Nondeterminator-3: A Provably Good Data-Race Detector That Runs in Parallel

by

Tushara C. Karunaratna

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
August 16, 2005

Certified by .... [illegible] ....
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ...... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Nondeterminator-3:

A Provably Good Data-Race Detector That Runs in Parallel

by

Tushara C. Karunaratna

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the implementation of a provably good data-race detector, called the Nondeterminator-3, which runs efficiently in parallel. A ***data race*** occurs in a multithreaded program when two logically parallel threads access the same location while holding no common locks and at least one of the accesses is a write. The Nondeterminator-3 checks for data races in programs coded in Cilk [3,10], a shared-memory multithreaded programming language.

A key capability of data-race detectors is in determining the series-parallel (SP) relationship between two threads. The Nondeterminator-3 is based on a provably good parallel SP-maintenance algorithm known as SP-hybrid [2]. For a program with n threads, T_1 work, and critical-path length T_∞, the SP-hybrid algorithm runs in $O((T_1/P + PT_\infty)\lg n)$ expected time when executed on P processors.

A data-race detector must also maintain an access-history, which consists of, for each shared memory location, a representative subset of memory accesses to that location. The Nondeterminator-3 uses an extension of the ALL-SETS [4] access-history algorithm used by its serially running predecessor, the Nondeterminator-2. First, the ALL-SETS algorithm was extended to correctly support the `inlet` feature of Cilk. This extension increases the memory-access cost by only a constant factor. Then, this extended ALL-SETS algorithm was parallelized, so that it can be combined with the SP-hybrid algorithm to obtain a data-race detector. Assuming that the cost of locking the access-history can be ignored, this parallelization also inflates the memory-access cost by only a constant factor.

I tested the Nondeterminator-3 on several programs to verify the accuracy of the implementation. I have also observed that the Nondeterminator-3 achieves good speed-up when run on a multiprocessor machine.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

Thanks to my supervisor Charles Leiserson for introducing me to this project. Thanks to Jeremy Fineman, in addition to Charles, for providing invaluable guidance on carrying out this project. In addition to technical aspects, Charles also provided invaluable help with the presentation of this document. Thanks to Jeremy for letting me borrow a couple of diagrams from his papers. Thanks to them both for being very patient. Thanks also to Bradley Kuszmaul, Gideon Stupp, and other members of the Supercomputing Technologies group for helpful comments.

Thanks to my family back at home for providing invaluable support. Thanks also to my host family in Boston for helping me in many ways during the last few years. There are other friends that I would like to thank as well.

This work was supported in part by the Singapore-MIT Alliance, by NSF Grant ACI-0324974, and by NSF Grant CNS-0305606. Any opinions, findings and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Contents

List of Figures

Chapter 1

Introduction

This thesis describes the implementation of a provably good data race detector, called the Nondeterminator-3, which runs efficiently in parallel. A ***data race*** occurs in a multithreaded program when two logically parallel threads access the same location while holding no common locks and at least one of the accesses is a write. Data races are common bugs in parallel programs and are often hard to track down. The Nondeterminator-3 checks for data races in programs coded in Cilk [3,10], a shared-memory multithreaded programming language.

Figure 1-1 illustrates a data race in a Cilk program. The procedures `foo1`, `foo2`, and `foo3` run in parallel, resulting in parallel accesses to the shared variable `x`. The accesses by `foo1` and `foo2` are protected by lock `A` and hence do not form a data race. Likewise, the accesses by `foo1` and `foo3` are protected by lock `B`. The accesses by `foo2` and `foo3` are not protected by a common lock, however, and therefore form a data race. If all accesses had been protected by the same lock, only the value 3 would be printed, no matter how the computation was scheduled. Because of the data race, however, the value of `x` printed by `main` might be 2, 3, or 6, depending on scheduling, since the statements in `foo2` and `foo3` are composed of multiple machine instructions which may interleave, possibly resulting in a lost update to `x`.

The Nondeterminator-3 succeeds the serially running Nondeterminator-2 [4]. Although data race detectors are only debugging tools, there are many reasons for building an efficient parallel data-race detector. Firstly, a parallel data-race detector would

```
int x;
Cilk_lockvar A, B;

cilk void foo1() {
  Cilk_lock(A);
  Cilk_lock(B);
  x += 5;
  Cilk_unlock(B);
  Cilk_unlock(A);
}

cilk void foo2() {
  Cilk_lock(A);
  x -= 3;
  Cilk_unlock(A);
}

cilk void foo3() {
  Cilk_lock(B);
  x++;
  Cilk_unlock(B);
}

cilk int main() {
  Cilk_lock_init(A);
  Cilk_lock_init(B);
  x = 0;
  spawn foo1();
  spawn foo2();
  spawn foo3();
  sync;
  printf("%d", x);
}
```

Figure 1-1: A Cilk program with a data race.

enable faster debugging of data races in parallel programs. An on-the-fly data-race detector that preserves the parallelism of an application program would enable the program to be run with race-detection options always turned on; a serially-running data-race detector, on the other hand, would not enable such real-time testing. The Nondeterminator-3 implementation also helps in demonstrating the possibility of using the underlying theoretical ideas to obtain a data-race detector that achieves good speed-up in practice.

Series-parallel parse trees and SP-hybrid

A key capability of data-race detectors is in determining the series-parallel (SP) relationship between two threads. The Nondeterminator-3 is based on a provably good parallel SP-maintenance algorithm known as SP-hybrid [2]. For a program with n threads, T_1 work, and critical-path length T_∞, the SP-hybrid algorithm runs in $O((T_1/P + PT_\infty)\lg n)$ expected time when executed on P processors.[1]

The execution of a multithreaded program can be viewed as a directed acyclic graph, or ***computation dag***, where nodes are either ***forks*** or ***joins*** and edges are

[1]In [9], the SP-hybrid algorithm has been improved to give better asymptotic bounds.

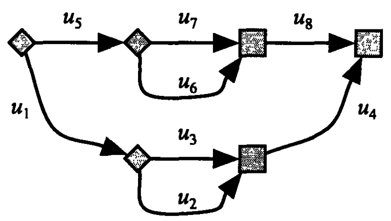

Figure 1-2: A dag representing a multithreaded computation. The edges represent threads, labeled $u_0, u_1, \ldots u_8$. The diamonds represent forks, and the squares indicate joins.

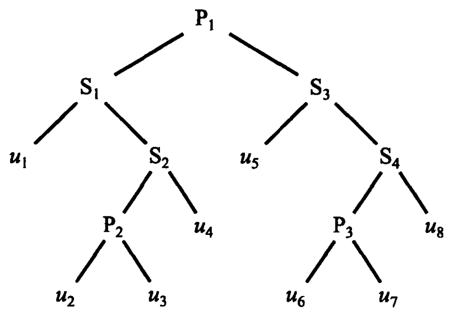

Figure 1-3: The parse tree for the computation dag shown in Figure 1-2. The leaves are the threads in the dag. The S-nodes indicate series relationships, and the P-nodes indicate parallel relationships.

threads. Such a dag is illustrated in Figure 1-2. A fork node has a single incoming edge and multiple outgoing edges. A join node has multiple incoming edges and a single outgoing edge. Threads (edges) represent blocks of serial execution.

For fork-join programming models, where every fork has a corresponding join that unites the forked threads, the computation dag has a structure that can be represented efficiently by a ***series-parallel (SP) parse tree*** [8]. In the parse tree each internal node is either an ***S-node*** or a ***P-node***, and each leaf is a thread of the dag.

Figure 1-3 shows the parse tree corresponding to the computation dag from Figure 1-2. If two subtrees are children of the same S-node, then the parse tree indicates that (the subcomputation represented by) the left subtree executes before (that of) the right subtree. If two subtrees are children of the same P-node, then the parse tree indicates that the two subtrees execute logically in parallel.

