Cilkprof: A Scalability Profiler for Cilk Programs

by

Kerry Xing

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

Master in Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author ..... Signature redacted
Department of Electrical Engineering and Computer Science
May 23, 2014

Certified by ..... Signature redacted
Charles E. Leiserson
Professor
Thesis Supervisor

Certified by ..... Signature redacted
I-Ting Angelina Lee
Postdoctoral Associate
Thesis Supervisor

Accepted by ..... Signature redacted
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Cilkprof: A Scalability Profiler for Cilk Programs

by

Kerry Xing

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master in Engineering in Computer Science and Engineering

Abstract

This thesis describes the design and implementation of Cilkprof, a profiling tool that helps programmers to diagnose scalability problems in their Cilk programs. Cilkprof provides in-depth information about the scalability of programs, without adding excessive overhead. Cilkprof's output can be used to find scalability bottlenecks in the user's code.

Cilkprof makes profiling measurements at the fork and join points of computations, which typically limits the amount of overhead incurred by the profiler. In addition, despite recording in-depth information, Cilkprof does not generate large log files that are typical of trace-based profilers. In addition, the profiling algorithm only incurs constant amortized overhead per measurement.

CilkProf slows down the serial program execution by a factor of about 10 in the common case, on a well-coarsened parallel program. The slowdown is reasonable for the amount of information gained from the profiling. Finally, the approach taken by Cilkprof enables the creation of an API, which can allow users to specify their own profiling code, without having to change the Cilk runtime.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Thesis Supervisor: I-Ting Angelina Lee
Title: Postdoctoral Associate

Acknowledgments

This thesis would not have been possible without the joint work of I-Ting Angelina Lee, Charles E. Leiserson, and William Leiserson. Angelina and Charles have been invaluable in their advice and guidance on the project. I am also extremely grateful for the help that Bradley Kuszmaul has given me on the project. He is extremely knowledgeable, and has helped me find problems when I didn't know where to look.

Contents

Chapter 1

Introduction

For improving performance, a profiler yields valuable insight into the bottlenecks of a program. Many profilers (such as gprof [22]) work well, but only for serial programs. Improving performance of a parallel program to achieve near-linear speedup can be difficult, especially because there are many types of performance bottlenecks that can occur in parallel programs, but not in serial programs. Indeed, common metrics for serial profilers, such as total time spent executing a particular function (or line of code) may not reveal the true performance bottlenecks in a parallel program. For example, it is possible that a parallel program is bottlenecked by a block of serial code even though the total time spent executing other regions of parallel code is greater.

This thesis presents Cilkprof, a profiler for Cilk programs, created in collaboration with I-Ting Angelina Lee, Charles Leiserson, and William Leiserson. Cilk provides the abstraction of ***fork-join parallelism***, in which dynamic threads are spawned off as parallel subroutines. Cilk extends C/C++ with three keywords: `cilk_spawn`, `cilk_sync`, and `cilk_for`. Parallel work is created when the keyword `cilk_spawn` precedes the invocation of a function. In a function F, when a function invocation G is preceded by the keyword `cilk_spawn`, the function G is ***spawned***, and the scheduler may continue to execute the ***continuation*** of F — the statement after the `cilk_spawn` of G — in parallel with G, without waiting for G to return. The complement of `cilk_spawn` is the keyword `cilk_sync`, which acts as a local barrier and joins together the parallelism specified by `cilk_spawn`. The Cilk runtime system ensures that statements after `cilk_sync` are not executed until all

```
1  int main() {
2      cilk_spawn foo();
3      bar();
4      cilk_sync;
5      baz();
6      return 0;
7  }
```

Figure 1-1: Portion of a Cilk program

functions spawned before the `cilk_sync` statement have completed and returned. Cilk's linguistic constructs allow a programmer to express the logical parallelism in a program in a processor-oblivious fashion. In addition, the keyword `cilk_for` indicates a parallel `for` loop, which allows all iterations of the loop to operate in parallel. Under the covers, `cilk_for` is implemented using `cilk_spawn` and `cilk_sync`.

For example, consider a portion of the program in Figure 1-1. On line 2, the function call to `foo()` is spawned, which implies it can run logically in parallel with the subsequent call to `bar()` on line 3. Then a `cilk_sync` keyword is reached on line 4, which implies that all prior spawned work in the function (ie. `foo()` in this example) must finish, before moving beyond the `cilk_sync` keyword. Finally, the code makes another function call to `baz()` on line 5.

If `foo()`, `bar()`, and `baz()` were large, complex functions, then optimizing the code may not be straightforward. Although `foo()` and `bar()` can execute in parallel, optimizing the wrong one could lead to minimal benefits. Furthermore, if the code were actually bottlenecked by `baz()`, then optimizing both `foo()` and `bar()` would not help much. Cilkprof reports metrics that help the programmer find bottlenecks.

The remainder of this chapter describes some existing tools and their deficiencies, followed by an illustration of sample Cilkprof output and the contributions of Cilkprof. The chapter ends with an overview of the thesis.

1.1 Cilkview

An existing tool, Cilkview [23], computes scalability metrics for the entire Cilk program. By default, Cilkview does not report these metrics over subcomponents of a parallel pro-

gram. Although there is an API to profile a particular interval, the programmer may need to profile over many different intervals to find performance bottlenecks in large programs. As a result, profiling with Cilkview can potentially be time-consuming.

Cilkview was implemented using Intel's Pin tool [32], which dynamically instruments the program as it runs. There are some drawbacks to Cilkview. First, because it uses Pin, it measures instructions, not running time. Second, Cilkview supports only serial executions. As a result, running Cilkview can take much longer than the production binary would take, especially if there are many processors available. In addition, if the user were to profile a non-deterministic program, then Cilkview would instrument the serial execution, which may not be a good representation of the non-deterministic parallel execution.

1.2 Cilkprof Prototype

Another existing tool, the Cilkprof prototype [25] was made to address the shortcomings of Cilkview. It reports scalability metrics with function-level granularity for a parallel program. As a result, the user would have a better understanding of where the scalability bottlenecks lie in the code.

Like Cilkview, the Cilkprof prototype also instruments using Pin, so it also reports the metrics using instruction counts, rather than the running time. The prototype also has other similar drawbacks. In particular, the prototype runs serially, so it is slow. As with Cilkview, it cannot simulate nondeterminism either.

1.3 Cilkprof - Sample Usage

Returning to the problem of profiling the code presented in Figure 1-1, Table 1.1 shows a simplified version of sample Cilkprof output [1]. The output suggests that `bar` is the main scalability bottleneck, because it has the lowest parallelism. Even though more total time is spent in `foo`, `bar` is far less parallel, so it will scale worse as the number of processors is increased.

[1]The actual Cilkprof metrics are described in Chapter 2, but the metrics in Table 1.1 can be computed from Cilkprof's output.

Caller	Callee	Total Time (ns)	Parallelism
`main`	`foo`	15263389798	31739
`main`	`bar`	4865148144	2
`main`	`baz`	626623777	238

Table 1.1: Sample Cilkprof Output

1.4 Contributions of Cilkprof

There are two main contributions of Cilkprof. First, Cilkprof performs profiling in an efficient manner. Second, Cilkprof provides a framework for instrumentation.

Efficient Profiling

Unlike the Cilkprof prototype, Cilkprof performs instrumentation at runtime, instead of relying on external instrumentation though a tool such as Pin. The runtime instrumentation allows for access to runtime data, and the ability to directly measure running time. Furthermore, Cilkprof supports parallel Cilk executions so that it can utilize the processors on a machine and simulate nondeterministic code.

Cilkprof's instrumentation was designed to add negligible overhead to the production binary. For profiling, users can link their code to a library that performs instrumentation. For production, users link to a library with no-ops for instrumentation, and the no-ops function calls can be optimized away via link-time optimization. As a result, production binaries should incur virtually no overhead.

