Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming

by

Seth Copen Goldstein

B.S.E. (Princeton University) 1985
M.S.E. (University of California–Berkeley) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science
in the
GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David E. Culler, Chair

Professor Susan L. Graham

Professor Paul McEuen

Professor Katherine A. Yelick

Fall 1997

Abstract

Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming

by

Seth Copen Goldstein

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

Many modern parallel languages support dynamic creation of threads or require multithreading in their implementations. The threads describe the logical parallelism in the program. For ease of expression and better resource utilization, the logical parallelism in a program often exceeds the physical parallelism of the machine and leads to applications with many fine-grained threads. In practice, however, most logical threads need not be independent threads. Instead, they could be run as sequential calls, which are inherently cheaper than independent threads. The challenge is that one cannot generally predict which logical threads can be implemented as sequential calls. In lazy multithreading systems each logical thread begins execution sequentially (with the attendant efficient stack management and direct transfer of control and data). Only if a thread truly must execute in parallel does it get its own thread of control.

This dissertation presents new implementation techniques for lazy multithreading systems on conventional machines and compares a variety of design choices. We develop an abstract machine that makes explicit the design decisions for achieving lazy multithreading. We introduce new options on each of the four axes in the design space: the storage model, the thread representation, the disconnection method, and the queueing mechanism. Stacklets, our new storage model, allows parallel calls to maintain the invariants of sequential calls. Thread seeds, our new thread representation, allows threads to be stolen without requiring thread migration or shared memory. Lazy-disconnect, our new disconnection method, does not restrict the use of pointers. Implicit and Lazy queueing, our two new queueing mechanisms, eliminate the need for explicit bookkeeping. Additionally, we develop a core set of compilation techniques and runtime primitives that form the basis for the efficient implementation of any design point.

We have evaluated the different approaches by incorporating them into a compiler for an explicitly parallel extension of Split-C. We show that there exist points in the design space (e.g., stacklet, thread seeds, lazy-disconnect, and lazy queueing) for which fine-grained

parallelism can be efficiently supported even on distributed memory machines, thus allowing programmers freedom to specify the parallelism in a program without concern for excessive overhead.

Contents

List of Figures **vii**

List of Tables **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 The Goal 3
1.3 Contributions 6
1.4 Road Map 6

2 Multithreaded Systems **8**
2.1 The Multithreaded Model 8
2.2 Using Fork for Thread Creation 11
2.3 Defining the Potentially Parallel Call 11
2.4 Problem Statement 13

3 Multithreaded Abstract Machine **14**
3.1 The Multithreaded Abstract Machine (MAM) 15
3.1.1 Threads 16
3.1.2 Processors and MAM 17
3.1.3 Threads Operations 20
3.1.4 Inlets and Inlet Operations 22
3.1.5 Thread Scheduling 24
3.1.6 Discussion 26
3.2 Sequential Call and Return 27
3.3 MAM/DF—Supporting Direct Fork 27
3.3.1 The MAM/DF Scheduler 29
3.3.2 Thread Operations in MAM/DF 30
3.3.3 Continuation Stealing 33
3.3.4 Thread Seeds and Seed Activation 35
3.3.5 Closures 40
3.3.6 Discussion 41
3.4 MAM/DS—Supporting Direct Return 41
3.4.1 MAM/DS Operations 44

3.4.2 Discussion 49
3.5 The Lazy Multithreaded Abstract Machine (LMAM) 50
3.6 Disconnection and the Storage Model 51
3.6.1 Eager-disconnect 52
3.6.2 Lazy-disconnect 54
3.7 Summary 55

4 Storage Models 57
4.1 Storing a Thread's Internal State 58
4.2 Remote Fork 59
4.3 Linked Frames 59
4.3.1 Operations on Linked Frames 60
4.3.2 Linked Frame Timings 60
4.4 Multiple Stacks 62
4.4.1 Stack Layout and Stubs 62
4.4.2 Stack Implementation 64
4.4.3 Operations on Multiple Stacks 65
4.4.4 Multiple Stacks Timings 69
4.5 Spaghetti Stacks 71
4.5.1 Spaghetti Stack Layout 72
4.5.2 Spaghetti Stack Operations 72
4.5.3 Spaghetti Stack Timings 74
4.6 Stacklets 76
4.6.1 Stacklet Layout 76
4.6.2 Stacklet Operations 76
4.6.3 Stacklet Stubs 78
4.6.4 Compilation 78
4.6.5 Timings 79
4.6.6 Discussion 79
4.7 Mixed Storage Model 80
4.8 The Memory System 81
4.9 Summary 82

5 Implementing Control 84
5.1 Representing Threads 84
5.1.1 Sequential Call 84
5.1.2 The Parallel Ready Sequential Call 85
5.1.3 Seeds 86
5.1.4 Closures 87
5.1.5 Summary 88
5.2 The Ready and Parent Queues 88
5.2.1 The Ready Queue 88
5.2.2 The Parent Queue 89
5.2.3 The Implicit Queue and Implicit Seeds 89
5.2.4 The Explicit Queue and Explicit Seeds 91

5.2.5 Interaction Between the Queue and Suspension 93
5.3 Disconnection and Parent-Controlled Return Continuations 94
5.3.1 Compilation Strategy 96
5.3.2 Lazy-disconnect 97
5.3.3 Eager-disconnect 97
5.4 Synchronizers 98
5.5 Lazy Parent Queue 100
5.6 Costs in the Seed Model 101
5.6.1 The Implicit Parent Queue 102
5.6.2 The Explicit Parent Queue 103
5.6.3 The Lazy Parent Queue 104
5.7 Continuation Stealing 105
5.7.1 Control Disconnection 105
5.7.2 Migration 106
5.8 Integrating the Control and Storage Models 107
5.8.1 Linked Frames 107
5.8.2 Stacks and Stacklets 107
5.8.3 Spaghetti Stacks 108
5.9 Discussion 110

6 Programming Languages 111
6.1 Split-C+threads 111
6.2 Example Split-C+threads Program 112
6.3 Thread Extensions to Split-C 115
6.3.1 Fork-set statement 115
6.3.2 Pcall Statement 116
6.3.3 Fork statement 116
6.3.4 Start Statement 116
6.3.5 Suspend Statement 117
6.3.6 Yield Statement 118
6.3.7 Function Types 118
6.3.8 Manipulating Threads 118
6.4 Split-Phase Memory Operations and Threads 119
6.5 Other Examples 120
6.5.1 Lazy Thread Style 120
6.5.2 I-Structures and Strands 122
6.6 Split-C+Threads Compiler 125
6.7 Id90 128

7 Empirical Results 129
7.1 Experimental Setup 130
7.2 Compiler Integration 130
7.3 Eager Threading 131
7.4 Comparison to Sequential Code 134
7.4.1 Register Windows vs. Flat Register Model 134

7.4.2 Overall Performance Comparisons 135
7.4.3 Comparing Memory Models 135
7.4.4 Thread Representations 139
7.4.5 Disconnection 143
7.4.6 Queueing 145
7.5 Running on the NOW 147
7.6 Using Copy-on-Suspend 152
7.7 Suspend and Steal Stream Entry Points 153
7.8 Comparing Code Size 153
7.9 Id90 Comparison 154
7.10 Summary 155

8 Related Work **157**

9 Conclusions **160**

A Glossary **163**

Bibliography **168**

List of Figures

1.1 *Examples of potentially parallel calls.* 4

2.1 *One example implementation of a logically unbounded stack assigned to each thread. The entire structure is called a cactus stack.* 9
2.2 *Logical task graphs.* 12

3.1 *The syntax of the formal descriptions.* 15
3.2 *Formal definition of a thread as a 7-tuple.* 16
3.3 *Formal definition of a processor as a 4-tuple.* 17
3.4 *Formal definition of the MAM as a 5-tuple.* 18
3.5 *Example translation of a function into psuedo-code for the MAM.* 19
3.6 *The thread exit instruction.* 20
3.7 *The yield instruction.* 21
3.8 *The suspend operation.* 21
3.9 *The fork operation for MAM.* 21
3.10 *The send instruction.* 23
3.11 *The ireturn instruction.* 24
3.12 *The legal state transitions for a thread in MAM.* 25
3.13 *The enable instruction.* 25
3.14 *Describes how an idle processor gets work from the ready queue.* 25
3.15 *Redefinition of the machine and thread for MAM/DF.* 29
3.16 *The legal state transitions for a thread in the MAM/DF without work stealing.* 30
3.17 *The fork operation under MAM/DF.* 31
3.18 *The dfork operation under MAM/DF transfers control directly to the forked child.* 31
3.19 *The dexit operation under MAM/DF transfers control directly to the parent.* 32
3.20 *The suspend operation under MAM/DF for a lazy thread.* 33
3.21 *The new state transitions for a thread in MAM/DF using continuation stealing.* 34
3.22 *Formal semantics for stealing work using continuation stealing.* 34
3.23 *Three possible resumption points after executing* `dfork`. 35
3.24 *Example thread seed and seed code fragments associated with* `dforks`. . . . 36
3.25 *Work stealing though seed activation.* 38

3.26 *Example translation of a function into psuedo-code for the MAM/DF using thread seeds.* 38
3.27 *Example of seed activation.* 39
3.28 *The seed return instruction is used to complete a seed routine.* 39
3.29 *Example translation of a function into psuedo-code for the MAM/DS using thread seeds.* 43
3.30 *Redefinition of a thread for MAM/DS.* 43
3.31 *A pseudo-code sequence for an* `lfork` *with its two associated return paths.* 45
3.32 *The lfork operation in MAM/DS.* 46
3.33 *The lreturn operation in MAM/DS when the parent and child are connected and the parent has not been resumed since it* `lfork`*ed the child.* 46
3.34 *The suspend operation in MAM/DS.* 47
3.35 *The lreturn operation when the parent and child have been disconnected by a continuation stealing operation.* 48
3.36 *The lreturn operation in MAM/DS when a thread seed in the parent has been activated.* 49
3.37 *The lfork operation in LMAM with multiple stacks.* 51
3.38 *Eager-disconnect used to disconnect a child from its parent when a linked storage model is used.* 52
3.39 *An example of eager-disconnect when the child is copied to a new stack.* 52
3.40 *An example of eager-disconnect when the parent portion of the stack is copied to a new stack.* 53
3.41 *The suspend operation under LMAM/MS and LMAM/S using child copy for eager-disconnect.* 53
3.42 *An example of how a parent and its child are disconnected by linking new frames to the parent.* 55
3.43 *The suspend operation for LMAM/MS and LMAM/S using lazy-disconnect.* 55
3.44 *The lfork operation under LMAM/MS using lazy-disconnect after a thread has been disconnected from a child.* 56

4.1 *A cactus stack using linked frames.* 59
4.2 *A cactus stack using multiple stacks.* 62
4.3 *Layout of a stack with its stub.* 63
4.4 *Allocating a sequential call or an* `lfork` *on a stack.* 65
4.5 *Allocating a new stack when a fork is performed.* 66
4.6 *Disconnection using child-copy for a suspending child.* 68
4.7 *Disconnection caused by seed activation.* 68
4.8 *Disconnection caused by continuation stealing.* 69
4.9 *Two examples cactus stacks using spaghetti stacks.* 71
4.10 *Allocation of a new frame on a spaghetti stack.* 72
4.11 *Deallocating a child that has run to completion.* 73
4.12 *Deallocating a child that has suspended but is currently at the top of the spaghetti stack.* 73
4.13 *Deallocating a child that has suspended and is in the middle of the stack.* 74
4.14 *A cactus stack using stacklets.* 76

4.15 *The basic form of a stacklet.* 77
4.16 *The result of a sequential call which does not overflow the stacklet.* 77
4.17 *The result of a fork or a sequential call which overflows the stacklet.* 77
4.18 *A remote fork leaves the current stacklet unchanged and allocates a new stacklet on another processor.* 78

5.1 *Pseudo-code for a parallel ready sequential call and its associated return entry points.* 86
5.2 *Example of a closure and the code to handle it.* 87
5.3 *An example of a cactus stack split across two processors with the implicit parent queue embedded in it.* 90
5.4 *Example of when a fork generates a thread seed, and when it does not.* 90
5.5 *Example of how an explicit seed is stored on the parent queue.* 92
5.6 *A comparison of the different implementations of suspend depending upon the type of parent queue.* 93
5.7 *Example implementation of two* `lfork`*s followed by a join.* 95
5.8 *Each fork-set is compiled into three code streams.* 96
5.9 *An example of applying the synchronizer transformation to two forks followed by a join.* 99
5.10 *The table on the left shows how a seed is represented before and after conversion. The routines that aid in conversion are shown on the right.* 101
5.11 *Example implementation of a parallel ready sequential call with its associated inlets in the continuation-stealing control model.* 105
5.12 *A parallel ready sequential call with its associated steal and suspend routines, helper inlets, and routines for handling thread migration.* 106
5.13 *A* `local-thread` *stub that can be used for either stacks or stacklets.* 108
5.14 *Code fragments used to implement a minimal overhead spaghetti stack.* 109

6.1 *An implementation of the Fibonacci function in Split-C+threads.* 113
6.2 *The new keywords defined in Split-C+threads.* 114
6.3 *New syntax for the threaded extensions in Split-C+threads.* 114
6.4 *Example uses of* `forkset`*.* 115
6.5 *Implementation of I-structures in Split-C+threads using* `suspend`*.* 123
6.6 *The flow graph for the sequential and suspend code-streams for a forkable function compiled with an explicit parent queue, the stacklet storage model, lazy-disconnect, and thread seeds.* 126

7.1 *The* `grain` *micro-benchmark.* 132
7.2 *Improvement shown by lazy multithreading relative to eager multithreading.* 132
7.3 *Memory required for eager and lazy multithreading.* 133
7.4 *Slowdown of uniprocessor multithreaded code running serially over GCC code.* 136
7.5 *Average slowdown of multithreaded code running sequentially on one processor over GCC code.* 137
7.6 *Comparison of the three memory models.* 138

7.7 *Comparison of continuation stealing and seed activation when all threads run to completion.* . . . 140
7.8 *The execution time ratio between the continuation-stealing model and the thread-seed model when the first* `pcall` *of the fork-set suspends.* . . . 141
7.9 *Comparing the effect of which of the ten* `pcalls` *in a fork-set suspends.* . . . 142
7.10 *Comparing thread representations as the percentage of leaves that suspend increases.* . . . 142
7.11 *Comparison of eager and lazy disconnection when all the threads run to completion.* . . . 143
7.12 *Comparison of eager and lazy disconnection as the number of leaves suspending changes.* . . . 144
7.13 *Comparison of eager and lazy disconnection when the first* `pcall` *in a fork-set with two (or ten)* `pcalls` *suspends.* . . . 144
7.14 *Comparison of the three queueing methods—explicit, implicit, and lazy—for* `grain` *when all threads run to completion.* . . . 146
7.15 *Comparison of the queueing mechanisms as the number of threads suspending increases.* . . . 146
7.16 *Speedup of* `grain` *on the NOW relative to a version compiled with GCC on a single processor.* . . . 147
7.17 *A breakdown of the overhead when running on one processor of NOW.* . . . 148
7.18 *Speedup of lazy threads (stacklets, thread seeds, lazy disconnection, and explicit queue) on the CM-5 compared to the sequential C implementation as a function of granularity.* . . . 149
7.19 *The effect of reducing the polling frequency.* . . . 149
7.20 *Speedup of* `grain` *for the explicit and lazy queue models.* . . . 150
7.21 *Speedup of an unbalanced* `grain` *where there is more work in the second* `pcall` *of the fork-set.* . . . 151
7.22 *Speedup of an unbalanced* `grain` *where there is more work in the first* `pcall` *of the fork-set.* . . . 151
7.23 *Speedup and efficiency achieved with and without copy-on-suspend.* . . . 152
7.24 *The amount of code dilation for each point in the design space.* . . . 154
7.25 *Which points in the design space are effective and why.* . . . 155

List of Tables

2.1 *Comparison of features supported by different programming systems.* 9

4.1 *Times for the primitive operations using linked frames.* 61
4.2 *Times for the primitive operations using multiple stacks.* 70
4.3 *Times for the primitive operations using spaghetti stacks.* 75
4.4 *Times for the primitive operations using stacklets.* 79
4.5 *Times for the primitive operations using a sequential stack for purely sequential calls, stacklets for potentially parallel and suspendable sequential calls with moderate sized frames, and the heap for the rest.* 80

5.1 *The cost of the basic operations using an implicit parent queue.* 103
5.2 *The cost of the basic operations using an explicit parent queue.* 104
5.3 *The cost of the basic operations using an lazy parent queue.* 104

7.1 *Comparison of basic thread operations using library packages versus compiler integration.* 131
7.2 *Comparison of* `grain` *compiled by GCC for register windows and for a flat register model.* 134
7.3 *The minimum slowdowns relative to GCC-compiled executables for the different memory models and where in the design space they occur for a grain size of 8 instructions.* 135
7.4 *Dynamic runtime in seconds on a SparcStation 10 for the Id90 benchmark programs under the TAM model and lazy threads with multiple strands.* . . . 155

Chapter 1

Introduction

1.1 Motivation

As parallel computing enters its adolescence its success depends in large part on the performance of parallel programs in the non-scientific domain. Typically these programs are less structured in their control flow, resulting in irregular parallelism and the increased need to support multiple dynamically created threads of control. Hence, the success of parallel computing hinges on the efficient implementation of multithreading. This dissertation shows how to efficiently implement multithreading on standard architectures using novel compilation techniques, making it possible to tackle a broad class of problems in modern parallel computing. The goal of this thesis to present and analyze an efficient multithreading system.

Many modern parallel languages provide methods for dynamically creating multiple independent threads of control, such as forks, parallel calls, futures, object methods, and non-strict evaluation of argument expressions. These threads describe the *logical parallelism* in the program, i.e., the parallelism that would be present if there were an infinite number of processors to execute the threads. The language implementation maps the dynamic collection of threads onto the set of physical processors executing the program, either by providing its own language-specific scheduling mechanisms or by using a general threads package. These languages stand in contrast to languages with a single logical thread of control, such as High Performance Fortran [38], or a fixed set of threads, such as Split-C [15] and MPI [21], which are typically targeted to scientific applications.

There are many reasons to have the logical parallelism of the program exceed the physical parallelism of the machine, including ease of expression and better resource utilization in the presence of synchronization delays, load imbalance, and long communication latency [44, 63]. When threads are available to the programmer they can be used to cleanly express multiple goals, co-routines, and parallelism within a single program. Threads can be used by the programmer or compiler to hide the communication latency of remote memory references. More importantly they can be used to hide the potentially long latency involved in accessing synchronizing data structures or other synchronization constructs. Load balance can also be improved when many threads are distributed among fewer processors on a parallel system. Finally, the semantics of the language or the synchronization primitives may allow dependencies to be expressed in such a way that progress can be made only by interleaving multiple threads, effectively running them in parallel even on a single processor [45].

To support unbounded logical parallelism, regardless of the language model, the underlying execution model must have three key features. First, the control model must allow the dynamic creation of multiple threads with independent lifetimes. Second, the model must allow switching between these threads in response to synchronization events and long-latency operations. Third, since each thread may require an unbounded stack, the storage model is a tree of stacks, called a *cactus stack* [29]. Unfortunately, a parallel call or thread fork is fundamentally more expensive than a sequential call because of the storage management, data transfer, control transfer, scheduling, and synchronization involved. Much previous work has sought to reduce this cost by using a combination of compiler techniques and clever runtime representations [16, 36, 44, 48, 49, 53, 57, 61, 63], or by supporting fine-grained parallel execution directly in hardware [3, 34, 50]. These approaches, among others, have been used in implementing parallel programming languages such as ABCL [65], CC++ [13], Charm [35], Cid [48], Cilk [7], Concert [36], Id90 [16, 49], Mul-T [39], and Olden [12]. In some cases, the cost of the fork is reduced by severely restricting what can be done in a thread. Lazy Task Creation [44], implemented in Mul-T, is the most successful in reducing the cost of a fork. However, in all of these approaches, a fork remains substantially more expensive than a simple sequential call.

1.2 The Goal

Our goal is to support an unrestricted parallel thread model and yet reduce the cost of thread creation and termination to little more than the cost of a sequential call and return. We also want to reduce the cost of switching and scheduling threads so that the fine-grained parallelism needed to implement unstructured parallel programs will not adversely affect the overall performance of these programs.

We observe that fine-grained parallel languages promote the use of small threads that are often short lived, on the order of a single function call. However, the advantages of fine-grained parallelism can be overwhelmed by the overhead of creating a new thread, which is inherently a more complex and expensive operation than sequential call. Fortunately, logically parallel calls can often be run as sequential calls. For example, once all the processors are busy, there may be no need to spawn additional work, and in the vast majority of cases the logic of the program permits the child to run to completion while the parent is suspended. Thus, a parallel call should be viewed as a *potentially parallel call*—that is, a point in the computation where an independent thread may, but will not necessarily be created. The problem is that in general when the thread is created there is no way to know whether it will complete without suspending or if it will need to run in parallel with its parent.

In Figure 1.1 we show three examples of forks that are potentially parallel calls. In Figure 1.1a `fork fib(n)` is a potentially parallel call because if all the processors are busy fib(n) can run to completion while the invoker is suspended. In Figure 1.1b, the the thread created by `fork consumer(queue)` can run to completion if the the shared queue, `q`, is never empty and `getItem(q)` never suspends. However, it is a potentially parallel call because if the queue becomes empty the thread running `consumer` will suspend and it needs to run in parallel with a thread that fills up the shared queue. The last fork of `task` in Figure 1.1c is a potentially parallel call because if `iptr` points to another processor, then the get operation (`i = *iptr;`) can suspend causing the thread running `task` to suspend. If, `iptr` points to a location on the same processor as is running the thread, then the operation will complete immediately, in other words, it can run to completion in serial with its parent.

We implement a potentially parallel call almost exactly like a stack-based sequential call to exploit the fact that most such calls complete without suspending. This is different than a typical implementation which assumes all parallel calls will run in paral-

```
int fib(int n)
{
   int x,y;

   if (n <= 2) return 1;
   forkset {
      // control leaves forkset when all forks in forkset have completed.
      x = fork fib(n-1);
      y = fork fib(n-2);
   }
   return x+y;
}
```

(a) `fork fib(n)`

```
void consumer(SharedQueue q)
{
   int item;

   while ((item = getItem(q)) != END)
   {
      ⋮
   }
}
```

(b) `fork consumer(queue)`

```
void task(int global* iptr)
{
   int x;

   ⋮
   i = *iptr;
   ⋮
}
```

(c) `fork task((int global *)&i)`

Figure 1.1: *Examples of potentially parallel calls.*

lel. The typical implementation allocates a child frame on the heap, stores the arguments into the frame, schedules the child. Later the child is run, stores its results in its parent frame, schedules the parent, and returns the child frame to the heap store. We, on the other hand, implement the potentially parallel call as a *parallel-ready sequential call.* We call our overall approach *lazy threads.* The call allocates the child frame on the stack of the parent, like a sequential call. As with a sequential call, control is transferred directly to the child (suspending the parent), arguments are transferred in registers, and, if the child returns without suspending, results are transferred in registers when control is returned to the parent. Moreover, even if the child suspends, the parent is resumed and continues to execute without copying the stack. Instead, the suspending child assumes control of the parent's stack, and further children called by the parent are executed on stacks of their own. In other words, the suspending child steals the thread out from under its parent. If work is needed by another processor, the parent can be resumed to spawn children on that processor.

The key to an effective implementation of lazy threads is to invoke the child—and suspend the parent—without performing any bookkeeping operations above those required by a sequential call and yet allow the parent to continue before its lazy child completes. In order to reach this goal we need to pay attention to four components of the invocation and suspension processes.

- How the thread state is stored. We investigate four storage models: linked frames, multiple stacks, spaghetti stacks, and stacklets.
- How work in the parent is represented. We study three different representations for work in the parent: thread seeds, closures, and continuations.
- How work is enqueued so it can later be found. We investigate three queuing methods: implicit queues, explicit queues, and lazy queues.
- How the child and parent are disconnected so that the parent may continue before the child completes. We study two disconnection methods: eager-disconnect and lazy-disconnect.

The advantage of lazy threads is that when they run sequentially (i.e., when they run to completion without suspending), they have nearly the same efficiency as a sequential call. Yet the cost of elevating a lazy thread into an independent thread of control is close to that of executing it in parallel outright.

Unlike previous approaches to lazy parallelism, e.g. load-based inlining and Lazy Task Creation [44], our code-generation strategy avoids bookkeeping operations like creating task descriptors, initializing synchronization variables, or even explicitly enqueueing tasks. Furthermore, our approach does not limit the class of legal programs as opposed to some previous approaches, e.g, [7, 19, 63]. Through careful attention to the program representation, we pay almost nothing for the ability to elevate a sequential call into a full thread on demand. This contrasts with previous approaches to lazy parallelism based on continuation stealing in which the parent continues in its own thread, forcing stack copies and migration.

1.3 Contributions

The main contribution of this thesis is the development of a set of primitives that can be used to compile fine-grained parallel programs efficiently.

- We introduce new memory management primitives, called stacklets and stubs, to more efficiently manage the cactus stack.
- We introduce new control primitives, called thread seeds and parent-controlled return continuations, to reduce the cost of scheduling and synchronizing threads.
- We analyze, for the first time, the four dimensional design space for multithreaded implementations: (1) The storage model (linked frames, multiple stacks, spaghetti stacks, and stacklets), (2) the thread representations (continuations, thread seeds, and closures), (3) the queue mechanism (implicit, explicit, or lazy), and (4) the disconnection method (eager or lazy). We show how each point in the space can be implemented using the same set of primitives.
- Using the techniques presented in this thesis, we reimplement several earlier approaches, allowing us to compare for the first time the different points in the design space in the same environment.
- We develop a new parallel language, Split-C+threads, which is one of the source languages for our experiments.
- We develop a compiler for a new multithreading language, Split-C+threads, a multithreaded extension of Split-C, by implementing our compilation framework in GCC.
- We implement a new back-end of the Id90 compiler, a second source language for our experiments, which produces significantly faster executables than previous approaches.

1.4 Road Map

Chapter 2 defines the type of multithreaded systems on which this thesis focuses. Chapter 3 presents an abstract multithreaded machine which embodies the principal concepts behind these systems. We use the abstract machine to explore the four axes of the design space: storage models, disconnection methods, thread representations, and queueing methods. In Chapter 4 we describe the implementations of the different possible storage models and how they affect the overall efficiency of multithreading. In Chapter 5 we describe the mechanisms used by the different control models and show how they interact with the storage models. Chapter 6 describes the two source languages, Split-C+threads and Id90, used in our experiments and the techniques used in their compilation. Chapter 7 presents experimental results which compare the different multithreaded implementations using our compilation techniques and the mechanisms introduced in Chapters 4 and 5. Chapter 8 discusses related work. Finally, Chapter 9 contains a summary and concluding remarks. A glossary of special terminology used in this dissertation can be found in Appendix A.

Chapter 2

Multithreaded Systems

In this chapter we define the scope of the multithreaded systems that we study in this dissertation. We begin by defining the attributes of a multithreaded system. We then describe the language constructs for thread creation and define the potentially parallel calls which can be optimized by our system.

2.1 The Multithreaded Model

A *multithreaded* system is one that supports multiple threads of control, where each thread is a single sequential flow of control. We classify multithreaded systems into three categories: limited multithreading systems, virtual software processor systems, and virtual hardware processor systems. Limited multithreading systems are those that support multiple threads of control, but in which each thread must run to completion or a thread does not have an unbounded stack. This thesis concerns itself primarily with virtualizing a software processor, i.e., a multithreaded system in which each thread is a virtual processor with its own locus of control and unbounded stack, but which does not support preemption. A thread in a virtual hardware processor system is a virtual processor in every sense, i.e., each thread has its own locus of control, an unbounded stack, and supports preemption.

There are five features that such a system must support to virtualize a hardware processor:

- **Creation:** It must be able to create new threads dynamically during the course of the program. This is in direct contrast to the single-program multiple-data (SPMD)

System	Features				
	Dynamic creation	Lifetime	Suspension	Stack	Pre-emption
Split-C [15]	no	fixed	no	Unbounded	no
Cilk [8]	Yes	$<$ parent	no	N/A	no
Filaments [19]	Yes	$<$ parent	no	N/A	no
C-Threads [14]	Yes	independent	yes	fixed	no
NewThreads [42]	Yes	independent	yes	fixed	no
Chorus [54]	Yes	independent	yes	fixed	no
Solaris Threads [56]	Yes	independent	yes	fixed	yes
Nexus [22]	Yes	independent	yes	fixed	yes
Lazy Threads	Yes	independent	yes	unbounded	no

Table 2.1: *Comparison of features supported by different programming systems. N/A indicates a feature which is not applicable. In particular, in systems which do not support suspension, threads always run to completion. Thus each thread does not need its own unbounded stack; all threads can share a single stack.*

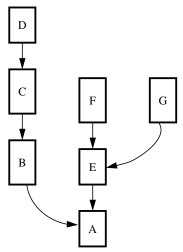


Figure 2.1: *One example implementation of a logically unbounded stack assigned to each thread. The entire structure is called a cactus stack.*

programming model where the number of threads is fixed throughout the life of a program (e.g., in Split-C [15] and Fortran-77+MPI [21]).

- **Lifetime:** Each thread may have an independent lifetime. If the lifetime of a thread is limited to the scope of its parent or is fixed for the duration of the program, then there are types of programs, described in Section 2.3, which cannot be represented.

- **Stack:** Each thread has a logically unbounded stack. Many multithreaded systems limit the size of a thread's stack (Chorus [54], QuickThreads [37], etc.), which in turn restricts the kinds of operations that a thread may perform. In contrast, we provide a logically unbounded stack. This does not require that the stack be implemented in contiguous memory. Figure 2.1 shows a logically unbounded stack implemented by linking heap-allocated frames for each function to the frame of the invoking function. Empirically this structure is tall and thin, so we call it a *cactus stack* [29].

- **Suspension:** Threads may suspend and later be resumed. This implies that they can interact with (and depend on) each other. Systems that do not support suspension (e.g. Cilk [7] and Filaments [19]) severely restrict the set of programs that can be expressed. For instance, these systems cannot use threads in producer/consumer relationships.

- **Preemption:** A thread may be preempted by another thread. Some multithreaded systems also allow preemption of threads, which allows fair scheduling of threads. Without preemption, threads may never get a chance to execute and without some thought programs can become livelocked. While preemption is essential for operating system tasks it is less important for application programs.

  If a system supports preemption, then a thread is not guaranteed to run indefinitely. In fact, the operating system may at any time remove a thread from its processor and start another thread. This is possibly the most controversial feature of multithreaded systems. While required in systems where threads are competing for resources, it may be a burden in systems that support multithreading among cooperating threads. In the former case, if threads are nonpreemptive, then one thread could block competing threads from ever executing. In the latter case, however, preemption is not needed to guarantee that a thread will run. For example, a thread that is computing a result needed by other threads will always get to run because the threads needing the result will eventually block. In addition, if preemption exists, then explicit action is required to ensure atomicity of operations. In the absence of preemption a thread makes all decisions as to when it can be interrupted, so atomicity is easier to guarantee.[1] However, care must be taken to avoid constructs like spinlocks.

