Labeling Schemes for Improving Cilksan Performance

by

Satya Holla

S.B. Computer Science and Engineering and Mathematics
Massachusetts Institute of Technology, 2024

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 2024

Authored by:	Satya Holla Department of Electrical Engineering and Computer Science August 17, 2024
Certified by:	Tao B. Schardl Research Scientist at CSAIL, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Labeling Schemes for Improving Cilksan Performance

by

Satya Holla

Submitted to the Department of Electrical Engineering and Computer Science
on August 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

While race detection algorithms like SP-bags have provably good theoretical properties, large overheads exist in practice, which urges the need for performance optimization. In this thesis, I propose labeling schemes as a method of circumventing many of the expensive operations in Cilksan, an implementation of the SP-bags algorithm. The proposed labeling schemes give strands of a parallel program labels during the execution of Cilksan, allowing Cilksan to shortcut the processing of certain memory accesses if the label comparison allows. I describe and prove correctness for two labeling schemes, the procedure labeling scheme and the prefix labeling scheme, implement both in Cilksan, and measure their performance. While the results show that the overhead of maintaining labels is too high in my implementation, the labeling schemes manage to circumvent many of the memory access operations, suggesting the merit of a more performant implementation of the same schemes.

Thesis supervisor: Tao B. Schardl
Title: Research Scientist at CSAIL

Acknowledgments

A heartfelt thanks to my wonderful advisor, Tao B. Schardl, for making my experience working with him educational and fun, and my friends and family, who helped keep me on track writing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cilk is a programming language which allows programmers to specify areas of code in their program which may run in parallel, after which its runtime system uses this information to efficiently schedule program instructions to processors. Cilk's model of parallel programming comes with an associated tool, Cilksan, which analyzes Cilk programs to find bugs called determinacy races. Cilksan has uniquely strong theoretical guarantees: namely, it is provably good (i.e., it is guaranteed to find a determinacy race in a program execution if one exists), and it has excellent asymptotic time and space complexity. However, many of the core operations involve time-intensive pointer chasing, and high overhead has been observed in practice. As it stands, it is often the case that some expensive checks performed by Cilksan are unnecessary. In this thesis, I introduce labeling schemes, which keep track of information on memory accesses and eliminate some of these unnecessary operations, aiming to improve the practical performance of Cilksan. In doing so, I prove the correctness of the presented labeling schemes and benchmark their effectiveness on several parallel programs written in Cilk.

```
cilk_scope {
    cilk_spawn foo();
    bar();
}
```

Figure 1.1: Example usage of Cilk parallel constructs. The `cilk_spawn` allows `foo()` to run in parallel with `bar()`, and the enclosing `cilk_scope` ensures both tasks have completed before control exits the scope.

1.1 Cilk

Cilk is a popular programming language developed in the 1990s at MIT, which extends C and C++ to allow for multi-threaded parallelism [1]. Cilk provides a wrapper around thread creation and scheduling procedures in C with two basic constructs:

- `cilk_spawn` is a keyword used before a function call, which allows the function to execute in parallel with the remainder of the code in the scope designated by a `cilk_scope`.
- `cilk_spawn` statements may occur within a scope designated by curly braces preceded by the keyword `cilk_scope`. At the end of the braces, a **sync** occurs, in which the program waits for all parallel threads spawned within the scope to complete before proceeding.

Figure 1.1 illustrates the use of these constructs. In the snippet, the functions `foo` and `bar` are allowed to run in parallel (i.e., they are **logically in parallel**). Notice that this description avoids saying that `cilk_spawn` actually creates a new thread, and instead says that it “allows" code to run in parallel. This distinction is intentional, as Cilk separates the specification of logical parallelism with the actual allocation and scheduling of threads. While the programmer is expected to specify (using the constructs mentioned above) which parts of the code *can* run in parallel, the Cilk runtime system is responsible for deciding how this code should be handled by the resources of the machine. In particular, Cilk may decide against parallelizing a spawn at runtime if all existing worker threads already have parallel work.

```
int main() {
    int x = 0;
    cilk_scope {
        cilk_spawn increment(x);
        increment(x);
    }
}
```

```
void increment(int &x) {
    int y = x + 1;
    x = y;
    // Equivalently, we could have
    // x = x + 1; but this makes the
    // read and write of x more explicit.
}
```

Figure 1.2: Example determinacy race on the variable `x`. Both calls to `increment(x)` are logically in parallel, and both calls read and write the value of `x`, leading to nondeterministic behavior.

1.2 Cilksan and race detection

A variety of tools exist for Cilk, including Cilksan, a program which can detect determinacy races, which are often bugs [2]. Cilksan uses the SP-bags algorithm [3] to guarantee detection of determinacy races on given inputs to the program.

Determinacy races occur when two regions of code access the same data with the access order not guaranteed, leading to nondeterminism in the execution of the program [2]. What this means depends on the model of parallelism; for our purposes, they arise when two logically parallel regions access the same memory location and at least one of the accesses is a write. In this case, the execution of the program is dependent on which of the two regions accesses the race memory location first, leading to the nondeterminism (although the code output may still be deterministic). Nondeterministic code execution is notoriously difficult to debug, making race detection in Cilk a valuable problem to tackle.

An example Cilk program which illustrates the dangers of determinacy races is shown in Figure 1.2. In the figure, a `cilk_scope` is created, with a call to `increment(x)` allowed to run in parallel with another call to `increment(x)`. Each call to `increment(x)` involves the following in order:

1. Read the value of x

2. Add one to this value, and write it to x

If the first call to `increment(x)` executes completely before the second call, or vice versa, the final value of `x` will be `2`. However, if the first call reads the value of `x` as `0`, then the second call also reads the value of `x` as `0`, and finally both calls compute the value of `y` as `1` and write this to `x`, the final value of `x` will be `1`. Since the two regions of code are executing in parallel, either case (or others) could possibly occur.

A variety of race detection algorithms exist [3]–[6], as well as practical optimizations to these [7]; these can be further categorized by what types of information they collect and use for detection. A popular strategy is **on-the-fly analysis** [4], whereby the race detector instruments the parallel program with code that detects and reports races during execution. Cilksan uses one such on-the-fly algorithm called SP-bags [3]. SP-bags runs a program's code serially, and tracks which areas of code are logically in series or logically in parallel with each other. Briefly, the algorithm maintains data structures for each region of code P, to track which other regions are logically in series and which other regions are logically in parallel with P. For each memory access, the data structure of the currently running code must be queried, to check whether the access might result in a determinacy race. While the asymptotic overhead of the SP-bags algorithm is good, the primary operations performed involve expensive pointer jumping, leading to large overhead. Section 2.2 explains the SP-bags algorithm in further detail.

1.3 My contributions

This thesis introduces **labeling schemes**, augmentations to SP-bags which store additional numerical values, called **labels**, during each memory access operation in Cilksan; these labels mark different points of the Cilk program execution. Before two memory accesses are compared in SP-bags to detect a potential race, the labels of each access are first used to more quickly determine whether the accesses are in series, and if it is therefore possible to shortcut the expensive SP-bags operation(s) involved in such a comparison.