Instrumentation Framework

A second contribution of this thesis is the framework for instrumenting Cilk programs. In particular, the instrumentation is acheived by injecting profiling functions, which is an approach that can be useful for other types of tools. As a result, it is possible to generalize the injected functions to create an API that allows users to write their own custom profilers, without having to modify the internals of Cilk.

1.5 Overview

The thesis has the following organization: Chapter 2 provides a basic overview of Cilk and its runtime system, Chapter 3 describes the basic design of Cilkprof, Chapter 4 describes the implemention details of Cilkprof, Chapter 5 analyzes the performance of Cilkprof, Chapter 6 covers limitations and future work, Chapter 7 discusses related work, and Chapter 8 concludes.

Chapter 2

The Cilk Programming Library

Cilkprof was built on a GCC 4.9.0 experimental Cilk Plus [24] [1] branch. This chapter presents a brief overview of Cilk, including a description of the computation dag model of Cilk, the metrics that Cilkprof measures, and the Cilk runtime system.

Computational Dag Model

It is convenient to represent an execution of a Cilk multithreaded computation[2] as a directed acyclic graph, or ***dag*** [7]. The vertex set of a dag G representing the program execution represents a set of ***strands*** — sequences of serially executed instructions containing no parallel control — and its edge set represents parallel-control dependencies between strands. Figure 2-1 illustrates such a dag, which can be viewed as a parallel program "trace," in that it involves executed instructions, as opposed to source instructions. Without loss of generality, the granularity of strands can be chosen depending on context to be as small as an instruction or as large as any serial execution, as long as it involves no parallel control. Assume that strands respect function boundaries, meaning that calling or spawning a function terminates a strand, as does returning from a function. Thus, each strand belongs to exactly one function instantiation. Let G denote both the dag and the set of strands in the dag. A strand that has out-degree 2 is a ***spawn strand***, and a strand that resumes the caller after a

[1] As of the official GCC 4.9 release, Cilk Plus will be in the main branch of GCC and will be included automatically without the need for programmers to download a special GCC branch. The experimental branch can be found at: `http://www.cilkplus.org/download#gcc-development-branch`

[2] The running of a program is generally assumed to be "on a given input."

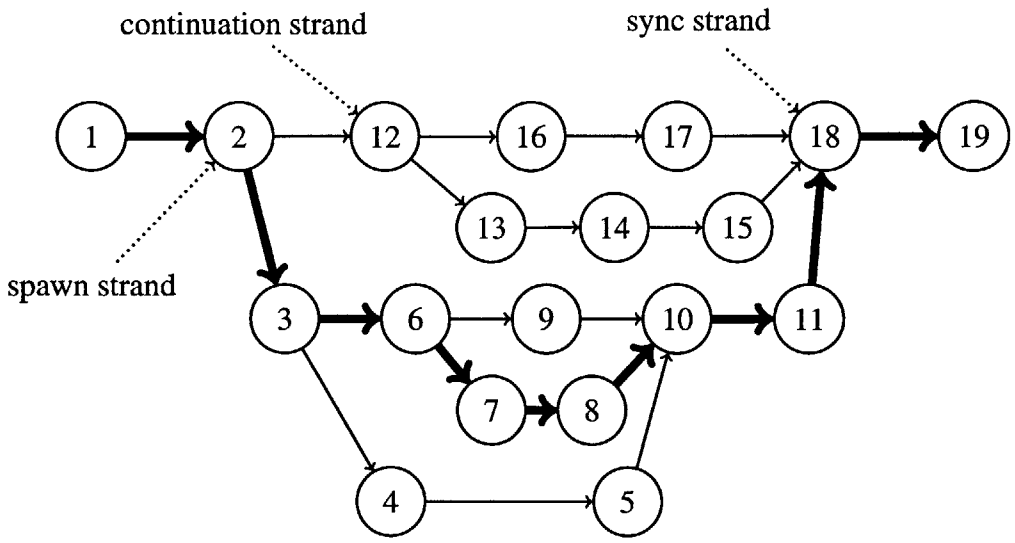

Figure 2-1: An example execution dag for a Cilk execution, labeled in English (that is left-to-right) execution order.

`cilk_spawn` statement is called a ***continuation strand***. A strand that has in-degree at least 2 is a ***sync strand***. For example, in Figure 2-1, strands 2, 3, 6, and 12 are spawn strands, strands 6, 9, 12, and 16 are continuation strands, and strands 10 and 18 are sync strands. Cilk's simple fork-join model produces only ***series-parallel*** dags [16], which enables fast algorithms for race detection and scalability analysis.

Cilk's work-stealing scheduler executes a user's computation near optimally [7] based on two natural measures of performance which can be used to provide important bounds [6, 8, 13, 21] on performance and scalability. Consider an ideal parallel computer in which each instruction takes unit time to execute, and let the runtime of computation G on P processors be denoted by T_P. For a computation G, the ***work*** $\text{Work}(G)$ is the total number of executed instructions in the computation, which for a deterministic computation is just its serial running time, that is, $T_1 = \text{Work}(G)$. For example, if each strand represents one instruction, then the work of Figure 2-1 is 19. The ***span*** $\text{Span}(G)$ is the length of the longest path of dependencies in G, which for a deterministic computation is its theoretical running time on an infinite number of processors, that is, $T_\infty = \text{Span}(G)$. For example, the span of Figure 2-1, which is realized by the highlighted path, is 10. Cilk's work-stealing scheduler can execute the computation on P processors in time $T_p \leq \text{Work}(G)/P + c_{\text{span}} \cdot \text{Span}(G)$,

where $c_{\text{span}} > 0$ is a constant representing the ***span overhead*** due to the cost of work-stealing. As is shown in [7], the expected number of steals incurred for a dag A produced by a P-processor execution is $O(P \cdot \text{Span}(A))$. The ratio $\text{Work}(G)/\text{Span}(G)$ is the ***parallelism*** of the computation, and Cilk guarantees linear speedup when $P \ll \text{Work}(G)/\text{Span}(G)$, that is, the number P of processors is much less than the computation's parallelism. Although these bounds assume an ideal parallel computer, analogous bounds hold when caching effects are considered [5], as well as empirically [18]. Work/span analysis is outlined in tutorial fashion in [11, Ch. 27] and [31].

Cilkprof Metrics

Cilkprof reports metrics about ***call sites***. A call site is a location in which one function invokes another function, either via a direct function call or via a spawn. This section defines the call site metrics that Cilkprof uses: work-on-work, span-on-work, work-on-span, and span-on-span.

Consider the parallel program corresponding to the dag in Figure 2-2. The critical path is bolded, and there are four instances where `foo` calls `bar`, whose subcomputations have been labeled as A, B, C, and D. The ***work-on-work*** of a call site is the sum of the work performed inside the function call, summed over the entire computation. In the figure, the work-on-work of the `foo` to `bar` call site would be the sum of the work in subcomputations A, B, C, and D. The ***span-on-work*** of a call site is the sum of the span of the child function, summed over the entire computation. In the figure, the span-on-work of the `foo` to `bar` call site would be the sum of the span in subcomputations A, B, C, and D. The ***work-on-span*** of a call site is the sum of the work of the child function, summed over the critical path (ie. that falls on the span). In the figure, the work-on-span of the `foo` to `bar` call site would be the sum of the work in subcomputations A and D. Analogously, the ***span-on-span*** of a call site is the sum of the span of the child function, summed over the critical path (ie. that falls on the span). In the figure, the span-on-span of the `foo` to `bar` call site would be the sum of the span in subcomputations A and D.