The SP-hybrid algorithm given in [2] takes as input an execution of a multithreaded program represented as a series-parallel parse tree.

Results

The main results obtained in this thesis are the following.

- I have translated the SP-hybrid algorithm from its abstract form which was given as a walk on the series-parallel parse tree in [2], into actual C code incorporated into the Cilk implementation.
- I discovered a deficiency in the serial ALL-SETS access-history algorithm [4], which was part of the Nondeterminator-2. The deficiency was that it could not satisfactorily handle the implicit atomicity guarantees provided by Cilk. For example, Cilk guarantees that inlets within a procedure instance all run atomically with respect to each other. An experienced Cilk programmer may assume these atomicity guarantees while writing code. The naive solution of simply avoiding the reporting of data races between implicitly atomic sections fails because this approach would preclude the detection of other data races. Other naive approaches, such as the use of an "inlet lock," also fail. My solution to this problem is an algorithm that involves expanding the access-history to maintain *two* representative thread ID's rather than just one.
- I have parallelized the extended ALL-SETS algorithm, so that it allows non-depth-first expansion of P-nodes. By combining this parallelization with the SP-hybrid implementation, I have obtained a data race detector for Cilk that runs in parallel.

Usage and organization of the program

The Nondeterminator-3 can be invoked at compile-time by passing the `-nd_sphybrid` flag to the `cilkc` compiler driver. For example, if `program.cilk` is the program given in Figure 1-1, then the commands

```
cilkc -nd_sphybrid program.cilk
./a.out
```

would produce an output similar to

```
--------------------------------------
Data race
   'x' (program.cilk:line 14) with
   'x' (program.cilk:line 20)
--------------------------------------
```

The work of the Nondeterminator is split between the `cilk2c` source-to-source compiler and the Cilk runtime system. When the `-nd_sphybrid` flag is passed to the `cilkc` compiler driver, `cilk2c` instruments the program with calls to SP-maintenance functions in the runtime system. These functions update the SP-maintenance data-structures as necessary. Each memory access is also instrumented so that it is notified to the access-history algorithm. The access-history algorithm makes calls to the functions that implement the SP-hybrid queries.

Organization of this document

The remainder of this document is organized as follows. Chapter 2 describes my implementation of the SP-hybrid algorithm. Chapter 3 presents my extension of the serial ALL-SETS algorithm to correctly support inlets, and also a parallelization of this extended algorithm. Chapter 4 provides some performance measurements of the data-race detector. Chapter 5 discusses related work. Chapter 6 offers some concluding remarks.

Chapter 2

Implementing the SP-hybrid algorithm

In this chapter, I present my implementation of the SP-hybrid algorithm. The SP-hybrid algorithm has been translated from its abstract form, which was given as a walk on the series-parallel parse tree in [2], into actual C code incorporated into the Cilk implementation. I will first give an overview of the data structures and types that are used. Then, I will give low-level pseudocode for my implementation of the SP-hybrid runtime functions.

2.1 Data structures used

The SP-hybrid algorithm partitions the threads into ***traces***, where a trace is a set of threads that have been executed by a single worker. A ***computation*** is a dynamic collection of disjoint traces. As the computation unfolds, each thread is inserted into a trace. Between different traces, the SP relationships are maintained using a shared SP-order data structure, sometimes referred to as the ***global*** tier. Within each trace, the SP relationships are maintained using an SP-bags data structure, sometimes referred to as a ***local*** tier.

I will first describe the data structures of the global tier. Then, I will do the same for the local tier. Finally, I will describe the state that needs to be maintained by each worker as it executes parts of the computation.

Global tier

The global tier is based on an order-maintenance data structure, which is an abstract data type that supports the following operations:

- CREATE_OM_STRUCT(): Creates and returns a new ordering.
- CREATE_OM_ELEMENT(L): Creates and returns a new element that can be inserted in the ordering L.
- OM_INSERT(L, X, Y): In the ordering L, inserts element Y immediately after element X.
- OM_MULTI_INSERT(L, Y_1, Y_2, X, Y_4, Y_5): In the ordering L, inserts elements Y_1 and Y_2 in this order immediately before X, and inserts Y_4 and Y_5 in this order immediately after X. This operation is supported by using a constant number of OM_INSERT operations.
- OM_PRECEDES(L, X, Y): Returns `TRUE` if X precedes Y in the ordering L, and otherwise returns `FALSE`.

The global tier consists of two linear orderings, *Eng* and *Heb*. A trace is represented by two elements, one in each of *Eng* and *Heb*. In my implementation, a *C* `struct` called `Trace` encapsulates the *Eng* and *Heb* elements. For any two traces X and Y in the global tier, X serially precedes Y if and only if both the *Eng* element of X precedes the *Eng* element of Y in the *Eng* ordering and the *Heb* element of X precedes the *Heb* element of Y in the *Heb* ordering. Traces X and Y are parallel if and only if neither X serially precedes Y nor Y serially precedes X.

Figure 2-1 shows the global-tier component of the SP-hybrid algorithm, written as Cilk-like pseudocode that operates on the series-parallel parse tree. This pseudocode was taken directly from [2]. SP-HYBRID accepts as arguments an SP-parse-tree node X and a trace U, and it returns a trace. The algorithm is essentially a tree walk which carries along with it a trace U into which encountered threads are inserted. The EXECUTE-THREAD procedure executes the thread and handles all local-tier op-

```
SP-HYBRID($X, U$)
      ▷ $X$ is a SP-parse-tree node, and $U$ is a trace
 1    if ISLEAF($X$)
 2       then ▷ $X$ is a thread
 3            $U \leftarrow U \cup \{X\}$
 4            EXECUTE-THREAD($X$)
 5            return $U$

 6    if ISSNODE($X$)
 7       then ▷ $X$ is an S-node
 8            $U' \leftarrow$ spawn SP-HYBRID($left[X], U$)
 9            sync
10            $U'' \leftarrow$ spawn SP-HYBRID($right[X], U'$)
11            sync
12            return $U''$

      ▷ $X$ is a P-node
13    $U' \leftarrow$ spawn SP-HYBRID($left[X], U$)
14    if SYNCHED()
15       then ▷ the recursive call on line 13 has completed
16            $U'' \leftarrow$ spawn SP-HYBRID($right[X], U'$)
17            sync
18            return $U''$

      ▷ A steal has occurred
19    create new traces $U_1$, $U_2$, $U_4$, and $U_5$
20    ACQUIRE($lock$)
21    OM-MULTI-INSERT($Eng, U_1, U_2, U, U_4, U_5$)
22    OM-MULTI-INSERT($Heb, U_1, U_4, U, U_2, U_5$)
23    RELEASE($lock$)
24    SPLIT($U, X, U_1, U_2$)
25    spawn SP-HYBRID($right[X], U_4$)
26    sync
27    return $U_5$
```

Figure 2-1: The global-tier component of the SP-hybrid algorithm, written as Cilk-like pseudocode.

erations. The SYNCHED procedure determines whether the current procedure is synchronized (whether a **sync** would cause the procedure to block), which indicates whether a steal has occurred. The SPLIT procedure uses node X to partition the existing threads in trace U into three sets, leaving one of the sets in U and placing the other two into U_1 and U_2. For a detailed description of this algorithm and its correctness, please refer to [2].

I will now describe the local-tier data structures and the Nondeterminator-specific state that needs to be maintained by each worker. Then, I will describe how to implement the SP-hybrid algorithm as part of the Cilk runtime system.