In the multithreaded systems we consider, logical parallelism is independent of physical parallelism. A multithreaded system does not specify whether the individual

[1] On a distributed memory machine, where local memory is owned by a processor and one processor cannot access another processor's local memory, atomicity is easily provided by ignoring the network. On shared memory machines this is harder, but for nonpreemptive systems, compilation techniques can provide atomicity when required.

threads are run on many hardware processors, on a single hardware processor by time-slicing, or on a hardware-multithreaded machine.

In short, a multithreaded system supports the dynamic creation of threads, each of which is suspendable, has an independent lifetime, and has a possibly unbounded stack. In addition, the system may allow threads to be preempted. In this thesis we are concerned with virtualizing the processor for a single cooperating task and thus we concentrate on multithreaded systems that do not support preemption. Our multithreaded system is aimed at application programs and not operating system tasks.

2.2 Using Fork for Thread Creation

Many constructs have been developed to create and manage multiple threads of control, such as forks, parallel calls, futures, active objects, etc. These constructs may be exposed to the programmer in library calls, as in P-Threads [33], or as part of the language, as in Mul-T [39] and Cilk [7], or, they may be hidden from the programmer but used by compiler to implement the language, as in Id90 [46].

Fork is the fundamental primitive for creating a new thread. When executed it creates a new thread, which we call the *child thread*, that runs concurrently with the thread that executed the fork, which we call the *parent thread*. While there are many methods to synchronize these threads, the most common is a join. The join acts as a barrier for a group of threads. When they have all executed the join, they join to become a single thread. All but one of the participating threads cease to exist after control has passed the point of the join. We call the threads that are synchronized by a single join a *fork-set*.

All of the other methods for creating threads can be implemented by means of fork. Henceforth, we use fork and parallel call interchangeably, and only they are mentioned explicitly.

2.3 Defining the Potentially Parallel Call

In this section we define the potentially parallel call by looking at the three kinds of threads that can be created in a multithreaded system: downward, upward, and daemon threads. This thesis concerns itself mainly with optimizing the potentially parallel call that

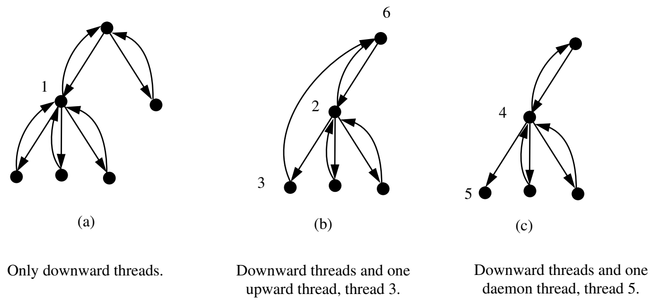


Figure 2.2: *Logical task graphs. Node 1 is a downward thread. Node 3 is an upward thread. Node 3 is a child of node 2. Node 6 is the youngest ancestor alive at the time of node 3's death. Node 5 is a daemon thread. Node 6 is a root thread.*

creates a downward thread. We also show how threads that ultimately require concurrency, including upward and daemon threads, can be implemented efficiently.

The *logical task graph* represents the interaction among all of the threads created during the life of a multithreaded program. Each node in the graph represents a thread. The directed edges represent the transfer of control between threads (See Figure 2.2). Each node has an edge to each of its children and to the youngest ancestor that is alive at its death. If no outgoing edge exists for a node, then the thread it represents was still executing at the end of the program. If no incoming edge ends at a node, then it is a root thread, i.e., one of the threads created at program startup. There can be more than one root thread.

By distinguishing three kinds of threads—downward, upward, and daemon—we are able to precisely define the potentially parallel call. A *downward thread* is one that terminates before its parent terminates, its ancestor edge points to its parent. All the threads in Figure 2.2.a are downward threads. An *upward thread* is one which terminates after its parent terminates.[2] For example, thread 3 in Figure 2.2.b is an upward thread, indicated by the upward edge going from thread 3 to thread 6, an ancestor other than its parent. A *daemon thread* is one which is not a root thread, but is still alive at the end of the program, i.e. has no outgoing edges.[3] For example, in Figure 2.2.c, thread 5 is a daemon thread.

When there are neither upward threads nor daemon threads created during an execution, the logical task graph equals its transpose, i.e., every edge participates in a

[2]The name "upward" comes about since the characteristics of upward threads mirror those of upward funargs.

[3]The name "daemon" is borrowed from Java [26].

simple cycle of length two. In this case, the only synchronization primitive needed for control is the join primitive. If a downward thread does not suspend, then the fork that created it could have been replaced with a sequential call. In other words, downward threads are those created by potentially parallel calls.

On the other hand, daemon and upward threads require that independent threads be created even if none of them suspend.[4] The creation of these threads always incurs the cost of a fork. Daemon and upward threads are not created with potentially parallel calls, but with parallel calls.

2.4 Problem Statement

This thesis presents a design space for implementing multithreaded systems without preemption. We introduce a sufficient set of primitives so that any point in the design space may be implemented using our new primitives. The primitives and implementation techniques that we propose reduce the cost of a potentially parallel call to nearly that of a sequential call, without restricting the parallelism inherent in the program. Using our primitives we implement a system (in some cases a previously proposed system) at each point in the design space.

[4]Similar to tail recursion elimination, some upward threads can be treated as downward threads. If an upward thread is to be joined with one of its ancestors, then before it terminates it must be given a continuation to the ancestor with which it will join. If the parent is a downward thread and creates a child thread as its last operation, it can give the child the parent's own return continuation and then terminate itself. In other words, the parent can wait for its child to complete before returning itself. For this reason, we treat the apparently upward child thread as a downward thread.

Chapter 3

Multithreaded Abstract Machine

In this chapter we describe the primitives needed to efficiently implement potentially parallel calls on a multithreaded system. We divide these primitives into two classes: storage primitives and control primitives. Storage primitives are used to manage the thread state and to maintain the global cactus stack. Control primitives are used to manage control flow through thread creation, suspension, termination, etc. These primitives support the potentially parallel call without restricting parallelism, yet bring the cost of thread creation and termination down to essentially the cost of the sequential call and return primitives.

As we formalize the multithreaded system, we introduce the abstract primitives that will be used in our implementations. We discuss the thread representation mechanisms: threads, closures, thread seeds, and continuations. We present the two basic methods for disconnecting a lazy thread from its parent and elevating the child to an independent thread: eager-disconnect and lazy-disconnect. We then formalize the four basic storage models that can be used to store thread state: linked frames, multiple stacks, stacklets, and spaghetti stacks. The different queueing methods are discussed in Chapter 5.

Our exposition proceeds from a basic machine (MAM) to one which efficiently executes potentially parallel calls (LMAM) in four steps. We begin by formalizing a basic multithreaded abstract machine (MAM). We next show how to make forks similar to calls by introducing a direct fork operation (MAM/DF). Next we introduce an abstract machine that makes the termination operation similar to the return mechanism (MAM/DS). Finally, we incorporate more efficient storage management into a lazy multithreaded abstract machine (LMAM). This last abstract machine has all the primitives necessary for efficient potentially parallel calls.

$<>$	Used to enclose a tuple.
inst[`ip`]	The instruction at address `ip`.
ϵ	Indicates that the field has no valid value.
$[)$	Represents a sequence which has been defined only on the values given, but which can expand infinitely, e.g., a stack.
$\top$	Can be any value of the correct type.
$\perp$	Used for fields with a null value.
$*x$	The contents of memory location x.
r_x	Register number x.
$x \leftarrow y$	x is assigned the value y.
$x \equiv y$	x is defined to be y.
$x = y$	x has the value y.

Figure 3.1: *The syntax of the formal descriptions.*

While there are many ways to distribute work among processors we focus on work stealing. *Work stealing* is a method of "pulling" work, i.e., threads, to idle processors. One of the focuses of this work is the representation of threads that are assigned to idle processors. We shall describe two methods of work stealing: continuation stealing and seed activation. Both continuation stealing and seed activation continue work in the parent while its child is still alive. Continuation stealing resumes the parent by resuming the parent's current continuation. Seed activation resumes work generated by the parent, the work is a child thread that would have been created if the parent's continuation had been stolen and the parent immediately executed a fork.

This chapter introduces a substantial amount of new terminology. The reader may want to refer to the glossary in Appendix A. It is suggested that on first reading the formal definitions and rewrite rules be skipped.

3.1 The Multithreaded Abstract Machine (MAM)

In this section we define a *multithreaded abstract machine* (MAM) and the components that make up all multithreaded systems, e.g. threads and processors. In MAM each thread operation is implemented in the most naive and direct manner possible. This will serve as the basis for the refinements required to make multithreading more efficient. It also models the multithreaded systems as implemented in much of the previous work.

Thread := $\langle \texttt{ip}, \texttt{sp}, t_p, \sigma, s, i_p, l \rangle$

Where	`ip`	is valid when the thread is not `running` and represents the address to continue at in the thread.
	`sp`	is valid when the thread is not `running` and represents the current stack pointer for the thread.
	t_p	is a pointer to its parent thread. If it has no parent thread (the case with daemon threads) then it is ϵ.
	σ	represents the current thread status. $\sigma \in \{\texttt{ready}, \texttt{running}, \texttt{idle}\}$. At any instance a thread can be in only one state.
	s	is the thread's stack.
	i_p	is the thread's return inlet, an instruction address. It points to the inlet used to return results to the parent. This field combined with t_p compromise the return register. Inlets are defined in Section 3.1.4.
	l	is either unlocked $(\cdot)$ or locked $(\otimes)$. A thread is locked to allow certain operations to be carried out atomically.

Figure 3.2: *Formal definition of a thread as a 7-tuple. The return register is broken down into its two components, the parent thread and the inlet address (fields three and six respectively). The state of the thread is included in the 7-tuple. Finally, a lock bit is included.*

3.1.1 Threads

In this work a *thread* is a locus of control that can perform calls to arbitrary nesting depth, suspend at any point, and fork additional threads. Threads are scheduled independently and are non-preemptive.

Each thread has an associated instruction pointer (`ip`), stack pointer (`sp`), stack (s), and return register(t_p and i_p). The *instruction pointer* points to the next instruction to be executed in the thread. The *stack pointer* points to the top of the thread's stack and the end of the current activation frame. The *return register* is used to return results to the parent and contains a continuation to an inlet in the thread's parent.[1]

The general registers are associated with the processor, not the thread. In particular, the general registers used to carry out the computation must be explicitly saved (or restored) when a thread exits (or is entered).

The formal definition of a thread, as shown in Figure 3.2, differs from the description in the previous paragraph to facilitate the definition of the operational semantics of

[1] Inlets are defined in Section 3.1.4.

$$\text{Processor} := \langle \texttt{ip}, \texttt{sp}, t, R \rangle$$

Where	`ip`	is the instruction pointer.
	`sp`	is the stack pointer.
	t	is the thread being executed by the processor. If no thread is being executed, then the thread executes the dispatch thread, which is indicated by the `ip` field having the address `wait`.
	R	is the set of general-purpose registers, $\{r_1, r_2, \ldots\}$.

Figure 3.3: *Formal definition of a processor as a 4-tuple.*

the MAM. In particular, each thread has an associated lock flag (l) and state field (σ). The lock flag is used to describe which thread and inlet operations must execute atomically with respect to other operations. The state field is used to describe the scheduling status of a thread: `running` indicates that the thread is assigned to a processor and currently executing, `ready` indicates that the thread is on the MAM ready queue and can be assigned to a processor, and `idle` indicates that the thread is neither running or ready. These states and thread scheduling transitions are fully described in Section 3.1.5.

Of course, any data from the thread needed for the operation of MAM can be stored in the thread's stack. We represent the thread as a separate tuple in order to make the rewrite semantics easier to understand.

3.1.2 Processors and MAM

A processor is the underlying hardware that threads execute on; it is what the threads are virtualizing. Each processor has an instruction pointer (`ip`), a stack pointer (`sp`), a running thread (t), and a set of general-purpose registers (R). The thread currently running on the processor has access to all the registers in R. We represent a processor as a 4-tuple as shown in Figure 3.3. Notice that the processor does not have a stack associated with it, but a stack pointer. The stack which is pointed to by `sp` is the one contained in t.

MAM is composed of a single global address space[2] (M and S) which is accessible by all threads, a set of (possibly only one) processors (P), and a ready queue (Q). The *ready queue* contains the set of threads that are ready to execute, i.e., those in the `ready` state. Despite its name, the ready queue is not necessarily a queue and nothing in the semantics

[2]The address space may be composed of distributed or shared memory.

$$\text{Machine} := \langle P, T, Q, S, M \rangle$$

Where	P	is the set of processors.
	T	is the set of all threads.
	Q	is the ready queue. This is the set of threads in the ready state.
	S	is the set of currently allocated stacks. Each $s \in S$ can grow infinitely. $(\bigcup_{s \in S} s) \subseteq M$ and $\forall s, t \in S, s \cap t = \emptyset$.
	M	is the entire global memory, including the heap and all of the stacks.

Figure 3.4: *Formal definition of the MAM as a 5-tuple.*

of MAM depends on it having any particular structure. It may also be distributed among the processors. We use the terms "enqueue" and "dequeue" to mean add to or remove from a data structure without requiring that the element be added at (or removed from) a particular location in the data structure. We represent the MAM formally as a 5-tuple as shown in Figure 3.4.

The code for a program on MAM is divided into two classes: codeblock and inlet. The *codeblock* code performs the actual work of the threads. The *inlet* code, described in Section 3.1.4, is the portion of the program that performs inter-thread communication. The program text is divided in this manner to highlight the fact that the codeblock code carries out the computation in the program while the inlet code is generated by the compiler to handle the communication.

We concern ourselves here with the instructions that affect multithreading. The instructions that may appear in a codeblock are `fork`, `exit`, `suspend`, `yield`, and `send`[3]. These instructions create, terminate, suspend, yield, and send data to a thread, respectively. The inlet instructions are `recv`, `enable`, and `ireturn`, which handle incoming communication to a thread. Standard instructions (e.g., arithmetic, logical, memory, control transfer, etc.) may be used in either codeblock or inlet code.

In order to give the reader an overview of how the individual instructions work together we now present an example program with a translation for the MAM. The reader is not expected to understand all the details of the individual instructions. Figure 3.5 shows an example translation of a simple function with two forks and a join. The first 9 lines are

[3]There is no explicit `join` instruction, as join is synthesized from more basic instructions.

```
int function()
{
     int a,b,c;

     a = fork X();
     b = fork Y();
     join;
     c = a+b;
     return c;
}
```

Codeblock Code

```
1: function:
2:   set     synch=2
3:   fork    inlet1 = X()
4:   fork    inlet2 = Y()
5:   suspend
6:   // enable will restart here.
7:   add     c = a + b
8:   send    tp, ip, c
9:   exit
```

Inlet Code

```
10:inlet1:
11:  recv    a
12:  add     synch = synch - 1
13:  cmp     synch, 0
14:  be      enab
15:  ireturn
16:
17:inlet2:
18:  recv    b
19:  add     synch = synch - 1
20:  cmp     synch, 0
21:  be      enab
22:  ireturn
23:
24:enab:
25:  enable
26:  ireturn
```

Figure 3.5: *Example translation of a function with two forks into psuedo-code for the MAM. t_p is the parent thread. i_p is the parent inlet.*

the codeblock code. They handle all the operations in the user code except for receiving the results from the spawned children. Lines 10–26 are the inlet code, which consists of two inlets. Each handles the result from a child thread and also schedules the thread when appropriate.

The join is accomplished by setting a synchronization variable, `synch`, to the number of forks in the fork-set, in this case 2 (in Line 2). When the children return they decrement the synchronization variable (in Lines 12 and 19) and, if the variable is zero, they enable the thread (in Line 25), indicating that the join has succeeded.

The function starts by setting the synchronization variable, and then the forks are performed followed by a `suspend`, which causes the runtime system to spawn another thread that is ready to run, i.e., a thread in the `ready` state. When the two children finish, the codeblock is resumed at Line 7. The thread finishes by sending the result, `c`, to an inlet, i_p, of its parent thread, t_p. The parent thread and inlet are both stored in the thread state when the thread is created.

$$
\frac{\begin{array}{llrcl}
\langle P, T, Q, S, M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \texttt{ip}, \texttt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\texttt{ip}] & = & \texttt{exit} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \texttt{running}, s, i, \cdot\rangle
\end{array}}{\begin{array}{llrcl}
\langle P, T', Q, S', M\rangle & \text{where} & p & \equiv & \langle \text{wait}, \epsilon, \epsilon, \epsilon\rangle \\
 & \text{and} & T' & \equiv & T - \{t\} \\
 & \text{and} & S' & \equiv & S - \{s\}
\end{array}}
$$

Figure 3.6: *The thread exit instruction.*

Each fork specifies the inlet that will be invoked when the child thread issues the `send` instruction. For example, `inlet1` on Line 10 is executed when the first child (the one executing the function `X`) finishes. The inlet stores the result into `a` as specified in the `recv` instruction on Line 11. Next the synchronization variable is decremented and tested against zero. If it is not zero, the inlet ends with an `ireturn` on Line 15. If the synchronization variable is zero, then both threads have finished and the codeblock can continue. The thread is made ready to run with by the `enable` in Line 25.

3.1.3 Threads Operations

In addition to standard sequential operations, there are four thread operations that can be performed by a thread: `fork`, `exit`, `yield`, and `suspend`. `fork` creates a new thread. `exit` terminates a thread and frees all the resources associated with it (see Figure 3.6). `yield` allows a thread to give up its processor by placing itself on the ready queue (see Figure 3.7). `suspend` causes a thread to give up its processor and marks the thread as idle (see Figure 3.8). There is no intrinsic join operation, nor is there any intrinsic operation for testing the status of a thread, but both can be synthesized.

`fork` creates and initializes a new thread (see Figure 3.9). `fork` has two mandatory arguments, the start-address and the inlet-address, and an optional list of parameters that is passed to the new thread.[4] We call the thread executing the instruction the *parent thread* and the newly created thread the *child thread*. When `fork` executes, the child thread's `ip` is set to the start-address and its `sp` is set to the base of its newly allocated stack. It is placed on the ready queue, and its parent continuation is set to the return continuation specified

[4] A more general model, such as TAM, separates thread creation from the passing of arguments. In the more general model, `fork` can only create the thread. `send` instructions are then used later to pass the new thread its arguments. TAM requires this more general model [16].

$$
\begin{array}{llrcl}
\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{yield} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{running}, s, i, \cdot\rangle \\
 & \text{and} & t' & \in & Q \\
 & \text{and} & t' & \equiv & \langle \mathtt{ip}', \mathtt{sp}', t'_p, \mathtt{ready}, s', i', \cdot\rangle \\
\hline
\langle P,T,Q',S,M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}', \mathtt{sp}', t', \epsilon\rangle \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{ready}, s, i, \cdot\rangle \\
 & \text{and} & t' & \equiv & \langle \epsilon, \epsilon, t'_p, \mathtt{running}, s', i', \cdot\rangle \\
 & \text{and} & Q' & \equiv & Q - \{t'\} \cup \{t\}
\end{array}
$$

$$
\begin{array}{llrcl}
\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{yield} \\
 & \text{and} & Q & = & \emptyset \\
\hline
\langle P,T,Q,S,M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t, \epsilon\rangle
\end{array}
$$

Figure 3.7: *The yield instruction. The first rule applies when there is at least one other thread that is ready to run. The second rule applies when there are no other threads in the ready queue; it is effectively a NOP.*

$$
\begin{array}{llrcl}
\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{suspend} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{running}, s, i, \cdot\rangle \\
\hline
\langle P,T,Q,S,M\rangle & \text{where} & p & \equiv & \langle \text{wait}, \epsilon, \epsilon, \epsilon\rangle \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{idle}, s, i, \cdot\rangle
\end{array}
$$

Figure 3.8: *The suspend operation. A new thread is scheduled on the processor by the Idle operation (See Section 3.1.5.*

$$
\begin{array}{llrcl}
\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{fork}\ i = \text{adr}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
 & \text{and} & t & \equiv & \langle \top, \top, \top, \mathtt{running}, \top, \top, \cdot\rangle \\
\hline
\langle P,T',Q',S',M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t, R\rangle \\
 & \text{and} & T' & \equiv & T \cup \{t_{\text{new}}\} \\
 & \text{and} & Q' & \equiv & Q \cup \{t_{\text{new}}\} \\
 & \text{and} & S' & \equiv & S \cup \{s_{\text{new}}\} \\
 & \text{and} & t_{\text{new}} & \equiv & \langle \text{adr}, s_{\text{new}}+n, t, \text{ready}, s_{\text{new}}, i, \cdot\rangle \\
 & \text{and} & s_{\text{new}} & \equiv & [\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n)
\end{array}
$$

Figure 3.9: *The fork operation for MAM.*

in the fork. If there are any arguments passed to the child thread, they are put at the base of the child thread's stack. The `sp` is set to the first free location in the child thread's stack. The child thread enters the ready state. Finally, control is passed to the next instruction in the parent thread.

In MAM, there are no intrinsic synchronization instructions. Instead, synchronization instructions are synthesized from the primitives of MAM. Since we are concentrating on downward threads, the most important synchronization instruction is join, which synchronizes a parent with a set of children that it forked. The parent continues only after all of its children have reached the join, i.e., when they have all completed. To implement this, each time a parent forks a child (or a set of children), it increments a counter variable called a *join counter*. The inlet that the parent passes to its child decrements the same join counter, making the parent ready (with `enable`) if the counter is zero.[5] The join operation is thus a test on the join counter. If it is not zero, then the parent will suspend.

3.1.4 Inlets and Inlet Operations

Inlets and their associated instructions generalize the data transfer portion of the sequential return instruction for threads. A conventional sequential return instruction implicitly performs two tasks. It transfers control from the child to the parent and it transfers results from the child to the parent in the processor registers. The calling convention specifies which processor registers (maybe just one) contain the results to be returned to the parent. The code executed in the parent by the return instruction (the code following the call that invoked the child that is issuing the return) may use these results immediately, or it may store them in the activation frame for later use. Just as a sequential call has a code fragment that follows it, in MAM each `fork` is associated with an inlet.

An *inlet* is a code fragment that processes data received from another thread, which sends the data to the inlet with a `send` instruction.[6] We call the thread that issues the `send` the *sending thread* and the thread to which the data is sent the *destination thread*. Because data and control are not transferred simultaneously the inlet typically has to perform some synchronization task in addition to storing the data in the destination thread.

[5] This works because inlets run atomically with respect to each other and the codeblock instructions.

[6] Of course, inlets may be used to receive data other than results from another thread. They are a general mechanism that enables one thread to communicate with another.

$$
\begin{array}{llrcl}
\langle P, T, Q, S, M \rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R \rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{send}\ t', \text{adr}, \text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, \epsilon, \mathtt{running}, \epsilon, i, \cdot \rangle \\
 & \text{and} & t' & \in & T \\
 & \text{and} & t' & \equiv & \langle \top, \top, \top, \top, s', \top, l \rangle \\
 & \text{and} & l & = & \cdot \\
 & \text{and} & \text{inst}[\text{adr}] & = & \mathtt{recv}\ \text{slot}_1, \text{slot}_2, \ldots, \text{slot}_n \\
\hline
\langle P, T, Q, S, M \rangle & \text{where} & p & \equiv & \langle \text{adr} + 1, \mathtt{sp} + 2, t', R \rangle \\
 & \text{and} & s'[\text{slot}_x] & \leftarrow & \text{arg}_x \ \ \forall x, 1 \leq x \leq n \\
 & \text{and} & *\mathtt{sp} & \leftarrow & \mathtt{ip} \\
 & \text{and} & *(\mathtt{sp} + 1) & \leftarrow & t \\
 & \text{and} & l & \leftarrow & \otimes
\end{array}
$$

Figure 3.10: *The send instruction.*

The inlet is run in the sender's thread, but it affects the destination thread's state. In other words, a new thread of control is not started for the inlet. Although an inlet runs in the sender's thread (and therefore on the sender's processor) it can be viewed as preempting the destination thread. Thus, thread operations like `fork` cannot appear in an inlet. Inlets run atomically with respect to the destination thread and other inlets.

MAM defines four instructions that are specifically aimed at inlets: `send`, `recv`, `ireturn`, and `enable`. `send` transfers values to another thread. `recv` is always the first instruction in an inlet; it indicates where the data values specified in the corresponding `send` are stored in the destination thread's frame. `ireturn` is used to return control from the inlet routine back to the sending thread. Finally, `enable` places an idle thread in the ready queue (if it is not already there).

Typically a thread in MAM ends with a `send` followed by an `exit`. The `send` transfers the data from the child to its parent. This invokes an inlet which stores the data from the child into the parent frame and then performs a synchronization operation. If the inlet determines that the parent should be placed on the ready queue, then it executes an `enable`. The inlet ends with an `ireturn` which transfers control back to the parent. The `exit` then terminates the child thread.

`send` specifies the destination thread, the inlet to execute, and the data to be sent to the destination thread. As shown in Figure 3.10, a `send` must be matched with a `recv` at the inlet address. `recv` and `send` must have the same number and types of arguments.

$$
\begin{array}{llrcl}
\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{ireturn} \\
 & \text{and} & t & \equiv & \langle \top,\top,\top,\top,\top,\top,l\rangle \\
 & \text{and} & l & = & \otimes \\
 & \text{and} & *(\mathtt{sp}-2) & = & \mathtt{ip}' \\
 & \text{and} & *(\mathtt{sp}-1) & = & t' \\
\hline
\langle P,T,Q,S,M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}', \mathtt{sp}-2, t', R\rangle \\
 & \text{and} & l & \leftarrow & \cdot
\end{array}
$$

Figure 3.11: *The ireturn instruction.*

In other words, a `send/recv` pair is used by threads in place of the processor registers used by sequential return to transfer data from the child to the parent. `send` carries out several operations atomically. First, it locks the destination thread to prevent other inlets and codeblock instructions from executing for the destination thread.[7] It then saves the sender's thread and `ip` on the sender's stack. Next it stores the arguments of `send` in the appropriate slots of the destination thread's stack, as specified by `recv`. Finally, it sets the sender processor's `ip` to the instruction following the `recv`. `send` may appear only in a codeblock.[8]

After `recv` completes, the inlet continues to execute instructions which may access the destination thread's state. Among these may be `enable` (see Figure 3.13) and `ireturn`. `ireturn` returns control from the inlet back to the sender's thread, unlocking the destination thread in the process (see Figure 3.11). `ireturn` may only appear in an inlet.

3.1.5 Thread Scheduling

During the life of a thread, it may be in one of three states: `ready`, `running`, or `idle`. A *ready thread* is one that the scheduler may run when there is a processor available. A *running thread* is one that is currently assigned to a processor. An *idle thread* is waiting on some event to become ready. The state transitions for a thread are shown in Figure 3.12.

[7] For readers familiar with active messages [62] the setting of a lock may be confusing. The use of a lock is strictly to facilitate the definition of the rewrite rules. In fact, the lock is implicit on networked machines because inlets, by construction, run atomically with respect to threads and other inlets.

[8] To avoid unnecessary complexity, without loss of generality, in the definition of MAM we disallow inlets from issuing `send` instructions. In particular, if inlets are allowed to issue `send`s then some mechanism is needed to prevent deadlock. See [62] for a discussion on allowing inlets to issue sends and how it relates to distributed memory machines.

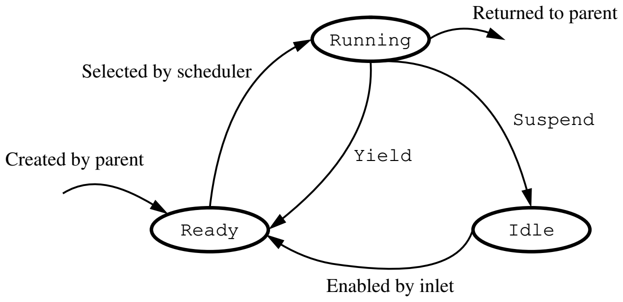


Figure 3.12: *The legal state transitions for a thread in MAM.*

$$\frac{\begin{array}{llrcl}\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P\\ & \text{and} & p & \equiv & \langle \mathtt{ip},\mathtt{sp},t,R\rangle \\ & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{enable}\\ & \text{and} & t & \equiv & \langle \mathtt{ip}',\mathtt{sp}',t_p,\sigma,s,i,\otimes\rangle\end{array}}{\begin{array}{llrcl}\langle P,T,Q',S,M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}+1,\mathtt{sp},t,R\rangle\\ & \text{and} & t & \equiv & \langle \mathtt{ip}',\mathtt{sp}',t_p,\sigma',s,i,\otimes\rangle\\ & \text{and} & \sigma' & \equiv & \begin{cases}\mathtt{ready} & \text{if } \sigma = \mathtt{idle}\\ \sigma & \text{otherwise}\end{cases}\\ & \text{and} & Q' & \equiv & \begin{cases} Q\cup\{t\} & \text{if } \sigma = \mathtt{idle}\\ Q & \text{otherwise}\end{cases}\end{array}}$$

Figure 3.13: *The enable instruction, which is always executed in an inlet. This instruction is a NOP if t is not in the idle state.*

$$\frac{\begin{array}{llrcl}\langle P,T,Q,S,M\rangle & \text{where} & p & \in & P\\ & \text{and} & p & \equiv & \langle \text{wait},\epsilon,\epsilon,\epsilon\rangle\\ & \text{and} & t & \in & Q\\ & \text{and} & t & \equiv & \langle \mathtt{ip},\mathtt{sp},t_p,\mathtt{ready},s,i,\cdot\rangle\end{array}}{\begin{array}{llrcl}\langle P,T,Q',S,M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip},\mathtt{sp},t,\epsilon\rangle\\ & \text{and} & t & \equiv & \langle \epsilon,\epsilon,t_p,\mathtt{running},s,i,\cdot\rangle\\ & \text{and} & Q' & \equiv & Q-\{t\}\end{array}}$$

Figure 3.14: *Describes how an idle processor gets work from the ready queue.*

When an idle thread becomes enabled, i.e., when some thread sends a message to one of its inlets which executes `enable`, it becomes a ready thread by being placed on the ready queue (see Figure 3.13). When a processor is idle, it steals work out of the ready queue by retrieving a thread from the ready queue (see Figure 3.14). The processor's `ip` and `sp` are then loaded with the thread's `ip` and `sp`. The processor continues to execute instructions from its running thread until the thread suspends (becomes `idle`), yields (becomes `ready`, placing itself on the ready queue), or exits. When the thread exits it frees its stack and ceases to exist.

3.1.6 Discussion

The abstract machine described here is the basis for most multithreaded systems that exist today. It can be implemented on a physical machine in many ways. In one such implementation, each thread is described by a data structure that contains the three special-purpose registers (`ip`, `sp`, and parent) and a pointer to a stack. A thread is mapped onto a processor by loading the `ip` and `sp` into the processor's `ip` and `sp`, and if necessary the registers are loaded from the register save area. Before a thread gives up the processor, the registers are saved in the register save area, and likewise the `ip` and `sp` are saved.[9] The fork operation creates a thread data structure, allocates a new stack, initializes the registers, and places the thread on the ready queue. The ready queue can be implemented as a queue, bag, or stack—the choice of the ready queue implementation is beyond the scope of this thesis. The other thread operations are easily implemented.

This naive implementation of MAM satisfies the requirements of a multithreaded system, but it is more costly than necessary since it treats every potentially parallel call as a fork of an independent thread. There are four areas of potential inefficiency:

- There may be many unnecessary enqueue and dequeue operations. For example, when a parent forks a thread and immediately executes a join operation, the child is put on the ready queue and the parent suspends whereupon the child is pulled off the ready queue and finally executed. Logically, however, the child could have been run immediately.
- The processor registers are not used for transferring data, arguments on call, and results on return between parent and child. This last inefficiency arises because MAM

[9] The description does not mandate a particular register save policy.

separates the transfer of data (the arguments in a `fork` or the results in a `send`) from the transfer of control (the actual execution of the child or parent).

