Dynamic Determinism Checking for Structured Parallelism

Edwin Westbrook[1] Raghavan Raman[2] Jisheng Zhao[3] Zoran Budimlić[3] Vivek Sarkar[3]

Kestrel Institute[1], Oracle Labs[2], Rice University[3]

westbrook@kestrel.edu, raghavan.raman@oracle.com, {jz10,zoran,vsarkar}@rice.edu

Abstract

Determinism is a powerful property, making it much easier to design, implement, and debug parallel programs when it can be guaranteed. There has been much work on guaranteeing determinism of programs, both through dynamic and static approaches. The dynamic approaches tend to have high overheads, while the static approaches can be difficult for non-experts to understand and can require high programmer effort in the form of typing annotations and code restructuring to fit the static type system. In this work, we present a compiler and runtime system that ensures determinism of application code, assuming that libraries satisfy their determinism specifications. In our system, the library writer includes determinism specifications in the form of annotations on the methods in the library. When these libraries are used by application programmers, our compiler will insert dynamic checks where necessary to ensure determinism at runtime. We demonstrate that our inserted dynamic checks lead to an average (geometric mean) slowdown of only 1.26× on 16 cores for our benchmark, thereby establishing the practicality of our approach.

1. Introduction

Writing parallel programs is hard because of the potential for *non-determinism*. Although non-determinism can be useful to concurrency experts, because it can enable writing of high-performance libraries with low amounts of synchronization, it is very difficult for mainstream application programmers to understand: even defining one of the most common forms of non-determinism, *data races*, requires an understanding of highly complex memory models [7, 16]. As a result, there has been a growing interest in parallel programming languages, type systems, compilers and runtime systems that ensure determinism for mainstream programmers [1, 2, 4–6, 17, 24].

At a high level, the problem of ensuring deterministic behavior of a parallel program[1] can be divided into two sub-problems:

1. Identify and verify key operations that commute with each other and/or themselves.
2. Ensure that only commuting operations are executed in parallel.

These two sub-problems correspond to two sorts of program code. The first applies to high-performance library code written by parallelism experts. These parallelism experts are likely to understand the various approaches [10, 12, 21] that have been put forth to solve this problem, since they are in the same domain of knowledge and use ideas and concepts already familiar to parallelism experts. Further, although these approaches may require a lot of verification effort, high-performance libraries do not change often (precisely because they are difficult to get right), so the cost of library verification can be amortized over multiple applications.

The second sub-problem applies to general application code, which represents the majority of the code written by "everybody else". There are also a number of approaches that apply to this problem, both dynamic [1, 2, 17]) and static [4, 5, 14]. Dynamic approaches, however, often yield a large overhead, which negates the benefits of parallel programming. Static approaches have no overhead, since all checking is done statically, but they can be difficult to use by "everybody else", who may not be familiar with the complex notions involved in these approaches. Further, they often have high annotation overheads, and generally require programs to be structured in a particular way, to use particular parallel patterns that are supported by the approach.

In this paper, we introduce a new approach to the problem of ensuring that only commuting operations are executed in parallel. In our system, library writers (typically, parallelism experts) write libraries with additional annotations that specify how the libraries can be used in parallel, while guaranteeing determinism. Specifically, libraries in our system include annotations on their methods that specify if the methods commute with each other and/or themselves. When these libraries are used in the application code, our compiler inserts dynamic checks where necessary to ensure that only commuting operations execute in parallel, thereby guaranteeing determinism. Programmers can use our libraries just as they would use any other Java or C++ library. Our approach also has low overhead, with a slowdown of only 1.26× on our benchmarks.