Cilkprof's metrics are useful for scalability analysis. Work-on-work represents a call site's contribution to the overall work. In particular, a call site with a large value of work-

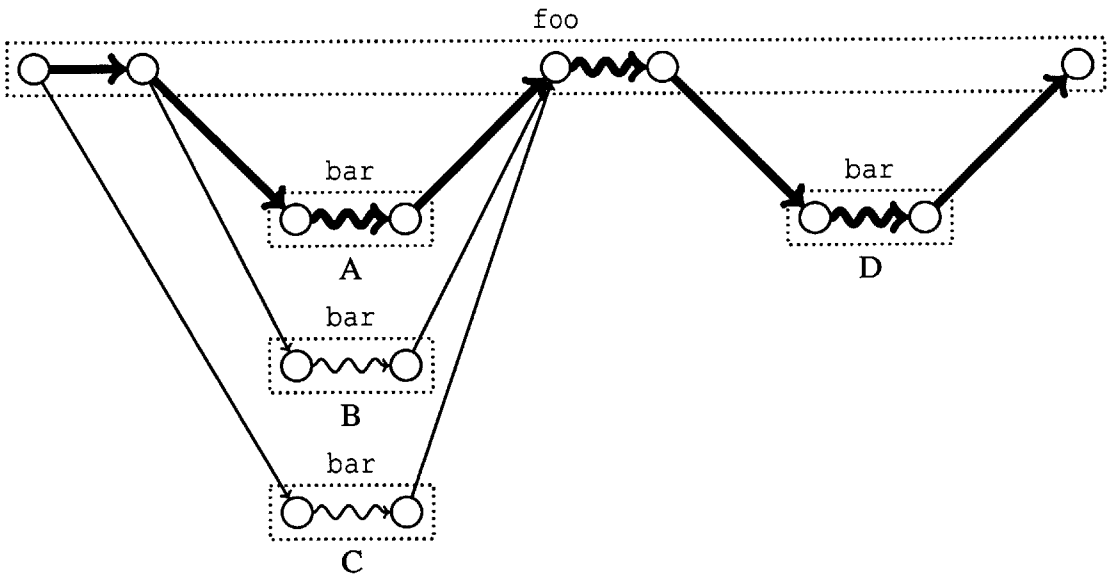

Figure 2-2: An example dag, for the definition of work-on-work, span-on-work, work-on-span, and span-on-span.

on-work adds a large amount of work to the program. Optimizing the called function can reduce the amount of overall work done by the program. The ratio of work-on-work to span-on-work can be used to provide a rough estimate of the parallelism of the called function, since it represents the ratio of total work to total span of the called function, summed throughout the entire computation. Thus, a call site with high span-on-work relative to its work-on-work is not very parallel. The span-on-span represents a call site's contribution to the overall span. In particular, a call site with a large value of span-on-span could potentially contribute a large amount of span to the overall program. The ratio of work-on-span to span-on-span can be used to provide a rough estimate of the parallelism of the called function over the invocations that lie on the critical path, since it represents the ratio of the total work to total span of the function, summed along the critical path.

Cilk Runtime System

For the purpose of describing the design of Cilkprof, a basic overview of the Cilk runtime system will be given.

To load-balance a computation, the Cilk runtime system employs ***randomized work-***

stealing [7, 18], which can be shown mathematically [7, 18] to yield provably good performance. With randomized work-stealing, a set of ***worker*** threads, typically one per processor, execute the user program, load-balancing the computation among them. When the user program executes a `cilk_spawn`, the worker posts the parent function frame to the bottom of a local deque (double-ended queue), and begins work on the child (spawned) function. The worker always consults the bottom of its local deque whenever it needs more work, pushing and popping it as in an ordinary serial stack-like execution. If it runs out of work, however, it becomes a ***thief*** and looks randomly among its peers for a ***victim*** with excess work. Upon finding a thief with a nonempty deque, the thief ***steals*** the topmost function frame from the victim's deque and resumes execution of the frame, which corresponds to resuming a caller after a `cilk_spawn` statement. When a Cilk computation is executed on a single processor, no steals occur, and the computation executes as an ordinary C/C++ program with the `cilk_spawn` and `cilk_sync` statements elided.

Each function that contains a Cilk construct (ie. `cilk_spawn`, `cilk_sync`, or `cilk_for`) is considered a ***Cilk function***. For each Cilk function, the Cilk runtime system maintains a ***Cilk stack frame*** object to do internal bookkeeping. For each spawned function, the Cilk compiler creates an additional Cilk function (with its own stack frame object) called a ***spawn helper***, which performs the appropriate internal bookkeeping while making the actual call to the child function. When a function [3] is spawned, the function is actually invoked via the spawn helper. Stack frame objects are linked to their parents with pointers. For example, if function `foo` spawns three calls to function `bar` and makes one direct function call to function `baz`, then Figure 2-3 shows the associated Cilk internal structure. The blank boxes correspond to the stack frame objects associated with the spawn helper functions, and the edges between the stack frame objects point from children stack frame objects to their parent stack frame objects.

Cilkprof uses the stack frame objects to store data, and to determine logical relationships between children and their parents (ie. direct function call vs spawn).

[3] The spawned function can be either a C function or a Cilk function.

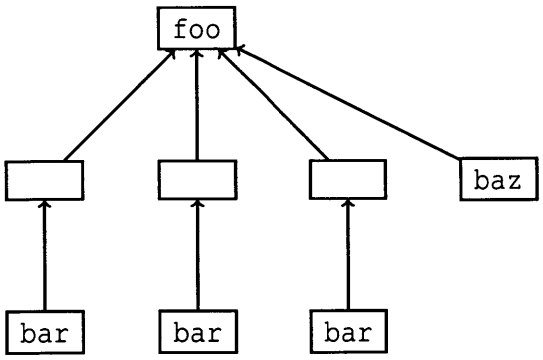

Figure 2-3: Internal structure

Chapter 3

Basic Cilkprof Design

This chapter presents the basic design of Cilkprof for serial instrumentation. The outline of the chapter is as follows: Section 3.1 describes the mechanism of instrumenting the Cilk runtime system. Section 3.2 describes the method of modeling the computational dag with the instrumentation. Section 3.3 explains how the profiling data (obtained from modeling the computational dag) is aggregated. Finally, Section 3.4 describes the data structures used.

3.1 Instrumentation

Cilkprof measures its data from compiler instrumentation. Specifically, Cilkprof uses a modified Cilk compiler that injects extra function calls into the Cilk runtime system. In particular, function calls were injected:

- after entering a Cilk function
- before a `cilk_sync`
- before exiting a Cilk function

The approach of injecting function calls has many benefits. First, since the performance measurements are measured as the program is executing, the instrumentation has direct access to the runtime data. Second, for production executables, the user can simply relink

the binaries with a Cilk library that does nothing for instrumentation. The function calls can be optimized out via link-level optimization, so the overhead for production executables is minimal.

3.2 Dag-modeling Algorithm

To model the dag appropriately, Cilkprof makes many changes to the existing runtime system. This section describes: internal data structures such as profiles and profile nodes, the changes made to Cilk stack frames, and the profiling algorithm.

Profiles

Cilkprof reports measurements for ***strands*** and ***call sites*** (as defined in Chapter 2). For a particular region of code, define the ***profile*** of the stack frame as a collection of strand and call site measurements, containing:

- work contribution of each strand to the overall work
- span contribution of each strand to the overall span
- work-on-work of each call site
- span-on-work of each call site
- work-on-span of each call site
- span-on-span of each call site

The main idea of the dag-modeling algorithm is to store the measurements in the profiles, and link the profiles of each Cilk stack frame to mirror the computational dag. The algorithm eagerly merges profiles whenever possible (ie. on sync). At the end of the computation, there will be one profile remaining, containing the strand and call site information for the entire program.

Profile Nodes

To link profiles together over the execution of the Cilk program, a data structure called a ***profile node*** was used. Each profile node corresponds to a strand in the computational dag. A profile node contains:

- `contents` - the associated profile of the profile node
- `parent` - the parent of the profile node
- `spawned_child` - the profile node corresponding to the spawned strand
- `continuation_child` - the profile node corresponding to the continuation strand

Profile nodes potentially have two children, so that they can appropriately model the logical dependencies of the dag.