Local tier

An SP-bags data structure maintains SP relationships within a trace. The SP-bags data structure is based on a disjoint-set data structure, which we view as an abstract data type that supports the following operations:

- CREATE_DS_ELEMENT(*data*): Creates a new set containing exactly one element. The element has a data field, in which it stores *data*. Returns the element.
- DS_GET_DATA(x): Returns the data associated with element x.
- DS_SET_DATA(x, *newdata*): Sets the data field of element x to *newdata*.
- DS_UNION(x, y): Performs a union of the two sets containing elements x and y. Returns the canonical element representing the resulting set.
- DS_FIND_SET(x): Returns the canonical element representing the set that contains element x.

For each trace, each procedure instance whose threads are in that trace is represented by a disjoint-set element. For each procedure instance that is currently executing or is in a worker's deque, two "bags" of procedure ID's are maintained.

- The ***s-bag*** of a procedure instance F contains the ID's of the descendant procedure instances of F that are in the same trace as the currently executing thread in F and that logically precede the currently executing thread in F.
- The ***p-bag*** of a procedure instance F contains the ID's of the descendant procedure instances of F that are in the same trace as the currently executing thread in F and that operate logically in parallel with the currently executing thread in F.

Associated with each trace are two `Trace` pointers, called the ***S-SET*** and the ***P-SET***, which point to the trace. In the canonical element representing an s-bag, the data field points to the S-SET. Similarly, in the canonical element representing a p-bag, the data field points to the P-SET. Thus, to determine whether an element is in an s-bag or a p-bag, we first find its canonical element and then check whether its data field points to the S-SET or the P-SET of the trace. To obtain the trace to which a thread belongs, we dereference the data pointer of the canonical element twice.

ND_state

Each active worker must keep track of certain Nondeterminator-specific state. This state includes the current trace (which is the trace containing the currently executing thread), the S-SET and P-SET of the current trace, the s-bag and the p-bag of the currently executing procedure instance, and the ID of the current procedure instance. We encapsulate this state in a C `struct` called `ND_state`.

The fields *current_trace*, *S_SET*, *P_SET*, *s_set*, *p_set*, *current_proc_id* are pointers to the current trace, S-SET, P-SET, s-bag, p-bag, and current procedure instance ID, respectively. We also need two additional fields, called *syncing_slow* and *next_sync_block_start_trace*, whose meanings are explained as we walk through the pseudocode in section 2.2.

SP queries may need to be performed from *C* code that does not have access to the worker state. One approach for providing the Nondeterminator state to such *C* code is by changing the signature of the function to accept an additional argument, the worker number. A global ND_state array can be kept, and the required object can be accessed by indexing the array with the worker number.

All calls to *C* functions will need to be changed to pass this additional argument, however, and it is not possible for the preprocessor to distinguish between a standard *C* function and one that has been instrumented by the Nondeterminator. Therefore, rather than changing any method signatures, we store and retreive the worker number in thread-local storage using the pthread methods PTHREAD_SETSPECIFIC and PTHREAD_GETSPECIFIC.

2.2 SP-hybrid runtime functions

This section lists and describes pseudocode for implementing the SP-hybrid algorithm as part of the Cilk runtime system. First, I will list and describe the pseudocode for updating the SP-hybrid data structures. Then, I will list and describe the pseudocode for performing the SP queries.

Implementation of SP-hybrid updates

The updates to the SP-hybrid data structures are implemented as additions to the default actions taken by the Cilk runtime system when spawns, returns, syncs, and steals are encountered. The pseudocode below shows only the Nondeterminator-specific additions and not the default actions that the Cilk runtime system already takes (such as pushing and popping Cilk activation frames, performing steals, and detecting whether a procedure's parent has been stolen).

When a spawn is encountered by a worker, this worker creates and initializes a new `ND_state` record for the procedure instance that is spawned. This work is shared between the procedures BEFORE_SPAWN and INIT_FRAME shown in Figures 2-2 and 2-3, respectively. The newly created record is initialized by setting the current trace, S-SET, and P-SET to be the same as those of the parent. A new procedure-instance ID is created and put into the s-bag. Pointers to this `ND_state` record are stored in the global `ND_state` array and in the child's activation frame.

```
BEFORE_SPAWN(frame)
    ▷ frame is the calling procedure's activation frame

1   nd_state[me] = MALLOC(SIZEOF(ND_state))
2   nd_state[me]→current_trace = frame→nd_state→current_trace
3   nd_state[me]→S_SET = frame→nd_state→S_SET
4   nd_state[me]→P_SET = frame→nd_state→P_SET
```

Figure 2-2: Pseudocode for BEFORE_SPAWN.

Figure 2-4 shows pseudocode for the Nondeterminator-specific actions that are taken when a spawned procedure returns to its parent. The actions taken depend on whether or not the parent procedure had been stolen. If the parent had not been stolen, then we first execute any inlet that is waiting for the result returned by the child. We then perform the usual SP-bags action, which is to move the contents of the child's s-bag to the parent's p-bag. The `ND_state` of the worker is also restored

```
INIT_FRAME($frame$)
    ▷ $frame$ is the child procedure's activation frame

1  $frame \rightarrow nd\_state = nd\_state[me]$
2  $nd\_state[me] \rightarrow s\_set$ = CREATE_DS_ELEMENT($nd\_state[me] \rightarrow S\_SET$)
3  $nd\_state[me] \rightarrow p\_set$ = NULL
4  $nd\_state[me] \rightarrow current\_proc\_id = nd\_state[me] \rightarrow s\_set$
5  $nd\_state[me] \rightarrow syncing\_slow$ = FALSE
6  $nd\_state[me] \rightarrow next\_sync\_block\_start\_trace$ = NULL
```

Figure 2-3: Pseudocode for INIT_FRAME.

to that stored in the parent so that the worker can resume executing the parent. On the other hand, if the parent had been stolen, then the `ND_state` record of the child must be stored along with any inlet that is waiting for the result returned by the child. The worker then automatically unwinds and returns to the worker pool. The worker that executes the inlet must restore the `ND_state` record from the inlets queue before executing the inlet, in order to have the correct state for the current procedure ID, S-SET, and P-SET. This restoration of state is necessary because logically, the inlet executes serially after the child's final sync block and before the child procedure syncs with the parent.

```
WHEN_RETURNED($frame$)
    ▷ $frame$ is the parent procedure's activation frame

 1  if $frame$ had been stolen
 2     then if there is an inlet waiting for the result
 3             then store $nd\_state[me]$ in inlets queue
 4             else FREE($nd\_state[me]$)
 5          return
 6  if there is an inlet waiting for the result
 7     then execute the inlet
 8  $frame \rightarrow nd\_state \rightarrow p\_set$
    = DS_UNION($frame \rightarrow nd\_state \rightarrow p\_set, nd\_state[me] \rightarrow s\_set$)
 9  DS_SET_DATA($frame \rightarrow nd\_state \rightarrow p\_set, frame \rightarrow nd\_state \rightarrow P\_SET$)
10  FREE($nd\_state[me]$)
11  $nd\_state[me] = frame \rightarrow nd\_state$
```

Figure 2-4: Pseudocode for WHEN_RETURNED.

Figure 2-5 shows pseudocode for the Nondeterminator-specific actions taken when a sync statement is encountered in a fast clone of a Cilk procedure. Since fast clones have never been stolen, we simply need to perform the usual SP-bags action which is to move the contents of the current procedure's p-bag into its s-bag.

```
AT_SYNC_FAST($frame$)
1  $nd\_state[me] \rightarrow s\_set$
       $= \text{DS\_UNION}(nd\_state[me] \rightarrow s\_set, nd\_state[me] \rightarrow p\_set)$
2  $nd\_state[me] \rightarrow p\_set$ = NULL
3  $\text{DS\_SET\_DATA}(nd\_state[me] \rightarrow s\_set, nd\_state[me] \rightarrow S\_SET)$
```

Figure 2-5: Pseudocode for AT_SYNC_FAST.