The idea of using labels to track series/parallel relationships is not new. In [4], Offset-Span Labeling labels each thread with a sequence of ordered pairs whose length is proportional to the depth of the thread's parallel nesting. These labels are then used to determine whether any two threads are in series or parallel. In English-Hebrew Labeling [5], two labels which serve the same purpose as in Offset-Span Labeling are given to each "sequential block" of code. English-Hebrew Labeling requires the program to either be run twice, once to determine the control-flow structure and assign one set of labels and another to assign the second and track memory accesses, or otherwise use labels whose length is proportional to the depth of nesting.

The labeling schemes presented here differ in that they are not intended to fully solve the problem of determining series/parallel relationships between threads. Instead, they are intended only as a quick verification that two threads operate in series. Whenever two threads are determined to not necessarily be in series, the standard SP-bags operations must still be run. By limiting the scope of the labeling scheme in this way, we can optimize our schemes for practical performance, and we are able to use smaller labels than those in English-Hebrew Labeling or Offset-Span Labeling, while still discovering a large fraction of the pairs of non-racing memory accesses.

The contributions of this thesis are:

- Defining the new method of labeling schemes
- Presenting two labeling schemes, the procedure labeling scheme and prefix labeling scheme, and proving their correctness
- Describing implementations for the above schemes
- Using various measures of performance to analyze the effectiveness of these implementations

1.4 Outline

Chapter 2 presents background on the SP-bags algorithm, as well as important information about Cilksan. Chapter 3 contains the contributions of the paper, namely the two labeling schemes, their proofs, implementations, and performance analyses. Chapter 4 gives additional background on related works. Chapter 5 discusses possible future research in optimizing Cilksan.

Chapter 2

Background

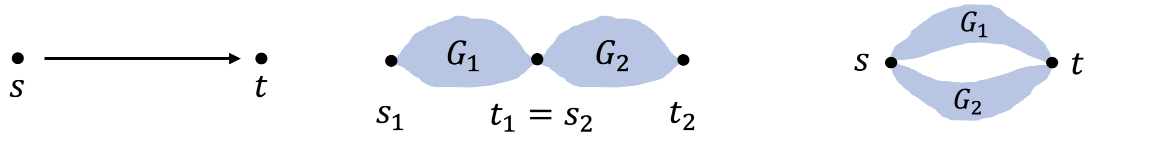
This chapter presents the background relevant to the later discussion of labeling schemes. Section 2.1 introduces a formalization of the structure of parallel programs useful for proving theoretical results. Section 2.2 states the SP-bags algorithm, as well as invariants maintained in the course of the algorithm. Finally, Section 2.3 discusses a deviation from the SP-bags algorithm found in Cilksan.

2.1 Cilk control-flow DAGs

The model of parallelism described by `cilk_spawn` and `cilk_scope`, **fork-join** parallelism, leads to the computation forming a highly structured DAG. In this section, I present the notion of a series-parallel DAG, and mention results showing their correspondence with Cilk control-flow DAGs.

To begin, we can subdivide Cilk programs to better understand their structure. The parallel constructs in a Cilk program break it up into units of instructions called **procedure instances**, which we will shorten to the term **procedure**.

Definition 1. *A* procedure *is a sequence (not necessarily continuous) of instructions in a Cilk program which begins at a* `cilk_spawn` *of a function (or, for the* main procedure*, at*



(a) Base (b) Series composition (c) Parallel composition

Figure 2.1: Each of the constructions of a series-parallel DAG

the start of the Cilk program), and ends when that function returns. During the runtime of procedure P, a ***cilk_spawn*** *of function instance x may create a new* subprocedure - *any instructions part of x's execution are not considered to be in procedure P.*

Each procedure can be further broken down into units called **strands**.

Definition 2. *A* strand *is a maximal continuous sequence of instructions in Cilk not containing any parallel constructs (i.e., not containing* `cilk_spawn`*,* `cilk_scope`*, or* `cilk_sync`*). As Figure 2.2 demonstrates, this is different from a procedure, since we consider* two *new strands to form when we encounter a spawn, the newly spawned strand and the* continuation strand.

In the Cilk model, the program can be represented by a **control-flow DAG** whose nodes represent locations in the code where logically parallel code is either spawned (with `cilk_spawn`) or synced (at the end of a `cilk_scope` scope), and whose edges are strands. Cilk control-flow DAGs turn out to have more structure than standard DAGs:

Definition 3. *A* series-parallel DAG *consists of a source node s, a sink node t, and is constructed recursively in one of the following ways:*

- *Base: the graph consists of a single edge from source s to sink t.*
- *Series composition: the graph G consists of two series-parallel DAGs G_1 and G_2 with disjoint edge sets, such that the source of G_1 is the source of G, the sink of G_2 is the sink of G, and the source of G_2 is the sink of G_1.*

```
...
int x = 0;                  // e_0
cilk_scope {
    x = 1;                  // e_0
    cilk_spawn foo();
    x = 2;                  // e_1
    cilk_spawn bar();
    x = 3;                  // e_2
}
x = 4;                      // e_3
...
```

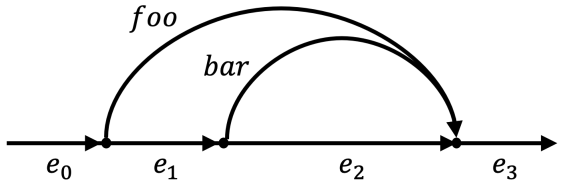


Figure 2.2: The code on the left corresponds to the series-parallel DAG on the right. In the DAG, each of e_0, e_1, e_2, and e_3 are strands in the same procedure. `foo` and `bar` are also procedures, which, as drawn, have no parallel constructs of their own, and hence contain only one thread.

- *Parallel composition: the graph G consists of two series-parallel DAGs G_1 and G_2 with disjoint edge sets, such that the sources of G_1 and G_2 are both the source of G, the sinks of G_1 and G_2 are both the sink of G.*

Note that we allow multiple edges between the same two vertices in these DAGs. Figure 2.1 illustrates each case above.

Theorem 1 ([3]). *A Cilk control-flow DAG is a series-parallel DAG.*

The following relationships between strands of the DAG will be useful to define for the following sections, in which I give correctness results of Cilksan's algorithm.

Definition 4. *Given a DAG corresponding to a Cilk computation,*

- *A strand e_1* precedes *another strand e_2, denoted $e_1 \prec e_2$, if there is a path in the DAG containing both e_1 and e_2 in that order. The $\prec$ relation is transitive.*

- *Strands e_1 and e_2 operate logically* in parallel, *denoted $e_1 \parallel e_2$, if $e_1 \not\prec e_2$ and $e_2 \not\prec e_1$.*

Algorithm 1 SP-bags

```
S/P-bag maintenance:
spawn of procedure $F$
    $S_F \leftarrow \textsc{Make-Set}(F)$    ▷ This operation creates an empty union-find data structure
    $P_F \leftarrow \emptyset$
sync in procedure $F$
    $S_F \leftarrow \textsc{Union}(S_F, P_F)$
    $P_F \leftarrow \emptyset$
return from spawned $F'$ to $F$
    $P_F \leftarrow \textsc{Union}(S_{F'}, P_F)$

Shadow memory maintenance:
write to shared location $\ell$ by procedure $F$
    if $\textsc{Find-Set}(reader(\ell))$ is a P-bag or $\textsc{Find-Set}(writer(\ell))$ is a P-bag then
        Report race
    $writer(\ell) \leftarrow F$
read of shared location $\ell$ by procedure $F$
    if $\textsc{Find-Set}(writer(\ell))$ is a P-bag then
        Report race
    if $\textsc{Find-Set}(reader(\ell))$ is an S-bag then
        $reader(\ell) \leftarrow F$
```

2.2 The SP-bags algorithm

This section describes the SP-bags algorithm [3], states theorems regarding its correctness, and provides reasons for its poor performance in practice.