Cilk Stack Frame Augmentation

To maintain the profile nodes over the execution of the Cilk program on a function-based granularity, each Cilk stack frame object is augmented with:

- `topmost_profile_node` - The profile node representing the currently executing strand in the computational dag.
- `function_profile_node` - The profile node representing the first executed strand in the current function.

The function profile node is stored so that the appropriate profiles can be merged when a `cilk_sync` occurs. (All spawned work in the current function must complete before moving beyond the `cilk_sync`.)

Profiling Algorithm

Using the injected function calls as described in Section 3.1, the algorithm for computing the profiles over the course of the program execution is outlined in Figures 3-1, 3-3, and 3-5.

```
1  after_enter():
2      // sf := the child that is entered
3      // parent := the parent of the frame that is entered
4
5      // Update parent, as necessary
6      if [sf has a parent]:
7          stop timer (for parent's strand)
8          record elapsed time into parent.topmost_profile_node
9      if [sf is spawned from the parent]:
10         // Create a new profile node for the current stack frame.
11         profile_node *new_profile_node = new profile_node
12         new_profile_node.parent = parent.topmost_profile_node
13         new_profile_node.spawned_child = NULL
14         new_profile_node.continuation_child = NULL
15
16         // Set the current stack frame profile node.
17         sf.topmost_profile_node = new_profile_node
18         sf.function_profile_node = new_profile_node
19
20         // Create a new continuation profile for the parent.
21         profile_node *new_topmost_profile_node = new empty profile for the
               parent
22         new_topmost_profile_node.parent = parent.topmost_profile_node
23         new_topmost_profile_node.spawned_child = NULL
24         new_topmost_profile_node.continuation_child = NULL
25         parent.topmost_profile_node.spawned_child = new_profile_node
26         parent.topmost_profile_node.continuation_child =
               new_topmost_profile_node
27         parent.topmost_profile_node = new_topmost_profile_node
28     else:
29         // Called by the parent - reuse the parent's profile.
30         sf.topmost_profile_node = parent.topmost_profile_node
31         sf.function_profile_node = parent.topmost_profile_node
32     start timer (to measure current strand for new_profile_node)
```

Figure 3-1: After entering a Cilk frame

Figure 3-1 contains the logic after entering a Cilk stack frame. Lines 6–8 charge the current timer measurement to the parent of the current frame. Lines 9–27 handle the case in which the parent spawned the current frame. In particular, a new profile node is created, and set as the root node for the current frame. Then, a new node is created for the continuation strand in the parent function, and the parent's topmost node is updated. Figure 3-2 illustrates this logic. Lines 28–31 handle the case in which the parent made a function call to the current frame. In this case, it is fine to continue accumulating measurements into the parent's profile, because the parent function strand and child function strand are logically in series. Finally, in line 32, the timer is started so that future execution costs can be charged to the current frame.

Figure 3-3 contains the pseudocode for the logic before a `cilk_sync`. Starting with the profile representing the currently executing strand, profiles are logically merged until the

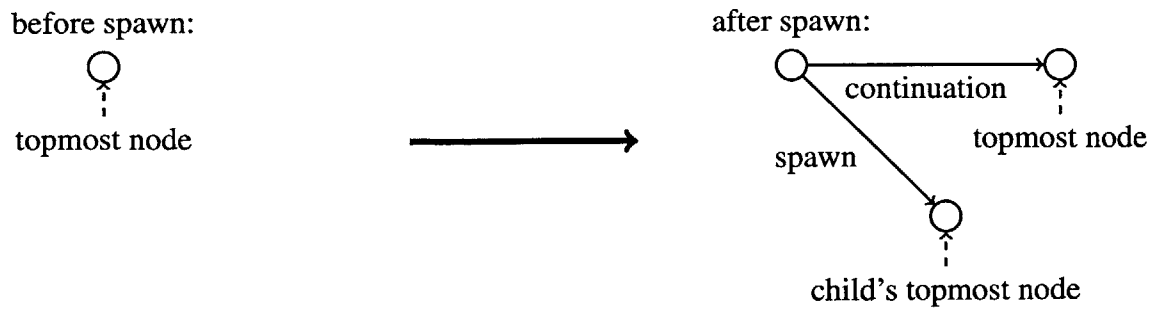

Figure 3-2: Spawning a child

```
1  before_sync():
2      // sf := the stack frame to be synced
3      stop timer (for the current strand)
4      record elapsed time into sf.topmost_profile_node
5      while sf.topmost_profile_node != sf.function_profile_node:
6          profile current_profile = sf.topmost_profile_node.contents
7          profile_node parent_profile_node = sf.topmost_profile_node.parent
8          profile_node sibling_node = parent_profile_node.spawned_child
9          if sibling_node != NULL:
10             // Two profiles in parallel.
11             current_profile = parallel_merge(current_profile,
12                                              sibling_node.contents)
13         // series combination
14         current_profile = series_merge(current_profile,
15                                        parent_profile_node.contents)
16         sf.topmost_profile_node = parent_profile_node
17         sf.topmost_profile_node.contents = current_profile
18     start timer (to start current BoC)
```

Figure 3-3: Before syncing a Cilk frame

profile corresponding to the strand representing function entry is reached. Lines 3–4 charge the current timer measurement to the current function. Additionally, the profiling data in all strands corresponding to the current function can be merged, because advancing beyond a `cilk_sync` implies that all spawned strands have completed. Lines 5–17 describe the merging process. Since the function profile node corresponds to the node that was set upon entering the current frame, line 5 causes the stack of nodes to merge until the profile in the function profile node contains all of the measurements made in the current function. Line 9 checks for a sibling of the topmost node, and lines 11–12 merges the two parallel profiles. Finally, lines 14–17 performs a merge of two profiles in series, removes the topmost profile node, and updates the new topmost profile node.

An illustration of the merge procedure is given in Figure 3-4. First, note that the left-

most node is the function profile node and the rightmost node is the topmost profile node. In the first iteration, the three rightmost profile nodes have their profiles merged. During the merging, the work is added together and the span is updated to reflect the span of the subcomputation. In the second iteration, the same process is applied to the three rightmost profile nodes. The process is continued until the function profile node contains the overall profile.

Figure 3-5 describes the logic before exiting a Cilk stack frame. Lines 3–4 charge the current timer measurement to the current function. Line 5 associates the time spent in the stack frame with the parent's call site. After line 7, any future computational cost is charged to the parent stack frame. Note that merging profiles is unnecessary here. Indeed, since a `cilk_sync` must have been called before exiting a Cilk function, profiles have already been merged.

3.3 Combining Profiles

The previous section assumed a black-box implementation for merging profiles. This section describes the details of performing series or parallel profile merges. In particular, this section describes the computation of work and span over subcomputations and the merging of strand measurements and call site measurements.

Work and Span of Profiles

The work and span of a profile can be computed recursively. For a strand with no children, the work and span of the associated profile are both equal to the time spent executing the strand. For computing the resulting profile from merging two profiles in series, the work and span are each summed across the profiles, because the computational subcomponents occur one after another. For computing the resulting profile from merging two profiles in parallel, the work is summed across the profiles. The span of the resulting profile is the maximum of the spans of the two constituent profiles, because the span represents the critical path. Using the recursive relation, the work and span can be computed over entire function calls.

Merging Strand Measurements in Profiles

The contribution of each strand to the overall work and span can also be computed recursively within profiles. For a profile corresponding to a strand, there is one measurement: the work and span of the strand is equal to the time spent executing it. When computing the resulting profile from merging two profiles in series, the work and span values for each strand are summed across the profiles, since the computational subcomponents occur one after another. When computing the resulting profile from merging two profiles in parallel, the work values for each strand are summed across the profiles. The span values of the profile with greater overall span are retained — the other profile's span values are discarded because the associated subcomputation did not fall on the critical path. Using this recursive relation, the work and span contributions of each strand to the overall work and span can be computed.