In more detail, our approach has been implemented as a language called Habanero Java with determinism (HJd), an extension of the Habanero Java (HJ) language [9] which itself is a task-parallel extension of Java. In addition to HJ, HJd includes *commutativity sets*, which are parallel library annotations that specify which operations commute. These annotations provided by the library implementer are trusted by HJd, and verifying them is beyond the scope of this paper. The HJd compiler translates these annotations into calls into the runtime system. When these library methods are called from the application code, our runtime system performs dynamic checks to ensure only commuting methods are executed in parallel. The dynamic checks are implemented using the Dynamic Program Structure Tree (DPST) data structure introduced by Raman et al. [19]. This structure captures may-happen-in-parallel information in a deterministic and efficient manner, and is the core of one of the fastest existing race-detection algorithms. Our approach guarantees determinism only up to the first failure. If there are no failures, our approach guarantees determinism.

The remainder of this paper is organized as follows. Section 2 introduces commutativity sets. Section 3 explains how commutativity sets are implemented in the form of permission regions in HJd. Section 4 describes the DPST structure and how it is used in HJd to perform dynamic checks. Section 5 evaluates a set of deterministic benchmarks and measures the performance impact of dynamic checks. Finally, Sections 6 and 7 give related work and conclude.

[1] We are concerned only with explicitly parallel programming in this paper; although the problem can be addressed somewhat by automatic parallelization, such approaches have major limitations that are beyond the scope of this paper.

```
1  class CountFactors {
2    int countFactors (int n) {
3      AtomicInteger cnt = new AtomicInteger();
4      finish {
5        for (int i = 2; i < n; ++i)
6          async {
7            if (n % i == 0) cnt.increment();
8          }}
9      return cnt.get ();
10   }
11 }
12
13 @ClassCommSets{"read","modify"}
14 final class AtomicInteger {
15   @CommSets{"read"} int get () { ... }
16   @CommSets{"modify"} void increment()
17                             { ... }
18   @CommSets{"modify"} void decrement()
19                             { ... }
20   @CommSets{"read","modify"} int initValue()
21                             { ... }
22   int incrementAndGet () { ... }
23 }
```

Figure 1. A Deterministic Counting Program

2. Determinism and Commutativity Sets

Following the seminal work of Steele [20], HJd ensures determinism by ensuring that only operations which commute with each other can run in parallel. Two operations *commute* iff a program that calls them cannot observe whether one completes before the other or if they ran in parallel. Ensuring that only commuting operations occur in parallel guarantees determinism because, no matter in what order the operations actually complete, the end results will be the same. As a simple example, two reads of an object field always commute with each other, since neither read can influence the other. Field writes, however, do not in general commute with either field writes or field reads. The Java Memory Model (JMM) [16] defines writes that occur in parallel with reads to be *data races*, which can lead to complex nondeterministic behaviors.

An example of a deterministic program calling commuting operations in parallel is given in Figure 1. The top half of the figure (before line 12) defines the method `countFactors()` that counts the number of factors (prime or otherwise) of the input `n` which are greater than 1 and less than `n`. This is done with a **for** loop that spawns a new task[2] for each index `i` in this range, using HJ's **async** construct. The loop occurs inside an instance of HJ's **finish** construct, which waits for all child tasks to complete before proceeding. The count of factors of `n` is maintained by incrementing the `AtomicInteger` object `cnt`. After all child tasks complete, the total value of `cnt` is returned. Note that `AtomicInteger` here is not the same class as in the Java standard library, but could be implemented as a wrapper for that class, containing the annotations discussed below to integrate it into the HJd system.

The reason that `countFactors()` is deterministic is that the only operations which run in parallel — other than primitive operations such as integer modulus and comparison with 0, which are obviously commutative — are calls to the `increment()` method of the `AtomicInteger` object `cnt`. If this method is implemented correctly, then it should commute with itself. Of course, verifying such concurrent properties is a difficult and complex task, and beyond the scope of this paper.

[2] We use the term "task" instead of "thread" to distinguish potential parallelism from OS threads that may be used to implement this parallelism.

Instead, HJd considers the `AtomicInteger` class to be a *concurrent library class* that has been implemented and verified independently. Note that any approach can be used for this verification. HJd views objects and methods of the concurrent library classes as primitives, trusting that they are correct. Given this correctness, HJd can then guarantee that programs built on top of such library classes are deterministic. All that is needed is for the library writer to give a *commutativity specification* of which methods in the library commute.