The SP-bags algorithm runs serially, despite its purpose in analyzing parallel programs. It processes the input Cilk program depth-first: that is, when encountering a `cilk_spawn`, it will explore the entire subcomputation stemming from this spawn before exploring the continuation. This order corresponds with how the input program would run on a single processor, without any parallel constructs. Algorithm 1 details the full algorithm, and is separated into two core routines which maintain information about the currently active procedures while processing the program.

The first of these routines is the S/P-bag maintenance, which tracks which procedures are in series/parallel with the current thread. When the SP-bags algorithm encounters a new subprocedure F via a `cilk_spawn`, it creates two new union-find data structures for the subprocedure called the **S-bag** and the **P-bag**, denoted S_F and P_F respectively. Upon processing parallel constructs, it modifies the S and P-bags with the three standard union-find operations MAKE-SET, UNION, and FIND-SET, in order to maintain the following invariant: When exploring any strand e, consider any previously explored strand f and its procedure P_f. If f is logically in series with e, then P_f is in an S-bag, and if f is logically in parallel with e, then P_f is in a P-bag. The exploration order of SP-bags guarantees that both of these cases will not simultaneously occur for a procedure.

The second routine is the shadow memory maintenance, which keeps track of previous accesses of each shared memory location in a structure called the **shadow memory**. The shadow memory is composed of two dictionaries called *reader* and *writer*. Each of these maintains, for every shared memory location ℓ in the original Cilk program, the procedure ID[1] of a procedure which previously read (resp: wrote) to ℓ. We refer to the procedure IDs stored in the shadow memory for location ℓ as $reader(\ell)$ and $writer(\ell)$. The operations on the shadow memory maintain the property that $writer(\ell)$ always stores the most recent writer to ℓ, and $reader(\ell)$ always stores some reader logically in parallel with the current strand if one exists - otherwise, it stores the most recent reader. It turns out that storing only these procedures at any given time for a memory location ℓ, rather than the entire history of accesses, suffices to determine if there is a race on ℓ, by checking for each new read whether $writer(\ell)$ is in a P-bag, and for each new write whether either $writer(\ell)$ or $reader(\ell)$ is in a P-bag.

I now state a result concerning the runtime of SP-bags.

Theorem 2 (Theorem 1 in [3]). *Consider a Cilk program that executes in time T on one processor and references ν shared memory locations. The SP-bags algorithm can be implemented to check this program for determinacy races in $O(T\alpha(\nu,\nu))$ time using $O(\nu)$ space, where α denotes the functional inverse of the Ackermann function.*

[1]Cilksan assigns these at runtime

This theorem shows that SP-bags is a nearly asymptotically optimal serially run race detector, as any serial race detector requires $\Omega(T)$ time. To see why, imagine a program which produces some Boolean output in optimal time T, then creates a race if this output is true - now, the output of the program should be the same as the output of the race detector, and this output again requires at least T time. While race detectors like SP-order [6], which runs in $O(T)$ time, achieve this optimal runtime, the extremely slow-growing factor of $\alpha(\nu, \nu)$ which separates the two algorithms can be considered constant for all practical purposes.

Rather than improving asymptotic runtime, this work focuses on reducing the number of S/P-bag operations done by the algorithm during reads and writes. These operations involve union-find data structures and require traversing trees. Despite good theoretical performance, the tree operations involve pointer chasing and therefore do not cache well, which may be the cause of Cilksan's slow runtime in practice. Empirical evidence agrees with this prediction - performance analysis on the same benchmarks used in Section 3.4 suggests that on average, Cilksan spends half its total time on reads and writes. Chapter 3 describes the approach of labeling schemes, which may allow SP-bags to shortcut these expensive operations.

2.3 Nested `cilk_scope`s and Cilk functions

Differences between Cilk and the model of computation proposed in the SP-bags paper [3] lead to several deviations seen in Cilksan from the SP-bags algorithm. This section discusses one such deviation, nested `cilk_scope`s, which significantly impacts the discussion of labeling schemes in Chapter 3.

The SP-bags paper [3] never mentions the parallel construct `cilk_scope`. Instead, they use a different sync construct, `cilk_sync`, which syncs *all* parallel threads spawned by the calling procedure. While the two are quite similar, an important difference is that `cilk_scope` allows for nested synchronization within a single procedure. For example, con-

```
 1 int x = 0;
 2
 3 void A() {
 4     x = 2;
 5 }
 6
 7 void CF() {
 8     cilk_scope {
 9         cilk_spawn B();
10         printf("nested");
11     }
12 }
13
14 int main() {
15     cilk_scope {
16         cilk_spawn A();
17         CF();
18         x = 3;
19     }
20 }
```

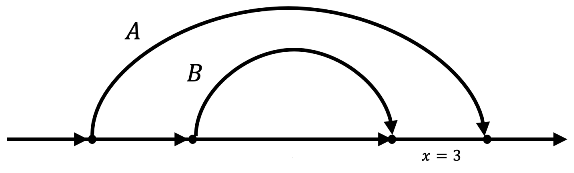


Figure 2.3: In the code on the left, a `cilk_scope` is nested inside another one, leading to the control flow DAG seen on the right.

sider the code in Figure 2.3. While inside a `cilk_scope`, we call another function containing a `cilk_scope`[2], leading to control-flow structure we could not have achieved using just `cilk_spawn` and `cilk_sync`.

Such nesting of `cilk_scopes` can lead to correctness issues if we attempt to naively apply SP-bags. Since the write to `x` in line 4 and the write to `x` in line 18 are logically in parallel, there is a race. However, suppose we try using the SP-bags algorithm and simply treat the end of a `cilk_scope` as a `cilk_sync`. We start with the `main()` procedure, then explore the procedure spawned at `A()`, return to `main()`, then enter the function `CF()`, which contains its own `cilk_scope` and spawned procedure `B()`. After line 10, both the `A()` and the `B()` procedures are in `main()`'s P-bag. Then, in line 11, a sync occurs at the end of `CF`'s `cilk_scope`, which causes `A()` and `B()` to move to `main()`'s S-bag. This causes the error

[2]It is also possible to directly nest two `cilk_scope`s, which leads to different behavior. However, this is not important to understand the remainder of the thesis, so we make no further mention of it.

- we still have not synced `A()` yet, but the sync indiscriminately moves the *entire* P-bag of a procedure to the S-bag, instead of just those procedures called within the scope. Now, in line 18, a race will not be detected.

Cilksan solves this by introducing **Cilk functions**, functions with any Cilk construct inside their body. When a Cilk function is called, Cilksan treats the call like a spawn to a new procedure whether or not the Cilk function was actually spawned, and creates a new **frame**, the Cilksan structure intended to track SP-bags procedures. In doing this, the newly created frame will have its own separate P-bag, so that syncs done within the body of the new frame will not affect the P-bag of the calling frame.

Algorithm 2 presents simplified version of what Cilksan does differently from SP-bags in this respect. With the exception of the routines involving Cilk functions in S/P-bag maintenance, Algorithm 2 and Algorithm 1 are identical.