Figure 2-6 shows pseudocode for the Nondeterminator-specific actions performed when a worker starts executing a slow clone of a procedure. First, we check the state of the field *syncing_slow*, which indicates whether execution of the procedure resumes at a sync point (this happens when a suspended procedure has been awakened as a result of all the children having returning to the parent). If it does, then we immediately return, and the necessary work will be done when AFTER_SYNC_SLOW is called at the sync point.

On the other hand, if the stolen procedure's execution resumes at a thread that begins immediately after a spawn, then we perform the trace-splitting: the central part of SP-hybrid's global-tier component. We assume that the current worker had acquired a "steal lock" of the victim, which we denote by *victim_steal_locks*[*victim*]. After the new traces have been inserted, we release this lock. The purpose of this lock is to ensure that no other steals from the victim will take place until all these new traces have been inserted; the SP-hybrid algorithm's correctness requires that "top-most" traces are split first. We then move the *s_set* of the stolen procedure to the newly created trace U_1, and the *p_set* to U_2. Finally, we reinitialize the current procedure's `ND_state` record and set the current worker's `ND_state` pointer to point to this record.

```
START_THREAD_SLOW($frame$)
 1  if $frame \to nd\_state \to syncing\_slow$
 2      then return

    ▷ Create and insert the new traces
 3  create new traces $U_1, U_2, U_4, U_5$
 4  ACQUIRE($OM\_insert\_lock$)
 5  OM_MULTI_INSERT($Eng, U_1, U_2, frame \to nd\_state \to current\_trace, U_4, U_5$)
 6  OM_MULTI_INSERT($Heb, U_1, U_4, frame \to nd\_state \to current\_trace, U_2, U_5$)
 7  RELEASE($OM\_insert\_lock$)
 8  RELEASE($victim\_steal\_locks[victim]$)

    ▷ Move the s-set set to $U_1$ and the p-set to $U_2$
 9  Trace ** $U_1\_SET$ = MALLOC(SIZEOF(Trace*))
10  $*U_1\_SET = U_1$
11  DS_SET_DATA($frame \to nd\_state \to s\_set, U_1\_SET$)
12  Trace ** $U_2\_SET$ = MALLOC(SIZEOF(Trace*))
13  $*U_2\_SET = U_2$
14  DS_SET_DATA($frame \to nd\_state \to p\_set, U_2\_SET$)

    ▷ Re-initialize the frame's $nd\_state$ using the new trace $U_4$
15  $frame \to nd\_state \to current\_trace = U_4$
16  $frame \to nd\_state \to S\_SET$ = MALLOC(SIZEOF(Trace*))
17  $frame \to nd\_state \to P\_SET$ = MALLOC(SIZEOF(Trace*))
18  $*frame \to nd\_state \to S\_SET = *frame \to nd\_state \to P\_SET = U_4$
19  $frame \to nd\_state \to s\_set$
      = CREATE_DS_ELEMENT($frame \to nd\_state \to S\_SET$)
20  $frame \to nd\_state \to p\_set$ = NULL
21  $frame \to nd\_state \to current\_proc\_id = frame \to nd\_state \to s\_set$
22  if $frame \to nd\_state \to next\_sync\_block\_start\_trace == NULL$
23      then $frame \to nd\_state \to next\_sync\_block\_start\_trace = U_5$

24  $nd\_state[me] = frame \to nd\_state$
```

Figure 2-6: Pseudocode for START_THREAD_SLOW.

```
BEFORE_SYNC_SLOW($frame$)
1  $frame \to nd\_state \to syncing\_slow$ = TRUE
```

Figure 2-7: Pseudocode for BEFORE_SYNC_SLOW.

The procedure BEFORE_SYNC_SLOW, shown in Figure 2-7, is called immediately before a slow clone checks whether all its spawned children have returned. If it is the case that some children have *not* yet returned, then the procedure would be suspended and would be reawakened only when all the children have indeed returned. To prevent the procedure START_THREAD_SLOW from performing any redundant splitting upon the reawakening of the procedure, we set the field *syncing_slow*.

The procedure AFTER_SYNC_SLOW shown in Figure 2-8 is called after all the children have returned to the parent and immediately before execution of the first thread of the next sync block. We first reset the field *syncing_slow*. It is also necessary to restore the `ND_state` pointer of the worker, because some inlets may have been executed immediately before the call to this procedure.

```
AFTER_SYNC_SLOW($frame$)
 1  $frame \rightarrow nd\_state \rightarrow syncing\_slow$ = FALSE
 2  $nd\_state[me] = frame \rightarrow nd\_state$
 3  if $frame \rightarrow nd\_state \rightarrow next\_sync\_block\_start\_trace \neq$ NULL
 4     then $frame \rightarrow nd\_state \rightarrow current\_trace$
              $= frame \rightarrow nd\_state \rightarrow next\_sync\_block\_start\_trace$
 5          $frame \rightarrow nd\_state \rightarrow S\_SET$ = MALLOC(SIZEOF(Trace*))
 6          $frame \rightarrow nd\_state \rightarrow P\_SET$ = MALLOC(SIZEOF(Trace*))
 7          $*frame \rightarrow nd\_state \rightarrow S\_SET = *frame \rightarrow nd\_state \rightarrow P\_SET$
              $= frame \rightarrow nd\_state \rightarrow current\_trace$
 8          $frame \rightarrow nd\_state \rightarrow s\_set$
              = CREATE_DS_ELEMENT($frame \rightarrow nd\_state \rightarrow S\_SET$)
 9          $frame \rightarrow nd\_state \rightarrow p\_set$ = NULL
10          $frame \rightarrow nd\_state \rightarrow current\_proc\_id = frame \rightarrow nd\_state \rightarrow s\_set$
11          $frame \rightarrow nd\_state \rightarrow next\_sync\_block\_start\_trace$ = NULL
12     else AT_SYNC_FAST($frame$)
```

Figure 2-8: Pseudocode for AFTER_SYNC_SLOW.

If no steals of the procedure instance had occurred in the preceeding sync block, then the field *next_sync_block_start_trace* would be equal to NULL. In such a case, the actions that need to be taken are the usual SP-bags actions, the code for which is already present in the function AT_SYNC_FAST, to which we simply jump. If one or more steals had indeed occurred in the preceeding sync block, then the field *next_sync_block_start_trace* would have been set at the first such steal, in the function START_THREAD_SLOW. In such a case, we modify the `ND_state` so that subsequent threads that descend from the current procedure instance (until the next steal of it, if any) will start getting inserted into this new trace.

Implementation of SP-hybrid queries

Now, I will list and describe the pseudocode for performing the SP-hybrid queries.

Figure 2-9 shows pseudocode for the query SP_TRACE_PRECEDES, which takes two traces U and V and returns `TRUE` if all the threads in U serially precede all the threads in V, and `FALSE` otherwise. This is determined by comparing the relative order of U and V in the *Eng* and *Heb* order-maintenance data structures.

```
SP_TRACE_PRECEDES($U, V$)
1  if OM_PRECEDES($Eng, U, V$) and OM_PRECEDES($Heb, U, V$)
2      then return TRUE
3      else return FALSE
```

Figure 2-9: Pseudocode for SP_TRACE_PRECEDES.

```
SP_ISLEFTOF($x, y$)
   ▷ We assume that the calling worker is currently executing thread $y$

1  $U$ = *DS_GET_DATA(DS_FIND_SET($x$))
2  $V$ = *DS_GET_DATA(DS_FIND_SET($y$))
3  if $U \neq V$
4      then return OM_PRECEDES($Eng, U, V$)
5      else return TRUE
```

Figure 2-10: Pseudocode for SP_ISLEFTOF.