An example of a commutativity specification for the `AtomicInteger` class is given in the bottom half of Figure 1, starting at line 13. Commutativity specifications are given as Java annotations. The first of these is the `@ClassCommSets` annotation on a concurrent library class, which gives a list of names of commutativity sets for the class. A *commutativity set* is a set of methods in the class that all commute with each other, when called on the same object. Each method of a concurrent library class can then be annotated with the `@CommSets` annotation, specifying which commutativity sets it belongs to.

The `AtomicInteger` class has two sets, called `"read"` and `"modify"`. `"read"` set contains the `get()` method and `"modify"` contains the `increment()` and `decrement()` methods. Intuitively, this means that an `AtomicInteger` can be read by multiple tasks in parallel, or it can be incremented and/or decremented in parallel, but, for example, an increment cannot happen in parallel with a read since the former could influence the result of the latter. The `initValue()` method, which returns the integer used to initialize the `AtomicInteger`, commutes with reads, increments, and decrements, so it belongs to both the `"read"` and `"modify"` commutativity sets. The `incrementAndGet()` method, in contrast, does not commute with itself or any of the other methods, and thus does not have a `@CommSets` annotation.

Two more examples of commutativity specifications are given in Figure 2, both of which define atomic blocking FIFO queues. The first, `AtomicQueue`, contains two methods, `add()` and `remove()`, for inserting into and removing from the queue. These methods commute with each other because, intuitively, objects will get removed in the same order no matter how inserts and removes are interleaved with each other, since `remove()` is blocking. Thus the methods are annotated as belonging to the same commutativity set, `"modify"`. The `add()` method, however, does not commute with itself, since swapping the order of two calls to `add()` changes the order of elements in the queue. Similarly, `remove()` does not commute with itself either. Thus, both of these are annotated with `@Irreflexive`, meaning that the "commutes-with" relation is not reflexive for these methods.

Because `add()` is marked as `@Irreflexive`, the `AtomicQueue` class cannot accommodate worklist-based algorithms, where multiple producers insert "units of work" onto the same worklist in parallel. Worklist-based algorithms, however, are (usually) deterministic, because they do not depend on the order in which the units of work are added: the same, deterministic consumer procedure is run on each unit of work added, irrespective of the order in which they are processed. To capture this common parallel pattern, Figure 2 additionally defines the `AtomicWorklist` class. This class omits the explicit `remove()` method; instead, the caller must supply a single `Handler` object that will process each object added to the worklist, possibly in parallel. HJd will ensure that the `handle` method of any user-supplied `Handler` object can be called in parallel. In turn, the `AtomicWorklist` class can then guarantee that calls to `add()` commute with each other. Setting the handler, however, does not commute with any other operations.

As a special case, past work on ensuring race-freedom [22, 23] can be captured in HJd by viewing field reads and writes for "normal objects" (i.e., objects that do not belong to concurrent

```
@ClassCommSets{"modify"}
final class AtomicQueue {
  @CommSets{"modify"} @Irreflexive
  void add (Object o) { ... }

  @CommSets{"modify"} @Irreflexive
  Object remove () { ... }
}

@ClassCommSets{"modify"}
final class AtomicWorklist {
  interface Handler () {
      void handle (Object o);  }

  @CommSets{"modify"}
  void add (Object o) { ... }

  void setHandler (Handler h) { ... }
}
```

Figure 2. Commutativity Specifications for Two Atomic Queues

library classes) as methods with commutativity set specifications. Intuitively, field reads commute with each other, while field writes do not commute with reads or with other writes. To model this commutativity behavior, all user objects in HJd implicitly have one commutativity set, `"fieldRead"`, that contains all field reads. Field writes do not belong to any commutativity sets, in a manner similar to the `incrementAndGet()` method of Figure 1.