For the remainder of the chapter, I use the following conventions:

- A *procedure* in a Cilk program retains the original definition as used in the standard SP-bags algorithm, whereas a *frame* is the Cilksan equivalent of a procedure. The difference between the two lies in the fact that Cilksan creates a new frame when entering a Cilk function, whereas the procedure remains the same.

- Before, we defined a strand as a maximal continuous sequence of instructions not containing a parallel construct. Since Cilk functions create new frames, we include entry into a Cilk function as a parallel construct, so that a new strand forms during this event.

Algorithm 2 Simplified model of Cilksan

```
S/P-bag maintenance:
spawn of frame $F$
    $S_F \leftarrow \textsc{Make-Set}(F)$
    $P_F \leftarrow \emptyset$
enter Cilk function frame $F'$ from frame $F$
    $S_{F'} \leftarrow \textsc{Make-Set}(F')$
    $P_{F'} \leftarrow \emptyset$
sync in frame $F$
    $S_F \leftarrow \textsc{Union}(S_F, P_F)$
    $P_F \leftarrow \emptyset$
return from spawned $F'$ to $F$
    $P_F \leftarrow \textsc{Union}(S_{F'}, P_F)$
return from Cilk function frame $F'$ to $F$
    $S_F \leftarrow \textsc{Union}(S_{F'}, S_F)$
```

```
Shadow memory maintenance:
write to shared location $\ell$ by procedure $F$
    if $\textsc{Find-Set}(reader(\ell))$ is a P-bag or $\textsc{Find-Set}(writer(\ell))$ is a P-bag then
        Report race
    $writer(\ell) \leftarrow F$
read of shared location $\ell$ by procedure $F$
    if $\textsc{Find-Set}(writer(\ell))$ is a P-bag then
        Report race
    if $\textsc{Find-Set}(reader(\ell))$ is an S-bag then
        $reader(\ell) \leftarrow F$
```

Chapter 3

Labeling schemes

This chapter presents the main contribution of the thesis, namely, the analysis of labeling schemes used to avoid unnecessary S/P-bag operations. Section 3.1 describes the motivation behind the labeling schemes along with useful definitions/conventions. Section 3.2 presents the procedure labeling scheme, along with its correctness proof and implementation details. Section 3.3 does the same for the prefix labeling scheme. Finally, Section 3.4 measures the labeling schemes' performance on various benchmark programs.

3.1 Motivation and definition

I begin by presenting an example program which illustrates the purpose of labeling schemes, and motivates their definition. Consider the Cilk program and associated control-flow DAG shown in Figure 3.1. Five total memory accesses take place, two of the variable `x` and three of the variable `y`. When run, Cilksan goes through the program and checks during each of the writes whether the previous accesses were from procedures in a P-bag or not. These include:

- A check while processing line 2 to see if the access in line 6 is in a P-bag
- A check while processing line 10 to see if the access in line 7 is in a P-bag

```
1  void foo(int& x) {
2      x = 5;
3  }
4
5  int main() {
6      int x = 3;
7      int y = 0;
8      cilk_scope {
9          cilk_spawn foo(x);
10         y = 1;
11     }
12     y = 2;
13 }
```

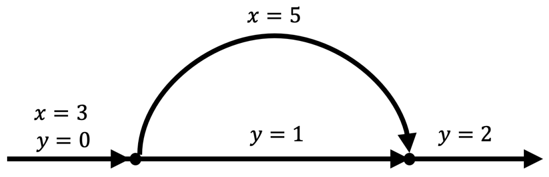


Figure 3.1: A Cilk program and associated control-flow DAG, with each strand's memory accesses depicted.

- A check while processing line 12 to see if the access in line 10 is in a P-bag

If we look at the example control-flow DAG, we see that some of these checks seem unnecessary. For example, the accesses in lines 7, 10, and 12 are from the *same procedure*. This means that they will always be logically in series. The line 2 access is from a subprocedure of the line 6 access, and occurs after the line 6 access. Examining the control-flow DAG shows that these two accesses will also always be in series.

In the above examples, we could use the information about the procedures and orders of these accesses to create a **fast path**, or shortcut, which bypasses the S/P-bag operations to check for races. Such fast paths can therefore reduce the performance overhead of Cilksan.

Labeling schemes are the method I will use in this thesis to implement the ideas above. By associating each strand with a value called a **label**, and storing the labels of procedures along with procedure IDs in shadow memory, we can use label information during memory accesses to determine if a fast path is possible. A formal definition is provided below:

Definition 5. *A* labeling scheme *consists of two modifications to the standard Cilksan implementation of SP-bags (see Algorithm 2):*

- *A modified S/P-bag maintenance algorithm which assigns labels to strands when parallel* *constructs are encountered.*

- *A modified shadow memory maintenance algorithm which, upon each memory access, compares the current strand's label to those in the shadow memory, creating a fast path if the label comparison determines that the two strands are in series.*

3.2 Procedure labeling scheme

The **procedure labeling scheme** labels each strand with the procedure that contains it. Because a `cilk_spawn` always starts a new procedure, two strands in the same procedure must always be in series, which is the crux of our fast path. Section 3.2.1 presents the algorithm pseudocode and proves its correctness, and Section 3.2.2 describes the structures in Cilksan used for implementation.

3.2.1 Algorithm and correctness

Algorithm 3 details the procedure labeling scheme pseudocode, Figure 3.2 shows an example of its use. The key modification in the S/P-bag maintenance is the assignment of labels to each new frame. When a spawn creates a new frame, the frame gets a new label generated from incrementing a global counter variable. However, when a Cilk function entry creates a frame, the frame inherits its label from the calling frame, as entry into a Cilk function does not create a new procedure. In the shadow memory maintenance, the modification is seen in lines 16, 23, and 25, which feature fast paths that allows us to circumvent FIND-SET operations[1].

I begin by proving some useful intermediate lemmas, then prove the correctness of Algorithm 3.

Lemma 1. *For any shadow memory location ℓ, after any call to Algorithm 3, $writer(\ell).label = wlabel(\ell)$ and $reader(\ell).label = rlabel(\ell)$.*

[1]Note that we assume that Boolean operations are short-circuited

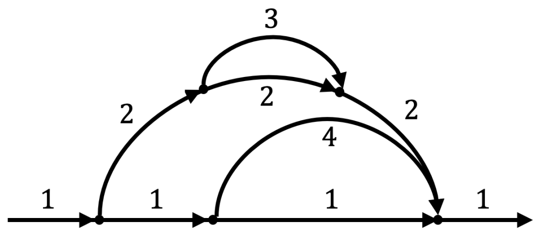


Figure 3.2: An example Cilk control-flow DAG illustrating the procedure labeling scheme

