 3.2 and 3.1 summarize our findings. "Work-sharing" idioms occur in pthreads programs, which lack OpenMP's higher-level constructs; e.g., they lock a global integer and save its value as a thread ID for later task assignment, before incrementing and unlocking it. Likewise, pthreads code has to use locks to update variables to

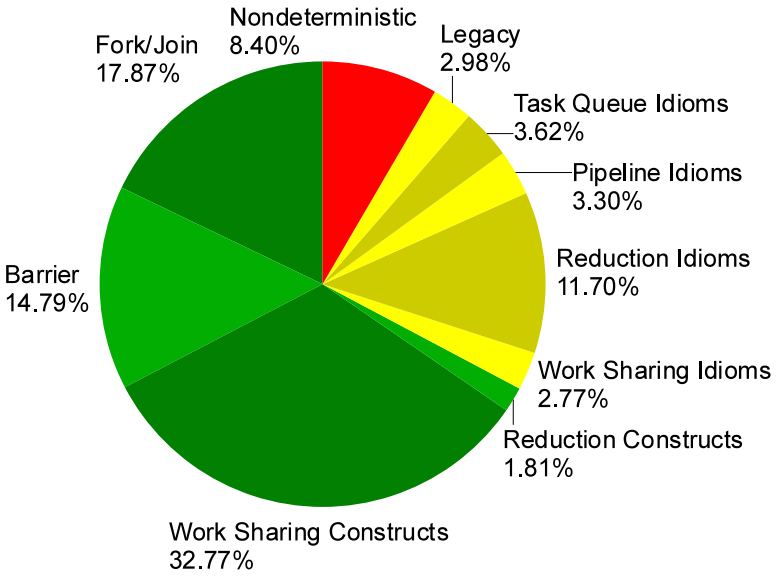


Figure 3.1: Types and uses of synchronization abstractions in SPLASH, NPB-OMP, and PARSEC programs. Items in green are naturally deterministic constructs. Items in yellow are uses of nondeterministic primitives to achieve deterministic higher-level goals.

achieve the equivalent of reductions. As Figure 3.3 shows, in the majority (74%) of cases of the use of these naturally nondeterministic synchronization constructs, the programmer was building a higher-level *deterministic* idiom.

Furthermore, we find that the majority of synchronization idioms used in all benchmarks taken together could be expressed using the deterministic set of OpenMP constructs, the same set to which DOMP restricts itself, as shown in Table 3.1. Of course, the standard OpenMP implementation of the deterministic set of its constructs is not, itself, deterministic. This is where DOMP comes in: DOMP provides a guarantee of deterministic semantics for this subset.

Note also that we include in Table 3.1 task queues because OpenMP has a `task` construct (currently available only in Fortran), which, *in principle*, could be implemented in a deterministic manner. In fact, none of the analyzed benchmarks using task queues implement them by means of the OpenMP `task` construct. Rather,

	OpenMP	Deterministic Constructs				Deterministic Idioms					Nondeterministic
		fork/join	barrier	work sharing	reduction	work sharing	reduction	pipeline	task queue	legacy	
barnes		1	6	-	-	2	1	-	-	1	2
fmm		2	13	-	-	1	1	3	-	15	8
ocean		1	40	-	-	1	3	-	-	-	-
radiosity		3	5	-	-	2	5	-	7	-	23
raytrace		1	1	-	-	5	-	-	-	6	2
volrend		5	15	-	-	5	-	-	-	1	6
water-nsquared		1	9	-	-	1	7	-	-	-	-
water-spatial		1	9	-	-	1	4	-	-	1	2
cholesky		1	4	-	-	1	-	-	2	4	-
fft		1	7	-	-	1	-	-	-	-	-
lu		2	10	-	-	2	-	-	-	-	-
radix		1	7	-	-	1	-	2	-	-	-
BT	√	12	-	37	-	-	2	-	-	-	-
CG	√	7	8	20	6	-	-	-	-	-	-
DC	√	1	-	-	-	-	1	-	-	-	-
EP	√	3	-	1	1	-	1	-	-	-	-
FT	√	8	-	8	1	-	-	-	-	-	-
IS	√	7	1	11	-	-	1	-	-	-	-
LU	√	12	4	71	3	-	2	5	-	-	-
MG	√	11	-	16	2	-	-	-	-	-	-
SP	√	13	-	38	-	-	2	-	-	-	-
UA	√	60	-	78	4	-	80	-	-	-	-
blackscholes	√	2	-	2	-	-	-	-	-	-	-
bodytrack	√	5	-	5	-	-	-	-	-	-	-
facesim		2	-	-	-	-	-	-	14	-	-
ferret		1	-	-	-	2	-	-	9	-	-
fluidanimate		13	14	-	-	-	-	-	-	-	15
freqmine	√	7	-	21	-	-	7	-	-	-	-
raytrace		1	3	-	-	-	-	-	2	-	-
swaptions		3	-	-	-	-	-	-	-	-	-
vips		1	-	-	-	-	-	-	-	-	
x264		2	-	-	-	-	-	-	-	-	6
canneal		1	1	-	-	1	-	-	-	-	-
dedup		5	-	-	-	-	-	17	-	-	-
streamcluster		5	34	-	-	-	-	4	-	-	-

Figure 3.2: Invocations of synchronization abstractions in source code of SPLASH, NPB, and PARSEC benchmark programs, broken down into naturally *deterministic constructs*, uses of nondeterministic primitives to build *deterministic idioms*, and genuinely nondeterministic idioms. No OpenMP benchmarks studied had any nondeterministic idioms.

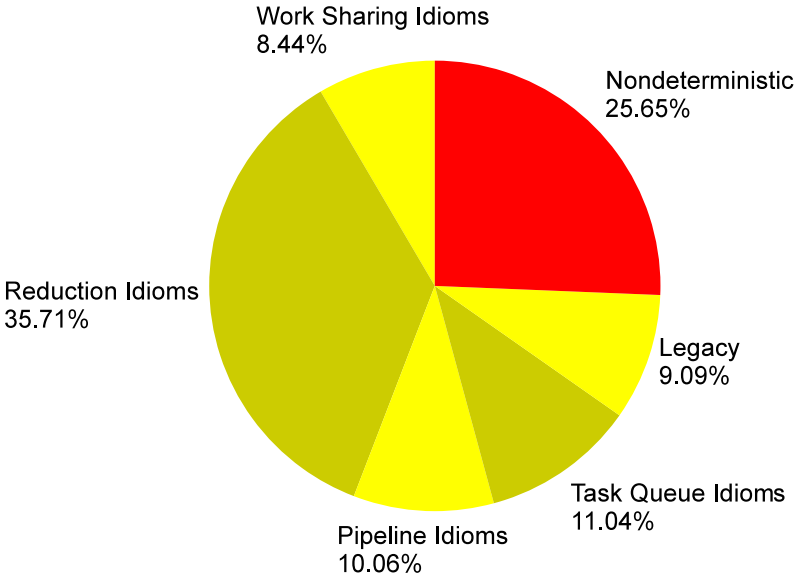


Figure 3.3: Naturally nondeterministic synchronization constructs, classified by idiom in which they are used.

In OpenMP?	Idiom	% Sync
Deterministic construct already in OpenMP	fork/join	17.87
	barrier	14.79
	work sharing	35.54
	simple reduction	1.81
	task queue	3.62
	Subtotal	**73.63**
Extensions to OpenMP	extended reduction	11.70
	pipeline	3.30
TOTAL		**88.62**

Table 3.1: The majority of idioms using synchronization in the analyzed benchmarks could be expressed with deterministic OpenMP constructs.

they all manage their task queues at the user level by assigning tasks to threads nondeterministically as the latter become available. Alternatively, the task queue idiom could use a task abstraction such as OpenMP's current one, but that would be implemented deterministically. For instance, the runtime can start a new thread for each task, leaving it to the OS's scheduler to assign threads to processors as usual.

OpenMP's *reduction* construct only supports built-in scalar types and simple operations, such as summing over an integer. This leaves many common, even simple, use cases unsupported. For example, the NPB benchmarks BT, EP, LU, and SP use a vector (array) to hold the sum of the threads' local vectors, with code such as this (from BT):

```
      do m = 1, 5
!$omp atomic
            rms(m) = rms(m) + rms_local(m)
      enddo
```

Likewise, DC finds the maximum over one object field and sums over several others dependent on this maximum. PARSEC freqmine finds a maximum over a field, freeing the non-maximal objects as a side effect.

DOMP's extended reduction construct expresses such idioms succinctly, while its deterministic implementation guarantees a fixed evaluation order, dispensing with *atomic* and other such nondeterministic constructs. Moreover, all non-OpenMP benchmarks that use reduction idioms would require an extended reduction if rewritten using OpenMP or DOMP, generally because they reduce on pointer types.

Figure 3.4 focuses in on the subset of analyzed benchmarks written in OpenMP, which includes all those in the NPB suite, as well as three in PARSEC. This chart highlights the relative importance of extended reductions to the project of a deterministic OpenMP.

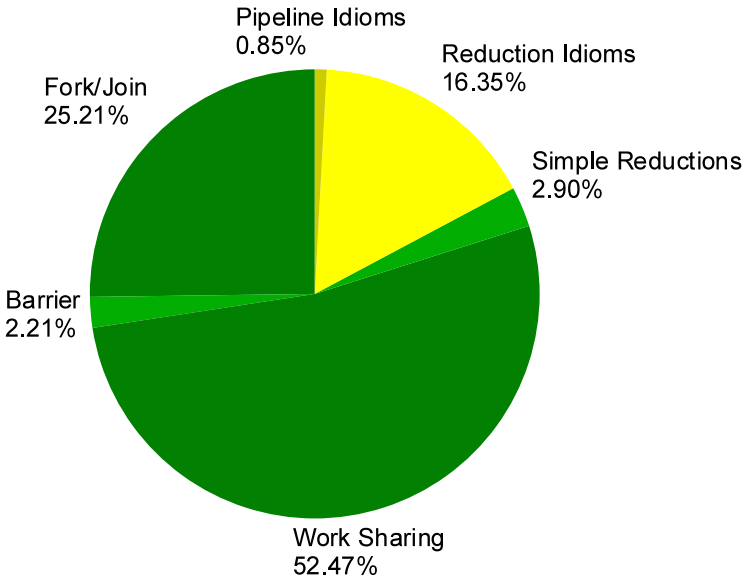


Figure 3.4: Types and uses of synchronization abstractions in those benchmarks that use OpenMP.

Both pthreads and OpenMP lack a high-level construct to represent a pipeline, having instead to resort to ad-hoc synchronization, for which OpenMP code uses the nondeterministic `flush` construct (NPB **LU**). Deterministic pipeline and task object constructs remain attractive DOMP extensions for future work.

The few remaining uses of synchronization abstractions, 8.7% in aggregate, may be irreducibly nondeterministic, and comprise a miscellany of idioms, from simulated user-program interactions (vips) to simulated particle interactions (fmm, fluidanimate) to user-level scheduling not formally implemented as a work queue (radiosity, volrend). Whether one might achieve the same goals using deterministic idioms is beyond our present scope.

The need for extended reductions and deterministic pipeline constructs account together for *all* instances of nondeterministic OpenMP constructs in the OpenMP benchmarks we studied. In NPB, they contribute 16.5% and 1% of total synchronization instances, respectively, thus again highlighting the potential usefulness of

extended reductions for a deterministic OpenMP.

Our analysis suggests, first, that a deterministic OpenMP, with an API consisting of the deterministic subset of OpenMP's constructs and an implementation that guarantees deterministic semantics, is a reasonable goal. It further suggests that generalizing OpenMP's reduction construct could substantially increase the portion of parallel applications that DOMP would support with its strictly deterministic abstractions. We describe this extension in Chapter 5, after a review of DOMP's overall design.

Chapter 4

Design and Semantics

DOMP builds on OpenMP [78] to offer a parallel programming model with both an expressive API and race-free, naturally deterministic semantics. DOMP retains most OpenMP core constructs, but excludes OpenMP's few nondeterministic ones. DOMP further extends OpenMP's API with a deterministic generalized reduction, which can replace the most common uses of its excluded nondeterministic constructs.

By "naturally deterministic," we mean such that program logic alone determines at what point in each thread's execution sequence it synchronizes with another thread—immune to the effects of the scheduler and hardware timing. The program defines a function from the input to the output and behavior. In order to support this level of determinism, DOMP follows the Working-Copies Determinism (WCD) programming model [11]. Since WCD is the theoretical foundation of DOMP, we here discuss WCD and its basis in the Workspace Consistency memory consistency model. Next we consider the rationale for implementing a deterministic version of the OpenMP API with Workspace Consistency semantics in the form of a user library. Finally, we review the features of DOMP, pointing out how they manifest WCD semantics.

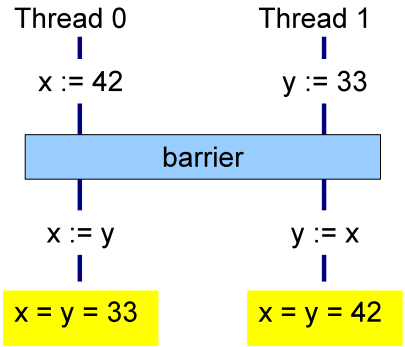
4.1 Working-Copies Determinism

DOMP enforces determinism according to the Working-Copies Determinism (WCD) programming model [11]. This model guarantees, not only that execution will always produce the same output and behavior on the same input, but that a program containing a data race will fail to execute to completion. WCD provides this guarantee by limiting communications between threads to the parent-child relation and to program-specified synchronization points such as *fork*, *join*, and *barrier*. Synchronization is then thoroughly deterministic. This arrangement follows the Workspace Consistency memory and programming model (WC) [11].

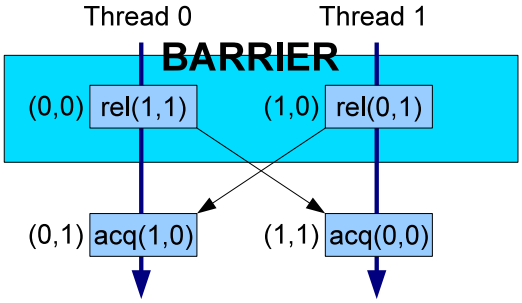
The essence of Workspace Consistency can be gleaned from a consideration of the "swap assignment" operation available in some languages, including Perl and JavaScript:

```
(x, y) := (y, x)
```

This construct implies no parallelism, but its semantics require, crucially, that both x and y on the right side be evaluated before either x or y on the left side receives its new value. Under Workspace Consistency, we get exactly the same results from a "parallel swap," as illustrated in Figure 4.1a. The runtime under WC ensures that



(a) Semantics



(b) Memory consistency model

Figure 4.1: The "parallel swap" construct under Workspace Consistency

x sees the old value of y and vice versa. The crucial point is that x "hands its value off," in effect, to y, and vice versa.

To explain this model both more generally and in greater detail, we use the traditional terminology for describing memory consistency models [43, 73]. Memory accesses in parallel programs fall into the two categories: *shared* and *private*, the former of which are the only ones requiring special treatment. Shared accesses, in turn, are *competing* if more than one thread accesses the same location and at least one such access is a write; otherwise, they are *non-competing*. Again, competing accesses are the ones of special concern, because only they could result in data races. Then, some competing accesses are *synchronizing*, while others are *non-synchronizing*. Synchronizing accesses ensure the safety of other competing accesses—-generally, by using some location in memory as a means to *pass a message* from one thread to another, as, for instance, when one thread sets a flag to signal that other threads may read the (data) value at location x.