- The registers may be unnecessarily saved and restored. For example, the system loads registers when the thread starts even though none are active.
- The thread may not need an entire stack.

3.2 Sequential Call and Return

Before considering a more efficient abstract machine, let us review sequential call and return. Observe that the efficiency of a sequential call derives from two cooperating factors. First, the parent is suspended upon call, and all of its descendants have completed upon return. Second, data and control are transferred together on call and return. The first condition implies that storage allocation for the activation frames involves only adjusting the stack pointer. The second condition means that arguments and return values can be passed in registers and that no explicit synchronization is required.

When a sequential call is executed, it passes its child a return address which is a continuation to the remaining work in the parent. In fact, since the parent can not continue until its child returns, the return address is a continuation to the rest of the program.

When a sequential return is executed it returns control to its parent through the return address. We can easily change the destination address because it is stored in memory. This indirect jump provides the only significant flexibility in the sequential call-and-return sequence, and we exploit it to realize a fork (and later a thread return) which behaves like a sequential call (and return).

3.3 MAM/DF—Supporting Direct Fork

In this section we present a multithreaded abstract machine with direct fork (MAM/DF), which eliminates two of the inefficiencies present in MAM when a child runs to completion. First, we eliminate extra scheduling, enqueueing, and dequeueing operations by directly scheduling the child on invocation and directly scheduling the parent when the child terminates. Second, we transfer data and control simultaneously to the child, so that we can use registers when a parent invokes a child. The key difference between MAM and

MAM/DF is that MAM/DF introduces a new way to create a thread: the `dfork` instruction. A `dfork` does not create an independent thread, but rather a lazy thread. In this section we define lazy threads and introduce the three possible new representations for threads: continuations, thread seeds, and closures.

The MAM/DF dfork behaves more like a sequential call, transferring control directly to its child on call, and receiving control when the child exits. When a parent forks a child, instead of creating a new independent thread for the child and placing it on the ready queue, it places itself on a parent queue and immediately begins execution of the child in a lazy thread on its processor. When a child completes without suspending, it returns control of the processor to its parent, which continues execution at the instruction after the fork, just as if the fork had been a sequential call.

MAM/DF keeps MAM's `fork` and `exit` instructions, and adds `dfork` and `dexit`, the direct scheduled versions of `fork` and `exit`. `fork` and `exit` are used to create and terminate upward and daemon threads. `dfork` and `dexit` are used for downward threads; `dfork` implements a potentially parallel call, and `dexit` terminates it.

Threads created by `dfork` are called *lazy threads.* A lazy thread is not independent on creation, but can become independent in one of three ways. First, it may suspend. Second, it may yield. Third, an idle processor may choose to "steal" the parent, running the parent concurrently with the child. In all three cases the relationship between the parent and child changes so that both may run concurrently as independent threads. When a lazy thread becomes independent, it or its parent (or both) must be modified to reflect the fact that the child will no longer be returning to a waiting parent.

When a child thread is a lazy thread we say that it is *connected* to its parent. If, through `suspend`, `yield`, or a steal operation, it becomes an independent thread, we say that it is *disconnected.* A thread started by `fork` is also an *independent thread.* In the rest of the dissertation we use the terms independent thread and lazy thread when the context is not sufficient to disambiguate our use of the word "thread."

Even though `dfork` transfers control directly to its child, we must allow for the fact that control may return to the parent before the child has completed, e.g., the child may suspend on a synchronization event. Thus, MAM/DF must be flexible enough to start a lazy thread which is expected to run to completion but instead suspends before doing so. To

$$\text{Machine} := \langle P, T, Q, Z, S, M \rangle$$

Where	Z	is the parent Queue.

$$\text{Thread} := \langle \texttt{ip}, \texttt{sp}, t_p, \sigma, s, i, f, l \rangle$$

Where	σ	represents the current thread status. $\sigma \in \{\texttt{ready}, \texttt{running}, \texttt{idle}, \texttt{is-child}, \texttt{has-child}\}$.
	f	is a flag, either true or false, that indicates whether the thread is connected to its parent (true) or is an independent thread (false).

Figure 3.15: *Redefinition of the machine and thread for MAM/DF. All elements not explicitly defined remain the same as in MAM (See Figure 3.3).*

handle this, MAM/DF has a parent queue in addition to the ready queue.[10] Furthermore, threads have an additional field (the connected flag) and can take on two additional states (`is-child` and `has-child`). See Figure 3.15 for a description of the changes to the Machine and Thread tuples.

In the Section 3.3.2 we describe the new operations in MAM/DF and significant changes to the original MAM operations. In Section 3.3.1 we outline the new scheduling methodology. In Sections 3.3.3 and 3.3.4 we address thread representations and new methods of work stealing.

3.3.1 The MAM/DF Scheduler

In MAM/DF threads may be scheduled by other threads or by the general scheduler. There are two queues of threads that are ready to run: a ready queue and a parent queue. The ready queue, as in MAM, has threads in the `ready` state. These are scheduled the same way they are scheduled in MAM using a rule similar to the MAM rule in Figure 3.14, which assigns a thread in the ready queue to an idle processor.[11] Threads in the *parent queue* are all in the `has-child` state and are connected to the child they last invoked with a `dfork`. Threads in the parent queue are scheduled either directly by an operation (`suspend`, `yield`, or `dexit`), or through work stealing.

[10] The parent queue, despite its name, need not be a queue and nothing in the semantics of MAM/DF requires it to have any particular structure.

[11] The change in the MAM rule is to add the disconnect flag to the Thread tuples.

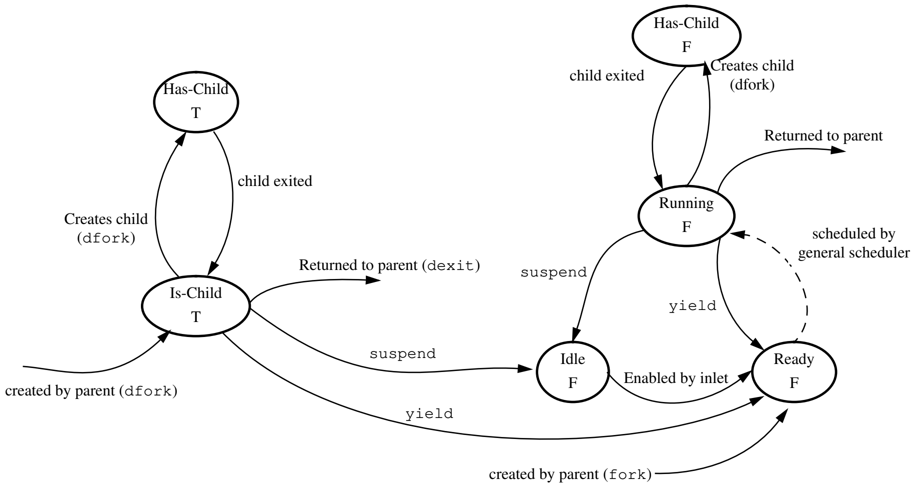


Figure 3.16: *The legal state transitions for a thread in the MAM/DF without work stealing. Each node in the graph represents a valid state and the value of the connected flag. The solid lines indicate transitions due to the execution of an instruction. The dashed line indicates a transition caused by an outside influence, in this case the general scheduler.*

Figure 3.16 shows the state transitions for a thread. Only threads in the `is-child` and `running` states are actually assigned to processors and executing. Threads in the `has-child` state are on the parent queue, while threads on the `ready` state are on the ready queue. Essentially, MAM/DF separates the `running` state of MAM into two states (`running` and `is-child`) and the `ready` state of MAM into three states (`ready`, `has-child` with connected flag true, and `has-child` with connected flag false).

If a child is created by `dfork`, then unless it suspends or yields it will remain on the left of the diagram, continuously moving between `is-child` and `has-child`. Once a thread becomes independent, it moves to one of the states on the right of the diagram. Only a thread in `running` or `is-child` can return to its parent. This is because only threads in these states are actually executing on a processor.

3.3.2 Thread Operations in MAM/DF

In MAM/DF, `fork` creates a new independent thread just as it did in MAM. However, as shown in Figure 3.17, the connected flag is set to false to indicate that the thread is independent and will not return control directly to its parent. We sometimes refer to `fork` as an *eager fork* since it always creates an independent thread.

$$
\begin{array}{llrcl}
\langle P, T, Q, Z, S, M \rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R \rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{fork}\ i = \text{adr}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
\hline
\langle P, T', Q', Z, S', M \rangle & \text{where} & p & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t, R \rangle \\
 & \text{and} & t_{\text{new}} & \equiv & \langle \text{adr}, s_{\text{new}} + n, t, \mathtt{ready}, s_{\text{new}}, i, \mathtt{false}, \cdot \rangle \\
 & \text{and} & s_{\text{new}} & \equiv & [\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
 & \text{and} & T' & \equiv & T \cup \{t_{\text{new}}\} \\
 & \text{and} & Q' & \equiv & Q \cup \{t_{\text{new}}\} \\
 & \text{and} & S' & \equiv & S \cup \{s_{\text{new}}\}
\end{array}
$$

Figure 3.17: *The fork operation under MAM/DF.*

$$
\begin{array}{llrcl}
\langle P, T, Q, Z, S, M \rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R \rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{dfork}\ i' = \text{adr}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \sigma, s, i, f, \cdot \rangle \\
 & \text{and} & \sigma & \in & \{\mathtt{running}, \mathtt{is\text{-}child}\} \\
\hline
\langle P, T', Q, Z', S', M \rangle & \text{where} & p & \equiv & \langle adr, s_{\text{new}}, t_{\text{new}}, R \rangle \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{has\text{-}child}, s, i, f, \cdot \rangle \\
 & \text{and} & t_{\text{new}} & \equiv & \langle \epsilon, \epsilon, t, \mathtt{is\text{-}child}, s_{\text{new}}, i', \mathtt{true}, \cdot \rangle \\
 & \text{and} & r_x & \leftarrow & \text{arg}_x,\ \ \forall x, \leq x \leq n \\
 & \text{and} & T' & \equiv & T \cup \{t_{\text{new}}\} \\
 & \text{and} & S' & \equiv & S \cup \{s_{\text{new}}\} \\
 & \text{and} & Z' & \equiv & Z \cup \{t\}
\end{array}
$$

Figure 3.18: *The dfork operation under MAM/DF transfers control directly to the forked child.*

The new `dfork` operation, shown in Figure 3.18, creates and then transfers control directly to a new child, a lazy thread. It does so by loading the arguments of the fork into the processor registers, saving a pointer to the parent thread on the parent queue, and then assigning the processor to the newly created child thread. The state of the new thread is `is-child` and its connected flag is set to true, indicating that it is connected to and can return control directly to its parent. The state of the parent becomes `has-child` indicating that it is no longer executing and is on the parent queue.

This abstract machine assumes a caller save protocol for saving registers. It is assumed that before the `dfork` is executed there were instructions that saved the active registers in the parent's stack. Upon return the registers will be restored. It should be noted that in order for the definition of this machine to make sense, there must be significant

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{dexit} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{is\text{-}child}, s, i, \mathtt{true}, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \mathtt{has\text{-}child}, s^p, i^p, f^p, \cdot\rangle \\
\hline
\langle P,T',Q,Z',S',M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p, R\rangle \\
 & \text{and} & t_p & \equiv & \langle \epsilon, \epsilon, t_p^p, \sigma, s^p, i^p, f^p, \cdot\rangle \\
 & \text{and} & \sigma & \equiv & \begin{cases} \mathtt{is\text{-}child} & \text{if } f^p = \mathtt{true} \\ \mathtt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & T' & \equiv & T - \{t\} \\
 & \text{and} & S' & \equiv & S - \{s\} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\}
\end{array}
$$

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{dexit} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{running}, s, i, \mathtt{false}, \cdot\rangle \\
\hline
\langle P,T',Q,Z,S',M\rangle & \text{where} & p & \equiv & \langle \text{wait}, \epsilon, \epsilon, \epsilon\rangle \\
 & \text{and} & T & \equiv & T' - \{t\} \\
 & \text{and} & S & \equiv & S' - \{s\}
\end{array}
$$

Figure 3.19: *The dexit operation under MAM/DF transfers control directly to the parent. The first rule applies when the child completes without suspending, i.e., while still in the* `is-child` *state. The second rule applies when the thread has been disconnected from its parent.*

compiler interaction. For example, a thread started by `fork` puts its arguments on the stack, while a thread started by `dfork` puts them in registers.

The `dexit` operation exits the thread and returns control from a child thread to its parent (see Figure 3.19). If the child thread is in the `is-child` state (the parent has to be in the `has-child` state), it is terminated and the parent thread is restarted on the same processor without any intervening operations by the general scheduler. The parent thread returns to the state it was in before the `dfork` of the child. This state is either `running` or `is-child`, depending upon the state of the parent's connected flag. If the child thread is in the `running` state, `dexit` behaves like `exit`;[12] it deallocates the thread's resources, but invokes the general scheduler on the processor.

[12]Recall that in MAM/DF control and data are passed simultaneously only on call. On return data is returned with a `send` and control with `dexit`.

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \texttt{ip}, \texttt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\texttt{ip}] & = & \texttt{suspend} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \texttt{is-child}, s, i, \texttt{true}, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \texttt{ip}^p, \texttt{sp}^p, t_p^p, \texttt{has-child}, s^p, i^p, f, \cdot\rangle \\
\hline
\langle P,T,Q,Z',S,M\rangle & \text{where} & p & \equiv & \langle \texttt{ip}^p, \texttt{sp}^p, t_p, \epsilon\rangle \\
 & \text{and} & t & \equiv & \langle \texttt{ip}+1, \texttt{sp}, t_p, \texttt{idle}, s, i, \texttt{false}, \cdot\rangle \\
 & \text{and} & t_p & \equiv & \langle \epsilon, \epsilon, t_p^p, \sigma, s^p, i^p, f, \cdot\rangle \\
 & \text{and} & \sigma & \equiv & \begin{cases} \texttt{is-child} & \text{if } f = \texttt{true} \\ \texttt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\}
\end{array}
$$

Figure 3.20: *The suspend operation under MAM/DF for a lazy thread.*

`suspend` and `yield` also behave differently depending upon whether the thread executing them is in the `is-child` or `running` state. If the thread is in the `running` state, the MAM versions of these operations apply. (See Figures 3.7 and 3.8.) If, however, the thread is in the `is-child` state, the child and parent threads must be disconnected. After a lazy thread suspends or yields, it becomes independent of its parent, and the parent may execute another `dfork`. Since a parent cannot have two children that expect to schedule the parent directly when they return, the suspending child is changed so that when it returns it will not schedule the parent directly. This change is indicated by setting the connected flag to false (see Figure 3.20).

3.3.3 Continuation Stealing

Continuation stealing is one of two methods by which a thread in the parent queue can resume execution before its child completes. The other, seed activation, is described in the next section. Continuation stealing resumes execution of a parent at its current continuation and breaks the connection between a parent and its child. In other words, it turns the child into an independent thread, removes the parent from the parent queue, and then resumes the parent at its current `ip`. Figure 3.21 shows the state transitions for MAM/DF when continuation stealing is used to effect work stealing.

The continuation that is stolen is the same one that the child would have used had it returned to the parent with `dexit`. This continuation exists because the child was started with `dfork`, which puts a pointer to the parent on the parent queue. The `ip` and

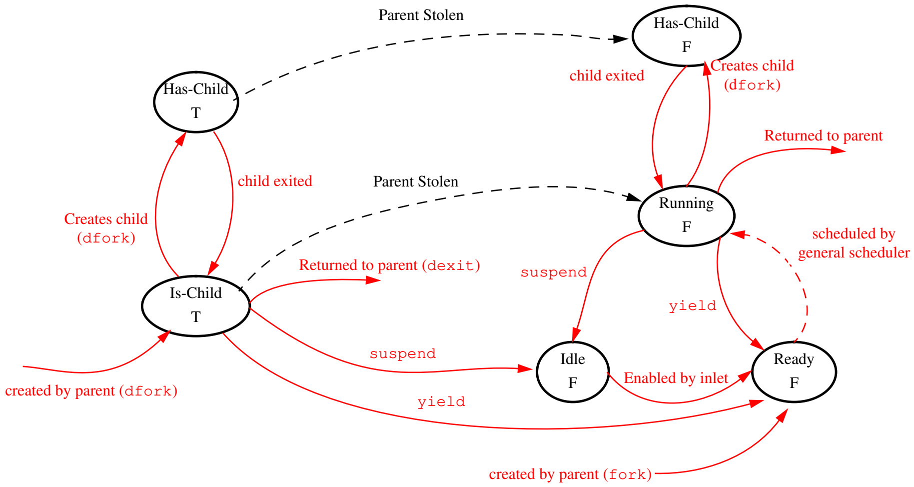


Figure 3.21: *The new state transitions for a thread in MAM/DF using continuation stealing. Here we highlight the new transitions introduced when a thread's continuation is stolen.*

$$
\frac{
\begin{array}{llrcl}
\langle P, T, Q, Z, S, M\rangle & \text{where} & p & \in & P \\
& \text{and} & p & \equiv & \langle \text{wait}, \epsilon, \epsilon, \epsilon\rangle \\
& \text{and} & t^c & \equiv & \langle \mathtt{ip}^c, \mathtt{sp}^c, t, \mathtt{has\text{-}child}, s^c, i^c, \mathtt{true}, \cdot\rangle \\
& \text{and} & t & \in & Z \\
& \text{and} & t & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t_p, \mathtt{has\text{-}child}, s, i, f, \cdot\rangle
\end{array}
}{
\begin{array}{llrcl}
\langle P, T, Q, Z', S, M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
& \text{and} & t^c & \equiv & \langle \mathtt{ip}^c, \mathtt{sp}^c, t, \mathtt{running}, s^c, i^c, \mathtt{false}, \cdot\rangle \\
& \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \sigma, s, i, f, \cdot\rangle \\
& \text{and} & \sigma & \equiv & \begin{cases} \mathtt{is\text{-}child} & \text{if } f = \mathtt{true} \\ \mathtt{running} & \text{otherwise} \end{cases} \\
& \text{and} & Z' & \equiv & Z - \{t\}
\end{array}
}
$$

Figure 3.22: *The rule for stealing work using continuation stealing when the child of the stolen thread is not currently running, i.e., the child is in the* `has-child` *state.*

`sp` fields in the parent thread are the continuation. If the child returns after the parent is stolen, it exits without scheduling the parent because it is now an independent thread.

Figure 3.22 shows the semantics of a continuation stealing operation when the child of the stolen thread is not currently running on the processor.The resulting configuration is very similar to the configuration of a parent after a child has suspended. The effect of continuation stealing is similar to that of a child suspending, i.e., the parent migrates. The main difference is that the child remains on its processor while the parent (and its ancestors) will now be scheduled on what was the idle processor.

```
dfork foo(...);
⋮ sequential code
```

case a: sequential code

```
dfork foo(...);
dfork bar(...);
⋮
```

case b: another `dfork`

```
⋮
dfork foo(...);
join
```

case c: a join

Figure 3.23: *Three possible resumption points after executing* `dfork` *foo.*

Any thread in the parent queue can be scheduled using continuation stealing. However, not all of them have useful work to perform. There are three different situations, shown in Figure 3.23, that the parent can be in when it is stolen. In the first two cases the parent will execute useful work, either by the sequential code as in case (a) or by creating a new thread in case (b). In case (c), however, it will suspend immediately after being scheduled because the join fails.

3.3.4 Thread Seeds and Seed Activation

A thread seed represents the work in a parent thread that can be forked off into a new thread. When a parent forks a child it leaves behind a thread seed which is a continuation up to the next fork in the parent. Seed activation is used by a processor looking for work, i.e., an idle processor. The idle processor finds a parent in the parent queue; which means that the parent has a descendant that is currently executing on a processor. The processor running the parent's descendant is interrupted and begins to activate the thread seed, i.e., it executes the continuation previously left behind in the parent, which forks the next child in the parent. This new child is picked up by the idle processor, the one that initiated the seed activation. In other words, seed activation is a method of making progress in the program by causing a thread in the parent queue to create a new thread. The new thread can then be stolen by another processor. As described below, seed activation uses thread seeds to instantiate nascent threads. A thread seed is the first part of a continuation to the rest of the program. It is the part that spawns a new thread.

Nascent Threads

A *nascent thread* is a thread that is ready to start execution, but has not yet done so. Nascent threads exist because `dfork` causes control to be transferred from the parent even when the parent has more work that it can perform, i.e., the instructions after the

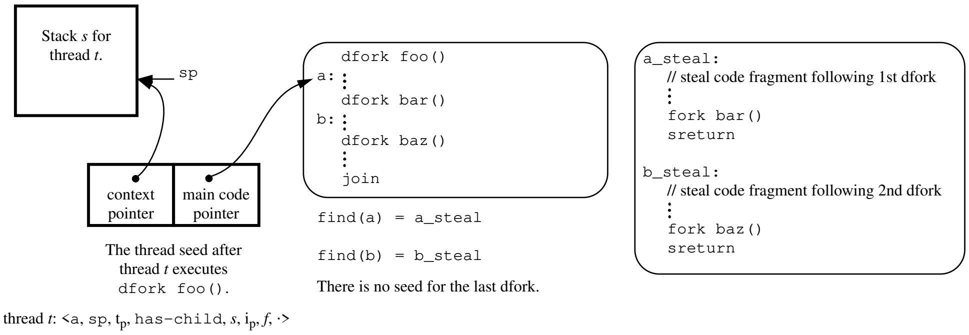


Figure 3.24: *Example thread seed and seed code fragments associated with* `dfork`*s. The thread seed shown is simply the first two fields of the thread tuple after thread t executes* `dfork foo()`*. Note that the code, if any, represented by the ellipsis is copied to the steal fragments.*

return continuation. Informally, threads that would have been created in MAM, but have not yet been created in MAM/DF, are nascent threads. Formally, a nascent thread is a thread in the logical task graph which is a younger sibling of a thread started by `dfork`.[13] For example, `dfork bar()` and `dfork baz()` are nascent threads after the execution of `dfork foo()` in Figure 3.24.

Thread Seeds

A *thread seed* consists of a continuation to the rest of the program and one or more partial continuations. A partial continuation is a continuation not to the rest of the program, but only to the portion of the program up to the next fork. It is given by a pointer to a context (i.e., an activation frame) and a set of related code pointers where each of the code pointers can be derived at link time from a single code pointer. The code pointer used to derive the other code pointers in the seed is the *main code pointer*. The main code pointer is a continuation to the rest of the program. The derived code pointers are partial continuations that point to code that is a copy of the code at the main code pointer up to and including the main code pointer. In other words, a thread seed can be viewed as a set of continuations where each continuation in the set has the same context and each of the code pointers in the set can be derived from the main code pointer. Or, more concretely, it is a set of entry points in a function, each of which performs a different, but related, task.

[13] A thread s is a sibling of thread t if s and t have the same parent. s is a younger sibling of t if t was created before s.

In MAM/DF the thread seed is represented by a pointer to a thread. The thread's `sp` is the context and the thread's `ip` is the main code pointer from which all the other code pointers in the thread seed can be derived. Figure 3.24 shows the thread tuple, and the portion of it that is the thread seed that represents the nascent thread that will be created by `dfork bar()`.

There is a thread seed for every `dfork` that has at least one `dfork` between it and its associated join (See Figure 3.24). Each thread seed in MAM/DF has two code pointers. The main code pointer points to the instruction following its `dfork`. This is the `ip` that the parent is set to after executing the `dfork` associated with the thread seed. Thus, the main code pointer for a thread seed is the same instruction pointer that would be saved by a sequential call instruction. In fact, the creation of a thread seed at runtime is nothing more than saving the return address of the call to the child thread. The second code pointer points to a code fragment for seed activation, called the steal code fragment. The steal code fragment is used to create the new work upon a work stealing request.

Figure 3.24 has a section of a codeblock which includes the steal fragments for the thread seeds associated with the first and second forks. The function `find()`, which is computed at compile or link time, returns the derived code pointers from the main code pointer. This function can be as simple as address arithmetic.

Seed Activation

The key characteristic of seed activation is that a nascent thread is elevated into an independent thread in the context of its parent. This contrasts with continuation stealing, where the parent itself is resumed. With continuation stealing the parent is migrated as opposed to seed activation where the new thread is migrated.

Seed activation (see Figure 3.25) takes several steps. We describe the procedure with reference to Figures 3.26 and 3.27. First a thread is chosen (t_p in Figure 3.27) and temporarily removed from the parent queue. The thread then begins executing, not at its current `ip`, but at the seed routine pointed to by the second code pointer, computed by `find(ip)` in Figure 3.25. The seed routine starts at Line 27 of Figure 3.26. This is state 2 in Figure 3.27. The processor that executes the seed routine is the one currently executing the lazy thread descended from the thread that owns the seed. (In the example, processor p is executing t_X, the first child of t_p.) In other words, the idle processor causes

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \text{wait},\epsilon,\epsilon,\epsilon\rangle \\
 & \text{and} & p^{\text{run}} & \in & P \\
 & \text{and} & p^{\text{run}} & \equiv & \langle \mathtt{ip}^{\text{run}},\mathtt{sp}^{\text{run}},t^{\text{run}},R\rangle \\
 & \text{and} & t^{\text{run}} & \equiv & \langle \epsilon,\epsilon,\top,\mathtt{is\text{-}child},\top,\top,\mathtt{true},l^{\text{run}}\rangle \\
 & \text{and} & l^{\text{run}} & = & \cdot \\
 & \text{and} & t^{c} & \equiv & \langle \top,\top,t,\mathtt{has\text{-}child},\top,\top,\mathtt{true},\cdot\rangle \\
 & \text{and} & t & \in & Z \\
 & \text{and} & t & \equiv & \langle \mathtt{ip},\mathtt{sp},t_p,\mathtt{has\text{-}child},s,i,f,\cdot\rangle \\
 & \text{and} & t & = & \text{ancestor}(t^{\text{run}}) \\
 & \text{and} & t^{c} & \neq & t^{\text{run}} \\
\hline
\langle P,T,Q,Z',S,M\rangle & \text{where} & p & \equiv & <\text{steal},\epsilon,\epsilon,\epsilon> \\
 & \text{and} & p^{\text{run}} & \equiv & \langle \text{seed}(\mathtt{ip}),\mathtt{sp}^{\text{run}}+2,t,R\rangle \\
 & \text{and} & *\mathtt{sp}^{\text{run}} & \leftarrow & \mathtt{ip}^{\text{run}} \\
 & \text{and} & *(\mathtt{sp}^{\text{run}}+1) & \leftarrow & t^{\text{run}} \\
 & \text{and} & t & \equiv & \langle \epsilon,\mathtt{sp},t_p,\mathtt{has\text{-}child},s,i,f,\cdot\rangle \\
 & \text{and} & Z' & \equiv & Z-\{t\} \\
 & \text{and} & l^{\text{run}} & = & \otimes
\end{array}
$$

Figure 3.25: *Work stealing though seed activation.*

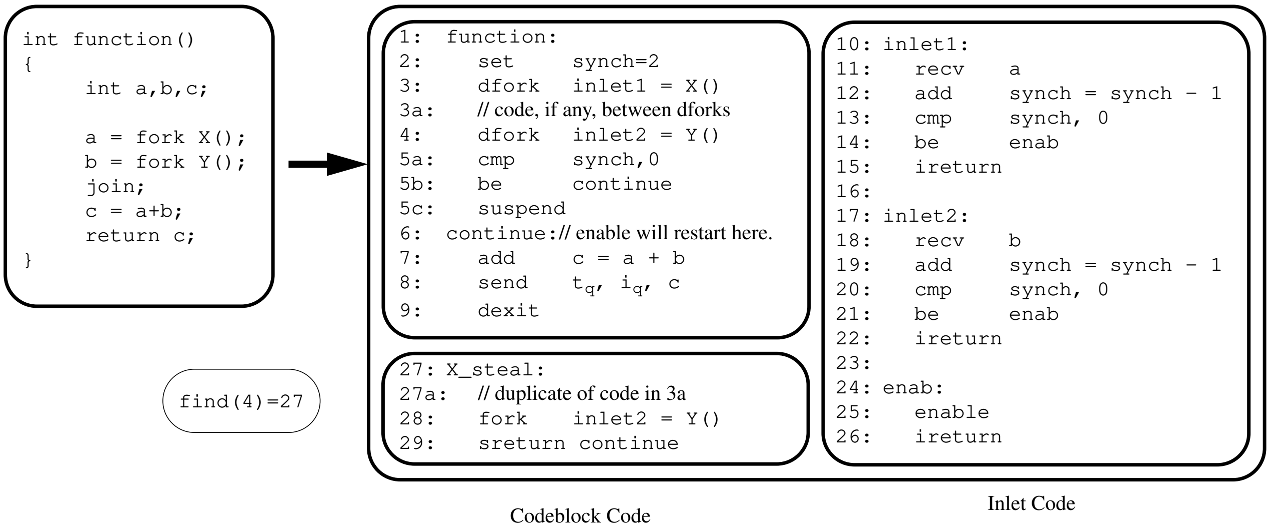


Figure 3.26: *Example translation of a function with two forks into psuedo-code for the MAM/DF using thread seeds. t_q is the parent thread. i_q is the parent inlet.*

1	MAM/DF	$\langle P \cup \{p\}, T, Q, Z \cup \{t_p\}, S, M \rangle$
	p	$\langle \mathtt{ip}, \mathtt{sp}, t_X, R \rangle$
	t_p	$\langle 4, \mathtt{sp}_p, t_q, \mathtt{has\text{-}child}, s_p, i_q, f_p, \cdot \rangle$
	t_X	$\langle \epsilon, \epsilon, t_p, \sigma_X, s_X, 10, \mathtt{true}, \cdot \rangle$
2	MAM/DF	$\langle P \cup \{p\}, T, Q, Z, S, M \rangle$
	p	$\langle 27, \mathtt{sp} + 2, t, R \rangle$
	t_p	$\langle 4, \mathtt{sp}_p, t_q, \mathtt{has\text{-}child}, s_p, i_q, f_p, \otimes \rangle$
	t_X	$\langle \epsilon, \epsilon, t_p, \sigma_X, s_X, 10, \mathtt{true}, \cdot \rangle$
3	MAM/DF	$\langle P \cup \{p\}, T \cup \{t_{\text{new}}\}, Q, Z, S \cup \{s_{\text{new}}\}, M \rangle$
	p	$\langle 29, \mathtt{sp} + 2, t, R \rangle$
	t_p	$\langle 4, \mathtt{sp}_p, t_q, \mathtt{has\text{-}child}, s_p, i_q, f_p, \otimes \rangle$
	t_X	$\langle \epsilon, \epsilon, t_p, \sigma_X, s_X, 10, \mathtt{true}, \cdot \rangle$
4	MAM/DF	$\langle P \cup \{p\}, T \cup \{t_{\text{new}}\}, Q, Z, S \cup \{s_{\text{new}}\}, M \rangle$
	p	$\langle \mathtt{ip}, \mathtt{sp}, t_X, R \rangle$
	t_p	$\langle 5a, \mathtt{sp}_p, t_q, \mathtt{has\text{-}child}, s_p, i_q, f_p, \otimes \rangle$
	t_X	$\langle \epsilon, \epsilon, t_p, \sigma_X, s_X, 10, \mathtt{true}, \cdot \rangle$
	t_{new}	$\langle \epsilon, \epsilon, t_p, \mathtt{running}, s_{\text{new}}, 17, \mathtt{false}, \cdot \rangle$

Figure 3.27: *Example of seed activation assuming t_X is executing function* `X` *on processor p. The numbers in the first field of the thread tuple represent the instruction pointer associated with the line numbers in Figure 3.26. State 1 occurs right before work stealing happens. State 2 is right after work stealing occurs. In this case,* $\mathtt{find}(4) = 27$. *State 3 is after the* `fork` *occurs in line 28. State 4 is after the* `sreturn` *has executed.*

$$
\begin{array}{llrcl}
\langle P, T, Q, Z, S, M \rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}', t, R \rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{sreturn}\ \mathtt{ip}_{\text{new}} \\
 & \text{and} & t & \equiv & \langle \epsilon, \mathtt{sp}, t_p, \mathtt{is\text{-}child}, s, i, f, l \rangle \\
 & \text{and} & *(\mathtt{sp}' - 2) & = & \mathtt{ip}' \\
 & \text{and} & *(\mathtt{sp}' - 1) & = & t' \\
 & \text{and} & t' & \equiv & \langle \epsilon, \epsilon, \top, \mathtt{is\text{-}child}, \top, \top, \mathtt{true}, l' \rangle \\
 & \text{and} & l' & = & \otimes \\
\hline
\langle P, T, Q, Z', S, M \rangle & \text{where} & p & \equiv & \langle \mathtt{ip}', \mathtt{sp}' - 2, t', R \rangle \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}_{\text{new}}, \mathtt{sp}, t_p, \mathtt{has\text{-}child}, s, i, f, l \rangle \\
 & \text{and} & l' & \leftarrow & \cdot \\
 & \text{and} & Z' & \equiv & Z \cup \{t\}
\end{array}
$$

Figure 3.28: *The seed return instruction is used to complete a seed routine.*

a running processor to be interrupted. The seed routine executes and forks off a new child, which is mapped to the idle processor though the MAM idle rule (See Figure 3.14). Finally, the `sreturn` at the end of the seed routine executes, which returns the parent thread to the parent queue and sets the parent thread's `ip` to the instruction following the `dfork` associated with the seed routine that was executed.