Algorithm 3 Prodecure labeling scheme

```
1:  **S/P-bag maintenance:**
2:  spawn of frame $F$
3:      $S_F \leftarrow$ MAKE-SET($F$)
4:      $P_F \leftarrow \emptyset$
5:      $globalLabelCt \leftarrow globalLabelCt + 1$          ▷ Each new procedure gets a new label
6:      $F.label = globalLabelCt$
7:  enter Cilk function frame $F'$ from frame $F$
8:      $S_{F'} \leftarrow$ MAKE-SET($F'$)
9:      $P_{F'} \leftarrow \emptyset$
10:     $F'.label = F.label$
11: sync in frame $F$, return from spawn, or return from Cilk function
12:     Same as in Algorithm 2
13: ________________________________________
14: **Shadow memory maintenance:**
15: write to shared location $\ell$ by procedure $F$
16:     if $rlabel(\ell) = wlabel(\ell) = F.label$ then
17:         continue
18:     if FIND-SET($reader(\ell)$) is a P-bag or FIND-SET($writer(\ell)$) is a P-bag then
19:         Report race
20:     $wlabel(\ell) \leftarrow F.label$
21:     $writer(\ell) \leftarrow F$
22: read of shared location $\ell$ by procedure $F$
23:     if $wlabel(\ell) \neq F.label$ and FIND-SET($writer(\ell)$) is a P-bag then
24:         Report race
25:     if $rlabel(\ell) \neq F.label$ and FIND-SET($reader(\ell)$) is an S-bag then
26:         $rlabel(\ell) \leftarrow F.label$
27:         $reader(\ell) \leftarrow F$
```

Proof. This follows from the fact that the algorithm always update $wlabel(\ell)$ at the same time as $writer(\ell)$, and similarly for $reader(\ell)$ and $rlabel(\ell)$. We use this lemma implicitly in statements like "the last strand to update $wlabel(\ell)$ and $writer(\ell)$". □

Lemma 2. *For strands $e \neq f$, $e.label = f.label$ if and only if e and f are contained in the same procedure, and are therefore logically in series.*

Proof. In the procedure labeling scheme, lines 5-6 gives a new label generated from incrementing the global counter to a strand upon a `cilk_spawn`, which ensures that two strands from different procedures will not have the same label. Since this is the only time we generate a new label (in the case of entry to a Cilk function, the label is inherited from the calling strand), strands in the same procedure must have the same label. □

Lemma 3. *For strands e, f, and g, if $e \parallel g$, then $e \prec f$ and $f \prec g$ cannot both be true.*

Proof. Immediate consequence of the transitivity of the $\prec$ relation. □

Lemma 4. *If strands e, f, and g are explored in that order, $e.label = f.label$, and $f \parallel g$, then $e \parallel g$.*

Proof. Suppose not, and $e \nparallel g$. Then either $g \prec e$ (which is impossible because e is explored before g), or $e \prec g$. In the latter case, we know from Lemma 2 and the exploration order that e and f are from the same procedure, with $e \prec f$. Since $f \parallel g$, g and f (and therefore g and e) are from different procedures, there must be some `cilk_spawn` leading to g encountered on e's procedure, which occurs after e is explored but before f is explored. But then the exploration order is e, g, f, a contradiction. □

Lemma 5 (Lemma 7 in [3]). *If strands e, f, and g are explored in that order, then*

$$(e \parallel f) \wedge (f \parallel g) \implies e \parallel g$$

Theorem 3. *The procedure labeling scheme correctly detects determinacy races.*

Proof. Armed with these lemmas, I first show that if a determinacy race exists, it will be detected. Consider strands x and y which both access memory location ℓ, with at least one access being a write. Suppose x precedes y in the depth-first exploration, and let us consider the earliest y with a determinacy race with x at ℓ. Consider each case:

1. Suppose the memory access in x is a write. Then, I show the following claims in order:

   (a) Immediately after x writes to ℓ, the last strand to have updated $\mathit{writer}(\ell)$ and $\mathit{wlabel}(\ell)$ is logically in parallel with y.

   - Let this last strand be w. Then, $w.\mathit{label} = x.\mathit{label}$, since for any write, line 20 updates $\mathit{wlabel}(\ell)$ to the label of the writer if they differ. Substituting $(w, x, y) \to (e, f, g)$ in Lemma 4 yields the result.

   (b) Immediately before y writes to ℓ, $\mathit{writer}(\ell)$ is in a P-bag, and $\mathit{wlabel}(\ell) \neq y.\mathit{label}$.

   - First, notice that if $\mathit{writer}(\ell)$ is in a P-bag, Lemmas 1 and 2 imply $\mathit{wlabel}(\ell) \neq y.\mathit{label}$. To prove that $\mathit{writer}(\ell)$ is in a P-bag, suppose not. Then, by (a), there must be some strand w explored after x which updated $\mathit{writer}(\ell)$, with $w \prec y$. Substituting $(x, w, y) \to (e, f, g)$ in Lemma 3, we find that $x \parallel w$, which contradicts y being the earliest strand racing with x at ℓ.

2. Otherwise, x's access is a read and y's access is a write. We can prove statements analogous to those in the first case:

   (a) Immediately after x's read is executed, the last strand to have updated $\mathit{reader}(\ell)$ and $\mathit{rlabel}(\ell)$ is logically in parallel with y.

   - When x writes to ℓ, lines 26-27 update $\mathit{reader}(\ell)$ and $\mathit{rlabel}(\ell)$ with the exception of two cases. One is if the previous strand w to update these is in a P-bag, in which case Lemma 5 applies, and $(w \parallel x) \wedge (x \parallel y) \implies (w \parallel y)$. The other is if the previous strand w satisfies $w.\mathit{label} = x.\mathit{label}$, in which case Lemma 4 applies, and again $w \parallel y$.

(b) Immediately before y writes to ℓ, $reader(\ell)$ is in a P-bag, and $rlabel(\ell) \neq y.label$,

- Again, we only need to prove the first statement, as it implies the second. Suppose $reader(\ell)$ is in an S-bag (and is therefore in series with x). Then, by (a), there must be some strand explored after x which updated $rlabel(\ell)$, such that it is in series with y. Let w be the earliest strand to be explored out of these. Now, our contradiction arises from the fact that w can only update $reader(\ell)$ if the previous entry was in an S-bag. To be precise, when w writes to ℓ, the previous entry in $reader(\ell)$ must be in a P-bag, as otherwise, we would have an even earlier strand in series with y which updated $reader(\ell)$ after x is explored.

In both cases, the final claim immediately proves that a determinacy race will be detected during y's memory access.

I now prove the opposite direction, that no determinacy race will be detected if none exists. Consider any execution which reports a race on location ℓ. Let us ignore memory locations other than ℓ, as different memory locations do not interact, and assume that the execution ends with the first race report of ℓ. First, notice that writes which were processed with a fast path do not change the state of the algorithm, so we can simply remove any fast path writes from the execution while maintaining the existence of a race report. Similarly, we can remove any reads for which a fast path skipped updates to $reader(\ell)$. Finally, any reads for which a fast path skipped the race report at ℓ can be considered to not have this fast path, as a race report of ℓ occurs in the execution anyway. What remains is a normal SP bags execution with a race at ℓ - by correctness of SP bags, there must be a race. □

3.2.2 Implementation

This section describes the implementation of the procedure labeling scheme in Cilksan [8].

First, I briefly describe some relevant data types in Cilksan:

- A `FrameData_t` object represents a frame in Cilksan.
- A `SimpleShadowMem` object represents the shadow memory.
- Each `SimpleShadowMem` contains one `SimpleDictionary` object representing *reader* and another representing *writer*.
- Each `SimpleDictionary` contains a table of `Page_t` objects which together encapsulate the memory the program uses. While `Page_t` objects represent large chunks, 2^{30} bytes to be precise, of contiguous memory, they do not necessarily align with the pages used by the operating system. The `Page_t` objects store procedure IDs during memory accesses as described in SP-bags.