Finally, we may classify a synchronizing access as either an *acquire* or a *release*. A thread performs an acquire in order to gain (non-synchronizing) access to some other location in memory, typically storing data; in a release, a thread signals that some other location in memory is available for access by one or more other threads. An acquire always involves a read. In a spinlock, for instance, the thread seeking access checks a flag repeatedly in a loop until the flag's value changes (say, from set to clear). The change in value signals the waiting thread that it has permission to access the desired memory location (holding data); this read is the acquire per se. As soon as the thread receives this signal, it writes to the flag again so as to signal other threads that they may not have access to the data and must wait. This write is considered non-synchronizing, since it does not signal that a memory location is available. When the current thread finishes, it must write again, clearing the flag so

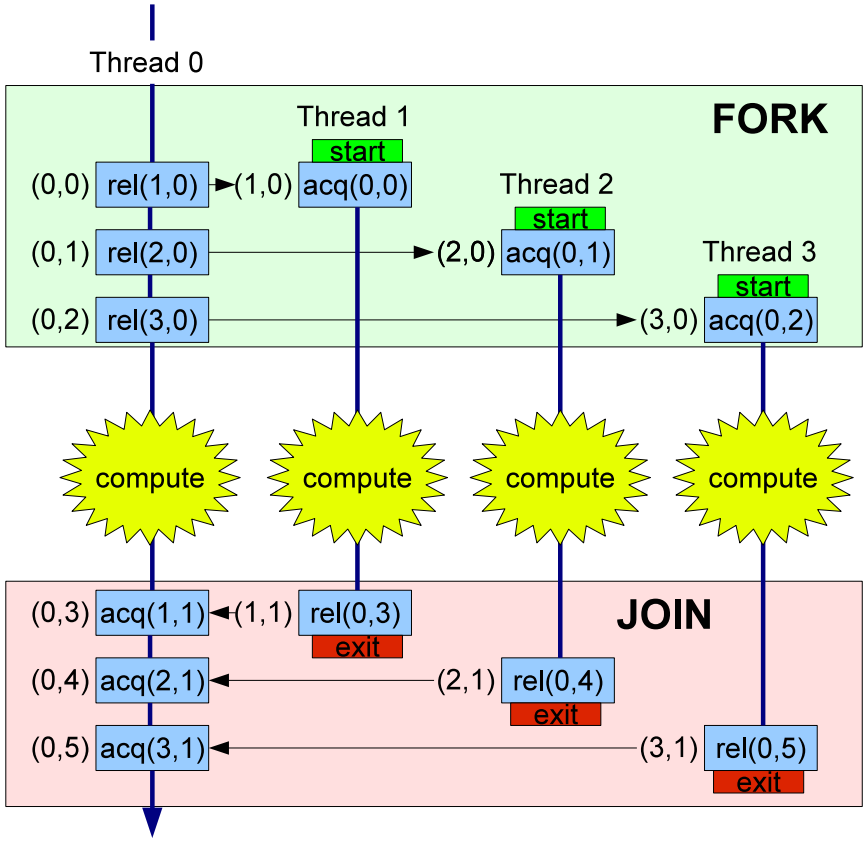


Figure 4.2: Pairing of releases and acquires following the Workspace Consistency model. The pair (*thread*, *event_number*) appears to the left of each synchronization event (blue rectangle) and identifies it uniquely. Each event has as its argument the identifier of the partnered event in another thread.

that some other thread may obtain the lock; this second write is a release. A release always involves a write. In an even simpler, non-exclusive scenario, a team of threads spins on a synchronization variable until one thread broadcasts to all of them that the data in some location are available, by setting a flag. The issuing thread's write to the flag is the release; each thread's read when the flag is set is its acquire.

Under Workspace Consistency, program logic pairs each release to a specific acquire, as illustrated in Figure 4.1b. In addition, one thread's writes do not become visible to any other thread until the next synchronization event, the release-acquire pair in which the writing thread releases the data. Synchronization accesses are

sequentially consistent (or, more precisely, processor consistent), meaning that the order of releases and acquires follows program logic and that all threads observe the same order. Finally, if two threads perform conflicting (non-synchronizing) writes, the implementation handles this condition deterministically, e.g., by always signaling it as a data race error.

Figure 4.2 illustrates Workspace Consistency in the fork-join synchronization construct with a team of four threads. The master thread (Thread 0) signals to each other thread in turn that all shared data are accessible at the same time that it creates those threads by calling `fork`. At the join, each thread in turn signals to the master that its version of shared data is ready for the master to read. In practice, for efficiency, we implemented the join in a binary tree pattern to increase parallelism (see 6.2). Figure 4.3 shows the corresponding synchronization pattern for a barrier, where the master's releases are not combined with the `fork` call.

Because synchronization in Workspace Consistency controls the visibility of data updates, this model constrains the flow of data to follow a strictly deterministic pattern. Returning to our example of the "parallel swap" (Figure 4.1b), before the barrier, Thread 0 sees the previous, not updated, value of y (say, 0), and Thread 1 does the same for x. After the barrier, x has the value 42 and y has the value 33 for *both* threads. Then, when each makes its respective assignment, the swap is complete.

Workspace Consistency gives us, in effect, both the convenience of shared-memory parallelism and the determinism of a message-passing system that takes the form of a Kahn process network [63]. The key feature of a Kahn network that makes it deterministic is that, at any given time, no node (processor, thread) is allowed to receive data from more than one communication channel. Whereas a system with shared channels (Figure 4.4) requires mutex locks to prevent data races and allows

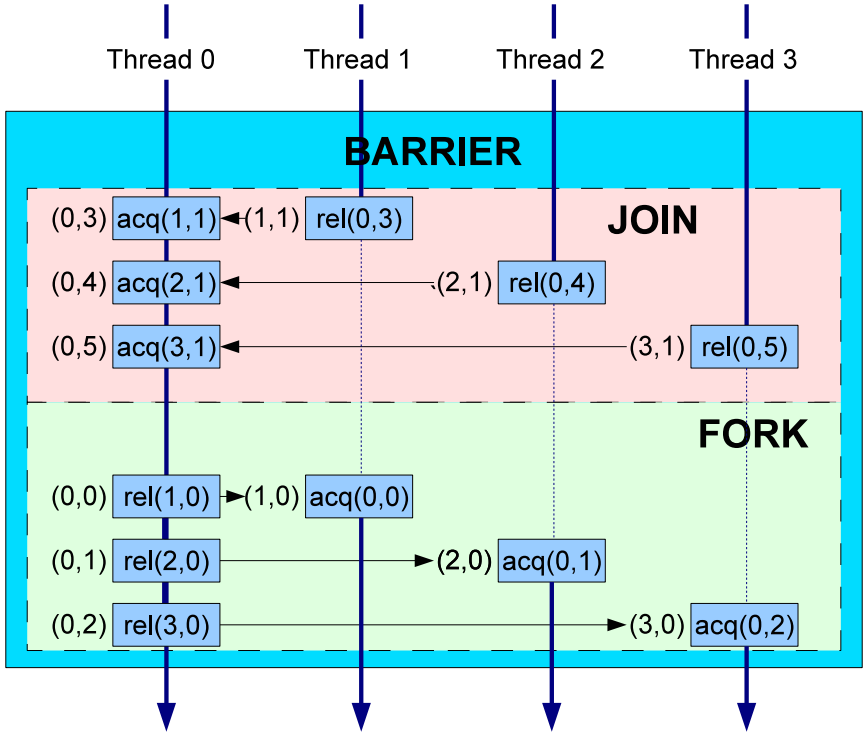


Figure 4.3: Pairing of releases and acquires in a barrier.

data to flow nondeterministically, a Kahn process network (Figure 4.5) constrains communication in such a way that the order in which data flows over time is always the same from one run to the next. Figure 4.5 shows nodes sending on only one channel at a time, but this is not strictly necessary. In fact, a single-producer, multiple-consumer data propagation pattern is compatible both with Kahn networks and with Workspace Determinism [11].

Working Copies Determinism then isolates data for each thread between synchronization events, in order to comply with Workspace Consistency's data visibility requirements. Every concurrent thread receives its own, private logical copy of shared state at the fork, and the restriction on communication prevents read-write conflicts. At the fork, the DOMP runtime also creates an additional *reference copy* of shared state, to remain untouched until a barrier or the join. At the barrier or join, the

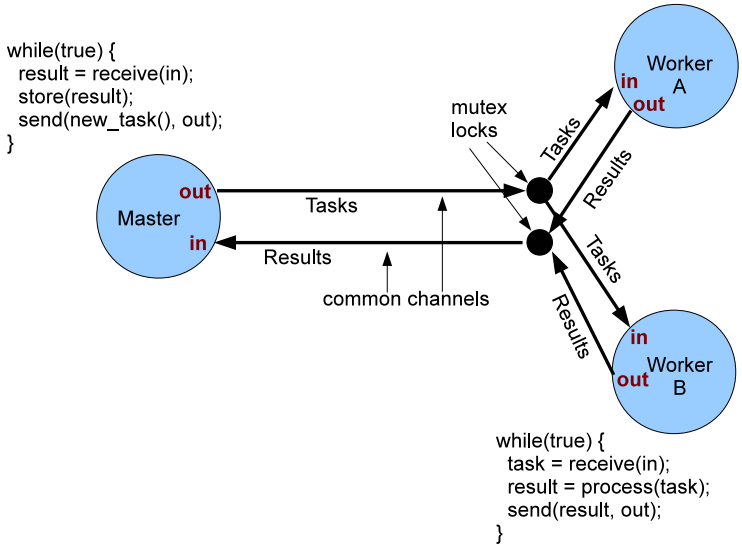


Figure 4.4: Nondeterministic network

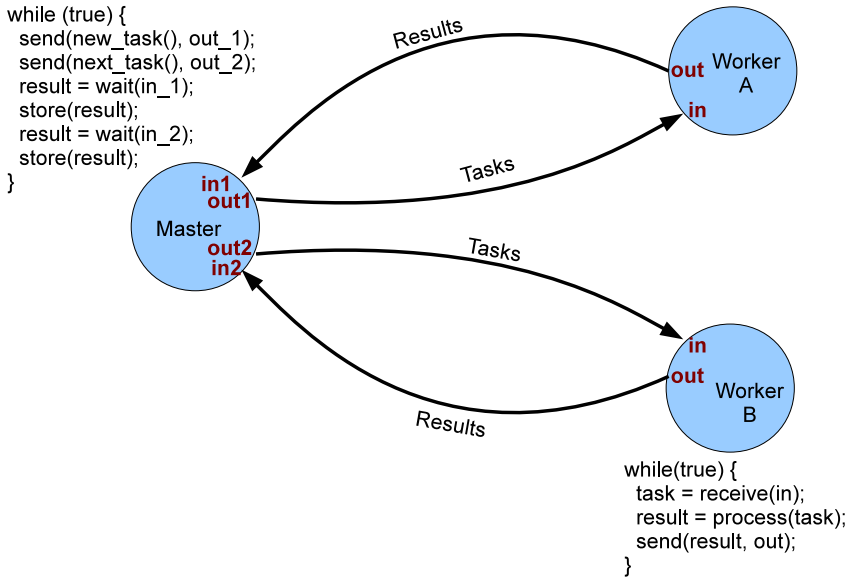


Figure 4.5: Kahn process network

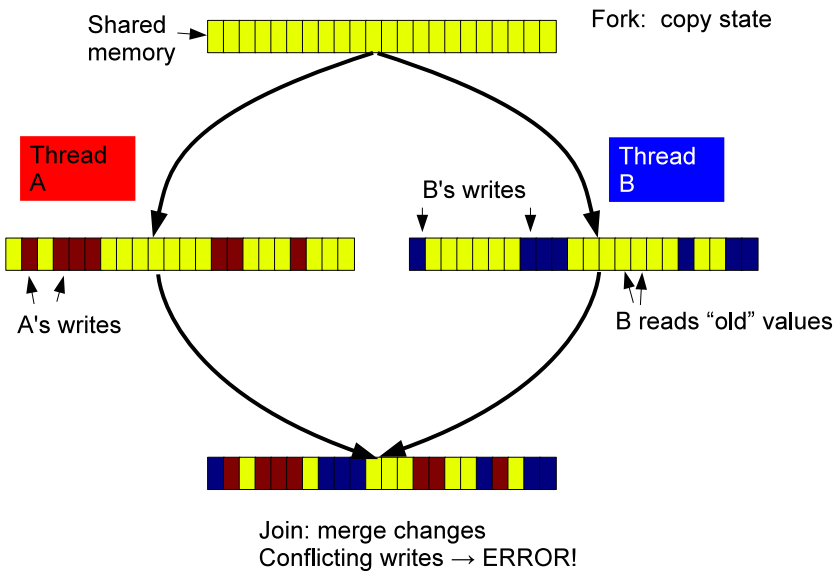


Figure 4.6: Working-copies determinism with 2 threads

parent thread compares and merges its own and its children's versions of shared state with the reference copy, signaling any conflicting write as an error. A barrier is effectively a join followed by a new fork, with the same number of threads resuming execution immediately after the barrier.

Figure 4.6 illustrates this sequence of events and its consequences on the visibility of data to threads.

DOMP, then, implements Working Copies Determinism while supporting most of the OpenMP API. In order to detect changes between synchronization points and to check for data races while merging updated versions of shared state, DOMP creates, not only a logical copy for each concurrent thread, but an extra *reference copy*. Figure 4.7 illustrates the role of these data versions in forking and merging.

DOMP's treatment of race conditions as errors is not the only possibility within a deterministic system. Alternatively, one may choose a scheme for resolving write conflicts in some known order, perhaps requiring programmer annotations, as in

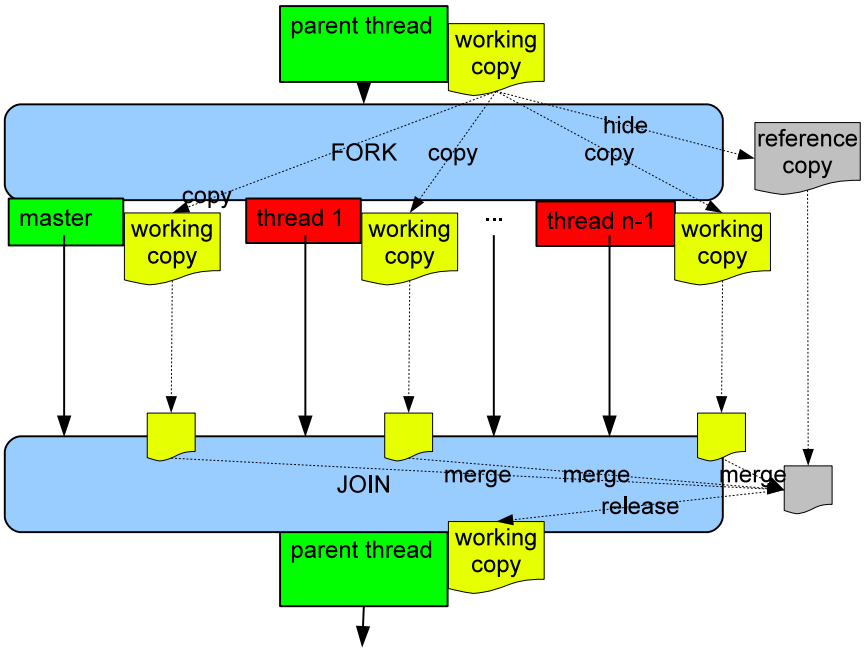


Figure 4.7: Sequence of events in a DOMP team of threads

Revisions [26]. In the WCD model, however, a write-write conflict is likely a sign of an error in program logic, which should not be silently resolved.