The seed routine finishes by executing an `sreturn` (See Figure 3.28). `sreturn` returns control of the processor to the descendant that was interrupted and also returns the thread to the parent queue, updating the `ip` of the thread to reflect the fact that it has made progress. When the parent is next resumed, either by its child (through `dexit`, `suspend`, `yield`) or by another steal request, it will continue execution at the point after the `dfork` of the nascent thread it just activated.

Seed activation requires support from the compiler in two ways. First, the compiler must construct the seed routines and the `find()` function for every `dfork`. Second, it must create two entry points for every thread, one for `dfork` and one for `fork`. Because `fork` creates the child, stores the arguments, and then continues in the parent, the `fork` entry point loads the arguments from the stack into registers and then continues at the `dfork` entry point, which assumes that the arguments are in registers. Two entry points are necessary because a routine which will be `dfork`ed in the main computation path may be forked from a seed routine.

3.3.5 Closures

The main drawback to thread seeds, as compared to continuation stealing, arises when an `dfork` is followed by sequential code (case (a) in Figure 3.23). The sequential code cannot be represented by a thread seed, so if a `dfork` starts the first thread, the total amount of parallelism available is reduced. In order to solve this problem we introduce another representation for a thread: the closure.

We introduce a new fork operation, `cfork`, that creates a closure, which becomes an independent thread when executed. The *closure* contains the necessary data (the instruction pointer and arguments specified in the fork instruction) to start the thread later. Closures are enqueued on a separate closure queue.

If a single fork is followed by sequential code (which can run in parallel with the forked child), we use `cfork` to create a closure instead of creating a lazy thread or an independent thread. When the join is reached, one of two things happens to the closure created for the child. The closure created by the `cfork` may have been stolen by another processor, in which case the the join fails, or succeeds depending on the child's completion. Otherwise the closure is still awaiting execution, in which case the join fails and the closure is instantiated.

3.3.6 Discussion

MAM/DF eliminates some of the overhead of multithreading by allowing a parent thread to schedule its child thread directly. This reduces scheduling costs by avoiding a general-purpose enqueue and dequeue and reduces memory costs by allowing arguments to be passed in registers. It also exposes the different policies allowed for work representing threads that can be stolen: continuations, thread seeds, and closures.

Seed activation in MAM/DF creates an independent thread, leaving a parent's pre-existing child still connected to the parent. We could have chosen to disconnect the pre-existing child and create the new child with a `dfork`. This more closely parallels what must happen with continuation stealing, where the pre-existing child is disconnected from the parent and the parent continues, which causes new children to be created with lazy threads. We have chosen to use `fork` in seed activation for reasons discussed in Chapter 5.

MAM/DF still has scheduling inefficiencies. A child cannot return control and data simultaneously to its parent. Instead, as in MAM, a child must return the results through an inlet and then return control with `dexit`. Furthermore, if both children execute sequentially, i.e., they are never disconnected from their parent, we perform unnecessary synchronization. We now define a machine that allows us to eliminate this overhead.

3.4 MAM/DS—Supporting Direct Return

In this section we modify our multithreaded abstract machine to allow a thread to directly schedule its child on creation and its parent on return. This optimization allows control transfer between threads to more accurately mimic that of sequential call and thus, when the semantics allow, to obtain similar efficiency. In particular, registers can be used to transfer both arguments and results.

We present new MAM/DS mechanisms which grant a lazy thread that runs to completion the ability to return results when it returns control to its parent. The major difference between MAM/DS and MAM/DF is that in MAM/DS the thread exit operation, `lreturn`, can specify return results, which will be transferred in registers directly to the parent as the thread exits. This change makes lazy thread exit similar to a sequential return instruction. The creation of lazy threads also changes. In MAM/DS `lfork` creates a lazy thread and it differs from `dfork` in that an inlet is not specified. Instead, the instruction following the `lfork` is implied as the inlet, just as in a sequential call.

To understand the ramifications of these changes we need to analyze the actions carried out on return from a child thread. A return operation consists of three independent operations. First, it must return the results to its caller. Second, it must update the synchronization status in the parent (which may have multiple outstanding children). Third, it must ensure that the parent restarts at the proper continuation address. In MAM and MAM/DF the first two steps are carried out by invoking an inlet on the parent thread. The third step is achieved because the parent maintains an `ip` which is the address at which it will restart when control is transferred to it, either from the general scheduler, from a work stealing request, or directly from the child.

If we compare these three independent operations to the sequential return instruction, we see that a return instruction combines all three actions into a single operation. It returns control to the parent at its return address passing results in registers. The instructions at the return address act as the inlet and save the results in the parent context. Then it continues to execute instructions in the parent. There is no synchronization step, since sequential calls always run to completion. The return address acts as both the inlet address and the continuation address.

Figure 3.29 shows a translation of a function with two forks into psuedo-code for MAM/DS using thread seeds. Lines 1–7a are executed when both children run to completion. The child created in line 3 (4) returns to line 3a (4a). No explicit synchronization is performed, and arguments and results are transferred directly in registers.

If the first child, `X`, suspends, control is transferred to line 30. This code fragment, lines 30–40, sets up the synchronization counter (line 31), changes `X`'s inlet address (line 32), forks `Y`, and then handles `Y`'s return (lines 34–40). The synchronization counter is set here because the order of return for the two children is no longer known. For the same reason, `X`'s inlet is changed to `inlet1` where it stores its result and performs synchronization on return. The reader should convince herself that whatever the order of `X`'s and `Y`'s return,

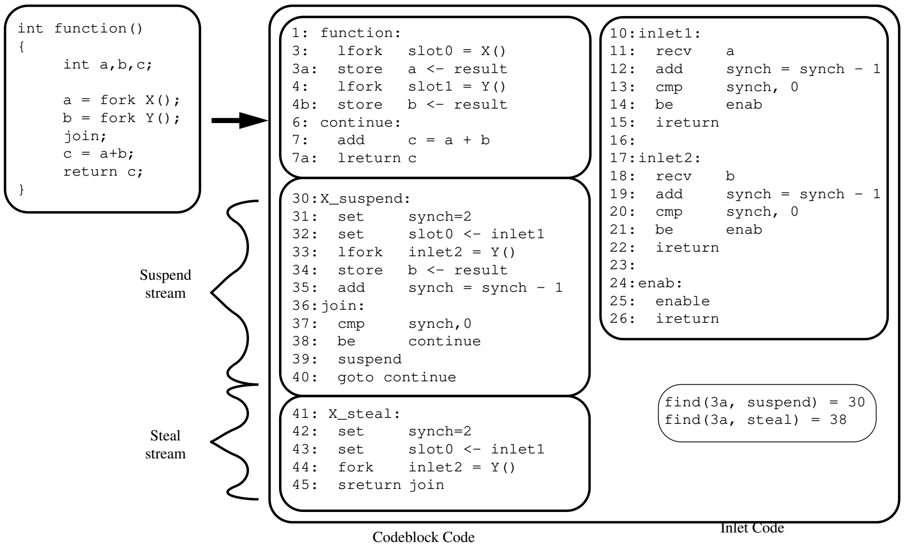


Figure 3.29: *Example translation of a function with two forks into psuedo-code for the MAM/DS using thread seeds.* `slot0` *and* `slot1` *are slots in the thread's activation frame used to hold inlet addresses.* `result` *is the register used to pass results from callees to callers.*

Thread := $\langle \mathtt{ip}, \mathtt{sp}, t_p, \sigma, s, i, f, l \rangle$
Where i is either an inlet address or an index into the parents inlet return vector.

Figure 3.30: *Redefinition of a thread for MAM/DS. All elements not defined remain the same as in MAM/DF.*

the proper synchronization is performed and eventually lines 6–7a will be executed, finishing the function.

If a steal request arrives while `X` is executing, the steal code fragment (lines 41–45) is executed. This code fragment activates the seed for Y as an independent thread, also setting up synchronization and changing `X`'s inlet address.

From the previous example we see that in MAM/DS each potentially parallel call has two associated return paths, one for the case when the child runs to completion, and the other for the case when the child has been disconnected from the parent. If the child runs to completion, it can use a mechanism similar to a sequential return, i.e., it returns to the instruction following the fork. If disconnected the inlet approach of MAM is used.

To support this dual-return strategy in MAM/DS, we introduce indirect inlets. An *indirect inlet* is simply an inlet invoked through an indirect jump. The key behind this change is that the indirect inlet address can be changed by the parent to reflect the parent's current state. We redefine the return register in a thread to be a pointer to an indirect inlet address (See Section 3.4.1). We also redefine the `find()` mapping: `find(adr, type)` maps the `ip` at `adr` to a code pointer determined by `type`: one of `suspend` or `steal`.

Thread seeds in MAM/DS have three code pointers: the main code pointer, the suspend code pointer, and the steal code pointer. When thread seeds are used to represent the nascent threads, `find` maps the thread's `ip`, the main code pointer of the thread seed, to one of two code pointers: `find(ip, suspend)` returns the suspend code pointer and `find(ip, steal)` returns the steal code pointer.

When continuations are used to represent the remaining work in a thread, `find` maps the `ip` in a thread into the continuation that will be stolen. In this case, `find` ignores the second argument.[14]

3.4.1 MAM/DS Operations

MAM/DS replaces `dfork` with `lfork` and `dexit` with `lreturn`. `lfork`, like `dfork`, creates a lazy thread and transfers control directly to the new thread. `lreturn` simultaneously returns control and data from the child to its parent when possible.

Indirect Inlets

The main challenge in implementing MAM/DS is to support two different return paths (one for return when the child is connected to the parent and one for return when the parent and child have been disconnected) with a single instruction. Instead of passing (to the child) the address of the inlet the child should invoke upon return, `lfork` passes the child the address of a location in the parent's frame which contains the inlet address. We call this location the *indirect inlet address*. Thus inlet invocation is performed via an

[14]By using a mapping function to derive the continuation to be stolen from the `ip`, we blur the distinction between continuation stealing and seed activation. In previous work, the actual continuation to be stolen is posted on a queue [44], so that two different continuations need to be saved: one used by the child for return, and one used by the system for suspension and work stealing. The mapping function eliminates the need for two continuations and allows us to focus on the more important difference between continuation stealing and seed activation, viz., how they treat the connection between the parent and child.

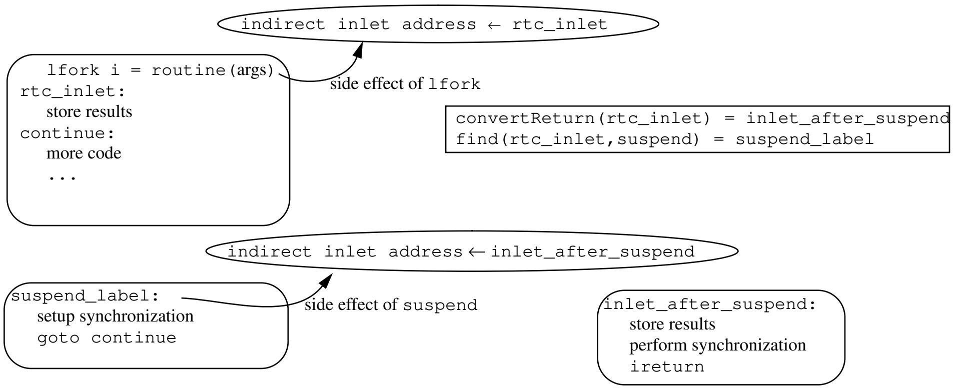


Figure 3.31: *A pseudo-code sequence for an* `lfork` *with its two associated return paths. If the child runs to completion, it returns to* `rtc_inlet`*. Otherwise, it runs the* `suspend` *routine and later returns to* `inlet_after_suspend`*. This example uses continuation stealing.*

indirect jump through the indirect inlet address.[15] The key behind this change is that the indirect inlet address can be changed by the parent to reflect the parent's current state.

When the child is invoked, its indirect inlet address is set to the address of an inlet that behaves like the code after a sequential call. (See inlet `rtc_inlet` in Figure 3.31.) If the child runs to completion, no synchronization is performed, and the inlet, `rtc_inlet` runs when the child returns control to the parent, storing the results and continuing. If the parent is resumed before the child returns (e.g., the child suspends), then the indirect inlet address is changed (by the routine at `suspend_label`) to contain a new inlet. The new inlet, `inlet_after_suspend` in Figure 3.31, stores the results and then performs the necessary synchronization. The inlet used after a child suspends must end with an `ireturn`. This ensures that control is transferred to the proper place when the inlet finishes.

Changes to Support Indirect Inlets

To support this new return mechanism many of the operations in MAM/DF are changed to manipulate the indirect inlet addresses: `lfork` passes its child an indirect inlet address, `send` can take an indirect inlet address, `lreturn` uses the indirect inlet address for return, `suspend` changes the indirect inlet stored in the indirect inlet address, and a work stealing request also changes the indirect inlet. `lfork` (see Figure 3.32) sets the indirect

[15]There is a unique indirect inlet address for each outstanding child. The number of indirect inlet addresses in a thread's frame is the maximum, over all joins in the thread, of the number of children that the join is synchronizing.

$$
\frac{
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{lfork}\ i' = \text{adr}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \sigma, s, i, f, \cdot\rangle \\
 & \text{and} & \sigma & \in & \{\mathtt{running}, \mathtt{is\text{-}child}\}
\end{array}
}{
\begin{array}{llrcl}
\langle P,T',Q,Z',S',M\rangle & \text{where} & p & \equiv & \langle adr, s_{\text{new}}, t_{\text{new}}, R\rangle \\
 & \text{and} & T' & \equiv & T \cup \{t_{\text{new}}\} \\
 & \text{and} & S' & \equiv & S \cup \{s_{\text{new}}\} \\
 & \text{and} & Z' & \equiv & Z \cup \{t\} \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{has\text{-}child}, s, i, f, \cdot\rangle \\
 & \text{and} & t_{\text{new}} & \equiv & \langle \epsilon, \epsilon, t, \mathtt{is\text{-}child}, s_{\text{new}}, i', \mathtt{true}, \cdot\rangle \\
 & \text{and} & r_x & \leftarrow & \text{arg}_x,\ \ \forall x, 0 \le x \le n \\
 & \text{and} & v[i'] & \leftarrow & pc+1
\end{array}
}
$$

Figure 3.32: *The lfork operation in MAM/DS.*

$$
\frac{
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{lreturn}\ \text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{is\text{-}child}, s, i, f, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \mathtt{has\text{-}child}, s^p, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & v^p[i] & = & \mathtt{ip}^p
\end{array}
}{
\begin{array}{llrcl}
\langle P,T',Q,Z',S',M\rangle & \text{where} & p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p, R\rangle \\
 & \text{and} & t_p & \equiv & \langle \epsilon, \epsilon, t_p^p, \sigma^p, s^p, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & \sigma^p & \equiv & \begin{cases} \mathtt{is\text{-}child} & \text{if } f^p = \mathtt{true} \\ \mathtt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & T' & \equiv & T - \{t\} \\
 & \text{and} & S' & \equiv & S - \{s\} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\} \\
 & \text{and} & r_x & \leftarrow & \text{arg}_x,\ \ \forall x, 1 \le x \le n
\end{array}
}
$$

Figure 3.33: *The lreturn operation in MAM/DS when the parent and child are connected and the parent has not been resumed since it* `lfork`*ed the child.*

inlet address to point at a code fragment immediately following the `lfork`. This is where a thread that runs to completion will return. `send` can either transfer data to an inlet specified by an address or to an inlet specified by an indirect inlet address.

The actions undertaken by `lreturn` vary according to the state of the child and the parent. If the child runs to completion and its parent has never had work stolen from it, `lreturn` mimics a sequential return (see Figure 3.33). It reclaims the resources of the thread executing the `lreturn` and returns results (in registers) and control to its parent by

$$
\begin{array}{lrrcl}
\langle P, T, Q, Z, S, M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{suspend} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{is\text{-}child}, s, i, \mathtt{true}, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \mathtt{has\text{-}child}, s^p, i^p, f^p, v^p, \cdot\rangle \\
\hline
\langle P, T, Q, Z', S, M\rangle & \text{where} & p & \equiv & \langle \text{seed}(\mathtt{ip}^p, \text{suspend}), \mathtt{sp}^p, t_p, \epsilon\rangle \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{idle}, s, i, \mathtt{false}, \cdot\rangle \\
 & \text{and} & t_p & \equiv & \langle \epsilon, \epsilon, t_p^p, \sigma, s^p, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & \sigma & \equiv & \begin{cases} \mathtt{is\text{-}child} & \text{if } f^p = \mathtt{true} \\ \mathtt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\} \\
 & \text{and} & v^p[i] & \leftarrow & \text{convertReturn}(\mathtt{ip}^p)
\end{array}
$$

Figure 3.34: *The suspend operation in MAM/DS.*

invoking the inlet at its indirect inlet address. Thus, when `lreturn` returns from a child that executed to completion, the indirect inlet address acts like the return address in a sequential call. We discuss the other two cases for `lreturn` below.

`suspend` (see Figure 3.34) resumes the parent at the continuation indicated by `find(ip, suspend)`. The reason we cannot continue at the parent's `ip`, as we did in MAM/DF, is that the code immediately following the `lfork` is the return inlet, which cannot be executed until the child thread actually exits. In addition to resuming the parent, `suspend` also changes the indirect inlet address for the parent's outstanding lazy child to an inlet that receives results for children that have suspended (See Figure 3.31). The work stealing operations, like `suspend`, also change the indirect inlet address for the parent's outstanding lazy child.

Resuming the Parent

When continuation stealing is used to resume the parent, the parent and child are disconnected and the parent continues at the point in the codeblock that follows the `lfork`'s associated inlet. (See Figure 3.31.) When the child finally returns to the parent, it cannot resume the parent, since the parent was already resumed when its continuation was stolen. Instead, `lreturn` runs the inlet in the indirect inlet address and frees the thread's resources, including the processor. Since the inlet ends with `ireturn` we can ensure that

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{lreturn}\ \text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{running}, s, i, f, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \sigma^p, s^p, i^p, f^p, v^p, \cdot\rangle \\
\hline
\langle P,T',Q,Z,S',M\rangle & \text{where} & p & \equiv & \langle v^p[i], \mathtt{sp}^p+2, t_p, R\rangle \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \sigma^p, s^p, i^p, f^p, v^p, \otimes\rangle \\
 & \text{and} & T' & \equiv & T - \{t\} \\
 & \text{and} & S' & \equiv & S - \{s\} \\
 & \text{and} & r_x & \leftarrow & \text{arg}_x,\ \ \forall x, 1 \le x \le n \\
 & \text{and} & *\mathtt{sp} & \leftarrow & \text{wait} \\
 & \text{and} & *(\mathtt{sp}+1) & \leftarrow & \epsilon
\end{array}
$$

Figure 3.35: *The lreturn operation when the parent and child have been disconnected by a continuation stealing operation.*

the processor is idled by making `ireturn` go to the general scheduler, represented by the address `wait` in Figure 3.35[16].

If seed activation is used to get work from the parent, then the routine at the suspend (or steal) code pointer will activate the nascent thread that the seed represents. After the nascent thread has been activated the parent's `ip` will point to the next thread seed, not to the inlet immediately following the `lfork` that activated the thread. The reason that the parent's `ip` does not point to the activated thread's inlet is that the parent can next be resumed either by one of its outstanding children or by a work stealing request. We can see that once a parent has activated a seed all of its children must be started with inlets that store their results, perform synchronization, and then continue in the parent at its current `ip`. Thus we lose some of the benefit of `lreturn` for threads that were activated from seeds. Though we pass the results from the child to the parent in registers, they may immediately be stored in the parent frame.

The behavior of `lreturn` for a thread whose parent has activated a seed ensures that when the inlet executes `ireturn`, the parent will continue at the proper point, which is the parent's current `ip`. It achieves this by placing the parent thread and `ip` on the parent thread's stack before executing the inlet. When the inlet executes `ireturn`, the parent will continue at the proper point (see Figure 3.36).

[16]The rules in Figure 3.14, Figure 3.22, and Figure 3.25 cause a processor with a `ip` of `wait` to get work from the ready queue, from the parent queue by continuation stealing, and from the parent queue by seed activation respectively.

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{lreturn}\ \text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{is\text{-}child}, s, i, f, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \mathtt{has\text{-}child}, s^p, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & v^p[i] & \neq & ip^p \\
\hline
\langle P,T',Q,Z',S',M\rangle & \text{where} & p & \equiv & \langle v^p[i], \mathtt{sp}^p+2, t_p, R\rangle \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \sigma^p, s^p, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & \sigma^p & \equiv & \begin{cases} \mathtt{is\text{-}child} & \text{if } f^p = \mathtt{true} \\ \mathtt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & T' & \equiv & T - \{t\} \\
 & \text{and} & S' & \equiv & S - \{s\} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\} \\
 & \text{and} & r_x & \leftarrow & \text{arg}_x, \quad \forall x, 1 \le x \le n \\
 & \text{and} & *\mathtt{sp} & \leftarrow & \mathtt{ip}^p \\
 & \text{and} & *(\mathtt{sp}+1) & \leftarrow & t^p
\end{array}
$$

Figure 3.36: *The lreturn operation in MAM/DS when a thread seed in the parent has been activated.*

3.4.2 Discussion

MAM/DS allows a potentially parallel call that runs to completion to take advantage of the sequential call and return mechanisms without limiting parallelism. It transfers control and data simultaneously on both invocation and termination.

This machine also clarifies the distinction between continuation stealing and seed activation. When continuation stealing is used to continue the parent before its child returns, the connection between the parent and the child is broken. This causes the child to use the two-step return process of an independent thread; `lreturn` behaves like a `send` followed by an `exit`.

When seed activation is used to continue the parent, it becomes a slave to its children. In other words, each of its returning children causes it to activate a thread seed until no more remain. When no more thread seeds remain, subsequent returning children fail at the join until the last child returns. When the last child returns it continues the parent after the join.

3.5 The Lazy Multithreaded Abstract Machine (LMAM)

With the lazy multithreaded abstract machine (LMAM) we complete our goal of a multithreaded abstract machine that can execute a potentially parallel call with nearly the efficiency of a sequential call. In LMAM an `lfork` starts a new lazy thread on the same stack as its parent thread. Combined with the direct scheduling of threads on call and return introduced in MAM/DS, this allows us to transfer arguments to the child in registers, allocate the child's frame on the stack of the parent, and, if the thread runs to completion, return the results in registers. In this section we introduce the two disconnection methods and the four storage models.

If a thread needs to run concurrently with its parent (because the child executes a suspend or the parent is stolen), then an independent thread is created for the child. This involves disconnecting the control between the parent and child and disconnecting the stack, i.e., creating a new stack so the child (or parent) can continue. The disconnection operation depends on the underlying storage model used to implement the cactus stack.

There are four storage models that we consider: linked frames, multiple stacks, stacklets, and spaghetti stacks. We divide the storage models into two broad classes: linked storage models and stack storage models. In the linked storage models, linked frames and spaghetti stacks, links between frames are explicit. Every thread frame has an explicit link to its parent. In these models disconnection is easy because, in some sense, the threads already have their own "stacks." In the stack storage models, multiple stacks and stacklets, the links between the frames can be implicit or explicit. The implicit links are between lazy threads and their parents. They are implicit since the parent of a lazy thread can be found by doing arithmetic on a stack pointer. The explicit links are between the stacks of the independent threads.

By fixing the storage model for threads, LMAM also dictates how the indirect inlet address is implemented. The indirect inlet address is stored in the same slot of the parent's activation frame used for a sequential call's return address. The link between a child thread and its parent is the pointer from the child to the indirect inlet address in the parent that the child will use to return to the parent. In other words, the return register is the link from a child to a parent. If this link is implicit, then the return register is never stored explicitly.

$$
\frac{
\begin{array}{lrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} \quad p & \in & P \\
& \text{and} \quad p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
& \text{and} \quad \text{inst}[\mathtt{ip}] & = & \mathtt{lfork}\ i' = \text{adr}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
& \text{and} \quad t & \equiv & \langle \epsilon, \epsilon, t_p, \sigma, s, i, f, \cdot\rangle \\
& \text{and} \quad \sigma & \in & \{\mathtt{running}, \mathtt{is\text{-}child}\}
\end{array}
}{
\begin{array}{lrcl}
\langle P,T',Q,Z',S,M\rangle & \text{where} \quad p & \equiv & \langle adr, \mathtt{sp}+F, t_{\text{new}}, R\rangle \\
& \text{and} \quad T' & \equiv & T \cup \{t_{\text{new}}\} \\
& \text{and} \quad Z' & \equiv & Z \cup \{t\} \\
& \text{and} \quad t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{has\text{-}child}, s, i, \top, \cdot\rangle \\
& \text{and} \quad t_{\text{new}} & \equiv & \langle \epsilon, \epsilon, t, \mathtt{is\text{-}child}, s, i', \mathtt{true}, \cdot\rangle \\
& \text{and} \quad r_x & \leftarrow & \text{arg}_x, \quad \forall x, 0 \le x \le n \\
& \text{and} \quad v[i'] & \leftarrow & pc+1 \\
& \text{and} \quad F & \equiv & \text{size of child frame}
\end{array}
}
$$

Figure 3.37: *The lfork operation in LMAM with multiple stacks.*

We thus have two operational semantics for LMAM. The first describes LMAM implemented with linked frames or spaghetti stacks; since the cactus stack is maintained with links, this semantics mirrors that of MAM/DS. The second describes LMAM implemented with multiple stacks or stacklets.

3.6 Disconnection and the Storage Model

When a lazy thread is elevated to an independent thread it must be disconnected from its parent. There are two disconnection methods: eager and lazy. *Eager-disconnect* allows the parent to invoke children on its abstract stack in exactly the same manner as it did before it had a disconnected child. *Lazy-disconnect* leaves the current abstract stack untouched and forces the parent to allocate new children differently than it did before it had a disconnected child.

In order to understand the difference between lazy- and eager-disconnect consider two of the actions taken by an `lfork` in LMAM (see Figure 3.37). First, it saves the indirect inlet address in the child's return register. This acts as the link between the child thread and its parent. Second, it creates an activation frame for the child. Because the `lfork` operation always stores the indirect inlet address in the same slot of the parent we will have to copy the disconnected child's indirect inlet address to a new slot. This allows the parent to invoke its children in the same manner independent of whether it has a disconnected

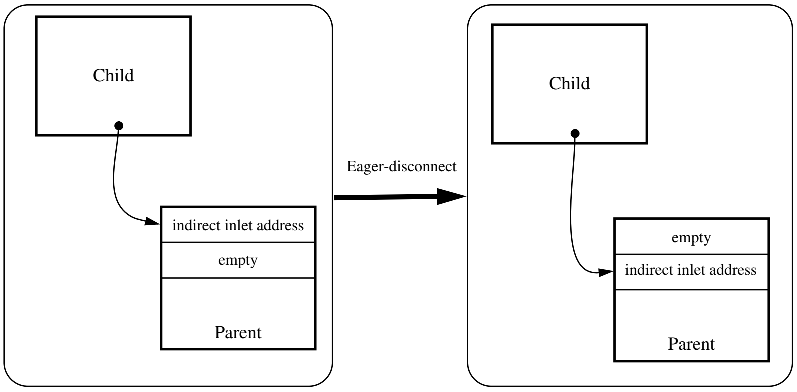


Figure 3.38: *Eager-disconnect used to disconnect a child from its parent when a linked storage model is used. In this case only the indirect inlet address needs to be "copied."*

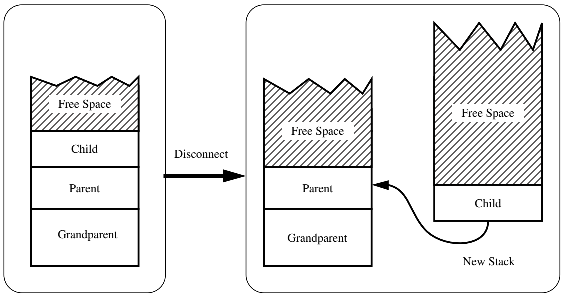


Figure 3.39: *An example of eager-disconnect when the child is copied to a new stack.*

child. Thus, there must be a separate slot in the activation frame for `dfork` in a fork-set. The child's return register also has to be updated. We call this method eager-disconnect (See Figure 3.38). Of course, in the stack storage models eager-disconnect also has to copy the child's activation frame onto another stack.

If we leave the child's indirect inlet address in its original place, then any future child invoked by the parent has to link to another slot in the parent's frame for its indirect inlet address. Lazy-disconnect does not copy the indirect inlet address. It also does not copy the child's activation frame, even in the stack storage models.