In my implementation of the procedure labeling scheme, each `Page_t` object maintains an array of 2^{30} 64-bit labels in which we may store labels during any memory access to any of the `Page_t` object's memory locations. While using 64-bit labels seems inordinate, 32-bits is insufficient, as there could feasibly be programs with more than $2^{32} \approx 4 \times 10^9$ spawns.

Each `FrameData_t` object has a label attribute. `FrameData_t` objects generated by a spawn have a new label generated during their creation from an incrementing global counter, whereas `FrameData_t` objects generated from entry into a Cilk function receive their label from their parent frame, just as in Algorithm 3. The logic for the fast paths resides in a function called `checkAndSetLabels`, which is called by Cilksan's read and write functions and reports whether these functions can return early. `checkAndSetLabels` handles large reads and writes by sequentially checking/updating each memory location in turn within the `Page_t` object.

3.3 Prefix labeling scheme

In the prefix labeling scheme, strands are labeled with vectors. The goal is for $e \prec f$ to be true iff *e.label* is a prefix of *f.label*, but the given algorithm ends up with a small exception to this. Section 3.3.1 describes the algorithm in pseudocode and presents theoretical results, and Section 3.3.2 describes the implementation of the scheme in Cilksan.

3.3.1 Algorithm and correctness

Algorithm 4 Prefix labeling scheme

```
1:  S/P-bag maintenance:
2:  spawn of frame $F$
3:      $S_F \leftarrow$ MAKE-SET($F$)
4:      $P_F \leftarrow \emptyset$
5:      $F'.label = F.label + [F.numChildren]$        ▷ This signifies appending to an array
6:      $F.numChildren = F.numChildren + 1$
7:  enter Cilk function frame $F'$ from frame $F$
8:      $S_{F'} \leftarrow$ MAKE-SET($F'$)
9:      $P_{F'} \leftarrow \emptyset$
10:     $F'.label = F.label$
11: sync in frame $F$, return from spawn, or return from Cilk function
12:     Same as in Algorithm 2
13: ________________________________________
14: Shadow memory maintenance:
15: write to shared location $\ell$ by procedure $F$
16:     if $wlabel(\ell) \neq F.label$ then
17:         $wlabel(\ell) \leftarrow F.label$
18:         $writer(\ell) \leftarrow F$
19:     if $rlabel(\ell)$ is a prefix of $F.label$ and $wlabel(\ell)$ is a prefix of $F.label$ then
20:         continue
21:     if FIND-SET($reader(\ell)$) is a P-bag or FIND-SET($writer(\ell)$) is a P-bag then
22:         Report race
23: read of shared location $\ell$ by procedure $F$
24:     if $wlabel(\ell)$ is not a prefix of $F.label$ and FIND-SET($writer(\ell)$) is a P-bag then
25:         Report race
26:     if $rlabel(\ell) \neq F.label$ and FIND-SET($reader(\ell)$) is an S-bag then
27:         $rlabel(\ell) \leftarrow F.label$
28:         $reader(\ell) \leftarrow F$
```

Algorithm 4 outlines the prefix labeling scheme. As before, we generate a new label only when a frame is spawned, rather than when a Cilk function is entered, so that again all strands within the same procedure share a label. However, unlike in the procedure labeling scheme, the prefix labeling scheme attempts to encode even more information into labels.

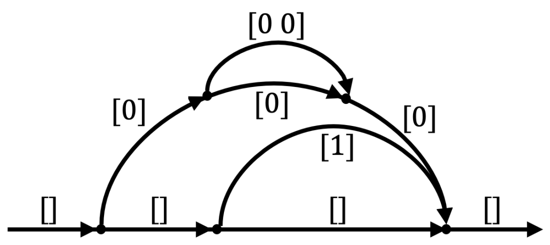


Figure 3.3: An example Cilk control-flow DAG illustrating the prefix labeling scheme

When `cilk_spawn` spawns a frame F' from frame F, line 5 gives F' a label one element longer than $F.label$; this new label contains $F.label$ as a prefix. Figure 3.3 shows an example execution of the prefix labeling scheme.

Before starting on the proof of correctness, we can see that the prefix labeling scheme strongly resembles the procedure labeling scheme. In fact, other than the way the labels are generated and the fast-passes of specifically the race reports, the two are identical in behavior. This will make our proof much simpler, as we can just refer to the previous section's results. There are two key lemmas we will need to do this:

Lemma 6. *Each label in Algorithm 4 maps to a single procedure, and vice versa.*

Proof. One direction of this is clear - any procedure can have at most one label among its strands, as new labels are generated only during a `cilk_spawn`. For the other direction, consider the Cilk control-flow DAG of any Cilk execution and any label Q given during this execution. We can use the fact that labels encode the history of their ancestor procedures to recover which procedure's creation generated Q. Suppose the last value in Q is n. Then, the procedure which generated Q was the nth subprocedure of its parent. By continuing this process for all of the values in Q until we reach the original `main()` procedure, we uniquely describe the location of Q's procedure in the Cilk control-flow DAG, which demonstrates its uniqueness. □

Lemma 7. *For strands e and f explored in that order, if $e.label$ is a prefix of $f.label$, then $e \prec f$.*

Proof. The same logic as in the proof of Lemma 6 shows that for procedures $E \ni e$ and $F \ni f$, if E's label is a prefix of F's label, then E is an ancestor of F (that is, there is some `cilk_spawn` statement within A which leads to B). Since e is explored before f, the aforementioned `cilk_spawn` statement must occur after e, so that $e \prec f$. □

We can now prove the correctness of Algorithm 4.

Theorem 4. *The prefix labeling scheme correctly detects determinacy races.*

Proof. Consider some Cilk execution and its associated labels generated by the procedure labeling scheme and the prefix labeling scheme. According to Lemma 6 and Lemma 2 in Section 3.2.1, both schemes assign the same labels to strands e and f iff they are from the same procedure. Furthermore, the procedure labeling scheme doesn't use any comparison between two labels except for the equality operator $=$, so replacing the label generation in the procedure labeling scheme with prefix labels does not change the algorithm behavior. After doing this, the algorithms only differ in their treatment of fast-passing the "Report race" statements (lines 25 and 30 of Algorithm 3 and lines 27 and 30 of Algorithm 4). Specifically, the only difference between the algorithms' behavior is that the prefix labeling scheme will never report a race against a previous read/write at ℓ if $rlabel(\ell)/wlabel(\ell)$ is a prefix of the current strand's label. But now, Lemma 7 shows that this fast-pass does not cause correctness issues. □

3.3.2 Implementation

This section describes the implementation of the prefix labeling scheme in Cilksan.

The prefix labeling scheme implementation resembles that of the procedure labeling scheme seen in Section 3.2.2 - again, we store an array of 2^{30} 64-bit labels within each `Page_t` object, and `checkAndSetLabels` houses the fast path logic seen in Algorithm 4. The main difference is in the implementation of prefix labels. As our prefix labels are only 64 bits, we cannot directly store the sequences of values used as labels in Algorithm 4, so a new method must be determined.

A first approach might be to label the first procedure `main()` as a `1`, then append a bit for each new child of `main()`, so the first child's label is `10` and the second child's label is `11`[2]. This runs into an issue if `main()` has more than two children, though, as there is no label left for the third child.