Merging Call Site Measurements in Profiles

The call site metrics (ie. work-on-work, span-on-work, work-on-span, and span-on-span) can be also computed recursively within profiles. For a profile corresponding to a strand that makes a function call, the work-on-work and work-on-span of the call site are both equal to the work of the called function. The span-on-work and span-on-span of the call site are both equal to the span of the called function. When combining two profiles that are logically in series, all four call site metrics are summed across call sites, because the subcomputations occur in series. When combining two profiles that are logically in parallel, the work-on-work and span-on-work metrics are summed across call sites, because they are summed over the entire computation. The work-on-span and span-on-span metrics corresponding to the profile with the greater span are retained while the other profile's work-on-span and span-on-span metrics are discarded, because the other profile's associated subcomputation did not fall on the critical path. Using this recursive relation, the call site metrics can be computed for the overall program.

3.4 Data structures

In this section, the data structures for the profiles will be described. A profile is represented as six sparse array data structures: one for each type of strand and call site measurement in Section 3.2. This section assumes the measurements are call site measurements, but the idea for strand measurements is analogous.

Assume that there are N distinct call sites over the lifetime of the execution. The sparse array consists of either a linked list or an array of size N. An empty sparse array is represented by an empty linked list, and new measurements are inserted into the linked list, until there are $\Theta(N)$ measurements in the linked list. When this happens, the linked list is converted into an array of size N. There is a global hashtable that maps each call site to a distinct call site ID. The IDs are used as indices to arrays, so that the ordering of the call sites within all arrays is the same. The arrays are big enough to store measurements for each call site, because it has size N, and there are only N distinct call sites over the lifetime of the execution.

Adding to a sparse array is relatively simple. The sparse array is represented by either a linked list or an array. If the sparse array is a linked list, a new node can be created and appended to the linked list. If the sparse array is an array, the call site ID can be found, and the corresponding measurement value can be incremented appropriately.

As described in Section 3.3, merging two profiles either involves adding the measurement values across corresponding call sites, or preserving one profile and deleting the other. The latter type of merge is trivial to implement efficiently. For adding measurement values across corresponding call sites, there are three cases:

- Case 1: Both sparse arrays are represented as linked lists. In this case, the linked lists are concatenated.

- Case 2: Both sparse arrays are represented as arrays. In this case, the contents of one array are added to the other array, across corresponding call sites.

- Case 3: One sparse array is represented by an array and one sparse array is represented by a linked list. In this case, the measurements in the nodes of the linked list

are added to the array.

Although some operations can be expensive, the overall cost of maintaining the profile is $O(1)$ time per measurement. This proposition can be proved with an accounting argument. When a measurement is added to the linked list, up to four credits are expended:

- One (type A) credit, for adding it as a node to the linked list.
- One (type B) credit, for initializing part of an array if/when the linked list is converted to an array.
- One (type C) credit, for moving it out of the linked list if/when the linked list is converted to an array.
- One (type D) credit, for moving it out of an array.

Adding a measurement to a linked list and adding to an array can be trivially done in $O(1)$ time. It can be verified that the other operations can be amortized to $O(1)$ time.

Lemma 1 *Converting a linked list to an array can be done in $O(1)$ amortized time.*

Proof. When the linked list is converted to an array, there must be $O(N)$ elements. Each of these elements has an associated type B credit and an associated type C credit. The $O(N)$ type B credits offset the cost of allocating a new array and initializing it to zeroes. The $O(N)$ type C credits offset the cost of moving the measurements from the linked list to the array. □

Theorem 2 *Adding measurements across two sparse arrays takes $O(1)$ amortized time.*

Proof. There are three cases:

Case 1: Both sparse arrays are represented by linked lists.

In this case, the linked lists can be concatenated in $O(1)$ time. If the linked list ever needs to be converted into an array, Lemma 1 shows that the additional amortized cost is only $O(1)$.

Case 2: Both sparse arrays are represented by arrays.

In this case, the measurements of one array are added to the second array. The first array has N elements, each with a stored type D credit. The N stored type D credits offset the cost of adding the measurements to the second array.

Case 3: One sparse array is represented by an array, and one sparse array is represented by a linked list.

In this case, each of the measurements in the linked list has an associated type C credit. The type C credits offset the cost of moving the measurements from the linked list to the array. □

Thus, all sparse operations take $O(1)$ amortized time. For space, it can be seen that the spawned child and continuation child of a profile node cannot both have children in a serial execution (since one call was made after the other). Thus, if the maximum depth of profile nodes is d, then the maximum space consumption is $O(Nd)$, corresponding to a size N array per profile. The space consumption tends to be much smaller than the size of a trace of the program's overall execution.

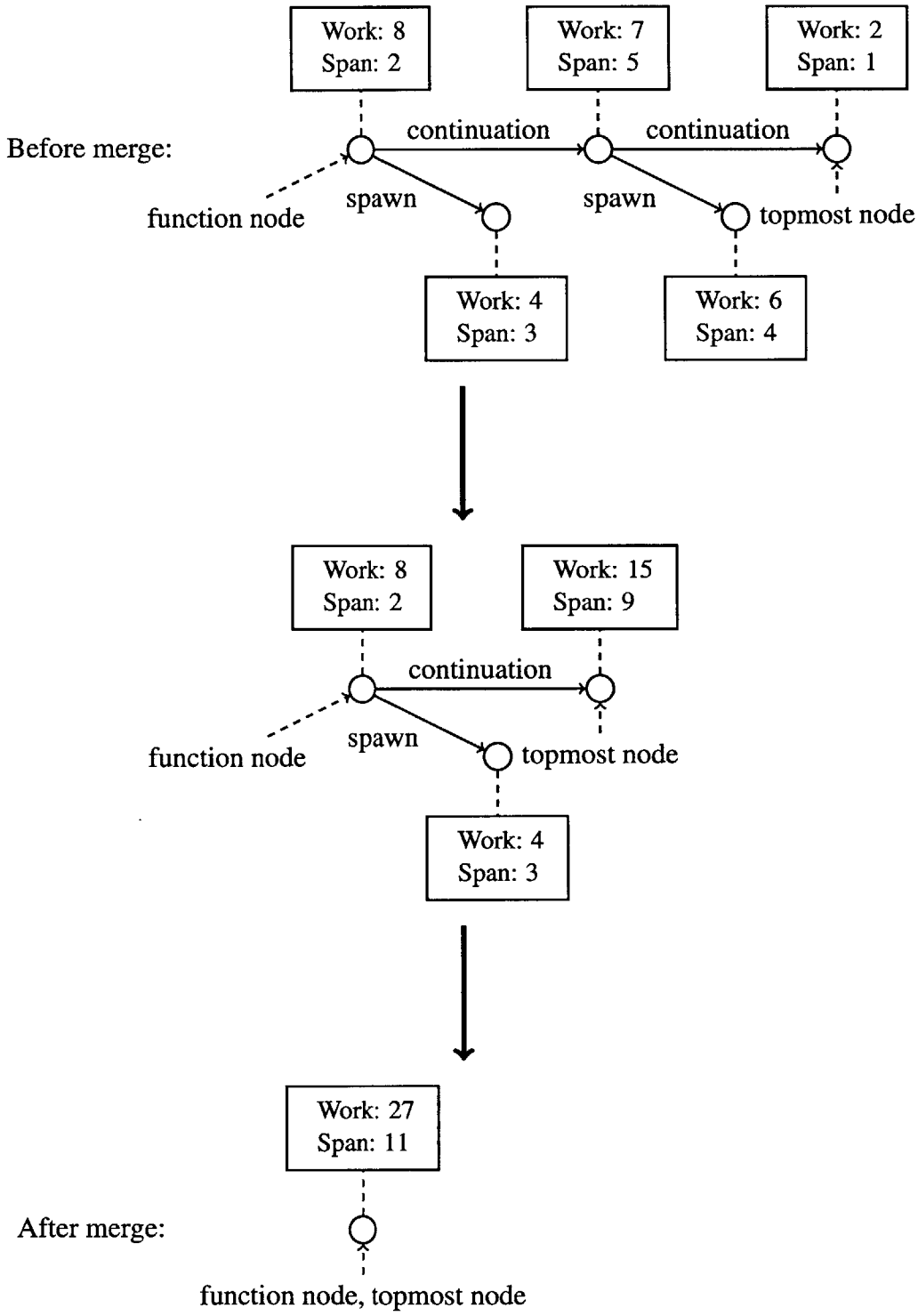

Figure 3-4: Merging profiles

```
1  before_exit():
2      // sf := the frame to be exited
3      stop timer (for current strand)
4      record elapsed time into sf.topmost_profile_node
5      record time spent in sf into sf.topmost_profile_node
6      if [sf has a parent]:
7          start timer (to start measuring parent's strand)
```