Note that direct field accesses to commutative library objects are disallowed for user-level code, as such fields are assumed to be part of the internals of the commutative library objects. Access to such fields can easily be provided by the library itself, however, using getter and setter methods, which can additionally be annotated with commutativity sets as necessary.

3. Language-Based Determinism-Checking

To check that only commutative methods are called in parallel, HJd extends a construct called a *permission region* introduced in previous work [22, 23]. Permission regions have the form

```
permit method(x) { ... }
```

where `x` is a variable and *method* is any method that can be called on `x`. This construct communicates to the runtime system of HJd that the code inside the curly braces should only be run if the current task is *permitted* to run method *method* on `x`; i.e., no other tasks could possibly call any methods on `x` that do not commute with *method*. To determine whether the current task can safely be given this permission, the runtime system performs a dynamic check, as described in Section 4. If the check succeeds, then code inside the curly braces is executed. Otherwise, the check fails, and an exception is thrown, indicating that some other permission region could potentially happen in parallel which allows methods to be called on `x` that do not commute with *method*.

Permission regions are not method-specific, but instead take into account the commutativity sets of the specified method. Specifically, any method belonging to at least all of the commutativity sets that *method* belongs to can also be called in a permission region for *method*. For example, if `x` is an `AtomicInteger`, then the permission region **permit** `increment(x)` allows `increment()`, `decrement()`, and `initValue()` to be called on `x`, since the set of commutativity sets for each of these methods is a superset of those for `increment()`. For a method with no commutativity set annotations, such as `incrementAndGet()`, all methods are allowed, since such a method can only be called if no other task accesses the given object at all.

```
class CountFactors {
  void countFactors (AtomicInteger cnt,
      int n) {
    finish { for (int i = 2; i < n; ++i)
      async { if (n % i == 0)
        permit increment(cnt) {
          cnt.increment(); } } }
  }

  void foo (AtomicInteger cnt) {
    permit set(cnt) { cnt.set (0); }
    countFactors (cnt, K);
    int res; permit get(cnt) {
      res = cnt.get (); } }
}
```

Figure 3. Example of HJd Permission Region Insertion

Permission regions are inserted by the HJd compiler where necessary, in order to ensure that a task always has permission to call a concurrent library method whenever such a call takes place. The goal of the insertion algorithm is to insert permissions regions in a manner that minimizes spurious failures, i.e., failures that have to do with where the regions are placed rather than with non-determinism in the underlying program. Note that users can fix such failures manually, by explicitly adding permissions to their code. This is a central benefit of making permission regions a construct in the language, as opposed to approaches like dynamic race detection, where checks are inserted into the code by the compiler with no chance for user involvement. A good insertion algorithm, however, will not require any such manual fixes, thereby increasing programmer productivity.

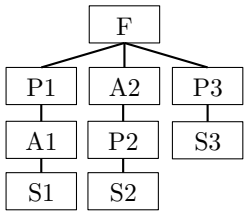
In previous work [22, 23], we showed that for preventing data-races, the best approach to inserting permissions regions is to add permission regions around variable scopes. This essentially creates regions of code that are as large as possible during which the current task holds permissions to the given object. This approach captures the intuition that programmers do not expect the contents of variables to be modified while they are in scope. This approach was experimentally validated on HJ benchmarks written without permissions in mind: for 11 HJ benchmarks totaling over 9,000 lines of code, only one modification was needed to prevent false positives.

For concurrent library objects, in contrast, our approach is to insert regions that are as small as possible, i.e., around each method call that requires concurrent library object permissions. As an example, consider the code in Figure 3. This is a slightly modified version of the `countFactors()` method of Figure 1 that takes `cnt` as an argument, along with a method `foo()` that sets `cnt` to 0, calls `countFactors()` with some `K`, and then reads the result from `cnt`. The permission regions in this code show how the HJd insertion algorithm works: each call to `set()`, `get()`, and `increment()` is directly inside a permission region to acquire permissions for the call. If, for example, either the `set()` or `get()` regions for the `cnt` object in `foo()` were any bigger, however, they would require the caller of `foo()` to hold permissions to call `get()` or `set()`, respectively, while `countFactors()` executed, which would not allow any of the child tasks spawned by this method to get permission to call `increment()`.