Because DOMP follows WCD, a programmer can retrofit legacy code, whether a sequential program to be parallelized or a standard OpenMP application, and uncover hidden data races that may be "benign" on test inputs but lead to incorrect results or heisenbugs when the code is deployed and encounters different inputs, or when it undergoes further development.

The WCD mechanism as described presupposes a particular *granularity* of comparison and merging. At the join, the runtime may compare and merge the various threads' copies of shared state by byte, by word, etc. Any choice of granularity will risks some false positives as well as false negatives. If we choose byte granularity, for instance, a program with a shared bitfield can raise a false positive. Meanwhile, suppose two threads share a 4-byte integer at location x, where the reference copy has 0, thread A has 1, and thread B has 16. Ignoring this race, the runtime will merge

bytes to produce 17 silently at location x. Ideally, a WCD system would be able to apply the granularity appropriate to each shared variable's type. In our prototype, we abstract granularity to make it easy to change, and use the byte as the default.

WCD's orientation toward hierarchical, fork-join parallelism makes for a convenient fit with OpenMP's general design. With its exclusion of the few nondeterministic features defined in OpenMP and its inclusion of generalized reductions, DOMP takes advantage of this design compatibility while affording the programmer a thoroughgoing deterministic parallel programming framework.

4.2 Accessible WCD

Workspace Consistency and Working-Copies Determinism serve as the foundation for the Determinator operating system [13], which has shown acceptable performance on a number of parallel programming benchmarks. One limitation of Determinator is in the narrowness of the API it can support—essentially, `fork`, `join`, and `barrier` only. We could have chosen to build a deterministic version of OpenMP, then, as a library for Determinator applications, to broaden its programming options. Such a choice would have presented us with considerable advantages, namely, that the DOMP support library would not have to manage the WCD runtime, including thread forking and joining, merging data while checking for data races, etc. (see 6.2). Since Determinator already implements copy-on-write at the OS level, we could have avoided the trapping, bookkeeping, and other complexities associated with user-level copy-on-write.

Implementing DOMP for Determinator remains, in our view, an important and realistic goal. However, the primary focus of the current project is to make WCD more *accessible* to programmers using familiar tools, environments, and platforms.

For this reason, we chose to implement DOMP first as a library for ordinary Linux systems, specifically as a modified version of GCC's OpenMP support library, libgomp. In so doing, naturally, we also hope to have solved some of the core design problems sure to be encountered in the development of a Determinator-based library, such as the proper semantics for DOMP's constructs and a workable way of integrating both deterministic simple and extended reductions into the merging process (see 5.1 and 6.2).

4.3 API

Building on the foundation of Working Copies Determinism, DOMP then implements most of the standard OpenMP API, including those constructs compatible with deterministic execution and excluding those that are not, while extending OpenMP with a generalized reduction construct. We here review these features.

4.3.1 Retained OpenMP Constructs

DOMP retains OpenMP's *parallel*, work-sharing (*loop*, *sections*, and *barrier*), and combined *parallel* work-sharing directives. In both OpenMP and DOMP, the *parallel* directive and its combined work-sharing variants represent a fork-join pair enclosing a structured block, creating and then joining a *team* of concurrent threads. Under DOMP, however, between any two synchronization points, no two concurrent threads may write a new value to the same shared variable (whether directly or through a pointer); the execution runtime treats such a data race as an error. Moreover, each thread's writes to shared variables remain invisible to all concurrent threads until the next synchronization point—such as a *barrier* or the closing *join*. These rules guarantee the controlled flow of data from thread to thread, as in a Kahn process

network, which is provably deterministic [63].

The *master* directive is naturally deterministic, since it appoints a single thread, the "master" (parent of the other team threads), to execute the code, and since OpenMP's implied *barrier* at the end controls data transfer to the team. Since *single* allows the scheduler to appoint an arbitrary thread to execute the block, which may differ from run to run, DOMP imposes deterministic semantics on *single* by making it synonymous with *master*. Moreover, if the programmer disables the implicit closing *barrier* with a *nowait* clause, the master's changes remain invisible to the team until the next explicit synchronization point.

4.3.2 Excluded OpenMP Constructs

DOMP's semantics excludes OpenMP's *atomic*, *critical*, and *flush* constructs as naturally nondeterministic, since they imply that concurrent threads can have conflicting memory accesses. As we have just seen in Chapter 3, programmers often use these nondeterministic constructs as low-level components of higher-level, deterministic idioms for which the parallel environment lacks suitable abstractions.

4.3.3 Extending OpenMP

In order to make it possible to express as many parallel programs as possible in a way compatible with strict determinism, DOMP offers a generalized reduction library function.

Standard OpenMP already has a reduction construct in the form of a *clause* modifying the directive opening a parallel block, but the reduction clause only supports built-in scalar types and simple arithmetic, bitwise, and logical operations. DOMP offers an extended reduction that accepts arbitrary types passed by reference and

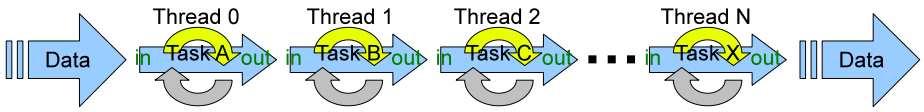


Figure 4.8: A simple or classic "conveyor belt" pipeline.

arbitrary, user-defined combining operations.

Since introduction of this feature accounts for the bulk of instances where we would wish to replace nondeterministic with deterministic code, we devote Section 5 to it.

The DOMP project would further benefit from two more extensions in the future: a distinct *pipeline* construct and a *task object* to facilitate deterministic work queues. Together, these extensions and the current core features would allow us to recast all OpenMP benchmarks we analyzed so as to use only deterministic constructs and to run deterministically under DOMP.

A *pipeline* is a sequence of repeated tasks, each dependent on the completion of a cycle of the task before it. With each task assigned to a different thread, data pass from thread to thread deterministically as each thread waits for input, processes it, and passes the output on, repeatedly until the pipeline is empty. Each sequence of task cycles performed on a given data item may be viewed as a single sequential operation merely divided into segments and rotated for processing from one thread to the next, and is therefore as naturally deterministic as a sequential program. Figure 4.8 shows a simple, "conveyor belt"-style pipeline, which we may regard as typical or classic. However, pipelines can have more complex structures and still retain it determinism for the same reason, as illustrated with a slightly more complex pipeline in Figure 4.9. For far richer examples of complex pipelines, see [37].

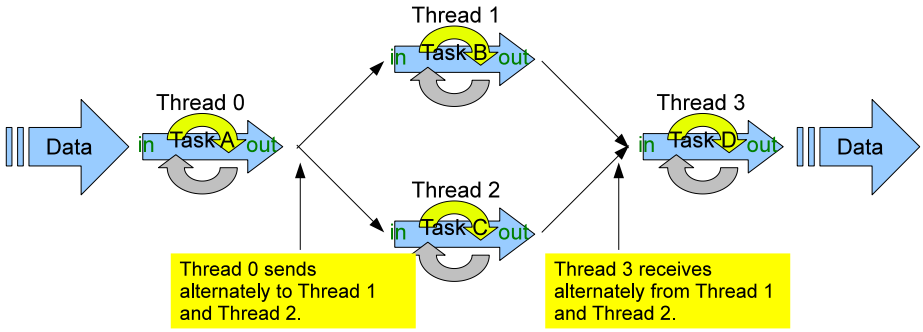


Figure 4.9: A slightly more complex pipeline. The controlled alternation of output from Thread 0 to Threads 2 and 3 and of inputs from those to Thread 4 maintains determinism.

The design of a pipeline construct could take one of a number of possible forms. OpenMP already has an `ordered` construct, which, when embedded in a parallel loop, causes the `ordered` block to be executed in the order of loop iterations, as if it were in a sequential loop. The first thread to encounter an `ordered` construct may execute it immediately, but every other thread must wait for the thread before it to finish the `ordered` block before it begins the block. Thus we could build a naturally deterministic pipeline by enclosing a loop with an `ordered` block within a larger loop handling the data flow into and out of the pipeline. OpenMP implies no barrier at the end of the parallel loop, so each thread is free to jump to the next iteration and wait only for the completion of the previous task on that iteration. For this to work, however, we would need to create a special version of the outer parallel loop that implies no work sharing but simply iterates over the data set:

```
#pragma omp parallel for pipeline ordered
for (i = 0; i < num_elements; i++) {
  fetch_element(i);
  switch(omp_get_thread_num()) {
  #pragma omp ordered
  case 0:
```

```
    do_task_A();
    break;
  case 1:
    do_task_B();
    break;
  // Etc.
  }
  store_element(i);
}
```

This approach requires modifications both to the standard OpenMP outer parallel loop and the `ordered` construct itself. OpenMP syntax requires the outer loop for an `ordered` construct, but it normally distributes iterations among threads according to either a default or a specified "chunking" schedule. Here, the outer loop must iterate over the data set, but the `ordered` block must "iterate" over threads rather than us the outer loop's iteration variable.

Alternatively, we could build a pipeline construct atop a modified version of OpenMP's `task` and `taskwait` constructs, which are nondeterministic in OpenMP's design. In particular, the OpenMP standard allows the thread that encounters a `task` either to execute it or defer it for another thread, nondeterministically. With the `taskwait` construct, whichever thread has created the task now waits for its completion. Instead, DOMP would have a designated thread that encounters a `task` construct, such as the master, fork a new child thread in order to execute the task. Then `task wait` would have similar semantics to those of the current standard, but the thread that waits would always be the same.

A further modification could name the *task object* and allow the creating thread to wait for its completion by name. This would enable DOMP to express futures [49]

and other non-hierarchical dependency graphs:

```
omp_task my_task;
#pragma omp task(my_task)
  {  /* Task code ... */ }
  // Other tasks
#pragma imp taskwait(my_task)
```

Then, any node in a pipeline graph, however complex, could express the task upon which it depends through the `task` and `task wait` constructs. This approach, while somewhat less intuitive and potentially more complex in implementation, also provides a more general solution for pipelines having arbitrary designs.

For the programmer, however, the tasks of a pipeline might seem most closely to match those of `section`s in the OpenMP `sections` construct, suggesting an intuitive API along the following lines:

```
#pragma omp sections pipeline
{
  while(more_work()) {
  #pragma omp section
    do_task_A();
  #pragma omp section
    do_task_B();
  // ...
  }
}
```

An implementation could use this API as mere syntactic sugar for one of the mechanisms discussed above, or manage low-level synchronization between `section`s directly.

DOMP could implement a deterministic *work queue* similarly to how it would the modified `task` and *taskwait* described above. The master thread would simply spawn a new thread for each task in the queue and would wait for each task in the same order. This approach effectively moves the nondeterminism of assigning physical processors to tasks away from the program and over to the operating system's scheduler, where it belongs, and where it should have no effect on the program's observable execution trace.

* * *

Although deterministic pipelines and work queues remain only imagined extensions to enable DOMP to accommodate as much existing OpenMP as possible, we have implemented the extended reduction, which addresses the largest missing element in the standard OpenMP API's ability to express deterministic idioms. Hence we describe this extension in greater detail in the next chapter.

Chapter 5

Extended Reductions

In addition to the standard reduction for scalar types and simple operations, DOMP includes a new library function to express an extended, generalized reduction over an arbitrary type, passed by reference, applying a user-defined combining operation. In addition, DOMP guarantees that it evaluates both standard and extended reductions in a *fixed order*, one that, provided the combining function is associative, corresponds intuitively to the order of evaluation the program would apply if the parallelizing directives were removed and the code were sequential. This semantic rule makes extended reductions useful for combining operations that are associative but not necessarily commutative, such as matrix multiplication, or floating-point computations on vectors. In all other respects, DOMP standard and extended reductions conform to the semantics of standard OpenMP reductions, but this conformity places special constraints on DOMP's extended reduction API and the combining operation the user supplies. We explore each of these points below, and then give an example of replacing nondeterministic with deterministic code using a DOMP extended reduction. These features implement concepts proposed in [10].

As discussed in 2.2, reductions have a long history [59], including in deterministic

parallel programming models [24], and we make no claim that DOMP's particular approach to reductions is either conceptually novel, or necessarily the "right" design for generalized reductions in OpenMP. We present this approach merely as one design point that attempts to remain consistent with OpenMP's design philosophy, and to balance various practical challenges such as flexibility, efficiency, determinism, support for multiple languages (C and Fortran), and equivalence of serialized and parallelized versions of code (i.e., "eraseability" of OpenMP pragmas).

5.1 API and Semantics

DOMP's syntax for reductions already supported by standard OpenMP is identical to the standard: a clause

`reduction(`*op* `:` *var_list* `)`

where *op* is one of the supported operators (such as +, ∗, or &) and *var_list* is a list of one or more variables, previously declared and initialized, to be subjected to reduction. The OpenMP standard further stipulates some behind-the-scenes manipulations so as to ensure a semantic equivalence between the reduction operation as a whole when run sequentially and in parallel, an equivalence that serves as one of OpenMP's guiding design principles. We return to these implementation details in Section 6.4.

OpenMP does not specify the order in which the implementation updates the original variable, so the order may well be—and typically is (as in GCC)—nondeterministic. Since all standard OpenMP supported operations are commutative as well as associative, however, this nondeterminism does not affect the result. Under debugging, how-

ever, this nondeterminism may leave intermediate states hard to reproduce. DOMP, by contrast, evaluates the reduction in a fixed order that, for an associative operation, corresponds exactly to the order the reduction would follow in sequential code, i.e., with the OpenMP parallelizing directive removed. Aside from easier debugging, this guarantee helps programmers to reason about error accumulation when reducing over floating-point types.

In order to preserve OpenMP's principle of sequential-parallel semantic equivalence, DOMP's *extended reduction* requires a user-defined *identity element*. For instance, for matrix multiplication, this would be the identity matrix of the same dimension as the reduction variable; for vector addition, a 0-vector of the same length. The requirement of an identity element for the user-defined combining operation's domain places an implicit constraint on possible operations, though typically not a burdensome one in our experience. Moreover, DOMP evaluates the reduction in a fixed order regardless of the operation's associativity or commutativity; associativity is required only to guarantee equivalence between sequential and parallel semantics.

DOMP's extended reduction API takes the form of a library function:

```
domp_xreduction(void*(*op)(void*,void*), void** var, void* idty, size_t size);
```

where *op* is the address of a user-defined combining operation, *var* is the address of a pointer to the reduction variable object, and *idty* is the address of the identity object. The user must create the *var* object in a *contiguous* block of memory, and likewise for *idty*. This is necessary in order for DOMP to integrate the reduction operation with its general merge-and-check operation as described in 4.1. The parameter *size* is the size of the reduction variable object, which, of course, should be the same as that of the identity object.