Figure 3-5: Before exiting a Cilk frame

Chapter 4

Implementation Details

This chapter describes a number of extensions and improvements that were implemented in addition to the basic design outlined in Chapter 3. In particular, this chapter describes handling recursion, dynamic resizing sparse arrays when the number of distinct entries is not known, and supporting parallel instrumentation.

4.1 Handling Recursion

An interesting situation arises when profiling a program that uses recursion. Indeed, if a function calls itself multiple times through a particular call site, then the work and span measurements corresponding to the call site may be double-counted. For example, consider a recursive Fibonacci implementation `fib`. An invocation of `fib(30)` calls `fib(29)` at a particular site. However, `fib(29)` calls `fib(28)` at the same site. Under the algorithm described in Chapter 3, the work and span incurred from `fib(29)` calling `fib(28)` would be double-counted. Indeed, it would be counted once for `fib(29)` calling `fib(28)` and once for `fib(30)` calling `fib(29)`. As a result of the double-counting, the resulting work and span metrics for call sites would be incorrect.

To address this potential problem, call site information is augmented with the ***Cilk height*** of the called function. Cilk height of a function invocation is equal to the maximum number of Cilk frames below the corresponding stack frame. With the augmentation of the Cilk height, the recursive calls are represented by different call sites, because the Cilk

height of each recursive call is different. Since it is common for the leaves of a call tree to correspond to a small number of helper functions, call sites with the same Cilk height are likely to exhibit similar behavior, so aggregating call site data across the Cilk height should be reasonable.

4.2 Dynamic Sparse Arrays

In Section 3.4, it was assumed that Cilkprof knew *a priori* the number of call sites it was profiling, so that it knew when to promote a linked list to an array. In this section, this assumption will be relaxed, and appropriate changes to the sparse arrays are described.

With the relaxed assumption, the sparse array now consists of both an array and a linked list. Along with the linked list, the largest call site ID I of the linked list measurements is stored. Whenever the length of the linked list equals (or exceeds) I, the linked list and array are merged into a new array.

Sparse array operations remain similar to the original ones. For adding to a sparse array, a new element is added to the linked list. When merging profiles, the linked lists are concatenated, and the contents of the smaller array are added to the larger array. The profile operations can be shown to use $O(1)$ amortized time.

Lemma 3 *The operation of emptying the contents of the linked list into an array takes* $O(1)$ *amortized time.*

Proof. Assume the same accounting scheme that was presented in Section 3.4.

Let L be the length of the linked list immediately before the operation. There are at least I elements in the linked list, so $L \geq I$. If the current array has size at least I, then it is already big enough to store the measurements in the linked list. Otherwise, a new array needs to be allocated and initialized to the existing measurements in the current array, which costs I. The cost of moving the linked list elements is L, so the total cost of the operation is at most $L+I$. However, each element in the linked list has an associated type B credit and an associated type C credit, so there are $2L$ stored credits, which is sufficent to offset the cost of the operation. □

Theorem 4 *Adding measurements across two sparse arrays can be done in $O(1)$ amortized time.*

Proof. The linked lists of each sparse array can trivially be concatenated in $O(1)$ time.

For the arrays, assume that the array sizes are S_1 and S_2, where without loss of generality $S_1 \geq S_2$. Then, the cost of moving the measurements from the second array into the first array is S_2. Since there are S_2 stored type D credits associated with the elements in the second array, the cost of adding the array elements into the first array is offset.

If the measurements in the linked list need to be added to the array immediately afterward, Lemma 3 shows that the operation takes $O(1)$ amortized time. □

As a result, profile operations take $O(1)$ time, and the space consumption is still $O(Nd)$, where N is the number of call sites and d is the maximum depth of profile nodes, since the arrays have size at most N.

4.3 Parallelized Cilkprof

This section describes the modifications needed to make Cilkprof work correctly for a parallel execution.

First, it is important to note that the algorithm for serial instrumentation presented in Section 3.2 is mostly correct for parallel instrumentation. There are no races on profile nodes and profiles, because the profile nodes and profiles are each local to a particular stack frame. The Cilk runtime system has an invariant guaranteeing that only one worker has ownership of a particular stack frame at a time. (See [17] for details about runtime invariants for a similar system.) Since new profile nodes are created whenever a function call is spawned, workers that perform work in parallel will operate on different profiles.

However, the serial algorithm presented in Section 3.2 does not account for many of the phenomena that can occur when running multiple workers in a parallel execution of the program. In each of the following cases, the profile data was saved and merged appropriately to ensure that profile data was not lost:

- A worker discovers that the parent stack frame was stolen by another worker. In this

case, the worker cannot pass the profile data to the parent frame, so it must save the data so that it can be merged later accordingly.

- A worker reaches the `cilk_sync` statement before other workers have finished all of the spawned function calls. In this case, the worker must wait for the other workers to finish their work before merging the profile nodes, because the profile data is not ready yet.

Chapter 5

Performance

This chapter reports the results of profiling serial and parallel executions. Table 5.1 describes the benchmarks used to evaluate the performance of Cilkprof. All of the measurements were taken on a 12 core Intel Xeon machine, clocked at 2.67GHz. The L1, L2, and L3 caches were 64KB, 256KB, and 12MB, respectively. L1 and L2 caches were private and the L3 cache was shared among CPUs.

5.1 Serial Performance

Table 5.2 shows the results for serial executions on the benchmarks. Ten measurements were taken and averaged for each benchmark. The standard deviation among the runs were within 2% of the average.

As described in Section 3.1, instrumentation is performed at each spawn and sync point, as well as the entry and exit points of each Cilk function. As a result, the instrumentation overhead is highest for programs that do little work per Cilk spawn statement. The results are consistent with this observation, because Fibonacci and knapsack use recursive functions that do not perform much work inside of the spawn calls. Most of the other benchmarks have coarsened base cases, so they have lower instrumentation overhead.