4. Dynamically Verifying Determinism

The Dynamic Program Structure Tree (DPST), introduced by Raman et al. [19], is a simple yet powerful run-time data-structure that allows the dynamic parallelism of a structured parallel program to be efficiently recorded and used for determining happens-before re-

```
1 finish {
2   permit increment(x) {
3       async { S1; }}
4   async {
5     permit get(x) { S2; }
6   }
7   permit increment(x) {
8       S3;
9   }}
```



(a) Example Program (b) Resulting DPST

Figure 4. An Example Program and its DPST

lation. Intuitively, the DPST of a program execution is an ordered rooted tree containing an internal node for each **async** and **finish** instance and a leaf node for each statement instance[3] in that execution. HJd implements permission regions by performing dynamic checks for happens-before on a modified version of the DPST that contains nodes for every permission region in the program in addition to the nodes for async, finish, and statement instances.

An interesting and powerful insight is that the DPST of a deterministic parallel program is also deterministic, meaning that its form for a given input is independent of the order in which tasks are executed. For HJd, the determinism of DPST means that the dynamic checks used to implement permission regions capture *any potential* non-determinism regardless of the actual task schedule of a particular run. In addition, the structure of the DPST ensures that both construction and queries of the DPST can happen without synchronization, leading to a very efficient implementation of dynamic permission checks.

We explain the DPST, as used in HJd, with the example of Figure 4. Figure 4(a) gives a small code fragment that uses variable `x`, assumed to be an `AtomicInteger` as defined in Figure 1. The outer-most construct is a **finish**, which contains, in sequence: a permission region for `increment()` containing **async**, which itself contains some statement `S1`; an **async** which contains a permission region for `get()` that performs `S2`; and, finally, a second permission region for `increment()` containing `S3`.

The DPST for this example is shown in Figure 4(b). DPSTs in HJd contain nodes for each **async**, **finish**, and permission region. The outer-most **finish** of the example becomes the top-most node of the tree, which is labeled F. The first permission region, which becomes the left-most child of F, is labeled P1, and has the first **async** as a child, labeled A1, with child S1. The second **async** becomes the second child of F, labeled A2, whose child is the second permission region, labeled P2, which has child S2. Finally, the last permission region becomes the third child of F, labeled P3, which has child S3.

Intuitively, node P2 potentially conflicts with both P1 and P3, since P2 acquires permission to call `get()` while P1 and P3 acquire permission to call `increment()`. To check whether these potential conflicts could lead to nondeterminism, the system must determine whether P2 could happen in parallel with either P1 or P3, i.e., whether there do not exist happens-before edges between P2 and each of P1 and P3.

We extend the may-happen-in-parallel algorithm from [19] to test whether one of two nodes, N and N', happens-before the other using the DPST. The first step is to determine the least common ancestor (LCA) A of N and N' on the DPST. If A is the node N itself, then N' occurs inside the scope of N, which leads to a conflict iff N and N' are in different tasks. This can be determined by examining the path from N to N': the two are in different tasks iff there is an **async** node along this path. The case for A being the node N' is similar. If A is neither of the two nodes N and N', the path from A to the left-most of N or N' is examined, and N happens in parallel to N' iff the first non-permission-region node on this path is an **async**. Otherwise, the left-most of N and N' happens before the right-most.