The first argument, *op*, is the address of a user-defined function that takes two arguments, each a generic pointer, and returns the first. The first argument always points to an object that serves as the *accumulator* for the operation. The function *op* always has the side-effect of updating this accumulator object. For instance, to compute the sum of real vectors (arrays of doubles) of dimension *DIM*, we would have:

```
void* vec_add(void* x, void* y) {
  double* a = (double*)x;
  double* b = (double*)y;
  double* end = a + DIM;
  while (a != end) *a++ = *b++;
  return x;
}
```

We explain the rationale for these design choices further in Section 6.4.

5.2 Converting Nondeterministic Code

For an example of OpenMP code that uses a nondeterministic construct as a mere component to build an extended reduction in the absence of language support for one, we turn to the NPB benchmark program EP (source `ep.f`). The program declares two double precision arrays, `q` and `qq`, both of size `nq`, with the former global (in a Fortran common block) and the latter thread-private. EP initializes `q` to 0 before the parallel block. Within the block, it assigns values to elements of `qq` from a pseudorandom function. Then, execution loops through the indices of both arrays, adding the value of the local array element to the corresponding element in the global array:

```
      do 155 i = 0, nq - 1
```

```
!$omp atomic
         q(i) = q(i) + qq(i)
 155  continue
```

Clearly, the programmer intended the array q to serve as a reduction variable, accumulating the sum of the thread-local vectors qq. The *atomic* construct prevents data races on q, but fits only in a nondeterministic programming model. Moreover, the evaluation order depends on OpenMP's assignment of outer loop iterations to threads, which is opaque to the programmer and may, itself, depend nondeterministically on a scheduler.

To convert this code easily to work with DOMP, we placed all necessary definitions in a separate file (`helper.c`), which we could then re-use for other programs having the same pattern (such as DC). We wrote this code in C merely because it was more familiar than Fortran, though DOMP supports generalized reductions in either language. The definitions include the following:

- An identity 0 array of size $n1$ on the heap, assigned to a global pointer `idty`.
- A `vec_add` function similar to the one shown above.
- A simple wrapper for `domp_xreduction`, shown below.

The C wrapper function is, in essence,

```
void xreduction_add(void ** input) {
        domp_xreduction(&vec_add,
          input, (void *)idty,
          nq * sizeof(double));
}
```

In `ep.f`, we defined a Cray pointer to q:

```
      double precision q_val
      pointer(q_ptr, q_val)
      q_ptr = loc(q)
```

Cray pointers are a nonstandard but widely supported extension to Fortran 77 that permits declaration of a variable containing the address of another variable.

Then, before the `omp parallel` directive shown above, we inserted

```
call xreduction_add(q_ptr, nq)
```

Finally, we replaced the `atomic` assignment loop shown at the beginning with a simple call:

```
call vec_add(q_val, qq)
```

Unfortunately, we could not simply leave the loop statement in place minus the `atomic` construct, because of peculiarities in the semantics of Cray pointers.

Although the procedure described above required several steps, it is easy to imagine labor simplifications. First, since many cases of extended reductions will be devoted to arithmetic operations on arrays, it would be simple enough to provide convenient library routines and data structures with all the necessary machinery behind the scenes, allowing the programmer to convert code from standard OpenMP or sequential to DOMP almost trivially in such cases. As it happens, the NPB benchmarks BT and EP use the same idiom, so we adapted them with the identical extended reduction. It might also be possible, in many cases, to infer automatically some of the manual changes shown here. We leave this possibility for future research.

In a slightly more complex case, less adaptable to a library API, we converted a `critical` block in the NPB benchmark DC that accumulates values in a global data structure `g` of type `g_t`:

```
#pragma omp critical
{
 if(g->tm_max<tm0) g->tm_max=tm0;
 g->failure += l->failure;
 if (!l->failure)
   g->num += l->num;
}
```

Although the values for these assignments come from two different sources—the thread-local data structure `l` of type `l_t` and the variable `tm0` of type `double`—the extended reduction's combining operation must express the same logic with a single data source, the second argument, of type `g_t`:

```
void* update_op(void* a_ptr,
  void* b_ptr) {
  g_t* a = (g_t*)a_ptr;
  g_t* b = (g_t*)b_ptr;
  if (a->tm_max<b_tm_max)
    a->tm_max = b->tm_max;
  a->failure += b->failure;
  if (!b->failure)
    a->num += b->num;
  return a_ptr;
};
```

We pass `update_op` to `domp_xreduction` before the parallel block, and simply remove the `#pragma omp critical` annotation.

In all cases, the logic of converting from nondeterministic code to deterministic extended reductions was straightforward.

Chapter 6

Implementation

We implemented DOMP by altering the Gnu OpenMP support library, libgomp, which comes packaged with GCC, as well as by making some changes to the OpenMP-handling code in GCC itself. The resulting DOMP support library is called *libdomp*. This approach makes it easy for a programmer to create or adapt code in a deterministic paradigm while using familiar tools and platforms. Moreover, we leveraged many elements of libgomp's internal API so as to concentrate on WCD-related components.

We shall first describe our basic threading framework and the machinery we used to implement Working Copies Determinism in user space. Given this approach, our major implementation challenges were (a) reducing the cost of the WCD algorithm described in Section 4.1 so as to allow acceptable performance; (b) providing backward compatibility of standard (simple) reductions with deterministic semantics; and (c) supporting the new extended reductions API. We review each in turn. Finally, we consider the limitations of our current implementation and how these might be resolved in the future.

6.1 User-Level WCD

In this section, we describe the threading model we use to support WCD, as well as our approach to heap memory, and, finally, our integration of DOMP code with the appropriate "hooks" in the libgomp's internal interface.

6.1.1 Threading Model

Working Copies Determinism requires that the runtime provide each thread with its own isolated logical copy of shared state, and also produce a *reference copy*, at the fork, as well as that it merge these copies while checking for write-write conflicts at the join (see 4.1). Thus for the underlying threading infrastructure, lightweight threads such as pthreads would present an ill fit: each thread's view of the address space should be the same, but the data held at a given location should be private to that thread.

The implementation of threads marks one of the many important differences between libdomp and libgomp. In libgomp, the underlying thread mechanism is pthreads, which all share the same address space. GCC transforms each parallel block into a separate function that accepts as its argument a pointer to a data structure holding copies of all shared stack and heap variables to which the function needs access. In turn, libgomp passes this function and data structure to `pthread_create`. While this design could, in principle, isolate each thread's copy of *value-type* variables in a manner consistent with WCD, it would have no way of preventing concurrent threads from writing to the same location by way of a pointer field in the data structure passed in as the argument, nor from similarly writing to the same global variable. For this reason, we chose to represent DOMP's threads by means of separate underlying *processes* created by Unix (or Linux) `fork`. At creation, then, each thread

```
for each data segment seg in (stack, heap, bss)
  for each byte b in seg
    writer = WRITER_NONE
    for each thread t
      if (seg[t][b]] ≠ reference_copy[b])
        if (writer ≠ WRITER_NONE)
          race_condition_exception()
        writer = t
    seg[MASTER][b] = seg[writer][b]
```

Figure 6.1: Naive implementation of merge loop.

gets its own complete and isolated copy of its parent's address space "for free" as part of the semantics of Linux `fork`. DOMP later joins these threads to the parent thread with `waitpid`. As in libgomp, the parent (or "master") thread creates, for a team of n threads, $n-1$ children, since it serves, itself, as one of the team members.

In this sense, DOMP adopts virtual memory techniques similar to those of Grace [18] and Dthreads [70] in order to achieve determinism; but, unlike these deterministic-scheduling-based projects, DOMP uses VM machinery for thread isolation following the WCD model.

Not only does Linux `fork` create a logical copy of shared state "for free," it even manages copy on write at system level, leaving the task of merging alone to libdomp. In a naive implementation, without further cost reductions, the merge routine requires three nested loops, as shown in Figure 6.1: it must examine each of the three distinct data segments of the shared address space—the stack, the heap, and the region for static variables (both initialized and uninitialized). Then, it must iterate over each unit of data (e.g., byte) in each data segment, and each thread's version of each unit. We discuss ways to reduce the cost of this operation and others in the next section. Important to note here is that, although Linux manages copy

on write for each independently-running thread, it does not do so, naturally, for DOMP's reference copy. Therefore, before creating the threads to make the team, the parent thread would have to make and set aside a full copy of all of its data segments.

6.1.2 Heap Memory

The parent thread, then, before spawning any children, must determine the bounds of its stack, heap, and static variables segments, and copy them (using, e.g., `memcpy`) to create the reference copy. The heap presents a potential problem in this case: popular implementations of `malloc` and its relatives, such as Doug Lea's *dlmalloc* [66], set memory aside for the heap in a "lazy" fashion, only upon the first call to an allocating function, and increase the size of the heap only incrementally. Thus we might easily encounter the case where the first parallel block of the program starts with no previous heap allocation calls in the program, and therefore no memory set aside for the heap. Suppose, then, that the program declares a pointer variable before the parallel block, but has one child thread allocate memory to it within the block. This works perfectly well in standard OpenMP, where all threads share the same address space, and it should likewise work in DOMP if the latter is to serve as a drop-in replacement for the former. However, in this scenario, at merge time, the child has a heap of some size, whereas the parent has none. How would the parent know to copy the child's heap into its address space and what the bounds of that heap are? Of course, we must also provide for the case where the parent allocates heap memory for a variable that a (single) child modifies in the parallel block.

The solution we chose was to allocate a fixed range of addresses for the parent's heap, and another one for each child, all before the parent creates the first child thread. In particular, we map a space sufficient for all of these heaps together with

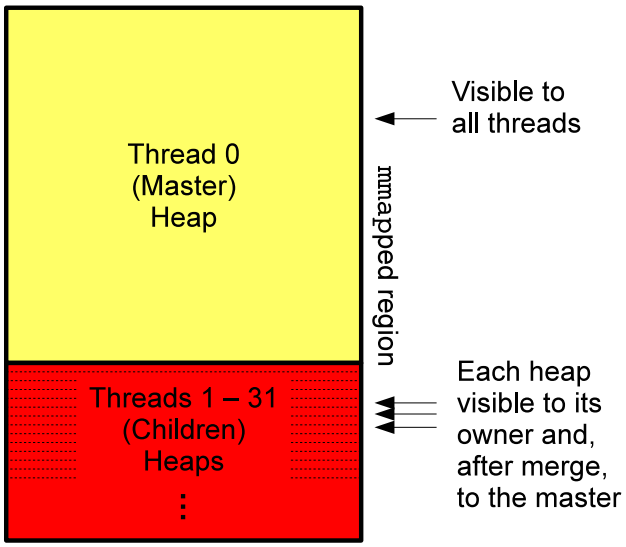


Figure 6.2: Structure of DOMP heaps allocated before any parallel execution.

In/After Block	Heap Owner	Can Do with Variables		
		See	Allocate	Free
in	master	all threads	all threads	all threads
	child	owner	owner	owner
after	master	master	master	master
	child	master	(none)	master

Table 6.1: Which threads can do what with variables from whose heap during and after a parallel block.

a single `mmap` call, thus minimizing system overhead. We set the `MAP_PRIVATE` flag, which gives the heap the same system copy-on-write semantics as the stack and static variables have after `fork`. The parent uses the heap bounds from this partitioned `mmap`ed region to create the appropriate reference copy at thread creation time and to find its own and its children's respective heaps at merge time. Since we found, in practice (e.g., in our benchmarks), that child threads update or merely read the parent's heap-allocated variables much more often than they allocate memory to pointers that they inherit, we apportioned more space to the parent's heap (as "main" heap) than to any child's heap. The specific proportions, however, were a result of trial and error and mere guesswork: 16 MB (2^{24} bytes) for the parent and 256 KB (2^{18} bytes) for each of a maximum of 31 children, giving a total of 24,903,680 bytes to be `mmap`ped. A better approach would entail the rigorous study of heap usage in a sample of a range of popular programs. Note, however, that the total space to be allocated by `mmap` under the standard C library implementation we used (*glibc* 2.11.1) must not exceed the maximum value of its `size_t` *length* parameter, or $2^8 - 1$ on the Linux x86-64 system we used. Figure 6.2 represents this arrangement, while Table 6.1 summarizes which threads have which capabilities with respect to which heap variables both within and after a parallel block.

Having each thread need to refer to its own heap space for heap operations leads us to define a global data structure for each thread, `thread_record_t`, one of whose fields is a pointer to that thread's heap. We list and discuss this and other fields and data structures below, in 6.5. Moreover, since any thread may call free on any variable whose space was allocated by itself or any other thread, the merging thread may encounter a conflict in malloc bookkeeping structures at merge time, if these consist of counters or bitfields, as is the case with the *dlmalloc* that provides the basis for DOMP's implementation of the heap. To prevent such spurious data races,

DOMP implements `free` by placing the address to be freed onto a queue belonging to the thread that owns the heap containing that address. This queue, then, is stored in the `queued_frees` field of a data structure accessible to all threads, as described further in 6.5.

Given Linux's copy-on-write behavior, examining child threads' address spaces at merge time poses its own challenges. For instance, the parent thread, which runs the merge routine represented in Figure 6.1, has no direct access to a list of the pages that the system has copied on write for each child thread. If it attempted to examine every page in the address range of the data segment, it would very likely provoke a segmentation fault by looking for a nonexistent page in memory. Instead, it would have to obtain the list of pages copied on write by examining the contents of the child thread's `/proc/`<*pid*>`/maps` file in the Linux `proc` pseudo-filesystem, where <*pid*> stands for that child thread's process ID. (Of course, the parent must first halt each child thread using the Linux `ptrace` call, as any thread must do in any scenario in which it examines the address space of another thread.)

The requirements of creating an appropriate "snapshot" reference before parallel execution and of examining child threads' data efficiently are easier to fulfill with the mechanism of trapping and trap handling that we describe in Section 6.2 as a cost-saving measure. Thus DOMP's implementation requirements intertwine cost reduction with basic practicality.

6.1.3 Using libgomp's Internal API

When GCC compiles OpenMP source code, it transforms the code of an internal block into a separate function, using a name mangling scheme to give it a unique name. In the containing function, then, in place of the block, it inserts a sequence of calls to `GOMP_parallel_start`, the newly-defined function, and `GOMP_parallel_end`,

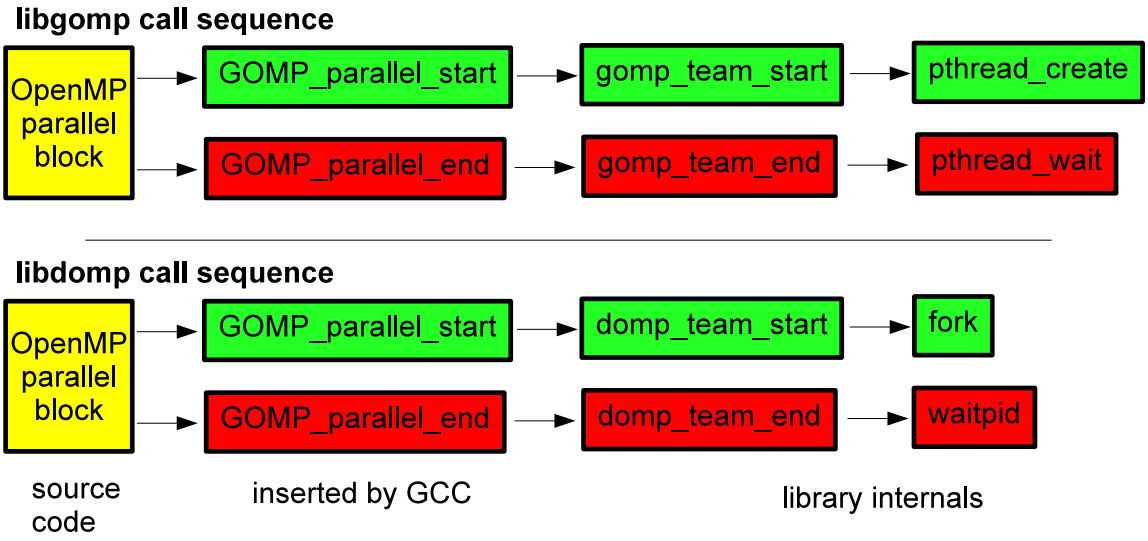


Figure 6.3: Comparison of libgomp's and libdomp's respective call sequences, using GCC's automatically-generated wrapper call.

in that order. In this way, the main (master) thread starts parallel execution for the rest of the team, then executes the parallel code itself as a member of the team, and, finally, waits for the rest of the team to finish.