3.6.1 Eager-disconnect

For all storage models, eager-disconnect copies the child's indirect inlet address to another slot in the parent's frame. In addition, in the stack models it must split and

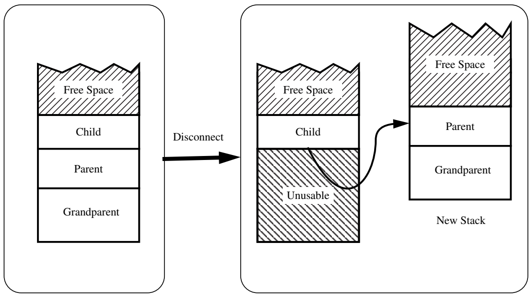


Figure 3.40: *An example of eager-disconnect when the parent portion of the stack is copied to a new stack.*

$$
\begin{array}{llrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\mathtt{ip}] & = & \mathtt{suspend} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \mathtt{is\text{-}child}, s, i, \mathtt{true}, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \mathtt{ip}^p, \mathtt{sp}^p, t_p^p, \mathtt{has\text{-}child}, s^p, i^p, f^p, v^p, \cdot\rangle \\
\hline
\langle P,T,Q,Z',S',M\rangle & \text{where} & p & \equiv & \langle \text{seed}(\mathtt{ip}^p, \text{suspend}), \mathtt{sp}^p, t_p, \epsilon\rangle \\
 & \text{and} & t & \equiv & \langle \mathtt{ip}+1, s_{\text{new}} + \mathtt{sp} - s, t_p, \mathtt{idle}, s_{\text{new}}, i, \mathtt{false}, \cdot\rangle \\
 & \text{and} & t_p & \equiv & \langle \epsilon, \epsilon, t_p^p, \sigma, s^p, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & \sigma & \equiv & \begin{cases} \mathtt{is\text{-}child} & \text{if } f^p = \mathtt{true} \\ \mathtt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\} \\
 & \text{and} & v^p[i] & \leftarrow & \text{convertReturn}(\mathtt{ip}^p) \\
 & \text{and} & S' & \equiv & S \cup \{s_{\text{new}}\} \\
 & \text{and} & s_{\text{new}}[x] & \leftarrow & *(s+x)\ \ \forall x, 0 \le x \le (\mathtt{sp} - s) \\
 & \text{and} & M[a/a'] & \text{where} & s \le a < \mathtt{sp} \\
 & & & \text{and} & a' = a + s - s_{\text{new}}
\end{array}
$$

Figure 3.41: *The suspend operation under LMAM/MS and LMAM/S using child copy for eager-disconnect.*

copy the stack. Eager-disconnect by child- or parent-copying causes the stack to be split and a portion of it to be copied to a newly allocated stack at the time of disconnection. Child-copying copies the portion of the stack used by the child to a newly allocated stack (See Figure 3.39). Parent-copying copies all but the child's frame to a newly allocated stack (See Figure 3.40).[17] In either case, since we copy the local data to a new location, we must either forbid pointers to data in a frame or scan memory and update any pointers to the region being copied. Eager-disconnect is currently employed by most lazy thread systems. [12, 44, 57].

[17] We can also copy only some of the parent's ancestors and leave behind frames which forward data to the new location.

The suspend operation in Figure 3.41 is an example of child-copying that splits the stack. In addition to changing the state of the child and parent, updating the indirect inlet address, and scheduling the parent, it allocates a new stack and copies the data for the child thread to the new stack.

The advantage of eager-disconnect is that the price is paid only once, at the time of the disconnect operation. The disadvantage is that in the stack storage models this one time cost can be quite high, and even more importantly, pointers to data in a frame cannot be passed to other threads.

3.6.2 Lazy-disconnect

Lazy-disconnect, a new method introduced in this thesis, logically disconnects the parent and child without requiring any copying at the time of disconnection. It allows us to use pointers to data in a frame without requiring that we scan memory to update them at disconnect time. Lazy-disconnect is the analog of lazy invocation. Just as a lazy thread call does not eagerly create a new thread, lazy-disconnect does not eagerly separate a child and parent, instead causing the parent to create a new stack for its future children in the same fork-set.

The lazy method does not perform any stack operations in any of the storage models when disconnection occurs, but instead incurs the cost of allocating a new stack for every fork or call made by the parent after it has been disconnected from its child. In other words, lazy-disconnect causes the memory model to devolve into the linked-frame model after a parent has been disconnected from its child. When the logical tree of stacks is cactus-like, having the parent create new stacks for each future child is not onerous, as the children are able to invoke their children using the more efficient stack- or stacklet-based calls.

In short, lazy-disconnect performs disconnection without copying either the child or the parent. Instead, the child steals the stack, causing all future allocation requests in the parent to be fulfilled by allocating a new stack (See Figure 3.42).

To support this method of disconnection, we change the definition of the stack to be a pair consisting of `top` and the former stack, $\langle \texttt{top}, [\,] \rangle$. `top` points to the highest location used in the stack itself.

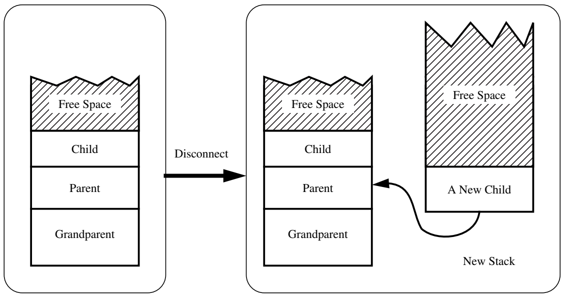


Figure 3.42: *An example of how a parent and its child are disconnected by linking new frames to the parent.*

$$
\begin{array}{lrrcl}
\langle P,T,Q,Z,S,M\rangle & \text{where} & p & \in & P \\
 & \text{and} & p & \equiv & \langle \texttt{ip}, \texttt{sp}, t, \epsilon\rangle \\
 & \text{and} & \text{inst}[\texttt{ip}] & = & \texttt{suspend} \\
 & \text{and} & t & \equiv & \langle \epsilon, \epsilon, t_p, \texttt{is-child}, s, i, \texttt{true}, \cdot\rangle \\
 & \text{and} & t_p & \in & Z \\
 & \text{and} & t_p & \equiv & \langle \texttt{ip}^p, \texttt{sp}^p, t_p^p, \texttt{has-child}, s, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & s & \equiv & \langle \texttt{sp}, \text{data}\rangle \\
\hline
\langle P,T,Q,Z',S,M\rangle & \text{where} & p & \equiv & \langle \text{seed}(\texttt{ip}^p, \text{suspend}), \texttt{sp}^p, t_p, \epsilon\rangle \\
 & \text{and} & t & \equiv & \langle \texttt{ip}+1, \texttt{sp}, t_p, \texttt{idle}, s, i, \texttt{false}, \cdot\rangle \\
 & \text{and} & t_p & \equiv & \langle \epsilon, \epsilon, t_p^p, \sigma, s, i^p, f^p, v^p, \cdot\rangle \\
 & \text{and} & \sigma & \equiv & \begin{cases} \texttt{is-child} & \text{if } f^p = \texttt{true} \\ \texttt{running} & \text{otherwise} \end{cases} \\
 & \text{and} & Z' & \equiv & Z - \{t_p\} \\
 & \text{and} & v^p[i] & \leftarrow & \text{convertReturn}(\texttt{ip}^p)
\end{array}
$$

Figure 3.43: *The suspend operation for LMAM/MS and LMAM/S using lazy-disconnect.*

When `top` equals `sp`, then `lfork` works as it does in LMAM/DS. After disconnection, `top` does not equal `sp` (see Figure 3.43). In this case, `lfork` must allocate a new stack for the child thread (see Figure 3.44).

3.7 Summary

This chapter culminates with the description of a lazy multithreading abstract machine which eliminates all of the unnecessary overhead found in MAM when potentially parallel calls run to completion.

- There are no enqueue, dequeue, or scheduler operations when `lfork` is used to create a lazy thread.

$$
\frac{
\begin{array}{lrcl}
\langle P, T, Q, Z, S, M\rangle & \text{where} \quad p & \in & P \\
& \text{and} \quad p & \equiv & \langle \mathtt{ip}, \mathtt{sp}, t, R\rangle \\
& \text{and} \quad \text{inst}[\mathtt{ip}] & = & \mathtt{lfork}\ i' = \text{adr}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n) \\
& \text{and} \quad t & \equiv & \langle \epsilon, \epsilon, t_p, \sigma, s, i, f, \cdot\rangle \\
& \text{and} \quad \sigma & \in & \{\mathtt{running}, \mathtt{is\text{-}child}\} \\
& \text{and} \quad s & \equiv & < \top, \text{data} > \\
& \text{and} \quad \top & \neq & \mathtt{sp}
\end{array}
}{
\begin{array}{lrcl}
\langle P, T', Q, Z', S', M\rangle & \text{where} \quad p & \equiv & \langle adr, \text{data}_{\text{new}} + F, t_{\text{new}}, R\rangle \\
& \text{and} \quad T' & \equiv & T \cup \{t_{\text{new}}\} \\
& \text{and} \quad Z' & \equiv & Z \cup \{t\} \\
& \text{and} \quad t & \equiv & \langle \mathtt{ip}+1, \mathtt{sp}, t_p, \mathtt{has\text{-}child}, s, i, \top, \cdot\rangle \\
& \text{and} \quad t_{\text{new}} & \equiv & \langle \epsilon, \epsilon, t, \mathtt{is\text{-}child}, s_{\text{new}}, i', \mathtt{true}, \cdot\rangle \\
& \text{and} \quad r_x & \leftarrow & \text{arg}_x, \ \ \forall x, 0 \le x \le n \\
& \text{and} \quad v[i'] & \leftarrow & pc + 1 \\
& \text{and} \quad s_{\text{new}} & \equiv & < \text{data}_{\text{new}} + F, \text{data}_{\text{new}} > \\
& \text{and} \quad \text{data}_{\text{new}} & \equiv & [) \\
& \text{and} \quad S' & \equiv & S \cup \{s_{\text{new}}\} \\
& \text{and} \quad F & \equiv & \text{size of child frame}
\end{array}
}
$$

Figure 3.44: *The lfork operation under LMAM/MS using lazy-disconnect after a thread has been disconnected from a child.*

- Processor registers are used to transfer arguments to the child and results back to the parent.
- Unnecessary register saves and restores are eliminated since the thread is never scheduled by the general scheduler except when it has previously suspended.
- Lazy threads do not require there own stack.
- No synchronization operations are performed between the parent and its children when the children are scheduled sequentially.

Chapter 4

Storage Models

This chapter describes the different concrete implementations for the storage portion of the abstract machine presented in the last chapter. We investigate four storage models for a global cactus stack: linked frames, multiple stacks, spaghetti stacks, and stacklets. Although the storage model and the control model interact, we delay a detailed discussion of this interaction until the next chapter. We explain how each model maintains the logical global cactus stack, how it allocates, deallocates, and disconnects frames, and what trade-offs it introduces. There are four kinds of calls, sequential, lazy fork, fork, and remote fork, each of which maps onto a different kind of allocation request in each storage model.

For each storage model we discuss four trade-offs: allocation efficiency, internal fragmentation, external fragmentation, and ease of implementation. Of these, the most important is the cost of allocating and deallocating an activation frame for the four different types of calls. Internal fragmentation, caused by leaving unused space within the basic allocation unit, is also important. Less important is external fragmentation, which results from allocating activation frames in different regions of the memory space. Finally, implementation details make certain storage-model variations impractical on some machines.

The models fall into two broad categories: linked storage models and stack storage models. The linked storage models include the linked frames and spaghetti stack models, which represent the parent-child relationship between lazy threads with explicit links. The stack storage models include the multiple stacks and stacklets models, which represent the parent-child relationship between lazy threads implicitly. Another view is that these four models are span a spectrum of possible implementations with linked frames on one end and multiple stack on the other. In between, spaghetti stacks and stacklets offer different compromises, with spaghetti stacks being closer to linked-frames and stacklets closer to multiple stacks.

4.1 Storing a Thread's Internal State

In this section, we discuss storage requirements for a thread's state variables, its `ip`, indirect inlet addresses for its disconnected children, and its portion of the ready queue.

In the previous chapter, we made an artificial distinction between the thread's stack and its state, i.e. its `ip`, `sp`, parent pointer, and return register. In fact, when a thread is not running it stores the thread state in its stack. A lazy thread also has state, although it may not have its own stack. It does, however, always have a base frame on some stack. It is in this base frame that we store the lazy thread's state.

Since all threads, lazy or independent, have a unique base frame, we use the address of the base frame to identify a thread. We sometimes find it convenient simply to refer to a thread by its frame.

In all of the memory models presented, each activation frame includes a field to store the thread's `ip`. Recall that a thread's `ip` is the address of the instruction at which it will begin operation when it is next scheduled, either by the scheduler or by an outstanding child. The return register is made up of the parent pointer and the indirect inlet address in the parent.[1] For lazy threads and sequential calls, this is the return address of the child, which is the same as the inlet address.

As discussed in Section 3.6, there must be a sufficient number of slots in the activation frame to hold the indirect inlet address of children that may be disconnected. In eager-disconnect, the slots are used to store the indirect inlet addresses for disconnected children. In lazy-disconnect the slots are used to store the indirect inlet addresses of children invoked after disconnection.

In the concrete implementation of LMAM we distribute the ready queue among the independent threads. Since the ready queue can include each independent thread at most once, it can be implemented by allocating a fixed number of frame slots per independent thread. We implement the ready queue as p linked lists, where p is the number of processors in the system. Thus we allocate one slot per thread activation frame and one ready queue

[1] Section 5.3 discusses the rationale for splitting the return register between the parent and the child.

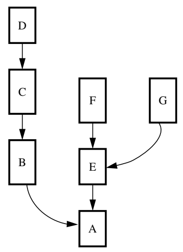


Figure 4.1: *A cactus stack using linked frames. Each frame has an explicit link to its parent*

pointer per processor. Since lazy threads can become independent we also reserve a slot in each lazy thread activation frame.

To summarize, every model requires that an `ip`, an indirect inlet address, and a slot for the ready queue be stored in each activation frame. For reasons given in Chapter 5, the return register is not directly realized. The parent pointer is included in the activation frame only in the linked frames and spaghetti stacks models. The `sp` is not stored in the thread's state in concrete implementations of any of the models.

4.2 Remote Fork

Remote fork, used only on distributed memory machines, is handled by sending messages to the remote processor, which performs the allocation request and deposits the arguments into the newly allocated frame. Since local fork has exactly the same storage requirements as remote fork, we discuss only local fork in this chapter.

4.3 Linked Frames

We first discuss mapping the cactus stack onto linked frames. This approach is the simplest and as we shall see the least attractive. It is also the most commonly used, e.g. in Cilk [7], Id90 [25], and Mul-T [39].

The cactus stack is represented explicitly by the links between the frames, as shown in Figure 4.1. Each activation frame, whether it is the start of a new thread or a sequential call, is allocated in a separate heap-allocated frame. The link between a child and its parent is explicitly part of the activation frame.

The linked-frames mapping imposes an expensive allocation and deallocation operation on every call and return. It produces no internal fragmentation, but can, produce external fragmentation, which can have negative effects on the processor cache and TLB, since frames related by control may be in completely different areas of the memory system.

4.3.1 Operations on Linked Frames

When using linked frames, all of the allocation requests (`fork`, `lfork`, and `call`) are treated uniformly. We allocate a child frame and link it to its parent frame. `exit`, `lreturn`, and `return` free the frame and return it to a pool of unused frames, which may be reused for subsequent allocation requests. Since each activation frame is physically separate from every other, disconnecting a lazy thread from its parent never requires copying.

The linked-frame mapping is simple to implement. However, it does nothing to optimize for the most frequent operations, the sequential call and the potentially parallel call that completes without suspension.

4.3.2 Linked Frame Timings

Here we discuss the cost of the basic operations. We present instruction counts whenever possible and cycle counts for library operations. We divide the instructions into three categories: memory reference instructions, branch instructions, and register-only instructions. We contend that on modern processors the most important metric is memory reference instructions. Thus, we try to minimize the number of memory reference instructions. In the case where we depend on a library call to do some of the work, we measure the cycle count to allow application to other processors. In all cases, we are concerned only with the costs of storage management; control costs are covered in the next chapter. In particular, we do not include the costs of manipulating the indirect inlet address on disconnection, since it depends on the queuing method chosen (See Section 5.6).

The timings are presented in Table 4.1. The first two lines of the table contain the cost of performing a sequential call and return in C.[2] This is used as a comparison against which we can judge the effectiveness of the linked-frame model. The next two lines show the cost of performing a sequential call in the linked-frame model. We then show the storage costs of allocating, scheduling, and freeing a thread. The allocation cost does not include

[2] In all cases we give the cost of a call with no arguments and a return with no results.

Operation	Instruction Count		
	Memory	Branch	Register
C call	0	0	1
C return	0	0	1
Sequential call	4	1	4
Sequential return	4	0	2
Create thread from pool	4	1	4
Schedule thread	0	0	0
Free thread	3	0	1
Create lazy thread	4	1	4
Lazy return	4	0	2
Eager-disconnect	0	0	0
Create lazy thread after lazy-disconnect	4	1	4

Table 4.1: *Times for the primitive operations using linked frames. These times do not include any control costs. They assume frames are allocated from a pool of frames and do not include the cost of a malloc.*

the cost of performing a malloc to allocate the thread. Instead, our allocation policy is to put freed frames in a pool of frames which can be reused. We assume that requests satisfied in the pool dominate those that are not in the pool and thus do not include the cost of doing a malloc.[3] Finally, we show the storage costs for lazy threads.

As we see from the table, the linked-frame model imposes a significant penalty on sequential calls over that of a traditional stack model. Furthermore, sequential calls have no advantage over forks. While `lfork` has no special advantage over `fork` in terms of storage, it also has no extra cost in the case when the child is disconnected from the parent.

Like call, return is also significantly more expensive than the traditional stack model. `lreturn` is more expensive than `exit` because it includes the cost of loading the frame pointer for the parent frame in addition to freeing the frame.

The data bears out the reasoning that the linked-frame model is not very efficient for sequential and potentially parallel calls. In addition, it creates external fragmentation. In spite of these problems, it is used often because it is easy to implement.

[3]This operation could be made more efficient by using a heap pointer as suggested in [2]. However, Appel's method requires garbage collection or a reclamation method that makes the resulting cactus stack look very similar to our spaghetti stacks.

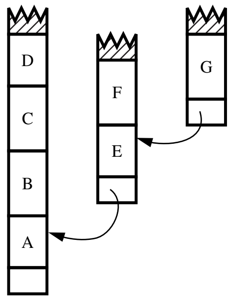


Figure 4.2: *A cactus stack using multiple stacks. There are only explicit links from the stubs at the base of a stack to the parent of the thread in the base frame of the stack.*

4.4 Multiple Stacks

The next approach we discuss is that of mapping the cactus stack onto multiple stacks. Each thread is assigned its own contiguous stack. The cactus stack is maintained with explicit links between stacks and implicit links within each stack (See Figure 4.2). The link between a lazy thread and its parent is implicit because a lazy thread is allocated on the same stack as its parent. The parent pointer is not stored, but is calculated by doing arithmetic on the stack pointer.

The main advantage of this approach is that an allocation request for `call` and `lfork` has the same cost as a sequential call in a single threaded system. The disadvantages are that it limits the number of threads in the system due to the large size of each stack, creates internal fragmentation, and increases the cost of creating an independent thread.

Multiple stacks have never to our knowledge been used in a lazy multithreaded system, but variations of the multiple stacks approach has been used in many thread packages, including pthreads [33], NewThreads [37], and Solaris Threads [56], and in some languages, e.g. Cid [48]. All of these systems fix the size of the stack and none provides protection against stack overflows.

4.4.1 Stack Layout and Stubs

Each stack in the multiple stacks approach is divided into two regions: the stub and the frame area (See Figure 4.3). The *stub* contains thread-specific data and maintains the global cactus stack by linking the individual stacks to each other. The *frame area* contains the activation frames of children invoked by either sequential calls or `lfork`s. Thus, the

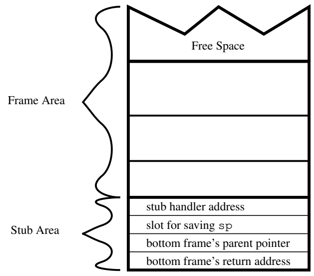


Figure 4.3: *Layout of a stack with its stub.*

cactus stack is maintained both explicitly, with links from the stack stubs to the parent thread of the bottom frame in the frame area, and implicitly, between lazy threads in the frame area.

The stack stub serves to maintain the explicit links needed between threads. It also stores an `ip` that can be accessed by the thread above it on the stack. When the thread returns, instead of testing its own state, it does an `lreturn` that assumes that the thread is lazy and is still connected to its parent. `lreturn` invokes the stub routine pointed to by the stub's `ip`. This routine, using the data in the stub's frame, then performs the necessary operations to return to the thread's parent.

The stack stub maintains information common to all the activation frames on its stack. When none of the threads on the stack are `running`, the stub contains the `sp` for the stack. The `sp` stored in the stub is loaded into the processor's `sp` when any of the threads in the stack are scheduled by the general scheduler. Every activation frame can calculate the stub for the stack on which it lives.[4] The stub also contains the parent pointer for the bottom frame in the stack. Since all other threads on the stack are descendants of the bottom frame, this is the only explicit link needed.

Stubs allow every thread, lazy or independent, to use the sequential return mechanism of `lreturn` even though a cactus stack is being used. The stack stub stores all the data needed for the bottom frame to return to its parent. When a new stack is allocated, the parent's return address and stack pointer are saved in the stub and a return address to the stub handler is given to the child. When the bottom frame in a stack executes a return, it does not return to its caller. Instead it returns to the stub handler. The stub handler

[4]This can easily be accomplished by fixing the size of a stack to be a power of two and using an exclusive-or and logical-and instruction.

performs stack deallocation and, using the data in the stack stub, carries out the necessary actions to return control to the parent, e.g. setting the `sp` to the parent's stack.

There are two different stub handlers in the multiple stacks model: the local-fork stub handler, which performs the actions described in the previous paragraph, and the remote-fork stub handler. The remote-fork stub handler uses indirect active messages [62] to return data and control to the parent's message handler, which in turn is responsible for integrating the data into the parent frame and indicating to the parent that its child has returned.

4.4.2 Stack Implementation

Since we do not have infinite memory in which to allocate stacks, we must decide how to simulate a virtually infinite stack. Many thread packages do this by allocating a relatively small (on the order of 4k-16k) fixed stack for each thread. This works well for some programs on some data sets but not at all for those that use a great deal of automatic data or have deeply nested routines. There are, however, many programs which, because the stack size depends upon the input data set, require large stacks.

Problems occur because these thread packages have no stack overflow detection. Stack overflow is especially problematic in multithreaded systems. If stacks are adjacent, a thread that overflows will write to the base of another thread's stack, which causes errors to occur when the other thread is scheduled and finally returns to the frame at the base of its stack. Even if the stacks are not adjacent, the error caused by a stack overflow may not surface until later in the program's execution.

Stack overflow protection can be added in one of three ways. First, sequential calls can check for stack overflow, which makes this model exactly like stacklets (See Section 4.6). Second, a guard page can be allocated for each stack. Third, a guard page can be created dynamically whenever a thread is scheduled. The latter two methods require that stacks be allocated on page boundaries and be multiples of the system page size. If a static guard page is allocated, then a page is wasted for each stack. With the dynamic method, the cost of switching to a new thread is raised substantially.[5]

[5]Guard pages require that the operating system allow the user to change the write protection of a page in memory, which is not supported on some machines. It is probably for this reason that thread packages do not support overflow detection.

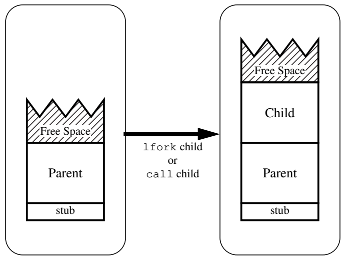


Figure 4.4: *Allocating a sequential call or an* `lfork` *on a stack.*

If there is already a copying garbage collector in the system, we can use it to avoid allocating large fixed regions. Instead, we can allocate a small stack and set a guard. When the stack overflows, we reallocate it to a larger region, copying the data, and adjusting any pointers to the data in the stack as appropriate. Of course, pointer adjustment would have to happen across the entire address space.

Without garbage collection, we have three different ways to implement multiple stacks. The fixed scheme allocates a small fixed stack per thread. The fixed scheme with static guard pages associates a static guard page to each stack. The fixed scheme with dynamic guard pages sets the guard page on each stack when the thread associated with it is running. The first scheme can be implemented using standard memory allocation primitives like malloc. The latter schemes require the ability to manipulate the read/write permissions of virtual pages.

4.4.3 Operations on Multiple Stacks

In this section, we discuss the implementation of the operations using multiple stacks. The multiple stacks model exploits the fact that each thread has its own stack. Sequential call and lazy fork both allocate the child frame on the stack of its parent. As a result, when a lazy thread needs to become independent, we must perform some kind of operation to disconnect the parent and child threads.

Allocation

The allocation operation for `call` or `lfork` allocates the child frame on the same stack as the caller. The operation is carried out by the child frame by adjusting the `sp` (See Figure 4.4).

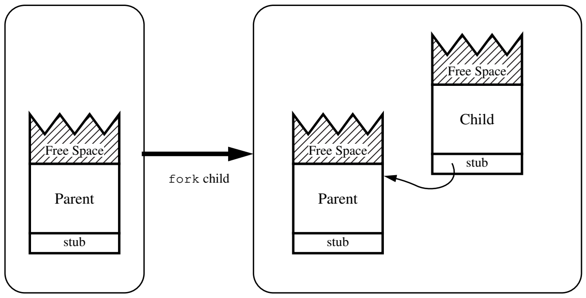


Figure 4.5: *Allocating a new stack when a fork is performed.*

The fork operation requires that a new stack be allocated and linked to the caller's stack (See Figure 4.5). The actual allocation requests will differ depending upon which stack scheme is used, but they are similar in all cases. If there is a free stack which has been allocated already, it is reused. Otherwise, a new stack area is allocated from the system using the appropriate library call, i.e., malloc or mmap, and if a static guard page is being used, it is set.

Deallocation

When a lazy thread (or sequential call) exits (or returns), it deallocates its frame area by adjusting the `sp`. When a thread exits, it returns its stack to the free pool of stacks.

Scheduling

Unlike the linked-frame representation, the multiple stacks model requires work to be performed whenever a thread is scheduled. In particular, the dynamic guard page scheme requires that the page above the stack be set so that if an overflow occurs, it will cause a fault. When a thread suspends, exits, or yields, the guard page is unprotected. In addition, in all the schemes the `sp` has to be set from the stub data.

Disconnection

Since a lazy thread is allocated on the same stack as its parent, if the lazy thread suspends it must be converted into an independent thread. This process requires that the parent and child each have its own logical stack. We now discuss the two options for disconnection; lazy and eager, identified in Section 3.6.

The lazy method avoids copying (and allows pointers to automatic variables[6]) by forcing all subsequent allocations by the parent onto new stacks. This means that subsequent sequential calls and lazy forks will always allocate a new stack for their frames. Since a suspending child is left unchanged, the `ip` that a parent presents to its future children must reside in a separate slot in the frame.

Eager-disconnect is significantly harder to implement than lazy-disconnect. When we copy an activation frame, we must be careful not to invalidate any pointers to the copied frames. Even if we disallow pointers to automatic variables within a frame, we must correctly maintain the pointers that comprise the cactus stack and the ready queue. Although we can update parent pointers that are on the local processor relatively inexpensively, updating remote parent pointers is costly and can lead to race conditions which are difficult and expensive to alleviate. To ensure that frames with remote children are not copied, we force any frame with a remote child to live at the bottom of a stack (creating a new stack if necessary) and enforce the invariant that the bottom frame in a stack is never copied.

This invariant implies that when a lazy thread suspends or yields, it must be disconnected from its parent by copying itself and its descendants to a new stack. This child-copying may require updating local pointers but never remote pointers. Since child-copying is used whenever a lazy thread suspends or yields, we also have the invariant that whenever a child suspends it is at the top of the stack.

Child-copying creates a new stack for the children copied and a stub to link the new child stack to the parent stack. The resulting configuration is exactly the same as if the bottom child of those copied had been eagerly forked.

When a child suspends and its parent is not on the parent queue, the suspend transfers control to an ancestor of the child, forcing many frames to be copied at once. The pointers to each of these frames will have to be updated. Figure 4.6 shows the effects of two suspend operations. In the first, the parent is on the parent queue and one frame is copied. In the second, the parent is not on the parent queue, forcing multiple copies.

The implementation of eager-disconnect differs depending upon the control model. When seed activation is used to steal work, a thread having its seed stolen is copied to a new stack to ensure that it is the bottom frame. Figure 4.7 shows the effect of activating a seed from a lazy thread.

[6] An automatic variable is one local in scope to a function, not to be confused with memory local to a processor.

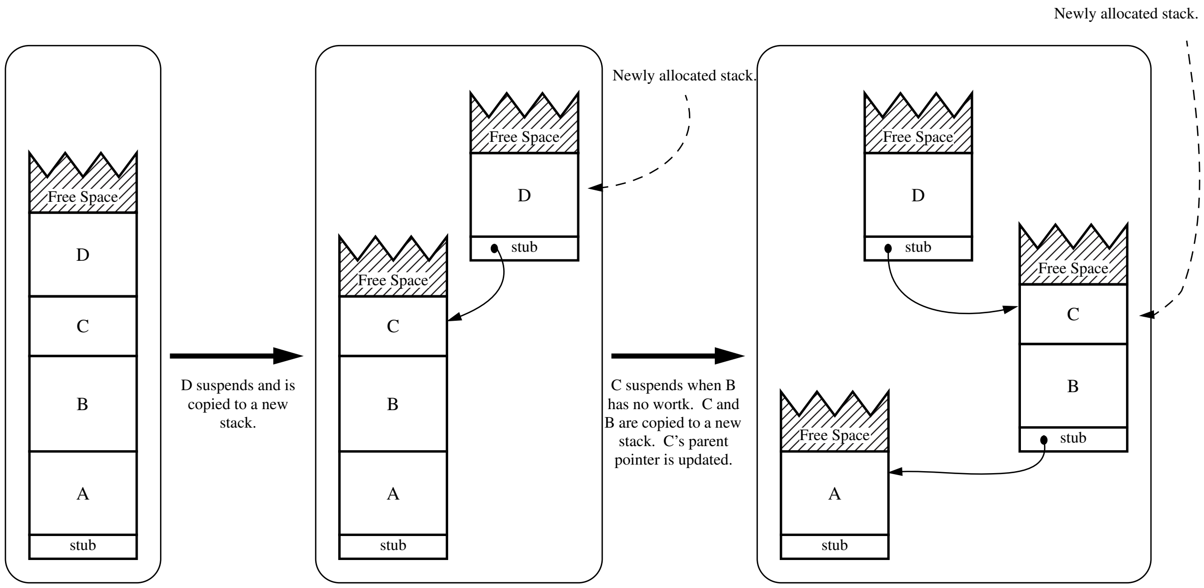


Figure 4.6: *Disconnection using child-copy for a suspending child. First D suspends, and then C suspends. When C suspends, B is not on the parent queue, so control is passed to A. This forces both B and C to be copied.*

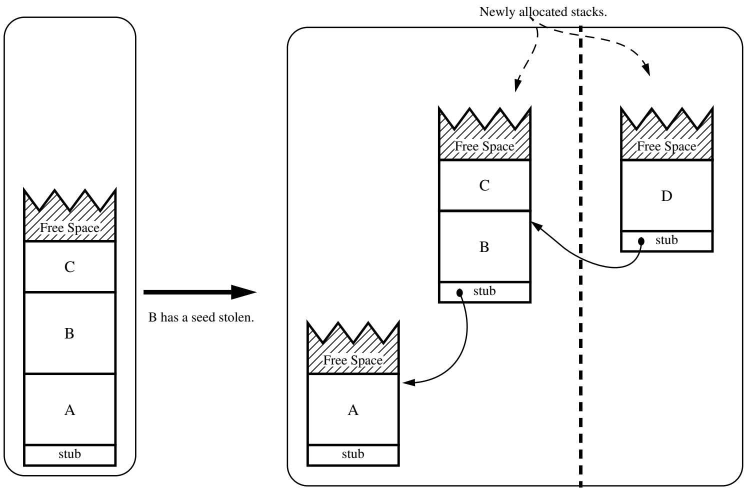


Figure 4.7: *Disconnection caused by seed activation also uses child-copy. Thread B has a seed stolen, causing it to be copied to new stack to maintain the invariant that a thread with a remote child is at the bottom of a stack.*

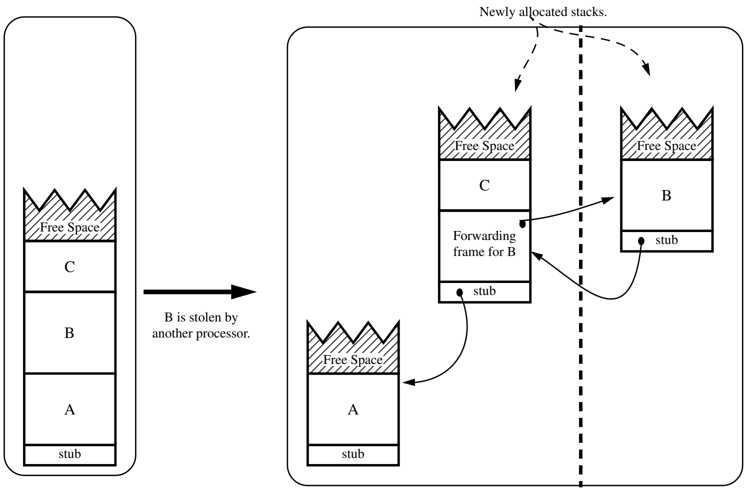


Figure 4.8: *Disconnection caused by continuation stealing. B migrates to a new processor leaving behind a forwarding frame.*

Since continuation stealing results in migration, we need to ensure that a forwarding frame is left behind after the frame is copied (See Figure 4.8). When a child of the stolen frame returns, the forwarding frame sends the results to the parent's new location. When the stolen thread completes, it sends its results back to the forwarding frame, which returns to the stolen frame's parent and then exits.