Figure 2-10 shows pseudocode for the query SP_ISLEFTOF, which takes two threads x and y where y is the thread that the calling worker is currently executing. It returns `TRUE` if thread x is encountered *before* thread y in a left-to-right walk of the SP parse tree, and returns FALSE otherwise. If x and y are in different traces, then we compare their order in the *Eng* order-maintenance data structure. If they are in the same trace, then we simply return TRUE, because the threads of a trace constitute a subcomputation that is executed by a single worker, and workers always execute a stolen subcomputation in the serial left-to-right order.

```
SP_PRECEDES($x, y$)
   ▷ We assume that the calling worker is currently executing thread $y$

1  $U\_SET$ = DS_GET_DATA(DS_FIND_SET($x$))
2  $U = *U\_SET$
3  $V$ = *DS_GET_DATA(DS_FIND_SET($y$))
4  if $U \neq V$
5     then return SP_TRACE_PRECEDES($U, V$)
6     else return ($U\_SET == nd\_state[me] \rightarrow S\_SET$)
```

Figure 2-11: Pseudocode for SP_PRECEDES.

Figure 2-11 shows pseudocode for the query SP_PRECEDES, which takes two threads x and y where y is the thread that the calling worker is currently executing. It returns TRUE if x serially precedes y, and FALSE otherwise. First, we obtain the "data field" of the canonical element containing the trace x and store it in U_SET. We then obtain the trace U that contains x by dereferencing U_SET. If x and y are found to be in different traces in line 4, then we use the query SP_TRACE_PRECEDES. On the other hand, if they are found to be in the same trace, then x serially precedes y if and only if x is in an s-bag of the trace containing thread y, which we determine by checking if U_SET is equal to the S_SET of the trace containing thread y.

Note that if we had obtained U by directly dereferencing the "data field" of the canonical element containing x, rather than by first storing the "data field" in U_SET, then in line 6 we would have had to reread the "data field." This code would have been incorrect, because a steal could have caused x to migrate from one trace to another in between the times that lines 4 and 6 are executed. Note also that we did not have to use this approach to obtain V, the trace containing thread y, because a

```
SP_PARALLEL($x, y$)
    ▷ We assume that the calling worker is currently executing thread $y$

1  $U\_SET$ = DS_GET_DATA(DS_FIND_SET($x$))
2  $U = *U\_SET$
3  $V$ = *DS_GET_DATA(DS_FIND_SET($y$))
4  if $U \neq V$
5      then return ¬SP_TRACE_PRECEDES($U, V$)
6      else  return ($U\_SET$ == $nd\_state[me] \rightarrow P\_SET$)
```

Figure 2-12: Pseudocode for SP_PARALLEL.

thread never migrates between traces while it is executing.

Figure 2-12 shows pseudocode for the query SP_PARALLEL, which is the same as SP_PRECEDES except at lines 5 and 6. At line 5, we return TRUE if and only if trace U does *not* precede trace V. (It cannot be the case that y precedes x, since x has already at least partially executed and y is still executing.) At line 6, we return TRUE if and only if U_SET is equal to the P_SET of the current trace.

A sequential consistency issue

We must ensure that whenever we change the "data field" of the canonical element of an s-bag or p-bag, dereferencing this data field twice would lead to the trace containing the s-bag or p-bag. Otherwise, an SP-hybrid query that is concurrently executing may not obtain the correct trace. Therefore, in START_THREAD_SLOW shown in Figure 2-6, the order of lines 10 and 11 and of lines 13 and 14 are important. To prevent these instructions from being reordered, a memory-fence instruction must be inserted in between each of these pairs of lines.

Garbage collection of thread IDs

To avoid a memory leak, we must free a thread ID whenever there can no longer be a reference to it from the access-history. We accomplish this by maintaining, for each thread ID, a count of the number of references to it from the access-history.

Chapter 3

Extending the All-Sets access-history algorithm

In this chapter, I present an extension of the serial `All-Sets` algorithm to correctly support the `inlet` feature of Cilk. Cilk guarantees that the threads of a procedure instance, including its inlets, operate atomically with respect to each other. Unfortunately, the parse tree does not capture the atomicity guarantees that involve inlets. The `All-Sets` implementation used by the Nondeterminator-2 approached this problem by the use of a fake global "inlet lock": when an inlet is entered, acquire this lock, and when it returns, release the lock. This approach is incorrect for two reasons. Firstly, it does not take into account the atomicity between inlet and non-inlet threads of the same procedure instance. Secondly, it precludes the data-race detector from detecting data-races between inlets of different procedure instances. A naive solution is to have a separate fake "inlet lock" for each procedure instance, but this approach leads to access-history lists with (nearly) unbounded length. My solution directly modifies the `All-Sets` algorithm to take the atomicity guarantees into account, and it increases the memory and time complexity by only a constant factor.

I also present `Parallel All-Sets`, which is a parallelization of the extended `All-Sets` algorithm. This algorithm is *parallel* in the sense that, unlike the serial `All-Sets` algorithm, it allows for non-depth-first expansion of P-nodes.

Extending All-Sets to support inlets

To extend the All-Sets algorithm to support inlets, we first make the following helpful definitions.

Definition For a thread e, we use ***frame(e)*** to denote the Cilk procedure instance that contains e.

Definition We say that threads e_1 and e_2 are ***strongly parallel*** (written as $\mathbf{e_1 \| \| e_2}$) if $e_1 \parallel e_2$ and $\mathit{frame}(e_1) \neq \mathit{frame}(e_2)$.

Definition An ***access*** is a 3-tuple $\langle e, H, l\rangle$, which denotes that thread e has read or written location l while holding the set H of locks.

A data-race exists between two accesses $\langle e_1, H_1, l\rangle$ and $\langle e_2, H_2, l\rangle$ if and only if $e_1 \| \| e_2$ and $H_1 \cap H_2 = \{\}$.

```
ACCESS($\langle e, H, l\rangle$)
 1  for each $\langle a, b, H', l\rangle \in \mathit{lockers}$
 2      do if $(H' \cap H = \{\})$ and $(a \| \| e$ or $b \| \| e)$
 3          then declare a data race
 4  if $\exists$ thread ids' $a$ and $b$ such that $\langle a, b, H, l\rangle \in \mathit{lockers}$
 5     then if $a \prec e$
 6             then if $\mathit{frame}(e) = \mathit{frame}(a)$
 7                     then $\mathit{lockers} \leftarrow \big(\mathit{lockers} - \{\langle a, b, H, l\rangle\}\big) \cup \{\langle e, b, H, l\rangle\}$
 8                     else $\mathit{lockers} \leftarrow \big(\mathit{lockers} - \{\langle a, b, H, l\rangle\}\big) \cup \{\langle e, a, H, l\rangle\}$
 9          elseif $(b \prec e)$ and $(\mathit{frame}(e) \neq \mathit{frame}(a))$
10             then $\mathit{lockers} \leftarrow \big(\mathit{lockers} - \{\langle a, b, H, l\rangle\}\big) \cup \{\langle a, e, H, l\rangle\}$
11     else $\mathit{lockers} \leftarrow \mathit{lockers} \cup \{\langle e, \mathit{initial}, H, l\rangle\}$
```

Figure 3-1: The Extended All-Sets algorithm, which handles inlets correctly.

Figure 3-1 gives pseudocode for the Extended All-Sets algorithm. In the pseudocode, *initial* denotes an imaginary "initial" thread that serially precedes all other threads and is part of a procedure instance different from those of all other threads. Unlike in the original serial All-Sets algorithm, the extended version stores a lockset together with *two* thread ID's in each locker.

Definition A ***locker*** is a 4-tuple $\langle a, b, H, l\rangle$, which denotes that accesses $\langle a, H, l\rangle$ and $\langle b, H, l\rangle$ have occurred. (The ID a is thought of as the "primary" thread ID, and the ID b is thought of as an "extra" thread ID.)