[3] DPST is defined with step instances as leaves in [19], but the same concepts apply to statement instances as well.

```
1 DoubleTaskNode get$$node;
2 DoubleTaskNode increment$$node;
3 DoubleTaskNode initValue$$node;
4 TaskNode private$$node;
5 boolean exclusive$$held;
6 TaskNode exclusive$$node;
7 AtomicLong start$$version, end$$version;
```

Figure 5. Additional DPST-Related Fields for the `AtomicInteger` Class of Figure 1

In order to implement a quick lookup of the permission regions that could potentially conflict with a given permission region, each object stores the permission regions that have succeeded in obtaining permissions to it. Of course, it would be impractical for an object to store every permission region that succeeded in acquiring a permission on it. Instead, objects only store a set of one or two permission nodes that *cover* the different permissions which can be acquired. The definition of covering, along with an adaptation of the theorem of Raman et al. [19] that a covering is sufficient to perform happens-before checks, are given as follows:

DEFINITION 1 (DPST Node Covering). *Let S be a set of DPST nodes (assumed to be in the same DPST). The singleton set $\{n\}$ of DPST node n is said to* cover *S iff each node in S, other than n, happens before n. The two-element set $\{n_1, n_2\}$* covers *S iff, for each $n \in S$, either n happens before one of n_1 and n_2, or $\mathbf{LCA}(n, n_i)$ is a descendant of or equal to $\mathbf{LCA}(n_1, n_2)$ for each $i \in \{1, 2\}$.*

THEOREM 1. *Let S and S' be any sets of DPST nodes such that S' covers S, and let n be any node not in S. If n does not happen before any node of S, then for all nodes $m \in S$, m happens before n iff all nodes $m' \in S'$ happen before n.*

Concurrent library classes are processed by the HJd compiler in order to add fields to track successful permission regions using coverings. The default root object, `hj.lang.Object`, is processed in a similar manner to track permission regions for field reads and writes by treating these as methods, as discussed in Section 2. To do this processing, the compiler first groups together all methods with the same commutativity sets that are not marked as `@Irreflexive`. For example, both `increment()` and `decrement()` in `AtomicInteger` belong to the same group, but the `add()` and `remove()` of the `AtomicQueue` class of Figure 2 do not. For each such group, the compiler adds a `DoubleTaskNode` field to the class, which maintains a covering of one or two DPST nodes for the the successful permission regions for any of the methods in the group. The compiler then adds a `TaskNode` field for each `@Irreflexive` method to maintain a covering of at most one DPST node; intuitively, only a single DPST node is needed because permission regions for an `@Irreflexive` method cannot happen in parallel. This process is illustrated in Figure 5, which shows the three compiler-inserted `DoubleTaskNode` fields for `AtomicInteger`. Each is named after the first method in its group, and includes `$$` to mark the fields as machine-generated and not user-accessible.

The compiler also adds five additional fields to every class. The first, `private$$node`, is a `TaskNode` field to track permission regions for methods not in any commutativity sets, such as `getAndIncrement()`. Such methods intuitively represent "private"

Name	Suite	Library Classes
health	**BOTS**	priority queue
floorplan	**BOTS**	min-accumulator, atomic integer
nqueens	**BOTS**	atomic integer
breadthFirstSearch	**PBBS**	worklist, min-accumulator
dictionary	**PBBS**	deterministic hash table
PriorityQueue	**HJ**	priority queue

Table 1. Benchmarks

permissions to use the object in any task-local manner. The next two, `exclusive$$held` and `exclusive$$node`, are a `TaskNode` and a **boolean** flag to track exclusive acquires and releases: if the flag is **true**, then exclusive permissions are currently being held on the object, and `exclusive$$held` stores the node where exclusive permissions were acquired; otherwise, the stored node indicates where exclusive permissions were released.

The final two fields added by the compiler, `start$$version` and `end$$version`, are `AtomicLong` objects used to implement Leslie Lamport's solution to the reader-writer problem [15]. This allows the DPST-related fields of an object to be examined and updated without using locks, thereby avoiding any potential blocking. See Lamport's paper, or Raman's thesis [18] for more details.