Benchmark	Input	Description
`fib`	37	recursive Fibonacci program
`cholesky`	2000/20000	Cholesky matrix factorization
`cilksort`	10^8	parallelized merge sort
`fft`	2^{26}	fast Fourier transform with Cooley-Tukey [10]
`heat`	2048×500	heat equation simulation
`knapsack`	35/1600	recursive knapsack solver
`lu`	4096	LU matrix decomposition
`matmul`	2048	square matrix multiplication
`nqueens`	13	n queens solver
`rectmul`	4096	rectangular matrix multiplication
`strassen`	4096	Strassen matrix multiplication

Table 5.1: Benchmarks

Benchmark	Original (seconds)	Instrumented (seconds)	Overhead
`fib`	2.489	569.903	228.969
`cholesky`	2.266	254.930	112.512
`cilksort`	16.893	25.608	1.516
`fft`	18.579	89.357	4.810
`heat`	7.421	9.665	1.302
`knapsack`	3.249	520.058	160.047
`lu`	16.826	78.379	4.658
`matmul`	10.966	21.522	1.963
`nqueens`	6.296	61.245	9.728
`rectmul`	33.474	221.850	6.628
`strassen`	20.477	21.907	1.070

Table 5.2: Serial performance

Benchmark	Original (seconds)	Instrumented (seconds)	Overhead
`fib`	0.236	240.761	1019.308
`cholesky`	0.334	119.485	357.526
`cilksort`	5.397	8.865	1.643
`fft`	2.177	34.401	15.805
`heat`	1.766	1.963	1.111
`knapsack`	0.420	359.949	856.818
`lu`	1.866	26.342	14.114
`matmul`	1.035	5.989	5.786
`nqueens`	0.579	25.862	44.690
`rectmul`	3.166	82.619	26.094
`strassen`	2.929	3.237	1.105

Table 5.3: Parallel performance

5.2 Parallel Performance

Table 5.3 shows the results for parallel executions on the benchmarks. As before, ten measurements were taken and averaged for each benchmark. The standard deviation among the runs were within 5% of the average, except for the nondeterministic `knapsack` benchmark.

Compared to the overhead observed for the serial executions, there is generally more overhead for parallel executions. Currently, the parallel instrumentation is completely unoptimized. In particular, the implementation uses two hashtables to store the mappings of strands and call sites to IDs for sparse array indexing. The hashtable accesses are currently synchronized with a global lock, which results in serialized hashtable accesses. Since the instrumentation involves many hashtable acceses, the hashtable is a significant bottleneck.

Table 5.4 shows the measured work for serial and parallel executions. Table 5.5 shows the analogous results for measured span. There is some blowup in the amount of measured work and span, partly due to the additional perturbation introduced by the parallel instrumentation. In particular, there is likely less memory locality in the parallel execution, due to the work-stealing involved. Furthermore, the memory allocator used is glibc [20], which may not handle parallel memory management well [4, 14, 15].

Table 5.6 and Table 5.7 show the measured parallelism for the serial and parallel executions, respectively. The measured parallelism values for serial and parallel executions are actually reasonably consistent with each other, after taking the instrumentation overhead

Benchmark	Serial	Parallel	Blowup
fib	567.948	1803.590	3.176
cholesky	254.243	913.693	3.594
cilksort	21.566	40.211	1.865
fft	88.858	255.761	2.878
heat	9.597	20.581	2.144
knapsack	518.517	2709.561	5.226
lu	77.830	187.908	2.414
matmul	21.420	49.393	2.306
nqueens	61.004	194.692	3.191
rectmul	221.255	577.963	2.612
strassen	21.427	28.894	1.349

Table 5.4: Overall work measured

Benchmark	Serial	Parallel	Blowup
fib	0.00093	0.00374	4.047
cholesky	0.48844	1.59390	3.263
cilksort	0.00603	0.01614	2.675
fft	0.03351	0.04938	1.474
heat	0.12519	0.20549	1.641
knapsack	0.00103	0.00401	3.880
lu	0.24302	0.50976	2.098
matmul	0.00769	0.02686	3.490
nqueens	0.00075	0.00263	3.512
rectmul	0.04162	0.09479	2.278
strassen	0.27297	0.33788	1.238

Table 5.5: Overall span measured

Benchmark	Work	Span	Parallelism
fib	567.948	0.00093	613887
cholesky	254.243	0.48844	521
cilksort	21.566	0.00603	3575
fft	88.858	0.03351	2651
heat	9.597	0.12519	77
knapsack	518.517	0.00103	501492
lu	77.830	0.24302	320
matmul	21.420	0.00769	2784
nqueens	61.004	0.00075	81409
rectmul	221.255	0.04162	5316
strassen	21.427	0.27297	78

Table 5.6: Measured parallelism from serial instrumentation

Benchmark	Work	Span	Parallelism
fib	1803.590	0.00374	481739
cholesky	913.693	1.59390	573
cilksort	40.211	0.01614	2491
fft	255.761	0.04938	5179
heat	20.581	0.20549	100
knapsack	2709.561	0.00401	675403
lu	187.908	0.50976	369
matmul	49.393	0.02686	1839
nqueens	194.692	0.00263	73979
rectmul	577.963	0.09479	6097
strassen	28.894	0.33788	86

Table 5.7: Measured parallelism from parallel instrumentation

into account.

Chapter 6

Limitations and Future Work

This chapter covers the limitations and future work of Cilkprof.

6.1 Limitations

There are many limitations to Cilkprof, including an unreliable mapping of profiling data to code, a space blowup for profiling large numbers of spawns, lost profiling data when profiling over multiple Cilk computations, the lack of optimization for parallel profiling, and nonsystematic error introduced from amortization arguments.

Mapping Profiling Data to Code

Cilkprof uses the GCC builtin function `__builtin_return_address()` to log the addresses for both strands and call sites. The compiler injects function calls into the Cilk Runtime System, so `__builtin_return_address()` will correctly return an address in the user's code.

Later, Cilkprof does post-processing to map the addresses to function names. In particular, Cilkprof uses the output of the objdump program to infer the function names from addresses. If the user compiles the code with the `-g` flag for debugging symbols, then the line number information is available, so the line numbers are also reported.

The current Cilkprof implementation assumes that the program is loaded at the default address. As a result, the current implementation will not work for programs loaded at a

custom address. This limitation also implies that dynamic library loading does not work.

In addition, due to the compiler's compilation of `cilk_for`, Cilkprof has trouble mapping return addresses to lines of code for `cilk_for`.

Sequential Spawn Space Blowup

The algorithm described in Section 3.2 creates new profile nodes every time a new function is spawned. Thus, the space used scale linearly with the number of unsynced strands, which could potentially be a problem. For example, if the user spawns millions of function calls in a `for` loop, then millions of profile nodes would be created, which would greatly increase the space consumption of the profiler. However, in this case, the space blowup can be mitigated by changing the loop to a `cilk_for`. Since `cilk_for` uses recursive doubling with intermediate syncs, the space consumption would shrink to logarithmic space usage, which would be much more reasonable. The Cilkprof prototype [25] uses a different algorithm that has a space bound.

Multiple Cilk Computations

The Cilk runtime system creates data structures such as Cilk stack frames as needed by the code, so once a Cilk computation is done (and the code returns from the root Cilk stack frame), the instrumentation data has to be written to disk. However, if there are multiple Cilk computations in a top-level `main` function, then the system will overwrite the data of the previous Cilk computation. Regardless, the entire program can be converted into one large Cilk computation by spawning a function that does nothing in `main`. As a result, the limitation has an easy work-around.

Non-optimized Parallel Profiling

As described in Section 5.2, the parallelized version of the profiler is not optimized. The hashtable actually has write-once semantics, with very few writes relative to the number of reads. The performance can be significantly improved by using a hashtable that has lockless reads. When a mapping is needed, the system can perform a fast lookup, and

perform the slow lookup if the fast lookup fails. Specifically, when looking for a hashtable mapping, the two paths are:

- Fast path - inspect the hashtable is inspected without a lock, and return the mapping if found.
- Slow path - lock the hashtable, and insert a new mapping into the hashtable if one does not already exist.

There should be a relatively small number of mappings over the lifetime of the program, so eventually all hashtable accesses should fall on the fast path.

Sparse Array Amortization

The sparse array data structure described in Chapter 3 and Chapter 4 uses an amortization argument to guarantee low overall overhead. However, the amortization can introduce non-systematic perturbations into the program to be measured. Thus, it would be beneficial for the sparse array operations to be deamortized.

6.2 Future Work

Future work includes visualizing Cilkprof's profiling data and creating a profiling API to allow users to perform custom profiling.

Visualizing Data

Currently, the profiling data is recorded in comma-separated value (CSV) format. Although it can be processed in a spreadsheet program, the user may still have to do some work to find the performance bottlenecks. The best visualization to present the Cilkprof data is still an area of active research.