GCC's generated code passes several arguments to `DOMP_parallel_start`, including a pointer to the newly-created separate function and a pointer to an ad-hoc data structure whose members correspond to all the shared variables to which the function will require access. Of course, this function-data pair is tailored to fit the API of `pthread_create`, libgomp's underlying threading framework. `GOMP_parallel_start` then calls the libgomp function `domp_team_start` with the same arguments, which, in turn, calls `pthread_create` once for each thread on the team besides the master, passing the same function and data pointers. It is then a fairly simple matter to switch libgomp's call inside `GOMP_parallel_start` from `gomp_team_start` to our own `domp_team_start`, as shown in Figure 6.3.

6.1.4 Work Sharing

We implemented the loop and `sections` OpenMP work sharing constructs, leaving the `task` construct for future work. Naturally, DOMP's implementation must be deterministic, meaning that the assignment of tasks to threads must be the same on every run of the same program with the same input.

In the case of the loop construct, fortunately, GCC's thread scheduling scheme is already deterministic, and GCC expresses this scheme directly in its generated code, by inserting instructions to have each thread query the total number of threads and its own ID with `omp_get_num_threads` and `omp_get_thread_num`, respectively. Then, with the default "static" scheduling setting, each thread divides the number of iterations by the number of threads and then assigns to itself the appropriate "chunk" of iterations based on its ID. OpenMP also offers a "dynamic" scheduling option, which is by nature nondeterministic (to allow load balancing), but we altered GCC's interpretation of the OpenMP parallel block's `schedule` clause so that `dynamic` means the same thing as `static`.

In the case of `sections`, things were not quite so simple: GCC assumes a *dynamic* schedule, with code that is awkward to alter in order to achieve a deterministic result. (The OpenMP standard stipulates that "[t]he method of scheduling structured blocks [i.e., the individual `section` blocks] among the threads in the team is implementation defined" [78]. However, we could still use GCC's internal API. In the object code, GCC emits calls to `GOMP_sections_start` and `GOMP_sections_end` at the beginning and end of the `sections` block, respectively, in a way analogous to `GOMP_parallel_start` and `GOMP_parallel_end`. Then, the special function representing the overall parallel block calls `GOMP_sections_next` in a loop, and uses the integer return value to determine the location to which to jump to

execute the appropriate `section` code, until `GOMP_sections_next` returns 0. We simply replaced libgomp's code for `GOMP_sections_start` and `GOMP_sections_next` with calls to libdomp's `domp_sections_start` and `domp_sections_next`. The former simply assigns the total number of `section`s (which GCC computes statically and passes as an argument to `DOMP_sections_start` to a field, `num_sections`, in the (globally scoped) thread-local `thread_record_t` data structure. Another field, `sections_done`, starts out at zero. After this initialization, `domp_sections_start` simply calls `domp_sections_next` for the first section assignment. DOMP's `domp_sections_next` assigns sections based on the thread ID, and adds the total number of threads to `sections_done`. If there are more `section`s to be done than there are threads, the next iterative call to `domp_sections_next` will return `sections_done` plus thread ID. If this sum exceeds `num_sections`, `domp_sections_next` returns 0 instead, which enables the executing thread to break out of the `section`s loop.

In this case, we clearly sacrifice some performance benefit from dynamic thread scheduling in order to uphold determinism. As it happens, none of the benchmarks we considered used the `sections` construct, so testing to measure this performance sacrifice remains for future work.

6.2 Reducing Cost

In a naive implementation, the most costly elements of DOMP are thread/process creation and merging. Thus the time cost of a program grows as a function of the number of parallel blocks (or iterations over a given block), the number of parallel threads per block (each with its thread creation overhead), and the size of the shared data to be compared and merged. This cost may be hard to offset with the benefit of parallelism.

Standard efficiency techniques improve this situation:

- Copy on write at page granularity
- Merge or copy pages only as needed
- Merge in parallel along a binary tree
- Create and keep a thread pool, to be destroyed at program end.

With thread creation now a one-time cost, overall cost grows with the log of the number of threads times the number of shared pages.

We discuss the details of each improvement technique in the following subsections.

6.2.1 Copy on Write

To create the reference copy, we use standard techniques to implement copy on write at the user level. At the start of a parallel block, in an initialization phase, the master thread finds the address bounds for the three shared data segments that will be in scope for the ensuing parallel execution: the stack from the contents of the stack pointer upon the call to `GOMP_parallel_start` up to a suitable upper limit, such as the glibc-defined symbol `__libc_stack_end`; the heap, including the entire `mmap`per range that contains both the main heap and the child heaps, as described in 6.1.2, and the static variables (or "bss") region. In order to make the bounds of the bss segment visible to DOMP's initialization code, we resort to a simple linker trick: we define the symbols `domp_bss_start` and `domp_bss_end` in two separate files, `head.s` and `tail.s`, respectively, and then link them to positions before and after the executable code, respectively.

Also during initialization, the main thread (at that point, the only thread) creates an array, called `threads`, of data structures called `domp_thread_t` to keep track

of such things as each thread's writes. Because sach thread will have to write to the `domp_thread_t` representing it and the master thread will have to read the whole array at merge time, the main thread creates the array in an `mmap`ped region accessible to all threads existing or subsequently created. When the master thread creates the child threads of the team, it increments a global thread count, which serves as the new thread's thread ID, starting with 1. This thread ID also serves as the index in the `threads` array to the `domp_thread_t` representing that thread.

Still during initialization, the main thread opens three files to hold the reference copies for the three data segments, storing their file descriptors in a globally-visible array, `ref_copies`, to which all threads will have access.

At the fork, then, we write-protect the data segments that represent shared data in scope for the concurrent threads. More precisely, we impose write protection on the heap and the bss segments, but not the stack, since write protecting the stack introduces too many runtime complications. Instead, we always assume that we must always merge stacks, and the main thread creates the reference copy for the stack during initialization. In practice, we find that this involves no more than two pages.

On the heap and bss segments, we impose write protection at the last moment, for reasons that will be clearer when we describe the thread pool in 6.2.4. Thus, immediately after calling `fork`, the master thread calls `mprotect` on all three data segments. Each new thread does the same on its own version of the data segments after it is created and just before it starts executing the parallel code.

During parallel execution, when any thread writes for the first time (i.e., for *its* first time) to a given page of shared data, the write provokes a `SIGSEGV` segmentation fault signal. DOMP's trap handler then finds the trapping address, rounds it down to the page boundary (using a page size of 4096 bytes), and lifts write protection for that page. It also stores the page address in the `pages_written` field (an array of

off_ts) of the domp_thread_t whose index in the threads array corresponds to its thread ID. Next, it determines to which data segment the page belongs, and creates a reference copy of that page—but only if no thread has done so before. To prevent such duplication (and to prevent any possible undefined behavior when two threads write the same copy "at the same time" to the same place in the reference copy file), the signal handler first checks a bit field representing reference-copied pages, ref_copy_locks, which is unique and globally visible to all threads, as explained further in 6.5 below. It uses an atomic bit-test-and-set instruction to check the bit corresponding to the trapped page. If the bit is clear, this instruction sets it—and the signal handler then copies the page to the reference copy file corresponding to the appropriate data segment, at the offset in the file equal to the offset of the page address from the start of the segment. This offset scheme prevents any possible conflicts between two threads writing to the same address range in any reference copy file.

When the signal handler returns, the trapping thread is now free to write to the page in question, and the reference_copies file for that page's data segment has a copy of the page in its state before any writes have occurred.

6.2.2 Merge or Copy As Needed

The "lazy" approach exemplified by copy on write applies equally well to the merge operation: we need only merge data together if we know that they may differ. More precisely, the master need only examine and merge with a given page of its own address space the corresponding page in any other thread's address space to which that thread has written—and, thanks to our copy on write mechanism, we have a record of these writes, one set for each thread, in the pages_written field of the domp_thread_t object representing that thread in the threads array.

For clarity's sake, we describe the merge operation here as if we had not implemented parallel merge, to be discussed later in 6.2.3. Before actual merging, the merge routine inserts the stack pages into each thread's `pages_written` array, including that of the master. It also sorts the individual `pages_written` arrays of all the respective threads, again including that of the master. Then, it goes through all the `pages_written` arrays together. If it finds a unique entry not belonging to the master, this means that only a single thread has written to that page. Therefore, the whole page can safely be copied over from its source to the master thread's address space. Only if it finds two or more entries for the same page must it execute the merge loop as shown in Figure 6.1—replacing "for each byte b in *seg*" with "for each byte b in *page*."

6.2.3 Parallel Merge

We found that an implementation with only the foregoing efficiency measures still provided unsatisfactory performance on the few benchmarks we tested. (We unfortunately did not keep records of these experiments, but we recall a worse than twofold slowdown.) The next major improvement came from merging updates in parallel. This not only enabled DOMP to parallelize the workload across more processors, it also reduced the workload and code complexity of each merge operation, since now each such operation only involves, at most, three sources of data: the current thread (or *self*, one other thread (*other*), and the reference copy. We then organized the parallel merge operation along a binary tree. For instance, for two threads, Thread 0 (the master) merges Thread 1. For four threads, Thread 0 merges Thread 1 while Thread 2 merges Thread 3; then Thread 0 merges Thread 2.

Figure 6.4 illustrates the pattern DOMP follows for parallel merging, with the root node of each (non-leaf) subtree labeled by the thread that merges data from

another thread into its own address space. Since now the master is not the only thread engaged in merging data into its own address space, we use the terms *up-buddy* and *down-buddy* for the *self* and *other* involved in each component merge operation, respectively. As we just noted, each merge now involves only a three-way comparison of the up-buddy, the down-buddy, and the reference copy. This removes one level of the nested loop in Figure 6.1.

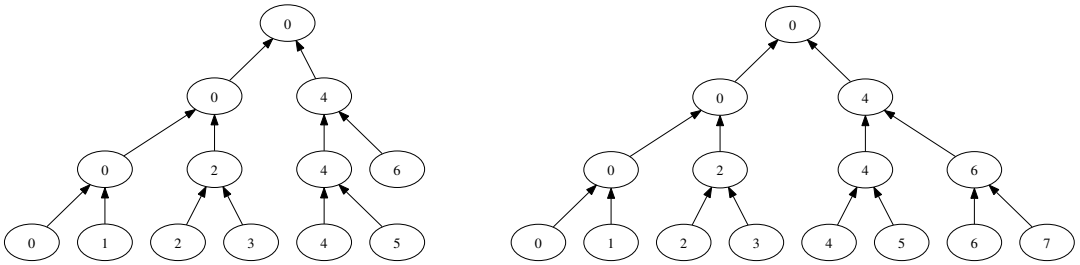


Figure 6.4: DOMP merge scheme for (a) 7 and (b) 8 threads. Efficiently parallel evaluation of reductions can coincide with merging.

In general, each even-numbered thread will serve as an up-buddy at least once, whereas odd-numbered threads can only be down-buddies. The down-buddies, by thread ID d, for a given up-buddy with even thread ID u in a team of n threads are given by the formula:

$$d = 2^p + u, \text{with integer } p \geq 1, d < n, \text{and} \begin{cases} 2^p < n, & \text{if } u = 0, \\ 2^p < u & \text{if } u \neq 0. \end{cases}$$

In order for the up-buddy to merge or copy data from the down-buddy, it first calls `ptrace` and `waitpid` on the down-buddy to synchronize with it, and then opens the down-buddy's address space as a file, using Unix's (Linux's) `/proc/<`*`pid`*`>/mem` pseudo-file. It stores the file descriptor in the `mem_fd` field of the `threads` entry representing the down-buddy, and closes the file at the end of the merge process. It

then copies data page by page into a local buffer for use in merging or copying into its own address space.

The initialization process remains the same as before, as described above in 6.2.1, since this design makes it possible for all threads to see each other's `domp_thread_t` in the globally-visible `threads` array. One small but important difference from the merge routine as described above, however, is how each up-buddy *accumulates* records of pages written before finishing and allowing itself, in turn, to serve as down-buddy to the lower-numbered thread that is its up-buddy. When an up-buddy thread finishes merging its down-buddy's data into its own address space, it inserts the addresses for pages that it has merely copied from its down-buddy (because only the down-buddy has written to those pages) into its own `pages_written` array. That is, although the up-buddy did not write directly to this page during parallel execution, it has, in effect, written to the page now by copying its down-buddy's updates onto it. Thus records of pages written filter up through the parallel merge process, at the end of which the master has a full list of all pages to which any thread has written in the foregoing parallel block.

6.2.4 Thread Pool

OpenMP is designed to allow an arbitrary number of parallel blocks within a program, and it is also possible to iterate over a parallel block. For example, the PARSEC benchmark *blackscholes* iterates over its one parallel block 100 times; the NPB benchmark *EP* has three within the main routine, and BT has 12 spread across various subroutines. With the design described thus far, every parallel block would incur the overhead of creating a new team of threads, with all of its attendant initializations. To avoid this overhead, we implemented a thread pool: the main thread initializes the global data structures and heaps and then creates the threads for the

team once for the program. When a child thread reaches the end of a parallel block, it waits for a signal from the master to restart *or* exit. If the message is to restart, the thread executes the function, with argument, to which the `function` field of the `global_vars_t` object points (see 6.5. If the message means to exit, the thread exits. At the end of the program, a clean-up routine that we register using the `atexit` call waits for each child thread (process) to exit and closes open files and pipes (to be discussed below in this section) before program termination.

We provide for changes in the number of threads from one parallel block to the next, although, in practice, none of the programs we encountered called for this feature. The feature depends on the main thread's initialization routine creating data structure entries and heaps for some global constant maximum number of threads (currently 32). Given that no program seems to need to change the number of threads from one block to the next, we could, in a future version of DOMP, improve memory efficiency by apportioning only the data structures and heaps necessary for the initial number of threads, which is generally known at program start.

Adding a thread pool to the design introduces the need for a whole new layer of inter-thread communication, which we support by means of Unix (Linux) *pipes*. When the main thread creates a new thread, it opens two pipes, one for messages from the child thread to its up-buddy and the other from the master to the child thread. Each message consists of a single byte to minimize overhead. The sequence of messages and related actions at the end of a parallel block, i.e., when a child thread returns from the special function representing that parallel block, is as follows:

1. If the thread is an up-buddy, it does the following for each of its down-buddies:

   (a) It waits for a "ready-to-merge" signal from the down-buddy.

   (b) When it receives the signal, it synchronizes with the down-buddy (using

`ptrace` and merges the down-buddy's data updates with the data in its own address space.