The two thread ID's in a locker together "represent" any past accesses to the same location while holding the same lockset. This notion is explained in the following definition.

Definition Consider some point during an execution of the program. Let $\langle e, H, l\rangle$, $\langle a, H, l\rangle$, and $\langle b, H, l\rangle$ denote any three (not necessarily distinct) accesses that have occurred so far. We say that the locker $\langle a, b, H, l\rangle$ ***strongly represents*** the access $\langle e, H, l\rangle$ if for any thread f that executes in the future, we have that $e|||f$ implies that either $a|||f$ or $b|||f$.

Before proving the correctness of the `Extended All-Sets` algorithm, we restate the following two important lemmas from [8].

Lemma 1 *Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk program, and suppose that $e_1 \prec e_2$ and $e_1 \parallel e_3$. Then, we have $e_2 \parallel e_3$.* □

Lemma 2 (Pseudotransitivity of $\parallel$) *Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk program, and suppose that $e_1 \parallel e_2$ and $e_2 \parallel e_3$. Then, we have $e_1 \parallel e_3$.* □

We now prove the correctness of the `Extended All-Sets` algorithm.

Lemma 3 *The following invariant holds after each access. Each locker $\langle a, b, H, l\rangle$ in* `lockers` *satisfies (i) frame$(a) \neq$ frame(b) and (ii) for any thread f that executes in the future, we have that $b \parallel f$ implies that $a \parallel f$.*

Proof. By induction on the number of accesses that have occurred. The base case is when no accesses have occured, in which case the invariant holds trivially. For the induction step, note that the access-history is modified only at lines 7, 8, 10, and 11. It is easy to check that (i) holds in each of these cases. With the help of Lemmas 1 and 2, it is also easy to check that (ii) holds in each of these cases. $\square$

The following three lemmas can be viewed as some form of "transitivity" of the relation "strongly represents".

Lemma 4 *Suppose that $\langle e, b, H, l\rangle$ strongly represents $\langle a, H, l\rangle$, and that $\langle a, b, H, l\rangle$ strongly represents $\langle d, H, l\rangle$. Then, $\langle e, b, H, l\rangle$ strongly represents $\langle d, H, l\rangle$.*

Proof. Let f denote any thread that executes in the future. If $d \| f$, then either $a \| f$ or $b \| f$. If $a \| f$, then either $e \| f$ or $b \| f$. Hence, if $d \| f$, then we have either $e \| f$ or $b \| f$, as required. $\square$

Lemma 5 *Suppose that $\langle e, a, H, l\rangle$ strongly represents $\langle b, H, l\rangle$, and that $\langle a, b, H, l\rangle$ strongly represents $\langle d, H, l\rangle$. Then, $\langle e, a, H, l\rangle$ strongly represents $\langle d, H, l\rangle$.*

Proof. Let f denote any thread that executes in the future. If $d \| f$, then either $a \| f$ or $b \| f$. If $b \| f$, then either $a \| f$ or $e \| f$. Hence, if $d \| f$, then we have either $a \| f$ or $e \| f$, as required. $\square$

Lemma 6 *Suppose that $\langle a, e, H, l\rangle$ strongly represents $\langle b, H, l\rangle$, and that $\langle a, b, H, l\rangle$ strongly represents $\langle d, H, l\rangle$. Then, $\langle a, e, H, l\rangle$ strongly represents $\langle d, H, l\rangle$.*

Proof. The proof is exactly the same as for Lemma 5. $\square$

Theorem 7 *The following invariant holds after each access. For any location l, any lockset H, and any access $\langle d, H, l\rangle$ that has occurred so far, there exists a locker $\langle a, b, H, l\rangle \in lockers$ that strongly represents the access $\langle d, H, l\rangle$.*

Proof. By induction on the number of accesses to l that have occurred. The base case is when no accesses to l have occurred, in which case the claim holds trivially. For the induction step, assume that the claim holds after some set of accesses to l has occurred. Let $\langle e, H, l\rangle$ denote the next access to l. To see that the claim also holds after this access, note that either *lockers* is left unmodified or exactly one of lines 7, 8, 10, 11 executes. We consider each case in turn:

- *lockers* is left unmodified: This event falls into at least one of the following two subcases.
  - $a \parallel e$ and $b \parallel e$: By Lemma 2, we have $e \parallel f$ implies that both $a \parallel f$ and $b \parallel f$. By Lemma 3(i), we know that $\mathit{frame}(a) \neq \mathit{frame}(b)$. Thus, if $e \| \| f$ then either $a \| \| f$ or $b \| \| f$. Consequently, $\langle a, b, H, l \rangle$ strongly represents $\langle e, H, l \rangle$.
  - $a \parallel e$ and $\mathit{frame}(e) = \mathit{frame}(a)$: By Lemma 2, we have $e \parallel f$ implies that $a \parallel f$. Moreover, since $\mathit{frame}(e) = \mathit{frame}(a)$, we have $e \| \| f$ implies that $a \| \| f$. Thus, $\langle a, b, H, l \rangle$ strongly represents $\langle e, H, l \rangle$.
- line 7 executes: By Lemma 1, we have $a \parallel f$ implies that $e \parallel f$. Moreover, since $\mathit{frame}(e) = \mathit{frame}(a)$, we have $a \| \| f$ implies that $e \| \| f$. Hence $\langle e, b, H, l \rangle$ strongly represents $\langle a, H, l \rangle$. Now apply Lemma 4 to see that the claim holds.
- line 8 executes: By Lemmas 3(ii) and 1, we have $b \parallel f$ implies that both $a \parallel f$ and $e \parallel f$. Therefore, since $\mathit{frame}(e) \neq \mathit{frame}(a)$, we have $b \| \| f$ implies that either $a \| \| f$ or $e \| \| f$. Hence $\langle e, a, H, l \rangle$ strongly represents $\langle b, H, l \rangle$. Now apply Lemma 5 to see that the claim holds.
- line 10 executes: This case is similar to the one where line 8 executes. The locker $\langle a, e, H, l \rangle$ strongly represents $\langle b, H, l \rangle$. Now, apply Lemma 6 to see that the claim holds.
- line 11 executes: The claim trivially holds in this case because the current access is the only access to l with lockset H that has occurred so far.

$\square$

Corollary 8 *The* `Extended All-Sets` *algorithm detects a data race in a computation if and only if a data race exists.*

Proof. We see from the condition checked in line 2 of ACCESS that a data race is reported in line 3 only if a data race exists between $\langle e, H, l \rangle$ and either $\langle a, H', l \rangle$ or $\langle b, H', l \rangle$.

Conversely, suppose that a data race exists between accesses $\langle d, H', l \rangle$ and $\langle e, H, l \rangle$, and that they occur in this order. By Theorem 7, we know that immediately before the access $\langle e, H, l\rangle$, there exists a locker $\langle a, b, H', l\rangle$ in *lockers* that strongly represents the access $\langle d, H', l\rangle$. Therefore, we have $H' \cap H = \{\}$ and either $a \| e$ or $b \| e$. So a data race is reported in line 3. □

Theorem 9 *Consider a Cilk program that references V shared memory locations, and in which the maximum number of locks held simultaneously is k and the maximum number of distinct locksets used to access any particular location is L. Then, the access-history takes space $O(kLV)$, and each call to* ACCESS *runs in time $O(kL)$.*

Proof. For any location l and any lockset H, there is at most one locker $\langle a, b, H, l\rangle$ in the set *lockers*. This property holds, because in an execution of ACCESS, if there is already a locker $\langle a, b, H, l\rangle$, then either this locker is replaced by another locker with the same lockset and location, or no change is made. Each locker takes space $O(k)$. Hence, the space taken by *lockers* is $O(kLV)$.