5. Results

We evaluated the performance impact of dynamic checks implemented using Dynamic Program Structure Tree in HJd. The benchmarks we used are listed in Table 1, including three OpenMP 3.0 BOTS benchmarks [11], two PBBS benchmarks [3], and one HJ priority queue micro-benchmark. All but the last were not originally written in Java; our versions are our interpretations of how to port these benchmarks to HJ. The main goal was to cover a number of common parallel library classes, including: the `AtomicInteger` class of Figure 1; a minimizing accumulator, that tracks the minimum value sent to it in parallel; `AtomicWorklist` of Figure 2; a priority queue version of `AtomicQueue` from Figure 2, where `add()` commutes with itself because the list is always sorted; and a deterministic hash-table.

We implemented HJd as an extension to the HJ compiler [9]. We ran our benchmarks on a 16-core (4 socket, 4 core per socket) Intel Xeon 2.4GHz system with 30GB of memory, running Red Hat Linux (RHEL 5) and Sun JDK 1.6 (64-bit). The JVM heap size was set to 6GB. The results obtained are given in Figure 6 as slowdowns relative to running the benchmark with no dynamic checks at all. For each benchmark, we also evaluated the performance impact of just the dynamic checks needed for the parallel library calls, and omitted the checks that would be inserted by our past work [23] to ensure race-freedom, e.g., that no user-defined object was written in parallel to being read. This was done to measure the performance overhead for parallel library classes, and to evaluate how well our approach would work as, for example, a debugging tool that just checks whether parallel library classes are called correctly. The results that check just the library classes are marked "partial" in Figure 6, while the full checks are marked as "full". The versions of the benchmarks with dynamic checks show similar scalability as the uninstrumented versions, with an average (geometric mean) overhead of 1.26× for the "full" case and 1.15× for the "partial" case on 16 cores. This is much lower overhead than even the fastest modern parallel dynamic race detectors, which generally exhibit slowdowns of at least 2-4× on 16 cores [19].

6. Related Work

There has been much work on dynamic checking of determinism (e.g., [1, 2, 17]). Kendo [17] provides a software-only system that provides deterministic multithreading of parallel applications. It enforces a deterministic interleaving of local acquisitions. To use this system, user need to convert parallel program to use Kendo APIs that perform runtime verification. Grace [2] also introduced a software-only runtime system that eliminates concurrency errors for fork-join based parallelism. It employs a novel virtual memory based software transactional memory to impose a sequential commit protocol, thus reduces user effort for converting original programs. In [1], Bergan et.al developed a compiler and runtime infrastructure that ensures determinism but resorts to serialization rarely, for handling inter-thread communication and synchronizations. It relies on compiler transformation to perform code instruction and redundancy optimizations.

There has also been a number of purely static approaches to ensuring determinism in the literature. Deterministic Parallel Java (DPJ) [4, 5] uses an effect type system to statically compute the memory accesses of a piece of code. Two pieces of code can run in parallel in DPJ only if their effects are guaranteed not to interfere with each other, i.e., neither can write to memory that the other might access. To summarize effects in types, DPJ uses an abstraction called *memory regions*, which give a logical tree-based structure to memory that mirrors tree-structured object graphs, allowing parallel tree traversals. Further, DPJ also contains a `commutative` annotation, that allows programmers to specify functions, much like our library methods, that are self-commutative. Unfortunately, tree-based memory regions make it difficult to parallelize access to object graphs that are not a tree, nor can it support objects moving around in an object graph, i.e., if the object graph changes dynamically. HJd, in contrast, can handle very complex parallel patterns by falling back on dynamic checks. It is unclear if effect-based type systems could be extended to support dynamic checks in this way.

There has also been much work on permission type systems to ensure the weaker guarantee of race-freedom. Permissions are closely related to linear type systems, which ensure that resources are not duplicated or deleted when doing so is disallowed. A number of systems for avoiding races have been based on linear types, since only one task can have permission on a linear pointer at a time. Haller and Odersky [13] describe one such system, Scala capabilities. A major breakthrough was Boyland's work on fractional permissions [8], which showed how a linear read/write permission could be split into fractional read permissions.