2. It sends a "ready-to-merge" signal to its up-buddy.

3. It waits for a "restart" signal from the master.

The master thread performs step 1 with respect to its down-buddies in `domp_team_end` (see 6.3), but does not take the steps 2–3, since, naturally, it lacks an up-buddy.

The master sends the "restart" message in `domp_team_start` (see 6.3), in place of initialization and creating the new threads, on subsequent parallel blocks after the first. Before doing so, it assigns to a global variable (the `function` field of the `global_vars_t` object—see 6.5) a pointer to the function to be executed in parallel. This is the special function that GCC has created to represent the current parallel block, a pointer to which GCC's emitted code passes as an argument to `GOMP_parallel_start`, which passes it to `domp_team_start`. Every thread, on restart, executes whatever function this field points to.

When the pooled threads restart, they must work with a fully updated version of the shared data, which they get from the master, since the master has updated its data at the last `domp_team_end` call. To support this data transfer, the master creates yet another globally-visible file at initialization, the `scratch` file. Furthermore, it creates an *extra entry* in the `threads` array to hold cumulative update information. (This is the "scratch" entry, and includes, among other things, a `pages_written` field.) Then, in `domp_team_start`, when it is called a second or later time, before the master sends out the "restart" signal, DOMP transfers the updated data from master to child threads as follows:

1. The master calls `copy_out`. This routine takes the following steps:

(a) The master copies its `pages_written` array to the `pages_written` field of the "scratch" entry in the `threads` array. Up to this point, the master has left its `pages_written` array untouched since its last call to `domp_team_end`, so this array contains a cumulative record of all pages updated in the previous parallel block.

(b) It clears its own `pages_written` field.

(c) Using the "scratch" `pages_written` array as a guide, it copies all pages updated in the previous parallel block from its own address space to the `scratch` file.

2. Each child thread calls `copy_in`, which takes the following step:

   (a) Using the "scratch" `pages_written` field as a *key*, it copies each page from the `scratch` file to the appropriate place in its own address space.

A further new complication is that the master thread may update shared data *between* parallel blocks. In order to accommodate this possibility in our copy-on-write scheme, the master must therefore write-protect its heap and bss segments at the end of `domp_team_end`, trap its writes as if in parallel mode, and record them—without making a reference copy. When the master calls `copy_out` in the next execution of `domp_team_start`, its `pages_written` array will include these records, along with those of pages updated during the previous parallel block. Of course, this does mean incurring the overhead of `mprotect` and trapping even after the end of the last parallel block of the program, but it is hard to avoid this waste without more advanced static analysis that will mark the final `domp_team_end` call for special treatment.

* * *

The foregoing efficiency improvement techniques all have some effect on the bookkeeping data structures DOMP uses to manage threading, synchronization, and merging. But, in addition to a core functionality made reasonably efficient by these methods, DOMP also supports both simple (OpenMP-style) and extended (generalized) reductions, which also require their own bookkeeping data fields. We therefore describe our data structures only after discussing DOMP's support for these two types of reduction.

6.3 Simple Reductions

While offering a new generalized reduction, DOMP naturally must also support the standard OpenMP reduction, with no changes to application source code, and with deterministic semantics. The OpenMP standard stipulates that the implementation must create, for each reduction variable, a private copy within the parallel block, initialized to the *identity value* for the given operator. Then, at the end of the parallel block, the implementation updates the original reduction variable with each new private copy, using the given combining operator again. This behind-the-scenes manipulation ensures the sequential-parallel semantic equivalence discussed in Section 5.1.

We were able to leverage GCC's compile-time handling of variable copying and initialization. The key change in DOMP was to replace GCC's nondeterministic, atomic update mechanism with a deterministic, fixed-order evaluation at merge time. GCC updates the shared reduction variable by simply turning the instruction representing the combining operation—say, ADD—into an atomic operation—e.g., LOCK ADD. In the x86 context, this means adding the LOCK prefix to the instruction for the combining operation. Instead, in place of the LOCK-plus-operation instruction,

```
int main(void) {
        int x = 0;
        int y = 2;
#pragma omp parallel num_threads(4) reduction(+:x)
        {
                x += y;
        }
        return x;
}
```

Figure 6.5: Minimal example program with a standard OpenMP reduction.

we have GCC emit a call to a new "built-in" function that we add to GCC's repertoire, `DOMP_reduction_info`. GCC passes to `DOMP_reduction_info` the following four arguments:

1. An integer that uniquely identifies the combination of the type of the reduction variable and the operation, e.g., integer addition.

2. An integer that uniquely identifies the type of the reduction variable. Although this information is redundant, it simplifies later lookups.

3. A `void` pointer to the local variable that GCC creates to take the value to be combined with the reduction variable. For instance, if the source code has `x += y`, this is a pointer to `y` (or to a variable GCC creates to represent `y` after various transformations).

4. A `void` pointer to the field representing the reduction variable in the data structure that GCC creates and passes to the special function representing the parallel block.

To illustrate our use and alteration of GCC's emitted code, we turn to a minimal example, whose source code is given in Figure 6.5. Figures 6.6 and 6.7 show an intermediate stage in standard GCC's transformations, simplified and edited for

```
main ()
{
  int y;
  int x;
  struct .omp_data_s.0 .omp_data_o.1;
  x = 0;
  y = 2;
  .omp_data_o.1.x = x;
  .omp_data_o.1.y = y;
  GOMP_parallel_start (main.omp_fn.0, &.omp_data_o.1, 4);
  main.omp_fn.0 (&.omp_data_o.1);
  GOMP_parallel_end ();
  x = .omp_data_o.1.x;
  y = .omp_data_o.1.y;
  return x;
}
```

Figure 6.6: Standard GCC's transformation of `main` in Figure 6.5.

readability. Figure 6.6 shows the transformation of the `main` function, including the calls to `GOMP_parallel_start` and `GOMP_parallel_end`, as described in 6.1.3. The the third argument to `GOMP_parallel_start` is the number of threads, in this case 4. Figure 6.7 shows the special function GCC creates to encapsulate the parallel block in the original source in Figure 6.5. Note in particular the call GCC inserts into this special function to the atomic function `__sync_fetch_and_add_4`. The 4 in this function's name refers to the number of bytes in the operands, which is, of course, appropriate for the 32-bit integer reduction variable. In turn, libgomp defines this atomic function and its many relatives as inline, enabling its replacement with a single atomic instruction. FIgure 6.8 shows the disassembly of the entire special function representing the parallel block, as compiled with standard GCC on an x86 machine running Linux, where it is easy to see the "lock add" instruction.

Figure 6.9 gives the corresponding intermediate representation of the special function from our altered, DOMP-enabled version of GCC. (The intermediate represen-

```
main.omp_fn.0 (struct .omp_data_s.0 * .omp_data_i)
{
  int x;
  x = 0;
  x = x + .omp_data_i->y;
  __sync_fetch_and_add_4 (&.omp_data_i->x, x);
}
```

Figure 6.7: Standard GCC's transformation of the parallel block in Figure 6.5.

```
<main.omp_fn.0>:
push   %rbp
mov    %rsp,%rbp
mov    %rdi,-0x18(%rbp)
movl   $0x0,-0x4(%rbp)
mov    -0x18(%rbp),%rax
mov    (%rax),%eax
add    %eax,-0x4(%rbp)
mov    -0x18(%rbp),%rax
lea    0x4(%rax),%rdx
mov    -0x4(%rbp),%eax
lock add %eax,(%rdx)
leaveq
retq
```

Figure 6.8: Disassembly (x86) of the code representing the parallel block in Figure 6.5.

```
main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i)
{
  int x;
  x = 0;
  x = x + .omp_data_i->y;
  DOMP_reduction_info (30, 4, &x, &.omp_data_i->x);
}
```

Figure 6.9: DOMP-enabled GCC's transformation of the parallel block in Figure 6.5.

tation of `main` is identical in this version.) Note in particular DOMP's replacement of `__sync_fetch_and_add_4` with the libdomp function `DOMP_reduction_info`.

The libdomp function `DOMP_reduction_info` stores the information from all four arguments in a `reduction_var_t` object, which has four corresponding fields. The `reduction_var_t` object, in turn, is an element of the thread's `reduction_vars` array, which is, itself, a field in the `domp_thread_t` object representing that thread in the global `threads` array. (See 6.5 below.)

In the case of the third variable—a pointer to the variable holding the value to be aggregated to the reduction variable—the current thread dereferences it and stores its value in the `value` field of the `reduction_var_t`. The `value` field must accommodate the value of a variable of any type and size allowed for an OpenMP reduction. Its therefore has a union type (a `common_value_t`, and the unique type identifier (`DOMP_reduction_info`'s second argument) enables `DOMP_reduction_info` to select the right member of the `common_value_t` by serving as an *index* into an array of functions, each of which casts the `void` pointer to the right type, dereferences it, and stores it in the appropriate `common_value_t` field of the next entry in the current thread's `reduction_vars` array.

In a somewhat similar fashion, during the merge process, the up-buddy goes through all of its own and its down-buddy's reduction variables (which should be matching lists) and combines their respective values using the appropriate operation.

This time, it finds the right operation by using the integer stored from the first argument to `DOMP_reduction_info` as an index into another array of functions, each function corresponding to one variable type, one variable size, and one combining operation. The array contains all the combinations of types, sizes, and combining operations allowable for a standard OpenMP reduction.

After the master has performed these reduction operations with respect to all of its down-buddies, as a final step, it goes through the same list of reduction variables and combines the current value of the reduction variable with its original value, accessible through the pointer stored as the fourth argument to `DOMP_reduction_info`, and stores the result in the latter location. This completes the reduction.

As just noted, the intermediate evaluation steps of reductions are mixed in with the merging process. As it happens, the binary tree pattern we use for efficient merging lends itself conveniently to the fixed-order evaluation of both standard and extended reductions, since the order in which data are merged from one thread to another follows the order of threads numbered from 0 (the parent or master). Thus, suppose we have 8 threads. Let z be the reduction variable, with z_f being its final value after reduction, and $z_0 \ldots z_{n-1}$ corresponding to the respective values of the local versions of variable z at the end of the parallel block (but before merging) for threads $0, \ldots, n-1$. We use $\square$ to represent the combining operation. Then

$$z_f = (((z_0 \square z_1) \square (z_2 \square z_3)) \square ((z_4 \square z_5) \square (z_6 \square z_7)))$$

Note that this evaluation order does not move the elements from their original order as in a sequential version of the program. The programmer can reason about the evaluation of a reduction using the "$\square$" operation, without the "$\square$" operator having to be commutative, though it must be associative for this equivalence to hold.

If the number of threads is fewer than the number of parallel loop iterations or sections in which a reduction occurs, GCC by default assigns threads to iterations (or sections) in contiguous chunks. For instance, if two threads execute eight parallel iterations, thread 0 gets iterations 0 through 3 and thread 1 gets the rest. Within each chunk, the code executes *sequentially*; and, given DOMP's compliance with OpenMP's sequential-parallel semantic equivalence, the results would be

$$z_f = ((z_0 \square z_1 \square z_2 \square z_3) \square (z_4 \square z_5 \square z_6 \square z_7))$$

This again preserves sequential order for associative operations.

6.4 Extended Reductions

To allow for arbitrary types and operations, we could not leverage GCC's compile-time machinery as we could for standard reductions, and must implement more of the required behind-the-scenes actions to occur at runtime. The DOMP built-in function `domp_xreduction` makes a copy of the identity object on the heap, a *scratch* object, and re-points the pointer to the reduction variable object `var` to point to this scratch object. (This is why the first argument to `domp_xreduction` has type `void**`.) It also records address and type information about the variable, identity, and scratch objects. At the fork, each thread gets a copy of the scratch object, and, during parallel execution, when any thread calls the combining operation, what looks like the first argument of that operation, the reduction variable, is actually a pointer to the scratch object, which is a copy of the identity object, so the result will be the same as the second argument. Therefore, this call to the combining operation effectively just copies the second argument to the scratch object. But, like any write,

this act of copying the second argument onto the scratch object traps, causing the signal handler to record the page.

At merge time, the up-buddy goes through all pages written by itself and its down-buddy. Before merging the data, it first resolves any extended reductions for that page, this time using the up-buddy's scratch object as the combining operation's first argument, the accumulator. Once within the merge loop, the up-buddy skips over the address ranges of the scratch objects, which otherwise appear as write-write conflicts.

Finally, the master thread calls the combining operation, this time with the original reduction variable as the first (accumulator) argument and its scratch object as the second. It restores the pointer back from the scratch object to the reduction variable, and deletes the scratch object.

This sequence fulfills the stipulations of the OpenMP standard for simple reductions, thus providing sequential-parallel semantic equivalence for associative combining operations, while ensuring an efficient, deterministic evaluation order.

6.5 Data Structures

In the foregoing sections, we have referred to several data structures used for various kinds of bookkeeping. Here, we review these data structures in detail.

DOMP uses three data structures that are global in some sense:

- A `global_vars_t` object, `g`, which the master creates in a separate mapped file before creating the thread pool, and which is therefore visible to all threads, at global scope.

- The `threads` array of `domp_thread_t` objects, which is a field of `g` and is thus also visible to all threads, at global scope.

```
typedef struct global_vars_t {
        struct segment_t data_segments[NUM_SEGS];
        struct func_t function;
        struct domp_thread_t * threads;
        int num_threads;
        int ref_copies[NUM_SEGS];  // Open file descriptors.
        int * ref_copy_locks[NUM_SEGS];
        struct xreduction_var_t xreduction_vars[DOMP_MAX_REDUCTION_VARS];
        int num_xreduction_vars;
        int scratch;   // Open file descriptor.
        bool in_parallel;
} global_vars_t;
```

Figure 6.10: The `global_vars_t` data structure.

```
typedef struct segment_t {
        off_t start;
        size_t length;
} segment_t;
```

Figure 6.11: The `segment_t` data structure.

- A `thread_record_t` object, one at global scope for each thread, whose values are different for each thread, and which is visible only to the thread to which it belongs.

The definition of the `global_vars_t` type is given in Figure 6.10.

The `segment_t` data structure has two fields, as shown in Figure 6.11.

The `data_segments` field has one such structure for each of the three shared data segments—stack, heap, and bss—which the threads use for `mprotect` calls to support copy on write and to guide merging (6.2.1).

The `function` field's `func_t` data structure also has two fields, as shown in Figure 6.12.

These two fields hold pointers to the current function and its argument data structure, respectively. Recall that, when GCC creates a special function to represent

```
typedef struct func_t {
        void (*fn)(void *);
        void * data;
} func_t;
```

Figure 6.12: The `func_t` data structure.

the parallel block, it also packages the variables from the outside scope to which the block refers into a data structure, and it passes pointers to both function and data structure to `DOMP_parallel_start`. (See 6.2.4.)

The `threads` array is composed of `domp_thread_t` objects, each representing a single thread, including the master, with an extra one at the end of the array to hold data the master uses for its `copy_out` routine (6.2.4. We return to this data structure below, after completing our tour of `global_vars_t`.

The `ref_copies` field is an array of three file descriptors for the three files holding the reference copy pages, which are global for all threads. The `ref_copy_locks` field is a bit field that the signal handler uses to ensure that any page is reference-copied only once, thus also preventing data races in the process of writing reference copy pages. (See 6.2.1.)