4.4.4 Multiple Stacks Timings

As the timings in Table 4.2 show, the multiple stack mapping introduces no overhead for sequential calls, `lfork`s, or `fork`s. However, when independent threads are needed, a significant overhead is incurred. For all schemes the cost of creating, suspending, and freeing a thread includes the cost of the stub used to link the thread to its parent. Notice that even though stubs are used, the cost of using threads in the multiple stack model is about the same as that in the linked-frame model (See Table 4.1).

Overflow protection introduces the largest overhead in the multiple stacks model. The dynamic scheme is clearly unacceptable because the call to protect the page on all scheduling operations costs four orders of magnitude more than any other operation. The static guard page avoids this problem at the cost of wasting memory and increasing the cost of allocating a stack from the system. Timing malloc and mprotect is imprecise at best.

Operation	Instruction Count			
	Memory	Branch	Register	System
C call	0	0	1	0
C return	0	0	1	0
Sequential call	0	0	1	0
Sequential return	0	0	1	0
Create thread from pool	5	1	1	0
Suspend thread (fixed)	2	1	0	0
Suspend thread (dynamic)	2	1	0	6600
Schedule thread (fixed)	0	0	0	0
Schedule thread (dynamic)	0	0	0	6600
Free thread (fixed)	3	1	3	0
Free thread (dynamic)	3	1	3	6600
Create lazy thread	0	0	1	0
Lazy return	0	0	1	0
Eager-disconnect(n-word frame)	2n+6	12	30	0
Create lazy thread after lazy-disconnect	5	1	1	0

Table 4.2: *Times for the primitive operations using multiple stacks. These times do not include any control costs. They assume stacks are allocated from a pool of stacks and exclude the cost of a malloc and mmap. "Fixed" denotes the fixed and static guard schemes. "Dynamic" denotes the dynamic guard page scheme. System calls are included as cycle counts. The eager-disconnect cost is the minimum cost.*

When a static guard page is used, however, it increases the cost of allocating a new stack by at least 30%.

The other significant increase in cost imposed by the multiple stacks model is due to eager-disconnect. Eager-disconnect causes at least $2n$ memory references, where n is the size of the frames copied. In addition, all pointers to the frames have to be updated, including the pointers in the ready queue and the links from parents to their children.

Although sequential and potentially parallel calls are efficiently implemented in the multiple stacks model, the inefficiencies inherent to supporting overflow protection and eager-disconnect make it a less than ideal candidate for a storage model. It introduces both internal and external fragmentation, and it is non-portable.

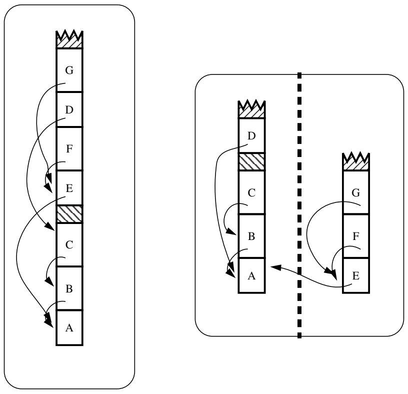


Figure 4.9: *Two examples of how the cactus stack could be represented using spaghetti stacks. In the first, the cactus stack is local to a single processor. In the second it is split across processors.*

4.5 Spaghetti Stacks

Spaghetti stacks are a compromise between the fork-centric linked-frame approach and the call-centric stack approach. By removing the requirement that all free space in a stack be above the current frame, the cactus stack can be implemented in a single stack. This means that no storage related disconnection operations are needed when a lazy thread becomes independent.

The use of a single interwoven stack for multiple environments was first introduced by Bobrow and Wegbreit [10]. It has since been used in a more simplified form in Olden [12] and in implementing functional languages [31]. In all cases, it has been accompanied by a garbage collector to perform compaction.

In a spaghetti stack, every allocation request is fulfilled by allocating space at the top of the stack. A frame in the middle of the stack can exit, however, and create free space in the middle of the stack. The spaghetti stack model thus requires a method to reclaim this fragmented freed space. It also requires that the global cactus stack be maintained explicitly, with links between every child and its parent (See Figure 4.9).

Unlike the linked-frame and multiple-stack approaches, in which each frame (or stack) is associated with a thread, spaghetti stacks are assigned to processors in the system so that each processor has its own spaghetti stack. The global cactus stack is maintained with global pointers between child frames on one processor and their parent frames on remote processors.

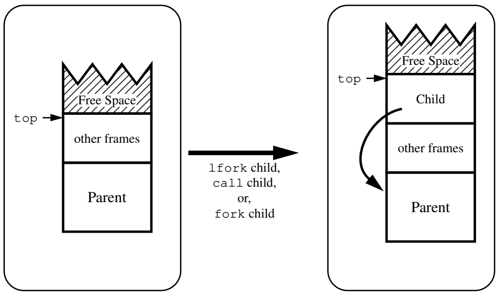


Figure 4.10: *Allocation of a new frame on a spaghetti stack.*

The main advantage of spaghetti stacks over multiple stacks is that no storage related disconnection operation is needed. In addition, allocation requests are relatively inexpensive compared to the linked-frame approach. The disadvantages are that it can be expensive to reclaim freed space and that internal fragmentation can consume a lot of memory.

4.5.1 Spaghetti Stack Layout

In addition to the stack pointer and frame pointer used in a single-threaded system, a spaghetti stack requires a top pointer (`top`), which points to the first free word above the last word used in the stack, i.e., the location to which `sp` would point in a traditional stack.

The cactus stack is maintained by links in each frame on the stack. A link from parent to child, even for sequentially called children, is required because the child frame may not be adjacent to its parent (See Figure 4.9).

4.5.2 Spaghetti Stack Operations

Allocation

The allocation operation is identical for `call`, `lfork`, and `fork`. The child allocates space for its frame by adding to `top` (See Figure 4.10). It saves the current `sp` in the base of its frame and then sets the `sp` to the top of its frame and the `fp` to the base of its frame. Remote fork allocates the remote stub in the same manner and then performs a local fork to allocate the thread.

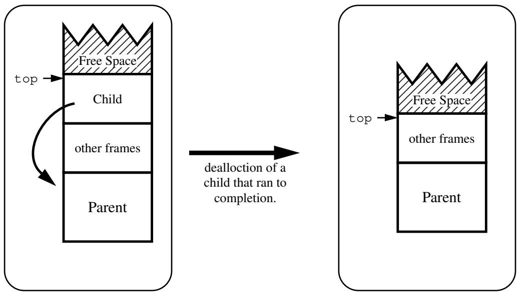


Figure 4.11: *Deallocating a child that has run to completion.*

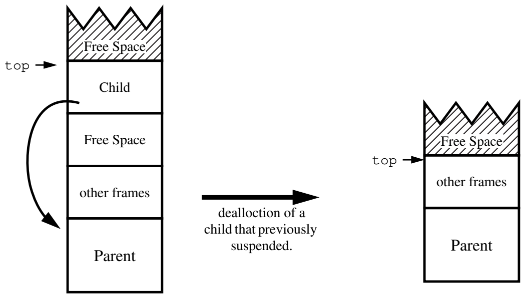


Figure 4.12: *Deallocating a child that has suspended but is currently at the top of the spaghetti stack.*

Deallocation

When a child returns or exits, it attempts to deallocate its frame. There are three possibilities: the child has run to completion, the child has suspended and is at the top of the stack, or the child has suspended and is not at the top of the stack.

If the child has run to completion, then it must be at the top of the spaghetti stack.[7] In this case, it deallocates its frame by setting `top` to the base of its frame (See Figure 4.11).

If the child previously suspended but is currently at the top of the stack, it checks the area below it to see if it is active or free. If it is active, then it sets `top` to the base of its frame. If the area below it has been marked as free, then it sets `top` to the base of the free area (See Figure 4.12).

If the child is not at the top of the stack, then it can mark the frame as free, but it cannot actually reclaim the space, since there are active frames above it in the stack. In

[7]This is only true if work distribution is performed by work stealing. If remote fork is used to create threads on other processors on a distributed memory machine, then we must consider additional cases.

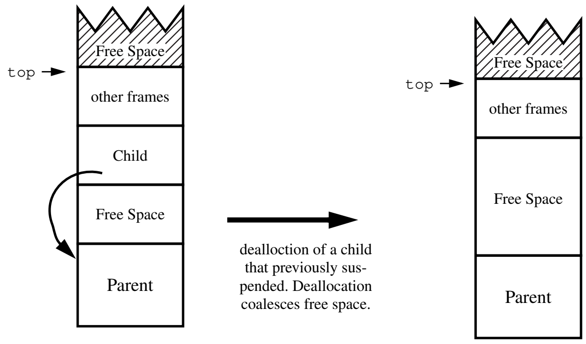


Figure 4.13: *Deallocating a child that has suspended and is in the middle of the stack. The child coalesces any free space below it into its frame.*

this case, the child marks its frame as free and checks to see if the frame below it is free. If the frame below is free, then both frames are merged into one free area. Later, when the frame above it exits, it will deallocate itself and the current frame (including any of the frames with which it was merged) as described in the previous paragraph (See Figure 4.13).

Whenever a frame is not at the top of the stack and is attempting to deallocate itself, it must have a suspended frame above it. This invariant lets us use PCRCs, as described in Section 5.3, to eliminate any reclamation overhead for frames that run to completion. In other words, if a frame runs to completion, it need not perform any checks, but can blindly set `top` to the base of its frame.

4.5.3 Spaghetti Stack Timings

As Table 4.3 shows, spaghetti stacks are a good compromise between linked frames and multiple stacks. In all cases, they are less expensive than linked frames, and all thread operations are cheaper on spaghetti stacks than on multiple stacks.

Sequential call and `lfork` on a spaghetti stack incurs the cost of saving a link to its parent, but it avoids allocation from a pool, which is expensive. It is only slightly more expensive than a sequential call on a stack.

The thread operations shown here include the cost of using a stub for independent threads and lazy threads invoked after either eager- or lazy-disconnect. Stubs allow us to reclaim the memory in the spaghetti stack without having to perform a test on every return.

Operation	Instruction Count		
	Memory	Branch	Register
C call	0	0	1
C return	0	0	0
Sequential call	1	0	2
Sequential return	1	0	1
Create Thread	3	0	4
Schedule Thread	0	0	0
Free Thread	3	1	3
Create Lazy Thread	1	0	2
lazy return	1	0	3
Eager-disconnect	0	0	0
Create lazy thread after disconnect	3	0	4

Table 4.3: *Times for the primitive operations using spaghetti stacks. These times do not include any control costs.*

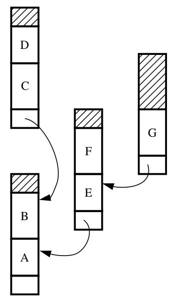


Figure 4.14: *A cactus stack using stacklets.*

4.6 Stacklets

Stacklets, introduced in [24], effect a different compromise between the linked-frame and stack approaches. Whereas spaghetti stacks incur extra cost when a function returns (to handle reclamation of free space), stacklets maintain a stack invariant, but incur extra cost on function call, to check for overflow.

A stacklet is a region of contiguous memory on a single processor that can store several activation frames. Unlike spaghetti stacks, each stacklet is managed like a sequential stack. As in the stack model, the global cactus stack is maintained implicitly (between frames in a single stacklet) and explicitly (with links between stacklets) (See Figure 4.14).

4.6.1 Stacklet Layout

Each stacklet is divided into two regions, the stub and the frame area. The stub contains data that maintain the global cactus stack by linking the individual stacklets to each other. The frame area contains the activation frames (See Figure 4.15).

4.6.2 Stacklet Operations

Stacklets exploit the fact that each independent thread begins at the base of a new stacklet while lazy threads are allocated on the same stacklet as their parent. Thus, lazy thread forks and sequential calls both perform a sequential allocation. A sequential allocation is one that requests space on the same stack as the caller. The child performs the allocation and determines whether the new frame fits in the current stacklet. If it does, `sp` is updated appropriately (See Figure 4.16). If not, a new stacklet is allocated and the child frame is allocated on the new stacklet (See Figure 4.17).

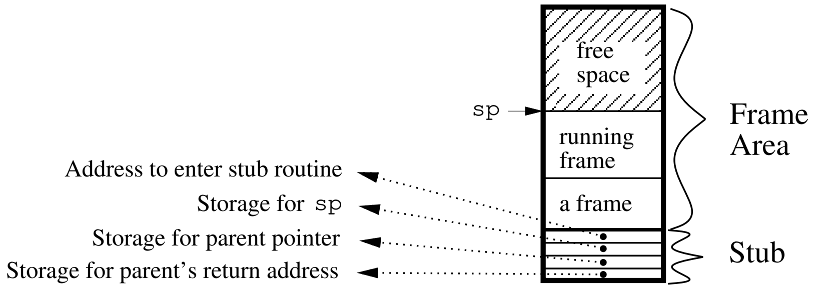


Figure 4.15: *The basic form of a stacklet.*

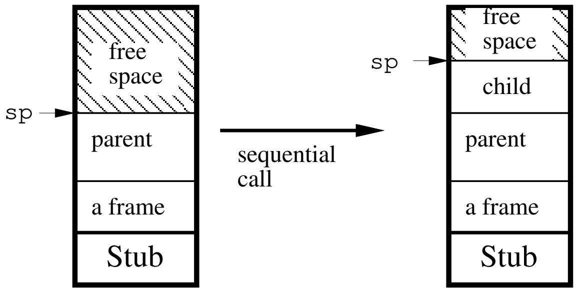


Figure 4.16: *The result of a sequential call which does not overflow the stacklet.*

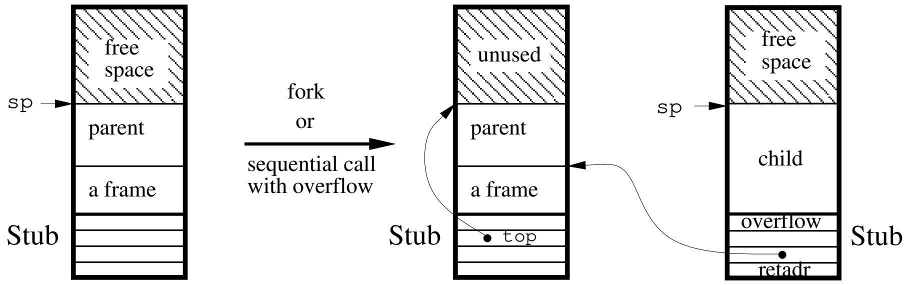


Figure 4.17: *The result of a fork or a sequential call which overflows the stacklet.*

An eager fork also causes the child to be allocated on a new stacklet. We can either run the child in the new stacklet immediately or schedule the child for later execution. In the former case, **sp** points to the child stacklet (See Figure 4.17). In the latter case, the **sp** remains unchanged after the allocation.

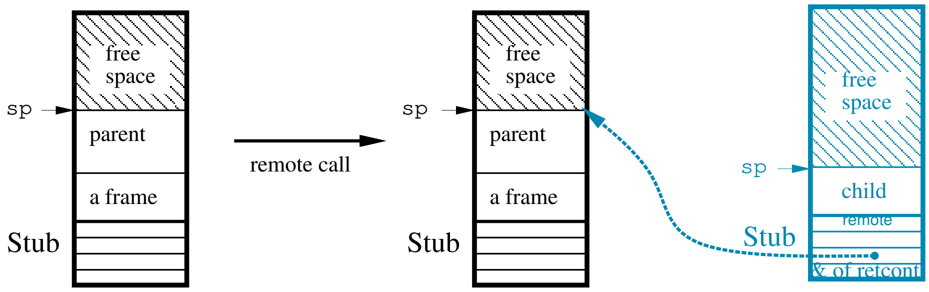


Figure 4.18: *A remote fork leaves the current stacklet unchanged and allocates a new stacklet on another processor.*

For a remote fork, there are no stacklet operations on the local processor. Instead, a message is sent to a remote processor with the child routine's address and arguments (See Figure 4.18).

The overhead in checking for stacklet overflow in a sequential call is three register-based instructions and a branch (which will usually be successfully predicted). If the stacklet overflows, a new stacklet is allocated. This cost is amortized over the many invocations that will run in the stacklet.

4.6.3 Stacklet Stubs

As with multiple stacks, stacklet stubs are used to link the individual stacklets that form the global cactus stack. The stub handler performs stacklet deallocation and, using the data in the stacklet stub, carries out the necessary actions to return control to the parent.

4.6.4 Compilation

To reduce the cost of frame allocation even further, we construct a call graph which enables us to determine for all non-recursive function calls whether an overflow check is needed [27]. Each function has two entry points, one that checks stacklet overflow and another that does not. If the compiler can determine that no check is needed, it uses the latter entry point. This analysis may insert code to perform a stacklet allocation to guarantee that future children will not need to perform any overflow checks.

Operation	Instruction Count		
	Memory	Branch	Register
C call	0	0	1
C return	0	0	1
Sequential call	0	1	5
Sequential call (overflow)	5	1	5
Sequential return	0	0	1
Create Thread from pool	5	1	6
Schedule Thread	0	0	0
Free Thread	5	1	1
Create lazy thread	0	1	5
Create lazy thread (no check)	0	0	1
lazy return	0	0	1
Eager-disconnect(n-word frame)	2n+6	12	30
Create lazy thread after lazy-disconnect	5	1	6

Table 4.4: *Times for the primitive operations using stacklets. These times do not include any control costs. They assume stacklets are allocated from a pool of stacks and exclude the cost of a malloc.*

4.6.5 Timings

Stacklets, like spaghetti stacks, offer a compromise between linked frames and multiple stacks. As seen in the timings in Table 4.4, sequential call and `lfork` behave similarly and require no memory references. However, when a stacklet overflows, or a new thread is created, we must create a new stacklet from the pool which costs slightly more than creating a new linked frame. Like multiple stacks, eager-disconnect is expensive.

We reject the use of guard pages to detect static overflow because the cost of taking an overflow fault is 60,000 cycles. The use of guard pages reduces the cost of sequential call and `lfork` by one branch and four register instructions. Thus, for the guard page scheme to pay off an overflow would have to be rarer than once every 12,000 calls.

4.6.6 Discussion

The main disadvantage of stacklets is that if a frame is at the top of a stacklet, each of its children must be allocated on a new stacklet. This boundary case is similar to the overflow case when using register windows. We can reduce the probability that this occurs

Operation	Instruction Count		
	Memory	Branch	Register
C call	0	0	1
C return	0	0	1
Sequential call	0	0	1
Sequential return	0	0	1
Create Thread (stacklet)	5	1	6
Schedule Thread	0	0	0
Free Thread (stacklet)	5	1	1
Create lazy thread (stacklet)	0	1	5
Create lazy thread (stacklet - no check)	0	0	1
lazy return (stacklet)	0	0	1
Create lazy thread after lazy-disconnect	5	1	6
Create thread or lazy thread (heap)	4	1	4
Free thread (heap)	3	0	1
Lazy return (heap)	4	0	2

Table 4.5: *Times for the primitive operations using a sequential stack for purely sequential calls, stacklets for potentially parallel and suspendable sequential calls with moderate sized frames, and the heap for the rest. These times do not include any control costs. They assume stacklets and heap-allocated frames are allocated from a pool and exclude the cost of a malloc.*

by increasing the size of each stacklet. Additionally, the compiler optimization mentioned in Section 4.6.4 eliminates many of the places where this boundary case can occur.

4.7 Mixed Storage Model

In all of the memory models except the multiple stacks model, sequential call and `lfork` cost more than a sequential call on a stack. For this reason we propose a fifth alternative, the mixed storage model, which combines a stack with stacklets and linked frames in a single system. In such a system, activation frames can be stored on the stack, a stacklet, or a heap, depending upon the type of call and the requirements of the child. The compiler chooses the least expensive storage mechanism for each activation frame. The least versatile and least expensive method is to use a stack, which is available only for purely sequential calls, i.e., for calls that can never suspend. The most expensive method is to store each frame in a heap-allocated memory segment. A heap-allocated frame store requires a link from child to parent, which is implicit when using a stack.

Between these two extremes, we use stacklets for frames that are allocated by forks. A stacklet combines the efficiency of a stack with the flexibility of heap-allocated frames for those frames that are or may become independent threads of control. This method allows us to assign to each thread its own logically unbounded stack without having to manage multiple stacks or require garbage collection. Our compilation strategy allows us to use a sequential stack for sequential activation frames, stacklets for small to moderately sized thread frames, and the heap for large thread frames.

4.8 The Memory System

Here we specify the possible memory hierarchies of the concrete implementation of the abstract machine. While we can implement LMAM on either a shared memory machine (SMM) or a distributed memory machine (DMM, either an MPP or a NOW), the underlying memory system will significantly affect the implementation and cost of the different mechanisms described here. There are three main areas of impact: locks, thread migration, and split-phase operations.

Many of the operations described have to be performed atomically. For instance, `lreturn` must remove the parent from the parent queue and return control to the parent atomically. If it does not, a steal operation may steal the parent thread from the parent queue at the same time. On a DMM using Active Messages [62], the steal operation is only initiated when the network is polled for a message. Since we control when the network is polled, we avoid the explicit manipulation of locks. On an SMM one of three approaches must be used: (1) the parent queue must be locked on every `lfork` and `lreturn`, (2) multiple queues maintained, or (3) active messages for work stealing requests simulated [55].[8] With all but the last technique, SMM implementations will introduce significant overhead.

On an SMM, it is easy to migrate a thread, while on a DMM the migration of threads requires sending all of the data in the thread's frame across the network. Since local pointers are not valid on another processor, we require either that all pointers be global or that no pointers be placed in the frame. When seed activation is used to represent nascent threads there is an alternative, for it is easy to "migrate" a nascent thread. A nascent

[8] The only synchronization point is the buffer used to store the steal requests. However, when a steal request is inserted into the buffer the processor inserting the message is by definition idle.

thread has no data associated with it except its arguments.[9] Thus, when a nascent thread is activated, it can be activated either locally or remotely.

Finally, on a DMM, split-phase memory operations are often used to hide the network latency. A split-phase read operation consists of two messages: a request and a response. The request must specify the address of the remote location and the address of the local destination. The response then stores the value into the local destination. If we allow child- or parent-copying disconnection, then the local destination may change between the request and response if the destination is in a frame that is copied due to disconnection. Thus on a DMM, neither child- nor parent-copying can be used unless we forbid thread operations during split-phase operations.

In the remaining chapters, we will focus on distributed memory multiprocessors. For this reason we will look only at systems that do not allow thread migration and perform lazy-disconnect. We will also generally skirt the issue of atomicity since we can easily guarantee it by polling the network controlling when handlers are permitted to execute.

4.9 Summary

In this chapter we have investigated five storage models for implementing thread state. The multiple stacks model is the most expensive for thread operations and the least for lazy thread and sequential operations. However, since we do not use garbage collection it does not provide an infinite logical stack per thread. For this reason we will not study it further.

Of the remaining four models, linked frames are the most expensive for sequential operations and moderately effective for thread operations. The mixed model, stacklets, and spaghetti stacks strike a compromise between sequential and thread operations.

In all cases, the timings we have presented depend heavily on integrating thread operations directly into the compiler. By integrating knowledge of the storage model into the compiler we are able to construct a function prolog and epilog that is tailored to the specific model being used. This is carried even further for the mixed model, where the size of the function's activation frame determines whether it is heap-allocated, stack-allocated, or stacklet-allocated. Furthermore, we discussed how the underlying memory model of the machine affects how atomicity is achieved.

[9] If an argument is a pointer it has to be a global pointer.

In all cases we have ignored the effects that the different thread representations and queuing methods impose on the storage model. We consider these effects in the next chapter.

Chapter 5

Implementing Control

In this chapter, we present the control portion of a concrete implementation of the abstract machinery introduced in Chapter 3 for the support of a lazy thread fork. We begin with the thread seed model and present the different means of invoking threads and representing nascent threads. We then explore the different ways in which nascent threads can be queued and dequeued. This leads us into a discussion of the implementation of disconnection. Before proceeding to discuss the continuation-stealing model, we show how to reduce the cost of synchronization and how to use lazy parent queues.

5.1 Representing Threads

As we saw in Chapter 3, the inclusion of a lazy thread fork requires that threads, or potential threads, be represented in many forms. What would have been a thread in MAM can begin life as a nascent thread, become a lazy thread, and even be transformed (by disconnection) into an independent thread. In this section we describe the concrete representations of nascent threads (as thread seeds and closures) and the implementation of an `lfork` (as a parallel ready sequential call).

5.1.1 Sequential Call

Although a sequential call does not start a new thread, it is the basis upon which we build all the representations for lazy threads. In this chapter, we are concerned mainly with the transfer of control between parent and child. For a sequential call, the child begins executing by saving its return address.[1] Instead of storing the return address in the child frame, the child stores the return address at a fixed offset from the top of the parent stack frame. The key feature we exploit is that the return address is in the same relative location in every parent frame.

The child returns to the parent by doing an indirect jump through the location that contains the return address. We exploit the flexibility of the indirect jump to implement `lfork` as a sequential call.

5.1.2 The Parallel Ready Sequential Call

The parallel ready sequential call implements the abstract `lfork` operation. It starts its child thread, a lazy thread, with a sequential call. The child stores its return address, just as a sequentially called child does, in the parent's frame. The child returns to the parent using the sequential return instruction.

To support disconnection, every return address has associated with it the addresses of two seed routines. These routines, `find(adr, susp)` and `find(adr, steal)` (See Section 3.4), are used to handle child suspension and the receipt of a work stealing request, respectively. There are many alternative implementation strategies for associating the seed routines with the return address: inserting a jump table or addresses between the call and normal return, defining fixed offsets, and a central hash table, to name a few. In all cases, given a return address, the seed routines should be easily computed, and if they are not needed they should not interfere with sequential execution.

For flexibility and ease of implementation, we insert jump instructions into the seed routines between the call instruction and code that is executed when the child returns. Figure 5.1 shows a sample instruction sequence for a parallel ready sequential call. Every thread codeblock begins with an adjustment to the return address just as in the Sparc. This adjusted return address points to the code immediately following the seed routines associated with the `lfork` that invoked it (See the prolog code in Figure 5.1).

The decision between implementations of the `find()` functions should be based on the cost of adjusting the return address, the cost of finding the seed routines, whether branch prediction is used on return, and the behavior of the cache. An ideal implementation allows a seed routine to be found and executed quickly without interfering with sequential execution.

[1] Leaf functions are not required to store their return address, but for the purposes of this discussion we assume they do.

```
...
call child                  // lfork child
jmp steal-fragment
jmp suspend-fragment
...                         // return here
```

A sample lfork instruction

```
retadr = retadr + 2             // function prolog
save retadr in a frame slot
// codeblock for child
...
...
load retadr from frame slot     // function epilog
jmp retadr
```

sample function prolog/epilog

Figure 5.1: *Pseudo-code for a parallel ready sequential call and its associated return entry points.*

As the sequential case occurs more frequently than finding a seed, an implementation should favor non-interference over finding the seed.

Inserting a jump table between the call and return requires the return address to be adjusted on every call and allows the seed routines to be found by doing address arithmetic. This is a particularly good method on architectures like the Sparc, which allow the return address to be adjusted as part of the return instruction, making the adjustment free. However, if the processor predicts return "branches" based on the address of the call instruction, then adjusting the return pointer causes a misprediction on return, increasing the cost of a sequential return. The inserted table also causes the sequential code to be larger, possibly leading to an increased instruction cache miss rate.

An alternate method to implement `find()` is for the compiler/linker to construct a hash table that maps return addresses to appropriate seed routines. This method produces no interference with sequential calls; the return address does not need to be manipulated and the code size is not increased. It does, however, increase the cost of finding seed routines.

A more complicated approach is to place the seed routines at a large fixed offset from the return address. This method does not interfere with the sequential call, and the seed routines can be found simply by using address arithmetic. However, the compiler/linker must interleave unrelated code and seed routines in order not to waste code space.

5.1.3 Seeds

A thread seed represents a nascent thread. It is a pair of pointers, a frame pointer and a code pointer. The addresses of the suspend and steal routines can be derived from the code pointer.

A seed is activated by loading the frame pointer and executing one of the routines associated with the code pointer. The suspend routine creates a new lazy thread. The steal

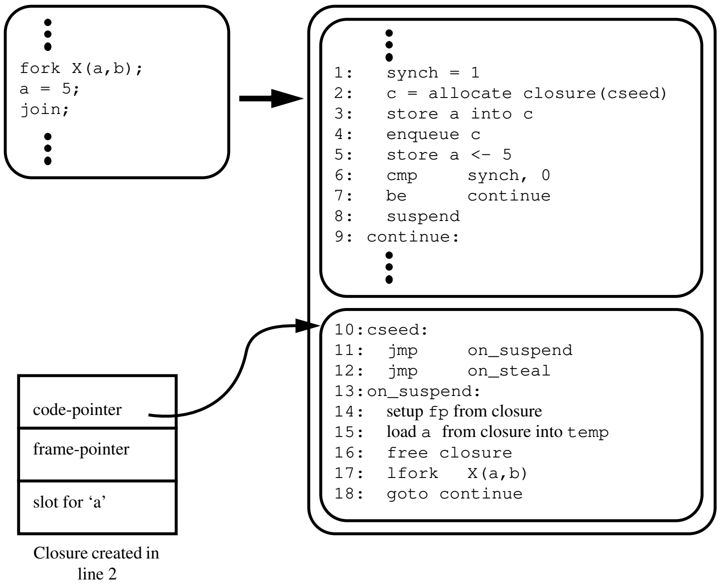


Figure 5.2: *Example of a closure and the code to handle it.*

routine starts a new independent thread and then exits. Both routines update the parent thread state to reflect the seed's activation.