HJd seems to be the first system to use permissions to ensure determinism, rather than just race-freedom. The main difference from previous permissions-based approaches is the idea of having different "ways" to split an exclusive permission, e.g., into either `increment()` or `get()` permissions, but not both. Our previous work [23] showed that this property is also key for gradual typing with permissions.

7. Conclusion

In this paper, we have introduced HJd, a compiler and runtime system that guarantees determinism. Rather than focus on determinism of high-performance parallel libraries, as much of past work has done, HJd instead focuses on ensuring that application code only calls such libraries in a manner that meets their concurrency specifications. These specifications are given with a new concept called *commutativity sets*, that capture which methods in a library class commute. HJd performs dynamic checks based on these commutativity sets annotations to guarantee that methods which do not commute are not called in parallel. In order to implement these dynamic checks efficiently, HJd extends the Dynamic Program Structure Tree (DPST) of Raman et al. [19]. The practicality of HJd has been validated with a suite of benchmarks, showing that not only can this approach handle a number of useful parallel library classes,

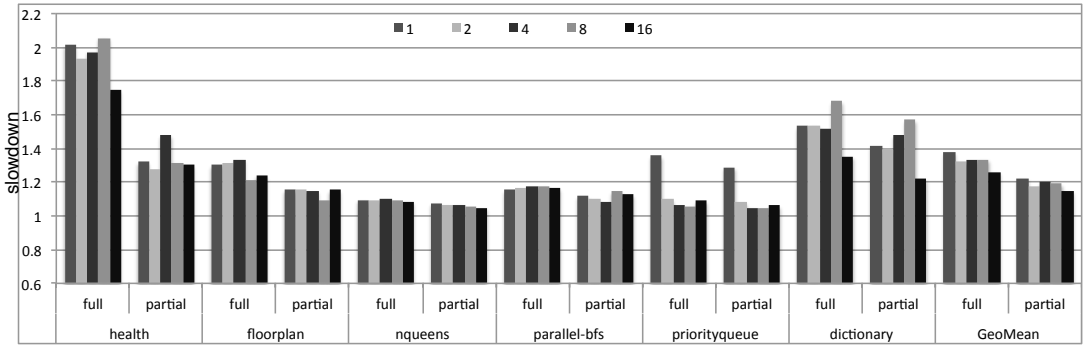


Figure 6. Benchmark slowdowns relative to running with no dynamic checks for 1, 2, 4, 8, and 16 cores

but also that, even with no user-supplied permission annotations, the overhead of dynamic permission checks is only around 1.26×.

References

[1] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[2] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *Proceedings of OOPSLA '09*, 2009.

[3] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPOPP'12*, pages 181–192, 2012.

[4] Robert L. Bocchino et al. A type and effect system for deterministic parallel java. In *OOPSLA'09*, 2009.

[5] Robert L. Bocchino et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL'11*, 2011.

[6] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism (HotPar)*, 2009.

[7] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *PLDI*, 2008.

[8] John Boyland. Checking interference with fractional permissions. In *SAS '03*, 2003.

[9] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old X10. In *PPPJ*, 2011.

[10] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proceedings of the 18th international conference on Computer Aided Verification (CAV)*, 2006.

[11] Alejandro Duran et al. Barcelona OpenMP Tasks Suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP'09*, 2009.

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, 2009.

[13] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP '10*, 2010.

[14] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic parallelism via liquid effects. In *PLDI'12*, 2012.

[15] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11), 1977.

[16] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05*, 2005.

[17] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3), 2009.

[18] Raghavan Raman. *Dynamic Data Race Detection for Structured Parallelism*. PhD thesis, Rice University, 2012.

[19] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI*, 2012.

[20] Guy L. Steele, Jr. Making asynchronous parallelism safe for the world. In *POPL*, 1990.

[21] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2009.

[22] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Permission regions for race-free parallelism. In *RV'11*, 2011.

[23] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Practical permissions for race-free parallelism. In *ECOOP*, 2012.

[24] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News*, 37(3), 2009.