To obtain the time bound, we will implement the set *lockers* as a table that is indexed by the location. Each element of the table is a list of triples $\langle a, b, H\rangle$. Note that the length of each list is at most L. When a location l is accessed, $O(L)$ set operations and $O(L)$ series-parallel queries are performed. Each set operation takes time $O(k)$ and each series-parallel query takes constant time. Hence, the time per call to ACCESS is $O(kL)$. □

Parallelizing the extended `All-Sets` algorithm

This section presents `Parallel All-Sets`, which is a parallelization of the extended `All-Sets` algorithm. This algorithm is *parallel* in the sense that it maintains the access-history correctly while allowing for parallel execution of the program. The following notation is used in my presentation of the algorithm.

Definition Let e_1 and e_2 denote any two threads. We say that e_1 `isLeftOf` e_2 if e_1 is visited *before* e_2 in a left-to-right walk of the parse tree.

Definition Let e_1 and e_2 denote any two threads. We say that e_1 `isRightOf` e_2 if e_1 is visited *after* e_2 in a left-to-right walk of the parse tree.

Figures 3-2, 3-3, and 3-4 show pseudocode that constitutes the `Parallel All-Sets` algorithm. To allow for a parallel execution of the program, we combine the serial `Extended All-Sets` algorithm with an approach similar to that used by Mellor-Crummey [12]: we keep "leftmost" and "rightmost" thread ID's for each (lock-set,location) pair, and we keep "primary" and "secondary" versions for both the "leftmost" and "rightmost" accesses.

```
ACCESS($\langle e, H, l\rangle$)
1  for each $\langle a_L, b_L, a_R, b_R, H', l\rangle \in lockers$
2       do if $(H' \cap H = \{\})$ and $(a_L \| e$ or $b_L \| e$ or $a_R \| e$ or $b_R \| e)$
3             then declare a data race
4  if $\exists$ thread ID's $a_L, b_L, a_R, b_R$
         such that $\langle a_L, b_L, a_R, b_R, H, l\rangle \in lockers$
5     then UPDATELEFT($\langle e, H, l\rangle$)
6          UPDATERIGHT($\langle e, H, l\rangle$)
7     else $lockers \leftarrow lockers \cup \{\langle e, initial, e, initial, H, l\rangle\}$
```

Figure 3-2: Pseudocode for the `Parallel All-Sets` algorithm.

```
UPDATELEFT($\langle e, H, l\rangle$)
1  Let $a_L, b_L, a_R, b_R$ be thread ID's
      such that $\langle a_L, b_L, a_R, b_R, H, l\rangle \in lockers$

2  if $(a_L \prec e)$ or $(e$ isLeftOf $a_L)$
3     then if $frame(e) = frame(a_L)$
4             then $lockers \leftarrow \big(lockers - \{\langle a_L, b_L, a_R, b_R, H, l\rangle\}\big)$
                        $\cup \{\langle e, b_L, a_R, b_R, H, l\rangle\}$
5             else $lockers \leftarrow \big(lockers - \{\langle a_L, b_L, a_R, b_R, H, l\rangle\}\big)$
                        $\cup \{\langle e, a_L, a_R, b_R, H, l\rangle\}$
6  elseif $((b_L \prec e)$ or $(e$ isLeftOf $b_L))$ and $(frame(e) \neq frame(a_L))$
7     then $lockers \leftarrow \big(lockers - \{\langle a_L, b_L, a_R, b_R, H, l\rangle\}\big)$
                 $\cup \{\langle a_L, e, a_R, b_R, H, l\rangle\}$
```

Figure 3-3: Pseudocode for UPDATELEFT of `Parallel All-Sets`.

Each locker contains the thread ID's $a_L, b_L, a_R,$ and b_R which are thought of as, respectively, the "leftmost primary," "leftmost secondary," "rightmost primary," and

```
UPDATERIGHT($\langle e, H, l\rangle$)
1  Let $a_L, b_L, a_R, b_R$ be thread ID's
       such that $\langle a_L, b_L, a_R, b_R, H, l\rangle \in lockers$

2  if $e$ isRightOf $a_R$
3     then if $frame(e) = frame(a_R)$
4              then $lockers \leftarrow \big(lockers - \{\langle a_L, b_L, a_R, b_R, H, l\rangle\}\big)$
                                    $\cup \{\langle a_L, b_L, e, b_R, H, l\rangle\}$
5              else  $lockers \leftarrow \big(lockers - \{\langle a_L, b_L, a_R, b_R, H, l\rangle\}\big)$
                                    $\cup \{\langle a_L, b_L, e, a_R, H, l\rangle\}$
6  elseif ($e$ isRightOf $b_R$) and ($frame(e) \neq frame(a_R)$)
7     then $lockers \leftarrow \big(lockers - \{\langle a_L, b_L, a_R, b_R, H, l\rangle\}\big)$
                        $\cup \{\langle a_L, b_L, a_R, e, H, l\rangle\}$
```

Figure 3-4: Pseudocode for UPDATERIGHT of `Parallel All-Sets`.

"rightmost secondary" thread ID's. In UPDATELEFT the thread ID's a_L and b_L are updated, and in UPDATERIGHT the thread ID's a_R and b_R are updated. We now update our notion of "strongly represents" as follows.

Definition Consider some point during an execution of the program. Let $\langle e, H, l\rangle$, $\langle a_L, H, l\rangle$, $\langle b_L, H, l\rangle$, $\langle a_R, H, l\rangle$, and $\langle b_R, H, l\rangle$ denote any five (not necessarily distinct) accesses that have occurred so far. We say that the locker $\langle a_L, b_L, a_R, b_R, H, l\rangle$ ***strongly represents*** the access $\langle e, H, l\rangle$ if for any thread f that executes in the future, we have that $e \| f$ implies that either $a_L \| f$ or $b_L \| f$ or $a_R \| f$ or $b_R \| f$.

The following theorem asserts the correctness of the `Parallel All-Sets` algorithm.

Theorem 10 *The following invariant holds after each access. For any location l and any lockset H, for any access $\langle d, H, l\rangle$ that has occurred so far, there exists a locker $\langle a_L, b_L, a_R, b_R, H, l\rangle \in lockers$ that strongly represents the access $\langle d, H, l\rangle$.*

Proof sketch. Rather than providing a formal, lengthy proof of the theorem, we will provide some intuition. Consider some point during an execution of the program. For any location l and any lockset H, the thread ID's a_L and b_L in the locker

$\langle a_L, b_L, a_R, b_R, H, l \rangle$ are those that would be present under an execution of the serial `Extended All-Sets` algorithm on a left-to-right execution of the threads that have occured so far. The thread ID's a_R and b_R in this locker are those that would be present under an execution of the serial `Extended All-Sets` algorithm on a left-to-right execution of the threads that have occured so far *with the left and right branches of all P-nodes swapped.*

Consider some access $\langle d, H, l \rangle$ that has occurred so far, and some future thread f such that $d \| f$. If f is to the right of d, then we have either $a_L \| f$ or $b_L \| f$. If f is to the left of d, then we have either $a_R \| f$ or $b_R \| f$. $\square$

The theorem directly implies that the `Parallel All-Sets` algorithm correctly detects a data race in a computation if and only if one exists.

The size of the access-history has grown by only a constant factor, but we need to deal with concurrent updates. In my current implementation, I simply have a lock for each location of the table, and the procedure ACCESS is performed while holding the relevant lock.

Chapter 4

Performance measurements of SP-hybrid updates

In this chapter, I provide some performance measurements of SP-hybrid updates. The experiments were run on an SMP system consisting of 4 1.4-megahertz AMD Opteron processors, each with 1024-kilobytes of L2 cache, 64-kilobytes of L1 data/instruction caches, running GNU/Linux.