The arguments used to start the nascent thread, if any, are stored in the parent thread frame, i.e., the frame pointed to by the frame pointer in the seed. These arguments cannot be modified between enqueuing and activating the seed. If the program can modify the arguments, a closure is used instead of a seed to represent the thread.

5.1.4 Closures

We use a closure when the arguments to the fork of a nascent thread may change in the parent between the time the fork is represented as a nascent thread and the time that the nascent thread is instantiated (See Figure 5.2). It consists of two pointers, a code pointer and an environment pointer. The code pointer, like the code pointer in a seed, is used to derive the suspend and steal routines which activate the closure.

The environment is a heap-allocated region of memory that contains the arguments, and a pointer to the frame that created the closure. The mutable arguments are placed in the environment. Any arguments guaranteed not to change are left in the frame.

When a closure is activated, the activation routine uses the frame pointer in the environment to change the state of the parent thread. The environment and possibly the frame are used as the source of the arguments to the new thread. The closure is freed when the new thread is created.

5.1.5 Summary

In the seed model we have two ways to represent nascent threads (seeds and closures) and one way to execute a lazy thread (the parallel ready sequential call). It is no coincidence that the return address of a parallel ready sequential call is a code pointer that could be used for a seed. In fact, we have also arranged for the code pointer to be stored in the parent's frame. The result is that the child has a pointer to a location which stores the code pointer. In other words, the child has a thread seed to the remaining work in the parent.

In the next section, we discuss the ways in which work (threads, thread seeds, and closures) is queued, found, and then executed in the system. Since closures are queued and dequeued in the same manner as thread seeds, we only discuss thread seeds. We show how thread seeds and parallel ready sequential calls interact and affect the parent queue and its representations.

5.2 The Ready and Parent Queues

The abstract machine has two queues of threads that are ready to run: the ready queue and the parent queue. The former contains threads in the ready state, which are scheduled by the general scheduler. The latter contains threads that have passed control to a child, i.e., threads in the `has-child` state.

5.2.1 The Ready Queue

In order to bound the amount of memory that the ready queue consumes, we distribute it among the activation frames. Any data structure that allows us to do this would be sufficient, and we choose to represent the ready queue as a linked list exhibiting LIFO behavior. Thus, the ready queue is maintained for each processor by a global register which points to the first ready frame. Each frame in turn points to the next frame in the ready queue.

Our implementation does not depend on the data structure chosen to represent the ready queue. This is important because the data structure can have a significant effect on execution time and storage requirements [9, 17]. The effect of the data structure for the ready queue is orthogonal to the goals of this thesis.

5.2.2 The Parent Queue

Unlike the ready queue, which is involved with independent threads only, i.e., threads that have already suspended and need to be scheduled, the parent queue holds nascent threads. More precisely, `lfork` requires that nascent threads in the parent be represented by enqueuing the parent thread on the parent queue. This means that we have to pay more attention to the cost of enqueuing and dequeuing, or we lose the benefit of lazy threads.

A thread seed can be enqueued on either an implicit queue (See Section 5.2.3) or an explicit queue (See Section 5.2.4). When a thread seed is implicitly enqueued we call it an implicit seed, and when it is explicitly enqueued we call it an explicit seed.

Whether implicit or explicit, the parent queue, is implemented here as a stack. Thus, when an item is enqueued on the parent queue, it is placed at the top of the parent queue. Furthermore, dequeue removes the most recently enqueued item.

5.2.3 The Implicit Queue and Implicit Seeds

Parallel ready sequential call uses the call mechanism itself to implicitly enqueue the parent's seed as it transfers control to the child. The seed is enqueued in the parent frame by the act of calling the child. When an implicit queue is used, we establish the invariant that a child always returns control to its parent, even if it suspends. This allows us to use the return address produced by the parallel ready sequential call as the parent's thread seed, embedding the parent queue in the cactus stack (See Figure 5.3).

The implicit queue is thus woven throughout the activation frames and managed implicitly through the call and return operations, eliminating the cost of managing the queue whenever the child runs to completion. We call a return address enqueued by a parallel ready sequential call an *implicit seed*. When a fork that is translated into a parallel ready sequential call is followed by another fork, the implicit seed created by the first of

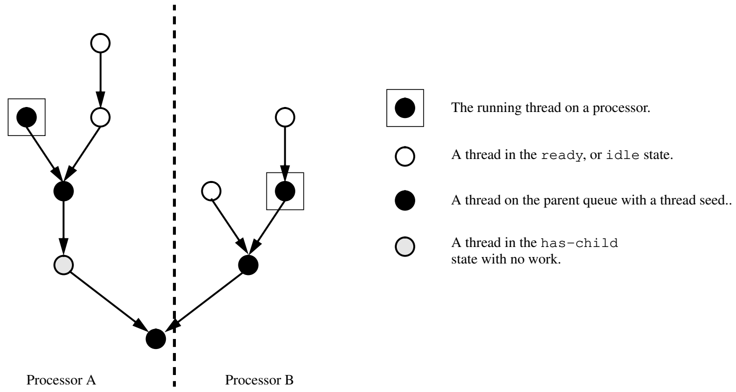


Figure 5.3: *An example of a cactus stack split across two processors with the implicit parent queue embedded in it. The threads on the parent queue are guaranteed to be ancestors of the currently running thread.*

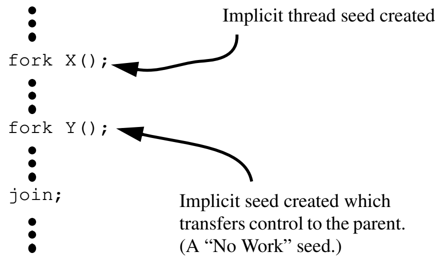


Figure 5.4: *Example of when a fork generates a thread seed, and when it does not.*

these two forks is a thread seed (See Figure 5.4).[2] If the code following the parallel ready sequential call does not contain a fork, then the implicit seed is used only to transfer control from the frame in which it resides to the frame's parent

The return address of the child thread, stored in the parent's frame, is an implicit seed that can be activated by either a suspending child or a work stealing request. On suspension, a child returns control to its parent, not at the return address, but to the seed routine for suspension, i.e. `find(ip, suspend)`. This removes the parent from the implicit parent queue and starts the suspend routine, which activates the seed and starts the next thread. If a work-stealing request occurs, it finds the implicit seed by searching the stack for a thread seed. Only the frames from the running thread to the root of the cactus stack need to be searched. All the other threads must either be `idle` or `ready`, which means they

[2]The parallel call does not have to immediately follow the parallel ready sequential call. It only must precede a join.

cannot have any nascent threads. When it finds an implicit seed that is a thread seed, it invokes the steal routine, found in the seed's frame. Since an implicit seed is stored in the frame that will invoke the nascent child thread, the seed itself is represented by a single word.

There are two drawbacks to an implicit queue, both of which derive from the fact that it contains only seeds created by parallel ready sequential calls. First, an implicit seed may be created even when there is no nascent thread in the parent. For example, the last `lfork` before a join creates an implicit seed that indicates that the parent has no threads available. In this case, the suspend routine causes the parent to suspend, recursively invoking its parent's suspend routine. Likewise, the steal routine for such an implicit seed invokes its parent's steal routine in search of a nascent thread to activate. The entire path from the currently executing child to the root may have to be scanned to verify that there are no nascent threads to invoke. The other disadvantage is that implicit seeds can only be created for nascent threads that follow a parallel ready sequential call. This means that parallel loops must be turned into loops of function calls, and that the sequential code following a lone fork must be encapsulated in a function call. Despite these two disadvantages, implicit seeds are useful because they cost nothing to enqueue, a side effect of invoking the lazy thread that precedes them.

5.2.4 The Explicit Queue and Explicit Seeds

An explicit queue is a data structure separate from the cactus stack that contains thread seeds. A thread seed on such a queue is called an *explicit seed*. An explicit queue solves the two problems of implicit queues by enqueuing the parent thread on an explicit parent queue. When an explicit parent queue exists, a suspending thread or a work stealing request may immediately find work by removing a seed from the parent queue.

Of course, explicit seeds may be enqueued on the parent queue at any time, not just when a parallel ready sequential call is made. This makes them useful not only for avoiding the stack walking needed by implicit seeds, but also for handling parallel loops and the case of a lone fork followed by sequential code. In order to handle parallel loops, we enqueue a seed before the loop starts which points to a routine which will fork off an iteration of the loop. After the seed routine creates the new thread, it replaces the seed in the queue. When the loop is finished, it removes the seed. The lone fork is represented as a

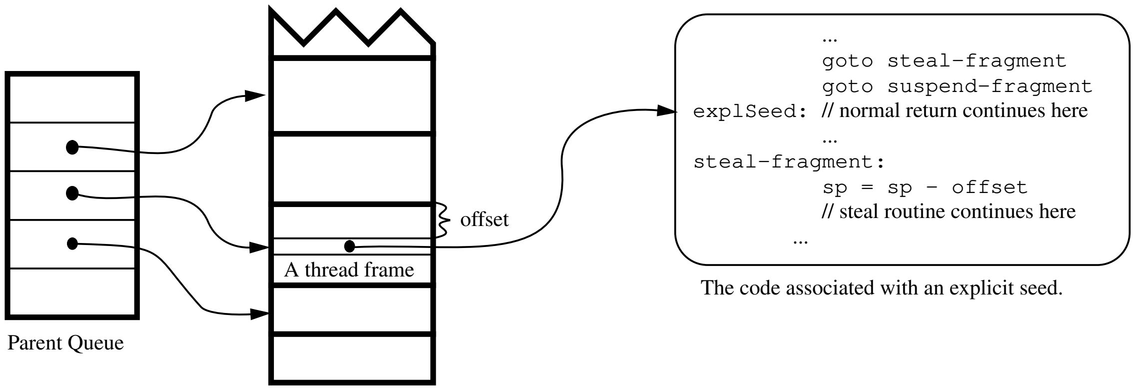


Figure 5.5: *Example of how an explicit seed is stored on the parent queue. The parent queue contains a pointer to a location in the thread's frame. The location contains a code pointer from which the seed routines can be derived. When invoked, the seed routines calculate the thread frame's stack pointer based on the offset, known at compile time, from the location to the frame's top.*

nascent thread and enqueued by the parent before it starts executing the sequential code. If no other processor has activated the seed when the parent reaches the join, it activates the seed and then continue. In this way, we never limit the parallelism in the original program, even though we are using seed activation as the underlying control model.

We implement the explicit queue as a fixed array that is managed by two global registers: front and back pointers.[3] We enqueue new seeds onto the front of the queue. If the seed is enqueued due to a parallel ready sequential call, then when the call returns it will pop the seed off the queue. If instead the child suspends, we also pop the seed off the front of the queue. On the other hand, a steal request takes a seed from the back of the queue. This allows work to be stolen from lower down in the cactus stack, and thus the work is more likely to be larger-grained.

An explicit seed is a single pointer which points to a slot in the frame of the thread that created it. The slot in the frame contains the code pointer from which the seed routines can be derived. For example, if seeds are represented by a jump table (See Figure 5.1), then the the code pointer for an explicit seed points to the instruction after the second jump (See Figure 5.5). The seed routines are invoked with the address of the seed itself, from which it is possible to calculate the base of the frame.

[3]Though this appears to fix the size of the queue, it can be dynamically expanded if needed. In any case, empirical results show a small queue, on the order of 100 words, is sufficient for most programs.

	Implicit Queue	Explicit Queue
Suspend:	`goto (*retadr)-1`	`goto (*popSeed())-1`

Implicit Seed:	`goto steal-routine`
	`goto suspend-routine`
	// continue	

Explicit Seed:		`goto steal-routine`
		`goto suspend-routine`
		`popSeed()`

No Work Seed:	`goto (*retadr)-2`
	`goto (*retadr)-1`
	// continue	

Figure 5.6: *A comparison of the different implementations of suspend depending upon the type of parent queue. Empty entries indicate that no such seed exists for the model. The units of address arithmetic are words.*

5.2.5 Interaction Between the Queue and Suspension

The implementation of the suspend operation depends upon which representation of the parent queue we choose. If an implicit queue is used, then the child always returns to its parent, which then decides which nascent thread to activate. If only the explicit queue is used, then the child immediately transfers control to its youngest ancestor with work, i.e. to the frame pointed to by the thread seed at the front of the queue. Figure 5.6 shows how suspend is implemented and what kinds of seeds are used depending upon the type of queue in the implementation.

When we have an implicit queue in the system, the child always returns to the parent, which means that sometimes the child returns to a thread that has no nascent threads. In this case, the seed left behind by the parallel ready sequential call that invoked the child is like the "No Work Seed" in Figure 5.6. As we can see from the figure, this seed recursively suspends the parent to its parent.

If we have only an explicit queue, then a child can never return to a parent without nascent threads. This is because such a parent would not push a seed on the queue in the first place.

We cannot use both queues in the system at the same time. The reason is that a child would never know if its parent is enqueued implicitly, and thus would always have to return to its parent. Yet if the parent enqueued a seed explicitly, it would not know if it was reached via an explicit dequeue (so that its seed is no longer on the explicit queue) or from a child suspending (with its seed still on the queue). While this could be fixed by setting and testing a flag, a more basic problem is that there is no explicit handle to the implicit queue. If for some reason a thread is scheduled which is not the child of the top parent on the implicit queue, then we lose the link to the implicit queue, possibly resulting in deadlock. In Section 5.5 we show how a lazy parent queue can be built which combines the best features of both the explicit and implicit queues.

5.3 Disconnection and Parent-Controlled Return Continuations

Disconnection is a key component of a lazy multithreaded system. It allows a thread to start a child as a lazy thread and later, if necessary, to disconnect from it, turning the child into an independent thread. In this section, we introduce the key mechanism that allows us to perform disconnection, parent-controlled return continuations, and discuss the differences between lazy-disconnect and eager-disconnect.

The address that a parallel ready sequential call gives its child acts like a sequential return address in that it acts as both the inlet address and the continuation address for the parent. When the child suspends, we need to break this bond between inlet and continuation addresses. In terms of the abstract machine, the parent must have a valid `ip` (its continuation), and the indirect inlet address for its child must be updated.

Three facts help us in this task. First, disconnection occurs only when the parent is resumed (by suspension or by a work stealing request). Second, the return instruction is an indirect jump, which we perform through a location in the parent frame. Third, as discussed in Section 3.4, every parallel ready sequential call has associated with it two inlets: one for normal return and one for return after disconnection.

When the parent disconnects from its child, it must change the indirect inlet address for the child. It does this by changing the return address for the child to its post-disconnection inlet. Since the parent controls the return continuation for the child, we call this mechanism *parent-controlled return continuations* (PCRCs). Later, when the child completes, it will run the inlet that reflects the fact that it has been disconnected from its parent. We are not done yet, since when the inlet returns, we must then continue execution in the parent at its current (at the time of return) `ip`.

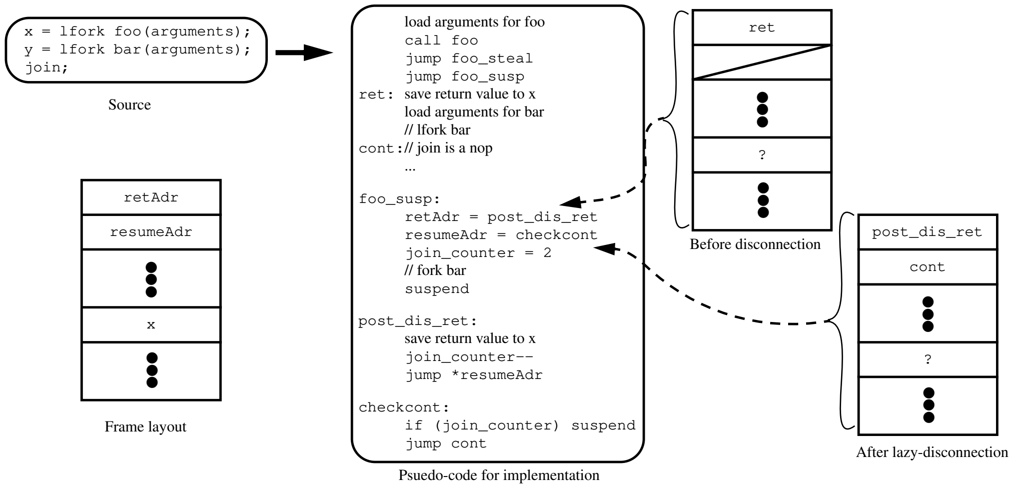


Figure 5.7: *Example implementation of two* `lfork`*s followed by a join. It shows the state of the thread frame before and after disconnection. It also shows the frames upon execution of the code up to the dashed arrows.*

We call the `ip` at which we continue the *resume address*. The resume address is stored in a predetermined field in the parent's frame. It is used only when a parent has been disconnected from its child. Thus, when disconnection occurs two things happen. First, the parent changes the PCRC to reflect the new inlet to be used by the child. Second, the parent stores a continuation that reflects the new state of the parent in its resume address field. The resume address now points to the code following the seed it is activating.

In other words, some action caused the parent to disconnect itself from its child. This action causes the parent to instantiate the nascent thread in the parent. When the child finally does return to the parent, we do not want it executing the `lfork` of the instantiated nascent thread. Instead it begins execution at the resume address.

Figure 5.7 shows an implementation of the `lfork` operation with its associated PCRC manipulation instructions under lazy-disconnect. If the child created by the `lfork foo()` runs to completion, it returns to `ret`, stores the return value to `x`, and then forks `bar()`. Before disconnection, no synchronization operations are performed. Also, the return address is both the inlet run (storing the return value `x`) and the continuation address (`ip` in the abstract machine).

After disconnection (in this case the code shown is for the child suspending), we see that the inlet used by the child has been changed to `post_dis_ret`, explicit synchronization is performed, and the `resumeAdr` field in the frame is set to `cont`. In other words, the

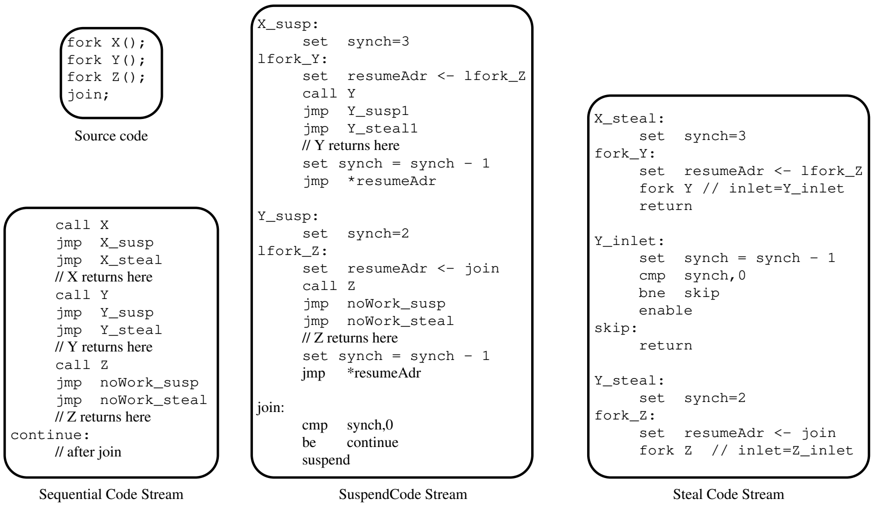


Figure 5.8: *Each fork-set is compiled into three code streams. The code fragments shown here do not include indirect inlet address manipulation.*

return address is now simply an indirect inlet address and the `ip` in the abstract semantics is now represented by the `resumeAdr` field.

A by-product of lazy-disconnect is that after disconnection, the parent thread continues to execute all the remaining `lfork`s before the next join differently than if no disconnection had occurred. This is because after disconnection, any progress in the parent has to be reflected in its resume address. Thus, the code for the second fork to the matching join in a series of forks has to be duplicated.

5.3.1 Compilation Strategy

The compilation strategy is to create three different code streams for each fork-set (See Figure 5.8): sequential, suspend, and steal. The sequential code-stream handles the case when all the parallel ready sequential calls run to completion. In this stream, each `lfork` is implemented as a call followed by the appropriate thread seed. The join operation is of course not implemented since no synchronization is needed if none of the calls is ever disconnected from the parent.

The suspend code-stream is entered when one of the original lazy threads suspends. In this stream, each `lfork` is also implemented as a call followed by a thread seed. Unlike the sequential stream, synchronization is performed explicitly, and after the lazy thread returns we find the next location to execute by jumping through the `resumeAdr` field. When this stream is entered, the suspending call will setup the synchronization counter and also modify its indirect inlet address to point to an explicit inlet (as in Figure 5.7). The join is explicitly implemented in this stream.

The steal code-stream is entered when a steal request arrives during execution of one of the parent's children. The steal routine is executed, which forks off the next child. This causes all future `lfork`s to be executed in the suspend stream. Since this stream is only entered via a steal request, there is no provision for forked children to return and execute anything other than their inlet. The future children are started when the currently outstanding child finishes and jumps through the `resumeAdr` field.

A slight optimization involves checking the number of outstanding children before activating a seed. If all the children have completed, then it continues execution back in the original sequence of code. If there are outstanding children, then it activates the seed with the post-disconnection code.

5.3.2 Lazy-disconnect

Lazy-disconnect is mainly a way to avoid thread frame copying when threads are disconnected from their parents. However, it also has an effect on the control model, since the frame slot in a child's parent that holds its return address is never moved.

If the parent has been disconnected, its return address field contains the inlet for the first child from which it was disconnected. Any further children that are invoked must have their own return address fields. Thus each seed activation is assigned a unique field in the frame to store the return address. Even though this field is never changed, we use it so that the child may return with a return instruction.[4] This eliminates the need for a child to determine how it was invoked, making concrete implementation of the abstract machine simpler than the rewrite rules would seem to indicate. The inlets for such children adjust the stack pointer in the same manner as an explicit seed routine (See Figure 5.5).

5.3.3 Eager-disconnect

When eager-disconnect is used, we copy the the disconnecting child's return address to a new slot, so that additional children can use the regular return address. This

[4]The children started in the duplicated code stream are invoked lazily, but their inlets always end with a jump through the resume address. Thus we never have to change their return addresses.

means that a parent has to maintain a pointer to its connected child. The parent uses this pointer to change the child's pointer to the return address in the parent slot. Figure 3.38 shows the configuration before and after disconnection using eager-disconnect with the linked-frames storage model.

The result of eager-disconnect is that the return address for a thread, even one invoked in the suspend code-stream, can change locations. Eager-disconnect also increases the cost of the parallel ready sequential call in the linked storage models, since we have to store a pointer from the parent to the child. The pointer is used to update the parent pointer in the child to reflect the new location of the child's return address.

5.4 Synchronizers

In this section we describe a compiler optimization, synchronizer transformation, which eliminates synchronization costs for forks and joins by always combining synchronization with return, even for disconnected threads. While there is no synchronization cost when all the forked children of a parent run to completion, we do incur synchronization costs once any one of them causes disconnection. We can minimize this synchronization cost by extending the use of PCRCs with judicious code duplication and by exploiting the flexibility of the indirect jump in the return instruction.

A *synchronizer* is an inlet which implicitly performs synchronization. The idea is to change the return address of a child to reflect the synchronization state of the parent. In this way, neither extra synchronization variables nor tests are needed. The amount of code duplicated is small since we need to copy only the inlet code fragments. Here we describe the synchronizer transformation for two forks followed by a join.

In the previous section we showed how to encode the inlet address and continuation address in the return address for the parallel ready sequential call. The return address for a child was changed at most once by the parent if the child suspended. Here we extend the encoding to include the synchronization state of the parent. If none of the children suspends, then each will return, and the parent will call the next child until the last child returns and the parent executes the code following the join.

If any child suspends, then not only must the inlet for that child change to reflect a new continuation address, but each subsequent fork must be supplied a return continuation that indicates that there is an outstanding child. Furthermore, when a child that previously

Main program text

```
lfork X
lfork Y
Join
continue after join
```

Inlets which handle return and suspension.

```
    pcrc(X)←inlet_X1
    Call X
inlet_X1.susp:
    pcrc(X)←inlet_X3
    pcrc(Y)←inlet_Y2
    Fork Y
inlet_X1.ret:

    pcrc(Y)←inlet_Y1
    Call Y
inlet_Y1.susp:
    suspend();
inlet_Y1.ret:
done:
    continue after join
```

Auxiliary inlets which handle synchronization.

```
inlet_X2.susp:
    suspend();
inlet_X2.ret:
    goto done;

inlet_X3.susp:
    suspend();
inlet_X3.ret:
    pcrc(Y)←inlet_Y1
    suspend();

inlet_Y2.susp:
    suspend();
inlet_Y2.ret:
    pcrc(X)←inlet_X2
    suspend();
```

Figure 5.9: *An example of applying the synchronizer transformation to two forks followed by a join. Each inlet,* i*, is represented by two labels, a suspension label (*i.susp*) and a return label (*i.ret*). In the pseudocode, we expose the operations that set a child's return address.* pcrc(X) *denotes the return address for the child* X*.*

suspended returns, it must update the return continuations for each of the outstanding children to reflect the new synchronization state. In the worst case this can require $O(n^2)$ stores for n forks. The synchronizer transformation also requires $O(2^n)$ synchronizers for n forks. For this reason we apply the synchronizer transformation only when $n = 2$.

In the case where two forks are followed by a join, the transformation is straightforward and cost-effective. As shown in Figure 5.9, the synchronization transformation generates five inlets for two forks: the first two (in the middle of the figure) are used when no suspension occurs, and the last three (at the right of the figure) are used when suspension does occur.

There are three cases to consider for the fork of X. First, if it returns without suspending, it returns to an inlet (inlet X1) that starts the fork of Y. Second and third, if it has suspended, then when it returns, either the second call will have been completed, or it will still be executing. In the former case its inlet (inlet X2) should continue with the code after the join. In the latter, it should suspend the parent (inlet X3).

There are two cases to consider for the second fork, the fork of Y. Either X has completed when it returns, or X's thread is outstanding. If X has completed, then the inlet (inlet Y1) continues with the code after the join. If X's thread is still outstanding, then the inlet (inlet Y2) changes the PCRC for X to inlet X2 and suspends the parent.

When no suspensions occur, inlets X1 and Y1 are executed, and the runtime cost of synchronization is zero. If the first fork suspends, it starts the second fork with inlet Y2 and sets its own inlet to X3. If X (Y) returns next, it sets the inlet for Y (X) to inlet Y1 (X2), which allows the last thread that returns to execute the code immediately after the join without performing any tests. In this case, the runtime synchronization cost is two stores, which is less than the cost of explicit synchronization. This optimization allows us to use the combined return address of a sequential call, even after disconnection.

5.5 Lazy Parent Queue

Before leaving seed activation and examining how we implement LMAM using continuation stealing, we revisit the representations for the parent queue. We have shown that there are advantages to the explicit seed queue, but it imposes a higher cost on threads that run to completion. Here we show how PCRCs allow us to combine the advantages of an explicit queue with the less expensive implicit queue in the same system.

The basic idea is that every seed starts off as an implicit seed. Then if a work stealing request arrives or a child suspends to a parent with no nascent threads, the system converts the current implicit seed queue into an explicit one. The resulting explicit queue maintains the same ordering of threads as that in the implicit queue, i.e., the oldest ancestor's seed is at the bottom of the queue. After this conversion, the implicit queue is empty, so the possibility of losing a handle to it if we transfer to another thread does not arise. Further, when a thread that was in the current implicit queue suspends, it obtains work from the explicit queue rather than traversing the tree to find work. In addition, a work stealing request can easily obtain work from the bottom of the newly created explicit queue.

Figure 5.10 shows how a seed is converted from an implicit seed into an explicit one. If a work stealing request arrives, it checks the explicit queue. If it is empty, the system invokes the steal entry point for the seed at the top of the implicit queue, which causes the `convert-routine` to start. The convert_routine traverses the implicit queue, converting each thread's seed to its explicit version. When the root of the cactus stack or an already converted thread is encountered, the recursion unwinds, and the newly created explicit seeds are stored in the order they would have been stored if explicit seeds had been used from the start. When all have returned, if the explicit queue is not empty, the steal

	Before Conversion	After Conversion
Suspend:	`goto (*retadr)-1`	`goto (*retadr)-1`

Seed for a nascent thread:	`goto convert_routine`
	`goto suspend_routine`
	`rest:`// continue
		`goto steal_routine`
		`goto suspend_routine`
		`popSeed()`
		`goto rest`

Seed when no nascent thread exists in thread:	`goto convert_routine`
	`goto conv_and_suspend`
	`rest:`// continue
		`return`
		// not used
		`popSeed()`
		`goto rest`

```
convert_routine:
   convert_flag = true
   call parent's convert_routine
   push converted seed
   change pcrc to new seed
   return
```

```
steal_routine:
   if (convert_flag == true) return
   // activate seed
```

```
conv_and_suspend:
   call convert_routine
   goto (*popSeed())-1
```

Figure 5.10: *The table on the left shows how a seed is represented before and after conversion. The routines that aid in conversion are shown on the right.*

entry point is invoked for the seed at the bottom of the queue. Otherwise it reports that there is no work. Likewise, if a child suspends and its parent has no nascent threads, it invokes the `conv-and-suspend` routine, which converts the implicit queue into an explicit one and then jumps to the top seed on the queue.

This entire scheme hinges on the use of PCRCs. Before conversion, the return address points to the preconverted implicit seed. After conversion, the return address of each converted child points to the converted explicit seed.

Once the conversion routine invokes a parent that has already been converted, we terminate the stack walk. Since we never pull seeds off the middle of the implicit queue, we know that once we have reached a seed that has already been converted, we have converted the entire implicit queue. As shown in Figure 5.3, the entire implicit queue is in one branch of the cactus stack.

The lazy parent queue also allows us to plant an explicit seed. The only caveat is that if we plant an explicit seed, we must convert the implicit queue below it into an explicit one.

5.6 Costs in the Seed Model

In this section we present the control related costs of performing the basic operations of thread creation, suspension, etc. in light of the various options presented in this chapter. Our main metric is the extra cost of an `lfork` above that of a traditional sequential call, e.g., a C function call. Since a lazy thread may end up suspending, we also define a secondary cost, the disconnection cost, as the incremental cost of converting a lazy thread call into an independent one rather than creating it as an independent thread ab initio.

There are three parts to the cost of an `lfork`: creation, linkage, and instantiation. Creation cost is the cost of creating the nascent form of the thread. Linkage is the cost of enqueuing the representation so that it can later be found and instantiated. The instantiation cost is the cost of turning the representation into an executing thread. We also compare `lfork` to a sequential call and to an eager fork.

As in our discussion about the storage model, we include the costs of `lfork`, `lreturn`, `suspend`, disconnection, and `lfork` after disconnection. We break down our analysis by the parent queue representation: implicit, explicit, or lazy. We further break down the differences by the kind of disconnection—lazy or eager—used to transform lazy threads into independent ones.

We describe the costs of the operations in terms of m, the cost of a memory instruction, b, the cost of a branch, r, the cost of a register-only instruction, and d, the number of frames traversed before a thread seed is found.

5.6.1 The Implicit Parent Queue

When the parent queue is implicit the creation and linkage costs of an `lfork` are always zero, since the seeds are created and enqueued implicitly, by calling the child. The overhead for an `lreturn` is also zero.