The `xreduction_vars` array holds global information about extended reductions, inserted *before* the parallel block in which the reductions occur, and therefore not subject to any data races. Concurrent threads read this information but do not write to it. (See 6.4.)

The `scratch` field is the file descriptor of the "scratch" file, which master and pooled threads use to transfer data before thread restart using `copy_out` and `copy_in` (6.2.4).

Finally, the signal handler uses the `in_parallel` flag to determine whether to make a full reference copy for a trapped page or merely to record it. The master

```
typedef struct domp_thread_t {
        int pid;
        struct pipes_t pipes;
        struct segment_t heap;
        void * heap_mspace;
        int mem_fd;
        off_t * pages_written;
        int num_pages_written;
        int num_reduction_vars;
        struct reduction_var_t reduction_vars[DOMP_MAX_REDUCTION_VARS];
        int num_queued_frees;
        struct queued_free_t free_queue[DOMP_MAX_QUEUED_FREES];
} domp_thread_t;
```

Figure 6.13: The `domp_thread_t` data structure.

sets `in_parallel` to true in `domp_parallel_start` and clears it (sets it to false) in `domp_parallel_end`. (See 6.2.4.)

We now turn to the `domp_thread_t` structure that forms each element of the global `threads` array, whose definition is shown in Figure 6.13.

Each up-buddy uses the `pid` field in order to find the `/proc/`<*pid*>`/mem` pseudo-file representing the down-buddy's address space. Later, the master uses the `pid` field in the `atexit` clean-up code to wait for each child thread's exit. (See 6.2.3, 6.2.4.)

The `pipes` field holds file descriptors for both ends of the pipe to communicate between the thread and its up-buddy and between the thread and the master. This arrangement allows the up-buddy to find the read end of its down-buddy's pipe to wait for its "ready-to-merge" signal. It also enables the master to iterate through the `threads` array when signaling all threads either to restart or to exit. (See 6.2.4.)

The `heap` and `heap_mspace` fields enable threads to avoid trampling on malloc-related bookkeeping structures in `copy_in` and merging, respectively.

The `mem_fd` field holds the file descriptor of the named file that the up-buddy opens with the contents of the down-buddy's `/proc/`<*pid*>`/mem` pseudo-file. (See

```
typedef struct reduction_var_t {
        void * orig;  /* Used only by the master */
        int index;
        int type_index;
        union common_value_t value;
} reduction_var_t;
```

Figure 6.14: The `reduction_var_t` data structure.

6.2.4.)

The `pages_written` array holds a list of page addresses of the pages to which this thread has written—or to which any of its down-buddies have written, after it has merged those down-buddies' changes into its own data. (See 6.2.1.)

DOMP's implementation of simple reductions (as in standard OpenMP) uses the `reduction_vars` field, an array, as described in 6.3. Each element of this array is a `reduction_var_t` data structure, shown in Figure 6.14. The `orig` field holds a pointer to the "original" reduction variable. (It actually points to the field in the data structure GCC creates to pass to the special function representing the parallel block; GCC also emits code to transfer the value back from this field to the original variable.) The fields `index` and `type_index` hold the integers uniquely identifying the type-size-and-operation combination and the variable type by itself, respectively. The `value` field holds the value to be aggregated to the reduction variable from the thread that this `domp_thread_t` data structure represents. The `value` field is a union type in order to hold the value of a variable of any of the types and sizes allowed for a standard OpenMP reduction. We show this union type in Figure 6.15.

Finally, the `domp_thread_t`'s `free_queue` array holds addresses and their respective heaps for freeing at merge time. More particularly, as noted in 6.1.2, calls to `free` do not execute the usual memory freeing routines, but rather place the "order to free" onto a queue. Later, the master thread calls `free_queues` in `domp_team_end`

```
typedef union common_value_t {
        bool val_bool;
        int8_t val_int8;
        int16_t val_int16;
        int32_t val_int32;
        int64_t val_int64;
        uint8_t val_uint8;
        uint16_t val_uint16;
        uint32_t val_uint32;
        uint64_t val_uint64;
        float val_float;
        double val_double;
        struct complex_t val_complex;
        struct double_complex_t val_double_complex;
} common_value_t;
```

Figure 6.15: The `common_value_t` union type, used in the `reduction_var_t` data structure's `value` field.

for the queue belonging to each thread in the `threads` array. This technique prevents spurious data races in *dlmalloc*'s bookkeeping structures (such as bitfields) that might otherwise occur during parallel merge. It also resolves in WCD fashion what would otherwise be a genuine data race, when one thread frees a variable to which another thread writes within the same parallel block: the write always comes first, since the free is delayed until the very end of synchronized events, after merging is complete.

Having reviewed the `global_vars_t` data structure and the `domp_thread_t` structure that populates the former's `threads` array, we finally turn to the thread-local data structure `thread_record_t`, shown in Figure 6.16.

The `id` field holds the thread's ID, numbered sequentially from 0 for the master and 1 for the first child.

DOMP uses the `num_sections` and `sections_done` fields in order to support the `sections` construct deterministically, as detailed in 6.1.4 above.

```
typedef struct thread_record_t {
        int id;
        unsigned num_sections;
        unsigned sections_done;
        void * heap;
        uint64_t down_buddies;  // Bit field.
        uint64_t max_down_buddy;
        int pipe_master;
        int pipe_up_buddy;
        struct sigaction old_act;
        int reduction_var_index_0;
        int reduction_var_index_1;
        int xreduction_var_index_0;
        int xreduction_var_index_1;
        struct domp_thread_t * this_thread;
} thread_record_t;
```

Figure 6.16: The `thread_record_t` data structure.

The `heap` field points to the start of the range in the `mmap`ped file for heaps that contains this thread's heap.

DOMP uses the `down_buddies` bitfield and `max_down_buddy` in order to store the identities of this thread's down-buddies (if there are any) so that they can be computed only once per parallel block and then used efficiently to find down-buddies in the merge process.

The fields ending in "`_index_0`" or "`_index_1`" provide handy storage when an up-buddy evaluates reductions (both simple and extended) in the merge process. The merge process goes by ascending page address, and the indices represent the start and end addresses of reductions within a given page.

Finally, the current thread uses its `this_thread` pointer to get convenient access to the `threads` entry representing it.

6.6 Limitations

The current implementation of DOMP supports its core features and enables us to perform experiments, as detailed in Chapter 7. However, it also has limitations that, in particular, hamper its potential for scaling to arbitrary numbers of threads and arbitrary input data sizes.

One of the main issues is the data structures DOMP currently uses for thread and shared memory bookkeeping, described in 6.5 above. In order for such metadata to be visible to all threads—useful in particular during parallel merge—DOMP stores these data structures in mapped files, which the parent thread must create before it spawns any other threads in the pool. In order to accommodate possible changes in the number of threads in different parallel blocks of the same program, we had to set an arbitrary maximum number of threads and initialize the data structure for that number. Thus a maximum number of threads is hard coded into DOMP, and can only be changed at the cost of recompiling. Clearly, there are further trade-offs: as the global data structures increase in size, the arrays containing them lose more cache locality.

Similarly, the parent thread carves its own heap and those of all possible future child threads out of a single region of virtual memory space, a mapped file. We use Linux's copy-on-write semantics for the `MAP_PRIVATE` flag to facilitate WCD semantics for the heap. As with the global data structures, DOMP maps this region once near the start of the program so that the parent's heap will remain visible to threads in the pool in second and later parallel blocks, and so that child heaps will be visible to the parent and to each other during parallel merge. This design is somewhat rigid, since it requires that we set aside some estimated maximum space for all heaps at program start, which the program cannot alter later on.

These constraints seem difficult to avoid in a user-level implementation, and highlight the advantages that Determinator has by implementing WCD at the OS level, since the OS routinely maintains bookkeeping for all processes and can manipulate address spaces more easily than user processes can. One possible option would be to have each child thread allocate its own heap (as a normal process would do) and write its contents to a named file before exiting. In addition, it would record the mapping of pointers to heap addresses and length of allocated space for each pointer and save these data in a file as well (which could be the same file). The up-buddy would then open and read the file and, in effect, reproduce the down-buddy's heap by allocating space in its own heap and copying each of its down-buddy's allocated blocks. Threads could take a similar approach to bookkeeping.This approach seems both complex and likely to be costly, but only experiments would establish the precise trade-off between these costs and the benefit of additional flexibility.

As we mentioned in 6.1.4, DOMP currently supports loop and `sections` work sharing constructs, but it does not support the `task` construct or its associated `taskwait` construct. We must leave these for future work.

The current implementation also does not support nested parallelism. In the standard benchmarks, this turns out not to pose a problem. However, it excludes a large class of potential parallel programs, including one so simple as a parallel quicksort following the natural pattern where recursion requires nesting.

The proposed solution for global data structures and heaps, though awkward, could also support nested parallelism.

* * *

Our current implementation, despite its limitations, enabled us to conduct experiments to evaluate DOMP's performance, which we describe in the next chapter.

Chapter 7

Evaluation

Our evaluation of DOMP included experiments to measure performance and scalability, as well as experiences in converting nondeterministic OpenMP to DOMP-compatible code. Because of the costs of enforcing determinism in the working-copies model, we do not expect DOMP to be competitive to GOMP in scalability and performance for *all* parallel applications. We hope to verify, however, that *some* parallel applications can be run under DOMP with little or no overhead compared with nondeterministic execution execution under GOMP. For applications with higher overheads, DOMP may still be useful during software development and debugging, even if the application is run in a nondeterministic environment on deployment for performance reasons: this should be possible for any DOMP application provided DOMP's extended reductions are "back-ported" to conventional nondeterministic OpenMP runtimes.

7.1 Performance and Scalability

In order to evaluate the performance and scalability of DOMP, we tested it against the standard libgomp implementation on one homemade, one novel, and nine stan-

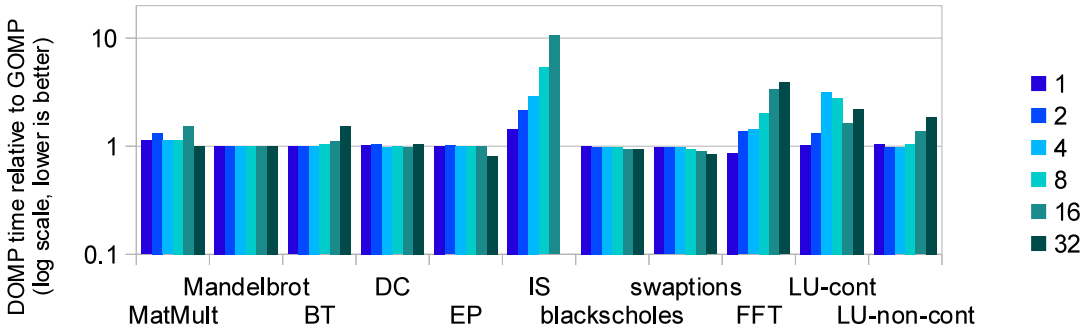


Figure 7.1: Benchmark running times, relative to standard (GOMP) times for the same benchmark and number of threads. A value of 1 means equal running times for DOMP and GOMP. IS (NPB) is a pathological outlier.

dard parallel benchmarks: **matrix multiplication** (2048×2048 matrices) (home-made); **Mandelbrot set** graphics creation (800×600 pixels, $50,000$ iterations) [48]; the PARSEC benchmarks **blackscholes** (10M input) and **swaptions** (128 swaptions, 1M trials); the NPB (3.3) benchmarks **BT** (input class "A"), **DC** ("A"), **EP** ("A"), and **IS** ("B"); and the SPLASH-2 "kernel" benchmarks FFT ($m = 18$), LU-contiguous-blocks ($n = 2048$), and LU-non-contiguous-blocks ($n = 2048$). MatMult, Mandelbrot, blackscholes, and the NPB benchmarks were already written in OpenMP; we converted the others from pthreads to OpenMP. We ran the programs with 1 thread and doubled the thread number up to 32, for IS, which had technical problems at 32 threads.

SPLASH2, NPB, and PARSEC are standard parallel benchmarks used for systems research. We tried to run a much larger range of these benchmarks, but, unfortunately, we ran into technical problems having to do with our `malloc` implementation, adapted from `dlmalloc`. We hope to be able to resolve these problems in a follow-up and have more benchmark results.

Figure 7.1 shows comparisons of DOMP's running times against those of the ref-

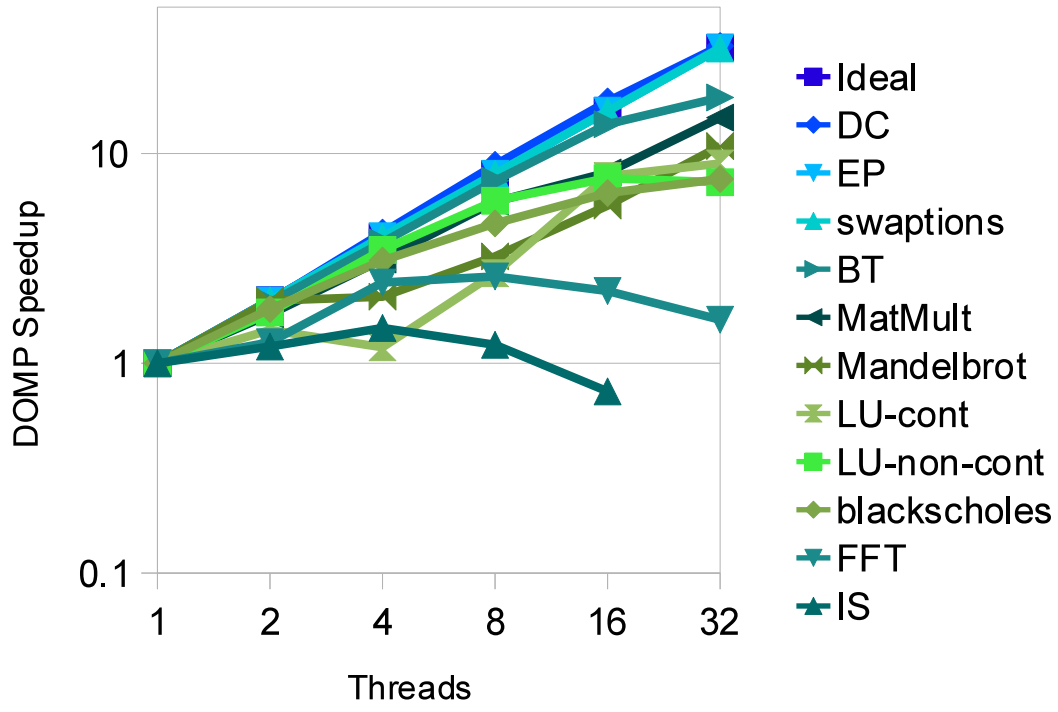


Figure 7.2: Speedups for all nine benchmarks under DOMP. Closer to the ideal curve is better. DC (NPT) appears slightly "better" than ideal because of mere noise.

erence implementation (GOMP), normalized to the latter. Six of the 11 benchmarks scale well relative to GOMP, deviating from its performance by $< 10\%$. MatMult performs less well with 2 and 16 threads, but well for 32, perhaps because of how the application splits the input matrix among threads. The three SPLASH-2 benchmarks do not scale well, increasing in running time relative to GOMP as the thread number increases, to almost 3x for 16 threads for FFT.