A solution to the above approach is to relabel the continuation strand after a spawn. For example, `main()` might start with the label `1`, and after spawning a child, the child's label is `10`, while the `main()` frame is relabeled `11` after the spawn. This might initially seem to upend some of the equality checking of labels we do in Algorithm 4, namely in lines 16 and 26, but notice that we can always recover a notion of "equality" of labels by stripping away any trailing ones from the label. The implementation I use essentially does this, but relabels the continuation less frequently by treating appends to a label as values in base 4. To illustrate this difference using previous example, `main()`'s first child's label becomes `100`, its second child's label becomes `101`, its third child's label becomes `110`, and finally we relabel the continuation of the third child's spawn as `111`. This variation of the scheme makes continuation labels smaller at the expense of making spawns slightly more expensive.

3.4 Performance analysis

This section presents the empirical results of implementing the labeling schemes earlier in the chapter. The labeling schemes are tested on three benchmarks:

- `parallelPrefix` takes an integer input n and computes the prefix sum of an array of size 2^n using the parallel prefix algorithm found in [9], using a base case of size 64.

- `cilksort` takes an integer input n and sorts an array of size n using a parallel sorting algorithm similar to merge sort.

[2]Note that these label values are in binary, not decimal

- `matmul` takes an integer input n and computes the product of two $n \times n$ matrices using a parallel cache-oblivious algorithm.

While the code for `parallelPrefix` is not public, the code for `cilksort` and `cholesky` can be found at https://github.com/neboat/opencilk-ppopp-23-ae. Three input sizes were chosen for each benchmark, aiming for a 1 to 10 second runtime for each input size with no labeling scheme. The input sizes grow exponentially in each case, to better illustrate how input size affects performance.

Table 3.1 shows the number of times during the execution of the code that each labeling scheme took a fast path, along with the total number of opportunities to do so. These were measured by counting the number of times the core labeling scheme procedure, `checkAndSetLabels` returned `false` to indicate it took a fast path, versus how many times `checkAndSetLabels` was called. We measure the number of `checkAndSetLabels` calls rather than the number of total memory accesses because other fast paths might already exist in Cilksan which circumvent our labeling scheme on some memory accesses. Table 3.2 shows the runtime of each benchmark/labeling scheme pair, along with the runtime without any labeling scheme and the runtime with an "occupancy bit optimization" explained below. While the fast path counts in Table 3.1 include the entire execution of each benchmark, including setup of data structures like arrays for `parallelPrefix` and `cilksort` and matrices for `matmul`, the runtimes in Table 3.2 measure only the algorithm runtime.

There are several points of note:

1. Both labeling schemes discovered safe fast paths in a large fraction of the calls to `checkAndSetLabels`, in all benchmarks except `matmul`.

2. Neither labeling scheme outperforms the runtime of the original Cilksan without labeling. Instead, the runtime is many times higher in each benchmark with either labeling scheme than without. This suggests that the overhead of maintaining a 64-bit label for each memory location is immense and overshadows the benefits of labeling schemes'

	Procedure LS	**%**	**Prefix LS**	**%**	**Total Number of Calls**
parallelPrefix 25	17825706	50.75	19399896	55.23	35127167
parallelPrefix 26	35651497	50.75	38798631	55.23	70254460
parallelPrefix 27	71303080	50.75	77596023	55.22	140509049
cilksort 3×10^6	49623958	38.95	106452010	83.56	127397698
cilksort 6×10^6	105946009	39.97	224343314	84.64	265065244
cilksort 1.2×10^7	198517738	35.60	473166534	84.85	557671331
matmul 256	37440	0.2219	61440	0.3655	16873472
matmul 512	299584	0.2219	507904	0.3762	134995968
matmul 1024	2396736	0.2219	4128768	0.3823	1080000512

Table 3.1: The number of times a fast path was used to shortcut S/P bag FIND-SET operations. The final column shows the total number of calls to `checkAndSetLabels` for each benchmark/input size. Each column with a "%" shows the ratio of the previous column to the final column as a percentage.

	Procedure LS	**Prefix LS**	**No LS**	**Occupancy Bit Optimization**
parallelPrefix 25	7.52	9.31	2.12	1.69
parallelPrefix 26	16.02	19.39	4.88	3.40
parallelPrefix 27	29.03	37.13	9.30	7.98
cilksort 3×10^6	6.90	6.29	1.61	1.61
cilksort 6×10^6	8.88	10.89	3.57	3.29
cilksort 1.2×10^7	18.79	21.02	7.25	6.97
matmul 256	2.17	2.15	0.17	0.16
matmul 512	5.26	4.70	1.20	1.13
matmul 1024	28.99	27.39	8.93	8.84

Table 3.2: Runtimes in seconds (rounded to nearest hundredth of a second)

fast paths. Chapter 5 discusses alternatives to the costly approach taken here which could potentially overcome this issue.

3. The percentage of calls to `checkAndSetLabels` in which the code takes a fast path is highly dependent on the specific parallel program and labeling scheme, and seems largely oblivious to the input size beyond a point. This can be seen in Table 3.1 in the fact that each benchmark/labeling scheme combination has roughly the same percentage for each of the three input sizes, but large variance exists amongst the benchmarks and labeling schemes. Furthermore, even the relative benefit of the prefix labeling scheme over the procedure labeling scheme seems very program dependent, as seen in that `parallelPrefix` sees a small increase from 50% to 55%, and `cilksort` sees a much more marked increase of 38% to 84%. This might suggest that algorithms or their implementations have specific parallel structures which hinder attempts at general optimizations.

4. The prefix labeling scheme outperforms or closely matches the procedure labeling scheme runtime in many of the runs of `cilksort` and `matmul`, benchmarks which also have a much larger number of fast paths taken for the prefix scheme than the procedure scheme. This correlation isn't strong enough to make definite claims, but might lend credence to the previous hypothesis that the poor runtime of our labeling schemes is due to a large fixed overhead of maintaining labels, rather than excessive logic in the fast paths; after all, the procedure labeling scheme has very little logic whereas the prefix labeling scheme has much more. This observation may also support the reduction of S/P bag operations as a fruitful avenue of performance engineering of Cilksan.

5. The occupancy bit (see Chapter 4) optimization in the final column of Table 3.2 mildly improves the runtime of Cilksan. As this optimization was inspired by the labeling schemes, I decided to include it in the thesis as an adjacent result without devoting

too much explanation to it. The optimization removes the clearing of the occupancy bit hashmap that occurs during a sync, motivated by the fact that the strand f after the sync is parallel to some strand g only if the strand e before the sync is parallel to g. Since Cilksan processes e immediately before f, we can keep e's occupancy bits while processing f.

Chapter 4

Related work

Labeling strands for the purpose of maintaining series/parallel relationships is well-documented in literature. In Nudler and Rudolph's English-Hebrew labeling [5], we produce two sets of labels which can jointly determine whether any two strands are in series or parallel. One set, called the English labels, mirrors the exploration order of the strands during a preorder traversal of the spawn call tree, visiting spawned strands from "left-to-right". The other set, the Hebrew labels, is the same, except strands are explored from "right-to-left". While the exploration order of an ancestor will always be smaller than a child in both English and Hebrew labels, two strands logically in parallel will be explored in different orders in the English and Hebrew scheme, which allows for differentiability between strands in series and strands in parallel. Furthermore, labels may be generated on-the-fly while executing in parallel, although in this case the label length grows linearly with the depth of nesting (as well as the number of syncs that are the first sync within their procedure). Crummey improves upon the asymptotic behavior of English-Hebrew labels with Offset-Span labeling [4]. This scheme is similar to prefix labeling (Section 3.3) in that each strand u which arose as the nth child of a forking strand v is labeled with a combination of n and $u.label$. We refer to n as the "offset". The scheme differs in that each offset is accompanied by the total number of strands spawned during the fork, which impacts the labels of strands that form after a sync.