Test Case			*Original Performance*		*With SP-hybrid updates*	
program	size	#processors	time(sec.)	speed-up	time(sec.)	speed-up
fib	37	1	19.6	1	50.4	1
		2	10.4	1.9	24.8	2.0
		3	7.01	2.8	18.4	2.7
		4	5.21	3.8	16.6	3.0
paraffins	23	1	5.81	1	15.6	1
		2	2.86	2.0	8.63	1.8
		3	1.96	3.0	6.82	2.3
		4	1.48	3.9	6.26	2.5
strassen	1024	1	4.23	1	9.44	1
		2	2.27	1.9	4.91	1.9
		3	1.61	2.6	3.41	2.8
		4	1.28	3.3	2.71	3.5

Figure 4-1: Performance measurements of SP-hybrid updates on a variety of test cases.

Figure 4-1 shows some recordings of performance. The first block of columns gives the various test cases used. The second block gives the performance recordings without the Nondeterminator turned on. The third block gives the performance recordings when the SP-hybrid updates are performed.

The most important column in this table is the final one, which shows the speed-up obtained with the SP-hybrid updates turned on. We see that the SP-hybrid updates seem to preserve the original parallelism of the user program. The anomalies are when the programs `fib` and `paraffins` are run on 4 processors — the speed-up in these cases are noticeably lower than the original speed-up.

During the performance testing, I observed that caching can have a significant impact on performance. At first, I had the `ND_state` records of the workers stored contiguously as an array. Since workers wrote to their respective `ND_state` records during each spawn and return, *false sharing* came into play — each such write by a processor would have invalidated the cache lines containing the array in each other processor. The speed-up for `fib` on 4 processors was only 1.6. Moving the `ND_state` records into the `CilkWorkerState` records nearly doubled the speed-up. It may be the case that such caching issues are the cause of the two anomalies mentioned above.

Unfortunately, I was unable to obtain performance measurements of the complete Nondeterminator-3 (that is, with the access-history enabled in addition to the SP-hybrid updates). The reason is because of an issue with porting to the 4-processor SMP system, which has word/address size of 64-bits. Some of the access-history code (which had been carried over from the Nondeterminator-2 implementation) performed address computations that had been hard-coded assuming 32-bit pointers, and I could not complete the porting in time.

Chapter 5

Related work

In this chapter, I summarize related work on race detection and SP-maintenance algorithms.

Static race detectors [14] analyse the text of a program to attempt to determine whether a program will ever produce a data race when run on all possible inputs. These detectors are conservative in the sense that they may report races that do not exist, since static analysis cannot fully determine the semantics of a program.

Most race detectors are dynamic tools, which detect potential races by executing the program on a given input. Some dynamic race detectors perform a post-mortem analysis based on program execution traces [7,11,13,15,16]. On-the-fly race detectors, like the one implemented in this thesis, detect races during execution of the program.

Netzer and Miller [17] distinguish between feasible races and apparent races. A ***feasible*** data race is one that can actually occur in an execution of the program. Netzer and Miller show that locating feasible data races in a general program is NP-hard. Most race detectors, such as the one implemented in this thesis, detect ***apparent*** races, which are an approximation of the races that may actually occur. These detectors typically ignore data dependencies that may make some apparent races infeasible.

Dinning and Schonberg's "lock-covers" algorithm [6] detects apparent races in programs that use locks. The All-Sets algorithm [4] of Cheng et al. improves the lock-covers algorithm by providing better time and space bounds. Cheng et al. also give a much more efficient algorithm, called Brelly, that detects violations of an "umbrella" locking discipline. The umbrella locking discipline precludes data races, and therefore Brelly can be used to detect data races in programs that obey this discipline. These algorithms require serial execution of the program. Mellor-Crummey [12] gives an access-history algorithm that can be used to detect determinacy races during a parallel execution of the program. The access-history algorithm used in the Nondeterminator-3 extends Mellor-Crummey's technique to support locks.

Savage et al. give an on-the-fly race detector called Eraser [19] that works on programs that have static threads, and enforces the simple locking discipline that a shared variable must be protected by a particular lock on every access. Eraser does not use an SP-maintenance algorithm, and hence it reports races between threads that operate in series. The umbrella discipline of Brelly is a generalization of Eraser's locking discipline. By keeping track of SP relationships, Brelly reports fewer spurious races.

Nudler and Rudolph introduced the English-Hebrew labelling scheme [18] for maintaining SP relationships. Their labels grow proportionally to the maximum concurrency of the program. Mellor-Crummey proposed the "offset-span" labelling scheme [12], which uses shorter label lengths but still not bounded by a constant.

Feng and Leiserson's SP-bags algorithm [8] is based on Tarjan's union-find data structure [20]. SP-bags inflates the space requirement by only a constant factor and the time by nearly as low as a constant factor. The time overhead is reduced to a constant factor in [9].

The SP-order algorithm [2] uses the technique of English-Hebrew labelling together with a centralized order-maintenance data-structure [1,5], and provides a constant factor inflation of time and space. The SP-hybrid algorithm [2,9] combines the SP-order and SP-bags algorithms, resulting in a SP-maintenance algorithm that runs in parallel.

Chapter 6

Concluding remarks

In this section, I offer some concluding remarks.

My implementation of the Nondeterminator-3 has some ineffiencies that are due to locking. Firstly, the order-maintenance data-structure used by my implementation of the SP-hybrid algorithm is currently a serial version which is locked on all operations. The SP-hybrid paper [2] gives a scheme for avoiding locking on queries, but in practice a more complicated scheme is necessary when implementing on machines that do not provide a guarantee of sequential consistency. Secondly, in the parallelized ALL-SETS algorithm, when an access to location l occurs, the access history list corresponding to location l must be locked. Finally, we also need to lock the SP-maintenance data-structures during garbage-collection. It would be interesting to see whether wait-free approaches lead to better speed-up in practice.

I believe that the `Extended All-Sets` algorithm can be generalized to obtain a more efficient way of supporting fake locks. The Nondeterminator data-race detectors provide fake locks to allow the user to protect accesses involved in apparent but infeasible races. Currently, fake locks are treated just like any other lock, and are placed in locksets. Using fake locks local to procedure instances could potentially cause a large blow-up in the number of distinct locksets. If we restrict to one the number of fake locks that can protect the same access, then we can generalize the `Extended All-Sets` algorithm by simply making a small change that would involve constructing a locker out of two accesses that held different fake locks.

It may also be desirable to allow the user to specify, via fake locks, which sets of inlets of a procedure instance commute with each other. Although Cilk guarantees that the inlets within a procedure instance commute with each other and with the parent procedure instance, the code may not commute. Therefore, a better alternative might be to allow the user to explicitly specify the atomicity guarantees that he/she assumes, via fake locks. If the sets of inlets that commute with each other are disjoint, then this stronger guarantee can be made by using at most one fake lock per access, and thus the `Extended All-Sets` algorithm could still be used.

It would also be desirable to extend the Brelly algorithm to correctly support the inlet feature of Cilk and to obtain a version that allows for parallel execution of the program.

Bibliography

[1] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 133–144, Barcelona, Spain, June 27–30, 2004.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[4] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, Puerto Vallarta, Mexico, June 28–July 2, 1998.

[5] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages

365–372, New York City, May 1987.

[6] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991.

[7] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '91*, pages 580–588, November 1991.

[8] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 22–25 1997.

[9] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, August 2005.

[10] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[11] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Analyzing traces with anonymous synchronization. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II70–II77, August 1990.

[12] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[13] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI), pages 135–144, Atlanta, Georgia, June 1988.

[14] Greg Nelson, K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Extended static checking home page, 1996.
`http://www.research.digital.com/SRC/esc/Esc.html`

[15] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[16] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II: 93–97, August 1990.

[17] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[18] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[20] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.