The most egregiously poor performer, IS, shows pathological scaling, with the 16-thread running time taking about 1.6 times that of 2 threads. Profiling with **gprof** shows that DOMP-related data copying between threads during the merge process accounts for about 56% of execution time. By contrast, in swaptions, BT, DC, and

EP, where DOMP performs well, these activities account for less than 1% of total time.

To investigate further, we instrumented the DOMP library to record the maximum number of distinct memory pages to which the master thread (thread 0) writes in any parallel block, as well as the total number of pages to which the master has written over the course of the program, running it with a single thread. As Table 7.1 shows, IS had both the highest maximum number of page writes to any parallel block and the greatest overall number of page writes. Matrix multiplication had a comparable per-block maximum, but this program has only a single parallel block, whereas IS has three. One of these latter, moreover, contains a barrier, further increasing the time spent copying and merging from thread to thread, as suggested in 4.1. This accounts for IS's high total of pages written. FFT has 7 barriers and the LU variants

Benchmark	**Max Pages**	**Total Pages**
MatMult	24578	24578
Mandelbrot	1	1
BT	4	1911
DC	2	3
EP	2	4
IS	34778	90100
blackscholes	9768	9768
swaptions	677	677
FFT	5	5
LU-cont	7	7
LU-non-cont	7	7

Table 7.1: Number of pages written when running single-threaded: maximum over all parallel blocks, and total over the whole program.

5, whereas DC, EP, and swaptions have none, perhaps helping to explain the former group's poor scalability as compared with the latter's.

7.2 Adapting Code for DOMP

DOMP served as a drop-in replacement for standard OpenMP with libgomp in 7 of the 11 benchmarks. BT and EP contain `atomic`, and DC `critical`, constructs, all of which we were able to replace with extended reductions, as described in 5.2. For better modularity, we wrote the extended reduction identity elements, combining functions, and associated wrappers in a separate file in C, with which we were more familiar than Fortran, though the reduction code could just as easily have been written in Fortran. Putting the extended reduction code in a separate file allowed us to re-use code, following the idea of a potential library of common extended reductions suggested in 5.2. Table 7.2 summarizes the extent of our changes to benchmark

	Total	DOMP Changes	Module	%
MatMult	109	0	0	0
Mandelbrot	105	0	0	0
BT	3589	16	30	1
DC	2809	3	48	2
EP	228	16	30	20
IS	634	0	0	0
blackscholes	359	0	0	0
swaptions	1780	0	0	0
FFT	1504	0	0	0
LU-cont	2484	0	0	0
LU-non-cont	1890	0	0	0

Table 7.2: Lines of code changed in adapting OpenMP programs to DOMP. **Total**: total lines of code in the original program, before modification. **DOMP Changes**: lines inserted, deleted, or replaced in any original source file. **Module**: size of external module supporting an extended reduction. %: Portion of total included in modification $((changed + total)/total)$.

source code in order to enable it to run with DOMP and extended reductions. We did not include makefiles or test-running scripts in these figures. In counting changed lines, we did not include those changes necessary to port pthreads benchmarks to

```
pragma omp parallel for
        for (i=0; i<numOptions; i++) {
            price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                         rate[i], volatility[i], otime[i],
                                         otype[i], 0);
            prices[i] = price;
        }
```

Figure 7.3: Benign data race in *blackscholes*.

OpenMP, since these changes were equally necessary for both GOMP and DOMP. DOMP-specific changes amount to 2% or less of the total lines of code, except in EP, where the original source is unusually short but requires two extended reductions.

7.3 Discovering Concurrency Bugs—Or Not

We had somewhat expected to uncover hitherto latent concurrency bugs by running DOMP on these well-known benchmarks, but found that they must have been long sifted enough to be rid of them. Early in development, we did discover a benign race in PARSEC *blackscholes*, represented in Figure 7.3.

The data race is on the variable `price`, which is defined outside of the parallel block and therefore shared. This was in an earlier version of PARSEC. We reported the data race to the PARSEC team, and they removed it in the next version by simply assigning directly to the elements of the `prices` array.

We did, however, test DOMP with artificial programs that had data races, and obtained the required error response.

Chapter 8

Conclusion and Future Work

Our analysis of synchronization in existing code bases suggests that a rigorously deterministic parallel programming model may be practical for much otherwise conventional software, using familiar platforms, languages, environments, and tools. Our encouraging experience with DOMP supports Working-Copies Determinism as a promising approach to realizing this goal.

In the short term, the DOMP project could go further in at least two directions. First, DOMP would benefit from the inclusion and implementation of pipeline and task queue constructs, as discussed in 4.3.3, as well as perhaps the OpenMP `task` construct itself. Such extensions would advance research further by solving the practical problems involved in making deterministic parallelism as accessible as possible. Secondly, changes in DOMP's design could possibly address at least some of its current limitations discussed in 6.6: a static limit on the number of allowable threads and a rigid heap mechanism that does not well accommodate the wide range of different demands that various programs place on this type of storage.

Pipelines, which appear as just another possible construct, could in turn open the way for a wide range of explorations of efficient deterministic processing, be-

cause of their kinship with dataflow languages and, underlying those, Kahn process networks. A DOMP implementation that could support a flexible range of different sorts of pipelines, extending to arbitrary execution graphs, could allow the programmer to express a program in any one of a number of different ways, all of which are deterministic—whether conceiving of the program as a distribution of tasks among workers or as a pattern of data flow or a combination of the two.

A deterministic task queue construct would undercut the programming assumption most likely to result in nondeterminism, which is the assumption that adaptive nondeterminism is necessary to maximize efficiency.

Another area of potential benefit for DOMP would be further improvements in DOMP's error output when it encounters a race condition. We currently print the address and data segment in which the race occurs and the pair of threads involved in the first race encountered. It would be far more useful to the programmer to have more clues as to the variable to which the address corresponds in the source code. We already have the structure in place easily to report the parallel block (identified by number in sequence from program start). Perhaps it would be useful to have a command line switch available to the programmer to cause DOMP to gather much richer information, such as the source code line number, using the same data as GDB uses. Thus, when the programmer encounters a race, he or she could re-run the code with the switch set, which might incur a higher overhead, but would be worthwhile in a debugging context.

The implementation of DOMP presented here as a library for Linux has the advantages of accessibility that we have noted. However, an implementation of the same API and underlying concepts to run on the Determinator operating system [13] would potentially expand the accessibility and appeal of Determinator for programmers and contribute to its further development as a full-scale operating system.

An implementation of DOMP for a strongly-typed language, perhaps specifically a language with a managed runtime virtual machine such as Java or C#, would open up attrractive possibilities. We could avoid the dilemmas inherent in the granularity of comparison during the merge operation as described in 6.6, since we could adjust the granularity to be appropriate to each type. Furthermore, such an implementation could be far more efficient than our current one, if it can restrict its comparing and merging to a list of known objects rather than to entire pages or address ranges. In principle, standard OpenMP can be implemented for such a language, as shown by JOMP, the implementation for Java [25].

Another area of future exploration is the exploitation of hardware in the present and future in order to make DOMP more efficient and scalable, for use in large many-core systems. Could some processors be set aside for synchronization and merging operations, and would we gain anything in performance? Although the merge operation must, as a whole, come *sequentially* after parallel execution (or between parallel blocks), elements of the merge process itself might be open to parallelization. For instance, processing of the reduction variables might go on at the same time as merging of pages (or objects). The possibility of custom, dedicated hardware might also be worth exploring. For instance, hardware that records which process has written to a location in memory could obviate the need for memory protections and trapping, with its attendant overhead.

These are a few possible avenues to explore. Nevertheless, the current state of the DOMP project has at least provided a reasonably practical instance of accessible deterministic parallelism, sufficient to conduct performance and development experiments. We hope that the resulting insights will benefit further research into practical and accessible parallel determinism.

Bibliography

[1] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, Feb 1982.

[2] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In Paul Hudak, editor, *Proceedings of the Haskell Workshop*, La Jolla, CA, pages 35–49, June 1995.

[3] H. Agrawal, R.A. De Millo, and E.H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, May 1991.

[4] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, Angers, France, pages 82–93, April 2003.

[6] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[7] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[8] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, October 2009.

[9] Amittai Aviram and Bryan Ford. Deterministic OpenMP for race-free parallelism. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, CA, May 2011.

[10] Amittai Aviram and Bryan Ford. A generalized reduction construct for deterministic OpenMP. In *3rd Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, London, England, March 2012.

[11] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A programming model for shared memory parallelism. In *2nd Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Newport Beach, CA, March 2011.

[12] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *ACM Cloud Computing Security Workshop (CCSW)*, Chicago, IL, October 2010.

[13] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010.

[14] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Minneapolis, MN, pages 382–400, 2000.

[15] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. Karma: scalable deterministic record-replay. In *International Conference on Supercomputing (ICS)*, Tucson, AZ, pages 359–368, 2011.

[16] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Seattle, WA, pages 168–176, 1990.

[17] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, March 2010.

[18] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, FL, October 2009.

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Toronto, ON, pages 72–81, October 2008.

[20] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, pages 207–216, 1995.

[21] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, California, March 2009.

[22] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, FL, October 2009.

[23] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent collections. *Scientific Programming*, 18(3):203–217, January 2010.

[24] Zoran Budimlić, Michael Burke, Kathleen Knobe, Ryan Newton, David Peixotto, Vivek Sarkar, and Edwin Westbrook. Deterministic reductions in an asynchronous parallel language. In *2nd Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Newport Beach, CA, March 2011.

[25] J. M. Bull and M. E. Kambites. JOMP—an OpenMP-like interface for Java. In *ACM 2000 Conference on Java Grande*, San Francisco, CA, pages 44–53, June 2000.

[26] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *25th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 691–707, October 2010.

[27] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleands, LA, February 1999.

[28] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, Nice, France, pages 10–18, 2007.

[29] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. Lreplay: a pending period based deterministic replay scheme. *SIGARCH Computer Architecture News*, 38(3):187–197, June 2010.

[30] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, Welches, OR, pages 48–59, 1998.

[31] Heming Cui, Jingyue Wu, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010.

[32] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[33] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *14th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.

[34] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *27th ACM Symposium on Principles of Distributed Computing (PODC)*, Toronto, Canada, pages 125–134, 2008.

[35] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[36] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, August 2006.

[37] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Design, Automation, and Test in Europe*, Munich, Germany, March 2008.

[38] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[39] A. A. Faustini. An operational semantics for pure dataflow. In *Ninth International Colloquium on Automata, Languages and Programming*, Aarhus, Denmark, pages 212–224, July 1982.

[40] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: the design of a parallel functional language implementation on workstations. In *1st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, PA, pages 169–179, May 1996.

[41] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20:537–576, November 2010.

[42] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, QC, Canada, pages 212–223, 1998.

[43] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*, Seattle, WA, pages 15–26, May 1990.

[44] Anwar Ghuloum, Amanda Sharp, Noah Clemons, Stefanus Du Toit, Rama Malladi, Mukesh Gangadhar, Michael McCool, and Hans Pabst. Array Building Blocks: A flexible parallel programming model for multicore and many-core architectures. `http://http://www.drdobbs.com/parallel/array-building-blocks-a-flexible-paralle/227300084`, September 2010.

[45] Gnu. Gnu libgomp. `http://gcc.gnu.org/onlinedocs/libgomp`.

[46] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[47] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[48] Brian "Beej Jorgenson" Hall. Command-line mandelbrot set generator with openmp support. https://github.com/beej71/goatbrot, 2011.

[49] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[50] Kevin Hammond. Parallel functional programming: An introduction. In *International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, Hagenberg/Linz, Austria, September 1994. World Scientific.

[51] C. L. Hankin and H. W. Glaser. The data flow programming language CAJOLE – an informal introduction. *ACM SIGPLAN Notices*, 16(7):35–44, July 1981.

[52] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[53] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *22nd Symposium on Principles of Distributed Computing (PODC)*, Boston, Massachusetts, pages 92–101, 2003.

[54] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th International Symposium on Computer Architecture*, San Diego, CA, pages 289–300, May 1993.

[55] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *20th International Symposium on Computer Architecture (ISCA)*, San Diego, California, pages 289–300, 1993.

[56] P. Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[57] National Instruments. LabVIEW. `http://www.ni.com/labview`.

[58] Intel Corporation. Intel® Threading Building Blocks reference manual, January 2012.

[59] Kenneth Iverson. A programming language. In *AIEE-IRE Spring Joint Computer Conference*, San Francisco, CA, pages 345–351, May 1962.

[60] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[61] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[62] Mark Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Research Report YALEU/DCS/RR-982, Yale University, New Haven, Connecticut, Aug 1993. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.3273&rep=rep1&type=pdf`.

[63] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, Amsterdam, Netherlands, 1974. North-Holland.

[64] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, pages 1–15, April 2005.

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

[66] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

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

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

[69] Charles E. Leiserson and Aske Plaat. Programming parallel applications in Cilk. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.3848&rep=rep1&type=pdf`, 1997.

[70] Tongping Liu, Charlie Curtsinger, and Emery Berger. Dthreads: efficient deterministic multithreading. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 327–336, October 2011.

[71] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics.

In *13th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, pages 329–339, March 2008.

[72] The MathWorks. Simulink. `http://www.mathworks.com/products/simulink`.

[73] David Mosberger. Memory consistency models. *SIGOPS Operating Systems Review*, 27(1):18–26, January 1993.

[74] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, and Gerard Basler. Finding and reproducing Heisenbugs in concurrent programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, California, pages 267–280, 2008.

[75] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[76] Rishiyur S. Nikhil. An overview of the parallel language Id (a foundation for ph, a parallel dialect of Haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

[77] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.

[78] OpenMP Architecture Review Board. OpenMP application program interface version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[79] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *Workshop on Parallel and Distributed Debugging (PADD)*, Madison, Wisconsin, pages 124–129, 1988.

[80] D. Patterson. The trouble with multi-core. *Spectrum, IEEE*, 47(7):28 – 32, 53, July 2010.

[81] Simon L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[82] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *IEEE 13th International Symposium on High Performance Computer Architecture, 2007*, pages 13 – 24, February 2007.

[83] John Reppy. Concurrent ML: Design, application and semantics. In Peter Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 165–198. Springer, 1993.

[84] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[85] Paul Roe. *Parallel Programming using Functional Languages*. PhD thesis, University of Glasgow, February 1991. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.7763&rep=rep1&type=pdf`.

[86] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Jose, CA, pages 161–172, New York, NY, 2007. ACM.

[87] 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*, 15(4):391–411, 1997.

[88] Michael L. Scott and Li Lu. Toward a formal semantic framework for deterministic parallel programming. In *2nd Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Newport Beach, CA, March 2011.

[89] Ali Sezgin and Ganesh Gopalakrishnan. On the definition of sequential consistency. *Information Processing Letters*, 96(6):193 – 196, 2005.

[90] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[91] David Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 2005.

[92] Sudarshan Srinivasan, Srikanth Kandula, Christopher Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[93] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, June 1997.

[94] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *6th Conference on Embedded Software*, Seoul, Korea, pages 142–151, October 2006.

[95] P. W. Trinder, K. Hammond, H.-W. Loidl, and Simon L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[96] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4&5):469–510, 2002.

[97] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: parallelizing sequential logging and replay. *ACM SIGPLAN Notices*, 47(4):15–26, March 2011.

[98] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[99] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38 –59, Winter 1994.

[100] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[101] WeiWei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010.

[102] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *30th International*

Symposium on Computer Architecture (ISCA), San Diego, California, pages 122–135, June 2003.