Offset-Span labels' length still grows linearly with the depth of nesting, but no longer has dependence on the number of syncs encountered on the path to a strand.

Strand labels can also be used for purposes other than series/parallel maintenance. Leiserson, Schardl, and Sukha describe an implementation of deterministic parallel random-number generation [10] called DotMix, whose goal is to eliminate nondeterminacy in multithreaded programs which use random number generation. DotMix uses spawn pedigrees, which again are similar to the prefix labels in Section 3.3, as unique identifiers for each procedure. DotMix generates pseudorandom numbers by combining a seed with a spawn pedigree, then hashing both to reduce the size of the pedigree with a pairwise-independent hash family, and finally applying other “mixing" functions to the result of the hash. The uniqueness of spawn pedigrees (a property also used in prefix labeling) is essential to the guarantees provided by DotMix.

Apart from labeling, race detection has seen asymptotic improvements since SP-bags. Bender, Fineman, Gilbert, and Leiserson's SP-order algorithm achieves more efficient series/parallel maintenance than the S/P-bag maintenance seen in SP-bags, eliminating the inverse Ackermann function $\alpha(\nu, \nu)$ from the runtime and yielding an optimal serial race detector. Rather than using a union-find data structure, the algorithm uses an “order-maintenance data structure” which maintains a total order of the strands of the parallel program with only constant-time operations. By keeping two such total orders and encoding a strands' English labels in one and their Hebrew labels in another, a more efficient English-Hebrew labeling is achieved. Utterback, Agrawal, Fineman, and Lee's work in [11] build upon this work with the WSP-Order, an algorithm that performs series/parallel maintenance in $O(T_1/p + T_\infty)$ time, which is asymptotically optimal. WSP-Order uses a parallelized version of SP-order, which requires a modified work-stealing scheduler and a parallel version of the order-maintenance data structure without additional asymptotic overhead.

Other work focuses on optimizing the access history maintenance (SP-bags' version of this is the shadow memory maintenance) of race detectors, rather than the series/parallel maintenance. Xu, Zhou, Yin, and Agrawal observe in [7] that the memory accesses of a strand are often done to contiguous chunks of memory at a time (for example, reading an array) and/or to the same memory location many times[1]. This observation motivates their use of spatial and temporal coalescing in race detection. Spatial coalescing involves the treatment of memory accesses in the access history as intervals rather than single locations. Their work uses a tree structure to store and maintain these intervals, and achieves an $O(\log n + T_1)$ bound on the total cost of their algorithm, where n is the number of intervals rather than the number of shared memory locations. Temporal coalescing employs a bit hashmap to track which locations have already been read from/written to during the execution of the current strand. The purpose of the bitmap is to ensure that only one access of each type (read or write) will be processed by the race detector for each strand. This bitmap exists in Cilksan as an "occupancy bitmap" [8], which, as currently written, clears all occupancy bits during any spawn or sync so that each strand has a fresh bit hashmap.

[1]This observation also explains why LRU caches work well.

Chapter 5

Conclusion

I conclude by discussing directions for future research on optimizing labeling schemes and Cilksan.

First and foremost, I believe the runtime of the labeling schemes can be improved by a more involved performance engineering of their implementations. Due to the scope of my project, the complexity of Cilksan, and system limitations, I did not significantly performance engineer the fast path logic, and I was unable to fully explore better methods for label maintenance than essentially a large array storing a label for each memory location. One possibility for improvement in this area is spatial coalescing of labels via storing them directly alongside memory accesses. Another promising possibility is to implement the procedure labeling scheme with no label maintenance overhead at all, by modifying frames to write a procedure ID instead of a frame ID into the shadow memory, and directly using the shadow memory entries as labels. While this seems simple, Section 3.4 shows that the procedure labeling scheme created many fast paths despite its simplicity, so this area of future work could potentially significantly improve Cilksan.

Another direction of research motivated by the results of Section 3.4 is improving the understanding of the structure of Cilk programs. Each of the benchmarks in Table 3.1 had very little variation among its input sizes in the percentage of memory accesses for which the labeling schemes used a fast path. However, this percentage varied greatly from benchmark to benchmark. This hints at some interesting structural properties of these algorithm implementations in Cilk. If structural properties that make programs amenable to a large number of fast paths can be granted during compile time, further research might be warranted in modifying the compiler to do so. As an example, suppose we have a strand e, after which a spawn creates a spawned strand f and a continuation strand g. The procedure labeling scheme works better if e and g have many memory accesses to the same locations than if e and f do - however, which strand is spawned and which becomes the continuation does not matter from the perspective of race detection, so the Cilksan compiler should be free to switch the two during compile time. The above discussion on structural properties that ease race detection is not limited to labeling schemes - the same argument might hold for occupancy bits.

References

[1] S. Amarasinghe, S. Devadas, and C. E. Leiserson, *6.1060 - software performance engineering*, Fall 2022.

[2] R. H. B. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992, ISSN: 1057-4514. DOI: 10.1145/130616.130623. URL: https://doi.org/10.1145/130616.130623.

[3] M. Feng and C. E. Leiserson, *Efficient detection of determinacy races in cilk programs*, 1997.

[4] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 24–33. DOI: 10.1145/125826.125861.

[5] I. Nudler and L. Rudolph, "Tools for the efficient development of efficient parallel programs," in *Proceedings of the first Israeli conference on computer systems engineering*, 1986, pp. 4–1.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, "On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04, Barcelona, Spain: Association for Computing Machinery, 2004, pp. 133–144, ISBN: 1581138407. DOI: 10.1145/1007912.1007933. URL: https://doi.org/10.1145/1007912.1007933.

[7] Y. Xu, A. Zhou, G. Q. Yin, K. Agrawal, I.-T. A. Lee, and T. B. Schardl, "Efficient access history for race detection," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 449–451, ISBN: 9781450380706. DOI: 10.1145/3409964.3461825. URL: https://doi.org/10.1145/3409964.3461825.

[8] T. B. Schardl and I.-T. A. Lee, "Opencilk: A modular and extensible software infrastructure for fast task-parallel code," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 189–203, ISBN: 9798400700156. DOI: 10.1145/3572848.3577509. URL: https://doi.org/10.1145/3572848.3577509.

[9] J. JáJá, *An introduction to parallel algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1992, ISBN: 0201548569.

[10] C. E. Leiserson, T. B. Schardl, and J. Sukha, "Deterministic parallel random-number generation for dynamic-multithreading platforms," *SIGPLAN Not.*, vol. 47, no. 8, pp. 193–204, Feb. 2012, ISSN: 0362-1340. DOI: 10.1145/2370036.2145841. URL: https://doi.org/10.1145/2370036.2145841.

[11] R. Utterback, K. Agrawal, J. T. Fineman, and I.-T. A. Lee, "Provably good and practically efficient parallel race detection for fork-join programs," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '16, Pacific Grove, California, USA: Association for Computing Machinery, 2016, pp. 83–94, ISBN: 9781450342100. DOI: 10.1145/2935764.2935801. URL: https://doi.org/10.1145/2935764.2935801.