Profiling API

The user may want to split strands or implement a custom profiler. The following sections describe these potential features in greater detail.

```
1  typedef struct __cilkrts_user_data __cilkrts_user_data;
2
3  struct __cilkrts_user_data {
4      __cilkrts_user_data *parent;
5      bool is_spawn_helper;
6      void *extra;
7  };
8
9  typedef struct __cilkrts_callbacks __cilkrts_callbacks;
10
11 struct __cilkrts_callbacks {
12     void (*after_enter)(__cilkrts_user_data *data);
13     void (*before_sync)(__cilkrts_user_data *data);
14     void (*before_exit)(__cilkrts_user_data *data);
15 };
16
17 CILK_API(void) __cilkrts_register_callbacks(__cilkrts_callbacks *cb);
```

Figure 6-1: Exposed API

Splitting Strands Currently, the strands that are reported by Cilkprof correspond to the regions in functions that start and end at one of the following points:

- the beginning of the function
- a point in which a function is spawned
- a point in which there is a call to a Cilk function
- a point in which a `cilk_sync` is reached
- the end of the function

However, these points may not allow for results at a fine granularity. In the future, a new API call (`CILK_SPLIT_STRAND()`) could be used, for splitting a strand into two pieces. The API would allow the user to profile at a finer granularity, at a cost of additional overhead.

Custom Profiling The function calls injected in Section 3.1 can be useful for tools other than Cilkprof. In particular, it is possible to implement the API presented in Figure 6-1, which would allow users to implement their own serial profilers. The API would allow users to attach custom profiling code into the important points of a Cilk program: the entry point of a Cilk function, the sync point of a Cilk function, and the exit point of a Cilk function.

Chapter 7

Related Work

There is a large amount of related work in the field of profiling parallel programs. Many of them [1–3, 9, 12, 28–30, 33–35] help the user pinpoint bottlenecks in parallel programs by collecting metrics such as processor utilization and idleness, but do not directly measure scalability. This chapter describes some of the other approaches taken by other tools.

Some tools, such as a particular version of HPCTOOLKIT [1], continuous monitoring systems [12], and DCPI [2] use some type of sampling to measure data. Sampling as a technique has some benefits. The authors of HPCTOOLKIT point out that eliminating the systematic error (introduced from instrumentation) is important. DCPI's use of sampling has the added benefit that it is general enough to use on any arbitrary process. In particular, there is no need to create instrumented versions of libraries. DCPI samples for hardware counts, to determine reasons for stalls. One continuous monitoring system [12] samples processor utilization profiles in 1 ms intervals, and merges them together. Interestingly, to save space, the system aggregates small measurements (that contribute less than a threshold, such as 10%) into an "other" category, unlike Cilkprof.

Many other tools such as Vampir [9, 28], Projections [29, 30], Paradyn [33], and Quartz [3] use instrumentation. Vampir has general instrumentation frameworks that allow users to do custom instrumentation, with support for many common parallel programming frameworks.

Projections is a tool for CHARM++ [27] programs, which use a message-passing model. It records its data in an event log. To avoid space blowup, Projections buckets

the events according to their timestamps, and doubles the bucket sizes to keep the number of buckets manageable. For analysis, Projections performs k-means clustering to find outliers in the behavior of the tasks that were run, to look for issues such as processor idleness.

Paradyn [33] uses dynamic instrumentation to record trace-style data according to the times in which they occur in the program. Because it uses a online heuristic algorithm for attempting to diagnose performance problems, Paradyn assumes that the programs run for a long time, unlike Cilkprof.

Quartz [3] performs function-level instrumentation, and reports measures the "function time" metric, which incorporates a heuristic that assigns higher cost to code that runs while other processors are idle. Cilkprof's metrics measure scalability more directly.

Some systems such as Tau [34] support both sampling and instrumentation. Tau has a general instrumentation framework similar to Vampir.

Kremlin [19, 26] was designed to help programmers parallelize serial programs by highlighting regions of code that can be parallelized fruitfully. Kremlin reports a "self-parallelism" metric (based on gprof's [22] self-time metric) to determine regions of code that can be parallelized to yield large speedups. In particular, Kremlin defines self-parallelism of a function in terms of the work of the function and the critical paths of the function and its children. Whereas Kremlin allows the user to find places to parallelize code, Cilkprof allows the user to verify that the code was parallelized correctly to achieve good scalability.

Chapter 8

Conclusion

Cilkprof measures information for strands and call sites. For strands, Cilkprof reports the work and span contribution to the overall work and strand. For call sites, Cilkprof reports the work and span contribution to the overall work and span. To measure the information, Cilkprof adds instrumentation at the critical points of Cilk's fork-join programming model. The measurements are stored in a sparse array, a hybrid linked list and array representation, that guarantees constant amortized time overhead per measurement while avoiding the space blowup typical of traces. In addition, the algorithm and data structures are localized so that the code is relatively easy to parallelize.

The design and implementation of Cilkprof created a framework on which other Cilk tools can be built. By adding such a framework, a general set of Cilk tools can quickly be built without having to modify the compiler or runtime system. Examples of such tools could include a parallelized race detector and a parallelized cache behavior monitor. Furthermore, user would be free to write custom profilers, which is something most other tools do not offer. Finalizing a design for a flexible profiling model for the user while maintaining good profiling performance will be a challenge, but when done well, can be invaluable for programmers.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov. 1997.

[3] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *SIGMETRICS*, pages 115–125. ACM, 1990.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[5] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 132–141, Honolulu, Hawaii, Apr. 1996.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, Feb. 1998.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[8] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, Apr. 1974.

[9] H. Brunst, M. Winkler, W. E. Nagel, and H.-C. Hoppe. Performance optimization for large scale computing: The scalable VAMPIR approach. In V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, editors, *ICCS*, volume 2074, pages 751–760. Springer, 2001.

[10] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, Apr. 1965.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[12] I. Dooley, C. W. Lee, and L. Kale. Continuous performance monitoring for large-scale parallel applications. In *16th Annual IEEE International Conference on High Performance Computing (HiPC 2009)*, December 2009.

[13] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, Mar. 1989.

[14] J. Evans. Scalable memory allocation using jemalloc. `https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`. [Online; accessed 20-May-2014].

[15] J. Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*. Citeseer, 2006.

[16] M. Feng and C. 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, June 1997.

[17] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, 2009.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[19] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting `gprof` for the multicore age. *SIGPLAN Not.*, 46:458–469, June 2011.

[20] GNU. The gnu c library (glibc). `http://www.gnu.org/software/libc/libc.html`. [Online; accessed 20-May-2014].

[21] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

[22] S. Graham, P. Kessler, and M. McKusick. An execution profiler for modular programs. *Software—Practice and Experience*, 13:671–685, 1983.

[23] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, 2010.

[24] Intel. Cilk plus. `http://www.cilkplus.org/download#gcc-development-branch`. [Online; accessed 18-May-2014].

[25] Intel. Download intel cilk plus software development kit. https://software.intel.com/en-us/articles/download-intel-cilk-plus-software-development-kit. [Online; accessed 18-May-2014].

[26] D. Jeon, S. Garcia, C. Louie, S. K. Venkata, and M. B. Taylor. Kremlin: Like gprof, but for parallelization. In *PPoPP*, San Antonio, TX, Feb. 2011.

[27] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[28] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing: Proceedings of the 2nd International Workshiop on Parallel Tools for High Performance Computing*, pages 139–155, 2008.

[29] C. W. Lee and L. V. Kale. Scalable techniques for performance analysis. Technical Report 07-06, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, May 2007.

[30] C. W. Lee, C. Mendes, and L. V. Kalé. Towards scalable performance analysis and visualization through data reduction. In *13th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2008.

[31] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[33] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov. 1995.

[34] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[35] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. *SIGPLAN Not.*, 44:229–240, February 2009.