The cost of suspension has two components. First, a seed must be located. Second, the parent must be disconnected from its child. Finding a seed requires d indirect jumps, each of which jumps to the suspend routine and loads the proper frame pointer,[5] which results in a total of $d(m + 2b + r)$ operations. When a work-stealing request arrives, it too must find a seed and perform a disconnection operation, all of which takes exactly the same amount of time as suspension.

Regardless of the disconnection method used, a synchronization counter must be initialized and a return address must be stored for the child, taking $2m + 2r$ steps. During lazy-disconnect, no other steps are performed. In eager-disconnect, we also have to change the child's pointer to the return address, incurring an additional cost of $2m + r$.[6]

[5] Loading the frame pointer is either a register or memory operation, depending on the storage model.
[6] Recall that we are only counting the control portion of the eager-disconnect operation.

Operation	Disconnection Model	
	lazy	eager
`lfork`	0	0
`lreturn`	0	0
`suspend`	$d(m+2b+r)$	$d(m+2b+r)$
disconnection	$2m+2r$	$4m+3r$
`lfork` after disconnect	$6m+3r+b$	$7m+3r+b$

Table 5.1: *The cost of the basic operations using an implicit parent queue. The cost of* `lfork` *also includes the costs of running its inlet on return.* m *is the cost of a memory reference.* b *is the cost of a branch.* r *is the cost of a register operation.* d *is the number of frames checked before work is found.*

`lfork`s after disconnection require setting a resume address and updating a synchronization counter before invoking the child with a parallel ready sequential call, incurring a cost of $3m+2r$. On return, the inlet must also update synchronization and then finish by jumping through the resume address, incurring a cost of $3m+1r+b$. These results are summarized in Table 5.1.

5.6.2 The Explicit Parent Queue

When the explicit parent queue is used, an `lfork` costs more than in the implicit queue because of the linkage cost. Every seed has to be enqueued, which costs $1m+1r$, assuming the head of the seed queue is kept in a register. However, suspend and work stealing can find a thread seed in O(1) time. Dequeuing a seed is a stack pop (and throw away) operation which takes $1r$ instructions. When a lazy thread runs to completion, it has an overhead of $1m+2r$ for the push and pop operations.

The cost of disconnection remains the same as that in the implicit queue model. An `lfork` after disconnection, however, incurs an additional cost of r for enqueuing the seed. The enqueue operation lacks a memory reference because the parent must have its seed on the queue if it is executing an `lfork` after disconnection. Since the seed model requires that all future executions of the parent begin with a jump through the resume address, the seed which is already on the queue is sufficient to represent the remaining nascent threads in the parent. The enqueuing of a seed after disconnection is effectively an "unpop" operation.

The costs of the basic operations using the explicit parent queue are summarized in Table 5.2.

Operation	Disconnection Model	
	lazy	eager
`lfork`	$2m + 2r$	$2m + 2r$
`lreturn`	0	0
`suspend`	$m + r + 2b$	$m + r + 2b$
disconnection	$2m + 3r$	$4m + 3r$
`lfork` after disconnect	$6m + 4r + b$	$7m + 4r + b$

Table 5.2: *The cost of the basic operations using an explicit parent queue.*

Operation	Disconnection Model	
	lazy	eager
`lfork`	0	0
`lreturn`	0	0
`suspend` before convert	$d(5m + 4b + 6r)$	$d(5m + 4b + 6r)$
`suspend` after convert	$m + b + r$	$m + b + r$
disconnection	$2(m + r)$	$4m + 3r$
`lfork` after disconnect	$6m + b + 4r$	$7m + b + 4r$

Table 5.3: *The cost of the basic operations using an lazy parent queue.*

5.6.3 The Lazy Parent Queue

The lazy parent queue has the same linkage costs as the implicit queue model, but on average the same suspension costs as the explicit queue model (See Table 5.3). When a thread suspends, it first invokes the conversion routine which walks down the tree, causing all thread seeds which represent nascent threads to be enqueued on the explicit queue until it reaches the root of the tree or a seed that has already been converted. Then the original thread executes the suspend by doing an explicit dequeue. The convert operation requires $d(5m+4b+6r)$ operations, which includes checking the convert flag, saving a new seed into the converted parent, executing the calls and returns to descend and ascend the tree, and finally saving the new seed on the queue. After disconnection the model behaves exactly like the explicit queue model. This is because a disconnected thread has already been converted to the explicit queue.

```
   call child          // lfork child
   jump steal_routine
   jump suspend_routine
   store results       // return here
   ...
after_join:
```

A parallel ready sequential call with its return inlet.

```
suspend_routine:
   retAdr = post_dis_ret
   join_counter = 1
   disconnect frames
   goto continue_parent
```

The associated suspend_routine. It continues the parent in the duplicated immediately after the call to `child`.

```
post_dis_ret:
   store results
   join_counter--
join:
   if (join_counter) suspend
   jump after_join
```

The associated post-disconnection inlet. Note that it goes directly to the join instead of jumping through a resume address.

Figure 5.11: *Example implementation of a parallel ready sequential call with its associated inlets in the continuation-stealing control model.*

5.7 Continuation Stealing

We now address how to implement the control of the abstract machine LMAM when using continuation stealing instead of seed activation. We use the same basic mechanisms, e.g., a parallel ready sequential call, multiple return entry points, and PCRCs, to implement the continuation-stealing model. The main difference is that the continuation-stealing model lacks representations for nascent threads. Instead, when disconnection occurs, execution continues with the remainder of the parent.

Every series of forks followed by a join is, as in the seed model, duplicated: One series handles the case when no disconnection occurs, and the other handles the forks that occur after disconnection. The only exception to this rule is when a single fork is followed by a large portion of sequential code. In this case, we do not duplicate the code, but explicitly synchronize. The reason for this exception is that we see no advantage to duplicating lengthy sections of code to save only four memory operations and a single test.

5.7.1 Control Disconnection

A parallel ready sequential call, as in the seed activation model, is associated with three return entry points: one for stealing, one for suspension of the child, and one for normal return. If a child suspends, it invokes its parent through the suspension entry point. This causes the PCRC to be changed so that when the child returns, i.e., when it completes, it returns to its post-disconnection inlet. It then disconnects the parent from its child and continues execution in the parent.

```
   call child          // lfork child
   jump steal_routine
   jump suspend_routine
   store results       // return here
   ...
after_join:
```

A parallel ready sequential call with its return inlet.

```
suspend_routine:
   retAdr = post_dis_ret
   join_counter = 1
   migrated = false
   disentangle frames
   goto continue_parent
```

The associated suspend_routine. It continues the parent in the duplicated immediately after the call to `child`.

```
post_dis_ret:
   if (migrated) {
      send results to new location
      suspend }
   store results
   join_counter--
join:
   if (join_counter) suspend
   jump after_join
```

The associated post-disentanglement inlet. It goes directly to the join instead of jumping through a resume address.

```
steal_routine:
   migrated = true
   disentangle frames
   send frame to newlocation
   return
```

The associated steal-routine. It sets the migrated flag and then ships the frame to its new processor.

```
recv_result:
   store results
   retAdr = forward2parent
   enable
   return
```

When the frame completes, it returns its results to this inlet.; which places the frame on the ready queue.

```
forward2parent:
   load results
   return
```

When the frame is scheduled it runs this routine, which returns the results to the thread's parent.

Figure 5.12: *A parallel ready sequential call with its associated steal and suspend routines, helper inlets, and routines for handling thread migration.*

Unlike the seed model, however, continuation stealing does not require us to store a resume address in the parent's frame. With continuation stealing, the parent continues executing after disconnection, so the continuation address for the parent is actually found in the processor's program counter. Instead of resuming the parent at a resume address, the post-disconnection inlet jumps directly to the join. If there are children outstanding, the join fails and the parent suspends, continuing execution in its parent (See Figure 5.11). Note that even if the parent of a suspended child has no nascent thread, it begins execution as a running thread.

5.7.2 Migration

In the seed model once a thread has been started on a processor it remains on that processor. Only nascent threads are migrated between processors. Thus a parent always knows if its children are remote and can customize its return inlet to handle a remote communication. Since there are no nascent threads in the continuation-stealing model, independent threads must migrate among processors to satisfy a work-stealing request. Aside from the storage related costs, the main effect of this is that child threads may migrate without the intervention or knowledge of the parent.

When a thread migrates to another processor, the frame that used to hold this thread remains behind. It is used to forward results from its outstanding children to the migrated thread and to forward the result received from the migrated frame to its parent. To implement this forwarding function we use a combination of PCRCs and a flag.

When a thread is migrated to another processor it may have outstanding children on the current processor. These children must have their results forwarded to the migrated frame on the new processor. The code fragments `suspend-routine` and `post-dis-ret` in Figure 5.12 show how this is handled. `suspend-routine` sets the `migrated` flag to false, indicating that no migration has taken place. The `steal-routine` sets the migrated flag to true and then migrates the frame. When the child returns, it uses the `post-dis-ret` inlet. This inlet first checks the `migrated` flag. If it is true, it sends the data to the new frame. Otherwise, it behaves normally. When the thread itself completes, it sends a message with the data back to its original frame, which causes `recv-results` to execute. `recv-results` stores the results and enqueues the frame on the ready queue. When the frame is run, it returns to its parent as if it were completing normally. Thus a parent does not have to notify its children when it migrates.

5.8 Integrating the Control and Storage Models

We have now described all the control mechanisms used to implement LMAM. In this section we relate these with the storage models presented in the previous chapter.

5.8.1 Linked Frames

Since the linked-frames mapping uses only explicit links, there are no special needs imposed by this mapping on the control model.

5.8.2 Stacks and Stacklets

Both stacks and stacklets have implicit and explicit links between activation frames, which require us to use stubs to construct the global cactus stack. There are several stub routines required by both control models. The two most important stubs assist in returning control and data from the child at the bottom of the frame area to its parent. When the child is local, it uses the `local-thread` stub, which simply finds the parent frame and

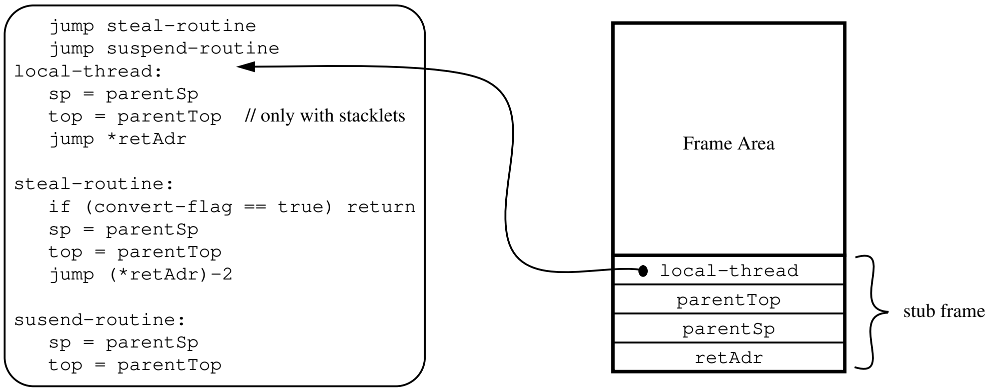


Figure 5.13: *A* `local-thread` *stub that can be used for either stacks or stacklets.*

invokes the proper inlet in the parent. When the child is on a different processor from its parent, it uses the `remote-thread` stub, which sends the data to the parent's processor.

The `local-thread` stub's return address is similar to the implicit seed planted by a parent with no nascent threads in it. In other words, it returns control to the proper return entry point in the parent depending on what return entry point the child uses to enter the stub (See Figure 5.13).

A `remote-thread` stub's return address is similar to a converted implicit seed planted by a parent with no nascent threads. Since the stub has no local parent, it transfers control to the general scheduler if its child suspends. Unlike the `local-thread` stub, the `remote-thread` stub is a custom routine created by the compiler for each codeblock. This is because it must both marshal the arguments from the parent and send the results back using active messages customized for the routine in question.

Both kinds of stubs indicate that the thread at the bottom of the frame area is an independent thread. Thus if we are using a lazy parent queue, the conversion routine will not proceed past the stub.

5.8.3 Spaghetti Stacks

A spaghetti stack presents special implementation challenges because of the need to limit the cost of reclaiming space on the stack. Since we do not have a garbage collector, we must reclaim the space as part of executing the program. Here we show how PCRCs can

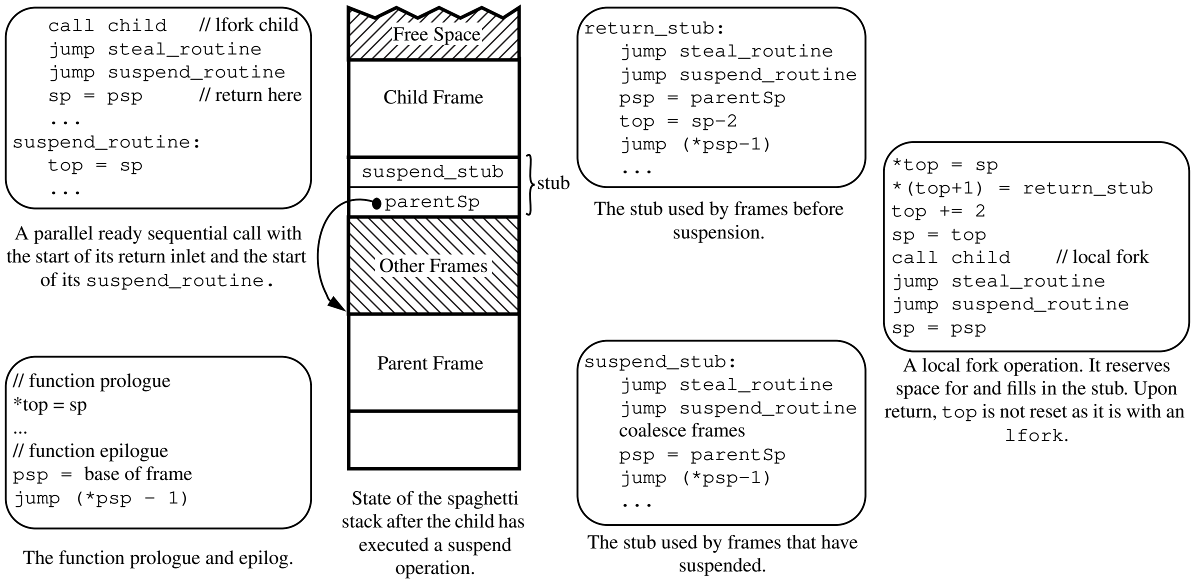


Figure 5.14: *Code fragments used to implement a minimal overhead spaghetti stack. On the left is the invocation routine for* `lfork` *and the function prolog and epilog. The stack in the middle shows the result of a parent forking a child. The routines to support suspend and subsequent* `fork`*s are the right.*

lower the reclamation cost to that of a stack when children run to completion, and we show how to keep the cost low when they suspend.

The scheduling policy of LMAM on a distributed memory machine implies that when a child runs to completion, it will be at the top of the spaghetti stack. Furthermore, a parallel ready sequential call is always made from a thread that is at the top of the stack. Thus there is no need to store a link between a child and its parent if the child is invoked by a parallel ready sequential call. Figure 5.14 shows code for a parallel ready sequential call that takes advantage of this invariant. For children that run to completion, the only additional overhead is the use of a temporary register, `psp`, which is needed to calculate the size of the frame by the coalesce routines. (The size of a frame is (`sp` − `psp`) at the time of return.)

When a thread suspends, `top` must be set, since `sp` no longer points to the top of the spaghetti stack. This setting of `top` occurs in the parent's `suspend_routine`, since we know, because of PCRCs, that the suspend routine is executed only on the first suspension of a child. Once a child has suspended, it can no longer reclaim the space for its frame on exit. However, since the function epilogue does not touch `top`, it will not reclaim the space in any case. Instead, reclamation is handled either by the post-disconnection inlet for the child, or upon the return of the frame above it (which will have a stub since it must have been started with a `fork`.)

A child invoked by a `fork` needs a link to its parent. This is provided in a stub, which is created by the parent as part of the `fork` operation (See routine at the right in Figure 5.14). The stub for such a routine is initially set to `return-stub`, which reclaims the space for the child upon exit. If, however, the child suspends, it changes the stub routine to `suspend-stub`, which coalesces the frames and frees them if they are at the top of the stack.

In short, by relying on the invariants created by our representation of the abstract parent queue, and the implementation of the `lfork`, `fork`, and `suspend` operations, we are able to reduce the cost of using a spaghetti stack to within one register move more than that of a normal stack for parallel ready sequential call.

5.9 Discussion

This chapter has introduced the mechanisms and compilation strategy needed to implement the control necessary to realize the lazy multithreaded abstract machine presented in Chapter 3. The four main mechanisms are the parallel ready sequential call, the PCRC, the realization of the `find` function, and the realization of the parent queue. Additionally we presented a method of eliminating synchronization costs with synchronizers.

The importance of these mechanisms combined with the compilation strategy of creating three code streams for each fork-set is the convergence of the implementations independent of the policy decisions (i.e., thread seeds vs. continuations, eager-disconnect vs. lazy-disconnect, implicit vs. explicit vs. lazy queue). The three code streams combined with the above mechanisms allow for the elimination of all synchronization costs whether or not thread seeds or continuations are used to represent the remaining work in a parent of a lazy thread. Furthermore, all bookkeeping operations can be avoided when lazy threads run to completion if an implicit or lazy queue is used to queue thread seeds or continuations. In fact, we find that the policy decisions have less impact then would appear to be the case from previous literature because we have used the appropriate implementation techniques.

Chapter 6

Programming Languages

In this chapter we describe two very different parallel programming languages for which we have implemented compilers that use the lazy threads model described in this dissertation: Split-C+threads and Id90. Split-C+threads, developed by the author, is an explicit parallel language based on Split-C [15], itself based on the imperative language C. Id90 [5] is an implicitly parallel functional programming language based on the dataflow model of computation.

6.1 Split-C+threads

Split-C+threads is a multithreaded extension to Split-C. Before introducing the thread extensions in Split-C+threads we briefly describe the features of Split-C that affect our extension and the compilation of Split-C+threads.

Split-C is a parallel extension of C intended for high performance programming on distributed memory multiprocessors. Split-C, like C, has a clear cost model that allows the program to be optimized in the presence of the communication overhead, latency, and bandwidth inherent on distributed memory multiprocessors. It follows a SPMD model, starting a program on each of the processors at the same point in a common code image.

Split-C provides a global address space and allows any processor to access an object anywhere in that space, but each processor owns a portion of the global address space. The portion of the global address space that a processor owns is its local region. Each processor's local region contains the processor's entire stack and a portion of the global heap.

Two pointers types are provided, reflecting the cost difference between local and global accesses. *Global pointers* reference the entire address space, while standard pointers reference only the portion owned by the accessing processor.

Split-C includes a split-phase assignment operator, `:=`. Split-phase assignment allows computation and communication to overlap. The := operator initiates the assignment, but does not wait for it to complete. The `sync` statement waits for all outstanding split-phase assignments to complete. Thus the request to get a value from a location (or put a value into a location) is separated from the completion of the operation.

The last attribute of Split-C that we are concerned with is the signaling store, denoted by `:-`. Store updates a global location, but does not provide any acknowledgement of its completion to the issuing processor. Completion of a collection of such stores is detected globally using `all_store_sync`, executed by all processors, or locally by `store_sync`.

Split-C+threads extends Split-C by allowing a programmer to move between the SPMD model and a multithreaded programming model. When a Split-C+threads program starts, it begins in the SPMD mode of Split-C. However the programmer may at any time initiate multithreading with the `all_start_threads` construct. When all of the threads have terminated, the program does an implicit barrier among all the processors and again re-enters the SPMD model of Split-C.

We begin our specification of Split-C+threads with a simple example and then proceed to describe the syntax and semantics of the new operators and how they interact with the Split-C global address space memory model.

6.2 Example Split-C+threads Program

In this section we present an example Split-C+threads program for naively computing the n^{th} Fibonacci number (See Figure 6.1). This example shows how to start a multithreaded segment in a Split-C program, how to declare a function that can become an independent thread, and how to invoke these threads in parallel.

The program starts execution with the invocation of `splitcthreads_main()` (on Line 19) which is the main entry point for all Split-C+threads programs. All processors assign `x` the value of the first argument. Then on Line 26, they all reach the `all_start_threads` call, which indicates the beginning of a multithreading portion of the program. There is

```
1    #include <threads.h>
2
3    forkable int  fib(int n)
4    {
5        int i;
6        int j;
7
8        if (n <= 2)
9            return 1;
10
11       forkset {
12           pcall i = fib(n-1);
13           pcall j = fib(n-2);
14       }
15       return i+j;
16   }
17
18   int
19   splitcthreads_main(int argc, char** argv)
20   {
21       int x;
22       int r;
23
24       x = atoi(argv[1]);
25
26       all_start_threads(0) r = fib(x);
27       onproc(0)
28           printf("fib of %d = %d", x, r);
29       return 0;
30   }
```

Figure 6.1: *An implementation of the Fibonacci function in Split-C+threads.*

an implicit barrier before and after the `all_start_threads` statement. The argument, 0, to the `all_start_threads` statement causes only processor 0 to start executing the `fib` call.

Instead of executing the call in the `all_start_threads` statement, the rest of the processors enter the main thread-scheduling loop, which schedules ready threads. If there are no ready threads, then it attempts to "steal" a thread from the parent queue of another processor. The default behavior is to attempt to get work from a random processor.

Processor 0 begins by executing a call to the function `fib` on Line 3. The function is given the type modifier `forkable` to indicate that it can become an independent thread. If `n` is greater than 2, then the function in turn creates two new logical threads with the `pcall` statements. The fork-set statement introduces a synchronization block which can contain `pcall` statements. A `pcall` statement indicates that the call on the right hand side can be executed in parallel. Execution will not continue out of the `forkset` block until all

all_start_threads	suspend
fixed	Thread
forkable	ThreadId
forkset	yield
pcall	

Figure 6.2: *The new keywords defined in Split-C+threads.*

$$
\begin{array}{rcll}
\text{stmt} & \rightarrow & \textbf{forkset}\ \text{stmt} & (6.1)\\
& | & \textbf{pcall}\ \text{pcall-expression}\ ; & (6.2)\\
& | & \textbf{pcall}\ (\ \textbf{fixed}\ )\ \text{pcall-expression}\ ; & (6.3)\\
& | & \text{pcall-expression}\ ; & (6.4)\\
& | & \textbf{fork}\ (\ \text{maybe-proc-expr}\ )\ \text{pcall-expression}\ ; & (6.5)\\
& | & \textbf{all\_start\_threads}\ (\ \text{maybe-proc-expr}\ )\ \text{pcall-expression}\ ; & (6.6)\\
& | & \textbf{suspend}\ ; & (6.7)\\
& | & \textbf{yield}\ ; & (6.8)\\
\text{pcall-expression} & \rightarrow & \text{lvalue assign-op function-name}\ (\ \text{arg-list}\ )\ ; & (6.9)\\
& | & \text{function-name}\ (\ \text{arg-list}\ )\ ; & (6.10)\\
\text{maybe-proc-expr} & \rightarrow & \text{expr} & (6.11)\\
& | & \epsilon & (6.12)\\
\text{expr} & \rightarrow & \textbf{ThreadId} & (6.13)\\
\text{type-qualifier} & \rightarrow & \textbf{forkable} & (6.14)\\
\text{type} & \rightarrow & \textbf{Thread} & (6.15)\\
& & & (6.16)
\end{array}
$$

Figure 6.3: *New syntax for the threaded extensions in Split-C+threads.*

the `pcall`s in the fork-set complete. In this case there are two `pcall`s in the fork-set. After the fork-set completes, i.e., the two threads are joined with the parent, the thread returns a value and terminates.

```
forkset {
  pcall i=fib(n-1);
  pcall j=fib(n-2);
}
```

(A)

```
forkset {
  for (i=0; i<n; i++)
    pcall sum+=sum( args);
}
```

(B)

```
forkset {
  pcall x = one();
  if (flag) pcall two();
  baz();
  pcall z = three();
}
```

(C)

Figure 6.4: *Example uses of* `forkset`.

When the original call to `fib` made in the `all_start_threads` statement completes, then all the processors are signaled that the multithreading section has completed and a barrier will be performed after which they will all continue, SPMD style, with the next statement. Note that the value of the call is only returned on processor 0.

6.3 Thread Extensions to Split-C

The main difference between Split-C and Split-C+threads is that Split-C+threads has an explicit parallel call, or fork operation, `pcall`. Scheduling capabilities are provided by suspend and yield statements and some library calls. In this section we describe the basic syntax and semantics of these statements. The new keywords defined in Split-C+threads are in Figure 6.2. The new syntax introduced in Split-C+threads is in Figure 6.3.

6.3.1 Fork-set statement

The fork-set statement introduces a statement block that can contain any statement, including `pcall`s and `fork`s, all of which must complete before control passes out of the block. The fork-set statement can only appear in a function that has been declared `forkable` (See Section 6.3.7). The statements are executed in the order in which they appear in the program text.

Figure 6.4 shows some example uses of `forkset`. In Figure 6.4A, two threads created by `pcall` are joined with their parent before control continues past the fork-set statement. In Figure 6.4B, n threads are created, one at a time by each iteration of the for statement. The enclosing `forkset` joins all of the n threads with the parent thread. In Figure 6.4C, first a thread is started to execute `one()`, then the `if` statement is executed, followed by a sequential call to `baz`. When `baz` completes the last `pcall` will execute. The fork-set joins all of the threads created before continuing in the parent.

6.3.2 Pcall Statement

The pcall statement initiates a potentially parallel call as a lazy thread. Every `pcall` must be in the scope of a fork-set statement. The function called must be declared `forkable` (See Section 6.3.7).

There are three variants of the pcall statement. The `pcall` in Rule 6.2 of Figure 6.3 creates a lazy thread which may be instantiated locally or stolen by another processor. This variant of `pcall` is a concrete representation of an `lfork`.

In Rule 6.3, the keyword `fixed` is used to indicate that the lazy thread started cannot be stolen by another processor. The rationale behind a `pcall` which creates a lazy thread that cannot be stolen is so programmers can initiate threads which have arguments which are pointers to local data in threads. Although the thread cannot be stolen, it can still suspend or help to hide latency.

Rule 6.4 describes a sequential call to a forkable function. This call, although run sequentially, can suspend. If a thread invoked with the last rule suspends, it also causes its parent to suspend.

6.3.3 Fork statement

The `fork` statement creates an independent thread. The programmer may specify the processor on which the thread is created or leave it to the runtime system to place the thread. In all cases the new thread is created eagerly.

`fork` can be used to create downward threads (by placing it in a `forkset`), upward threads, or daemon threads. In the latter two cases, the programmer has to construct her own synchronization mechanism from the Split-C+threads primitives. The programmer must be careful to detect the termination of the child. Otherwise `all_start_threads` may complete before the child thread itself terminates (See Section 6.3.4).

6.3.4 Start Statement

When the `all_start_threads` statement (See Rule 6.6) is invoked, an implicit barrier is performed, and then the program begins executing in multithreading mode. Before this, and after control returns to the statement following this statement, the program is in SPMD mode. If a processor is specified by this statement, then only the processor specified receives an active thread; the other processors enter the scheduling loop and attempt to steal work. If no processor is specified, then all the processors invoke the code in the pcall-expression.

The use of `all_start_threads` allows a programmer to construct mixed-mode programs. The program can switch back and forth between SPMD and multithreaded modes of execution. Since `all_start_threads` allows every processor to start a function, some affinity between data and threads can be created by starting functions on local data.

The multithreaded portion of the program started by `all_start_threads` comes to an end when the function it calls completes. Control passes to the next statement in the program after all the processors have completed their work and executed a barrier. If only a single processor was started (i.e., an argument was specified to `all_start_threads`) then it notifies the other processors that each of them should exit its scheduling loop and perform a barrier. If all of the processors were started, then each enters a barrier when the function it invoked finishes.

Since the only synchronization built in is the join mechanism in the fork-set statement, the programmer must include a check that threads started with `fork` complete before returning from the function invoked by `all_start_threads`. In other words, `all_start_threads` does not check to make sure that threads created by `fork` terminate before it completes.

6.3.5 Suspend Statement

The suspend statement (See Rule 6.7) causes the current thread to release the processor and become idle. The next thread run is from the parent queue, or if no threads are in the parent queue, it is a thread from the ready queue. Which ready thread is run on the processor depends on the scheduling policy and the control model of the compiler. The default scheduling policy is for the last thread enqueued on the ready queue to be the next one scheduled. If both the parent and ready queues are empty, the processor will initiate a work stealing request.

6.3.6 Yield Statement

The yield statement (See Rule 6.8) causes the current thread to release the processor and be placed on the ready queue. If there is a nascent thread on the parent queue it is run next. If the parent queue is empty, then a thread from the ready queue is run. If no other threads are available, then the thread executing the yield continues to execute.

6.3.7 Function Types

In order to distinguish between functions that can form threads and those that are guaranteed to run to completion we introduce a new type modifier, `forkable`. It is only used to qualify function types. A function that has type modifier forkable may be used in a pcall expression (See Rule 6.9).

If a function is not forkable, i.e. it is sequential, then it may not use any of the extensions presented here except the start statement (See Rule 6.6). It is a runtime error for a sequential function that has a forkable ancestor on the stack to execute the start statement. A forkable function may not use `barrier` or any expressions involving signaling stores (`:-`).

A forkable function that takes local pointers as arguments may only be invoked by one of the pcall statements in Rule 6.3 or Rule 6.4. This is to ensure that a thread with local pointers does not start on a processor other than the one on which those pointers are valid.

6.3.8 Manipulating Threads

The programmer can also construct her own synchronization primitives. `ThreadId` always evaluates to a pointer to the current thread. It has type `Thread`. The function `enq(Thread x)` will enqueue thread `x` on the ready queue. The current implementation of Split-C+threads does not allow the programmer to declare any variables of type `Thread` when eager-disconnect is in use for the stacklet memory model. The reason for this limitation is that when threads are disconnected, the activation frame of a thread may be moved, which in turn changes the thread's identifier. In other words, `ThreadId` may not remain the same over the life of the thread for eager-disconnect under stacklets.

6.4 Split-Phase Memory Operations and Threads

One of the main advantages of Split-C is that it allows the programmer to hide the latency of remote references with split-phase assignments. Such an assignment is initiated with one operation (e.g, `:=` or `:-`) and concludes with another (e.g., `sync` or `all_store_sync`). When there is at most one thread per processor this works quite well. However, when there is more than one thread per processor we must examine split-phase assignment more closely.

`all_store_sync` must be abandoned in this case.[1] The basis for the operation of these signaling stores is that all the threads in the system are participating in the operation. At the very least it requires the ability for all the threads in the system to execute a barrier. This allows the synchronization counters to be reset. Then all the threads in the system perform some operations, including the store operation, and finally they all wait until all the stores have completed.

With the split-phase assignment operators there are two possible semantics. When the `sync` operation occurs and the outstanding assignments have not yet completed the processor can either spin or suspend. If it spins, then it will wait for the operations to complete before proceeding in the current thread. This closely matches the Split-C semantics, and threads will not interfere with each other. If, on the other hand, the processor suspends, which will allow the latency of the remote operation to be hidden by running another thread, then each thread must have its own synchronization counter. If they did not have unique counters, then an outstanding operation from one thread could cause another to wait even though its operation had completed. In practice the latency is low enough that this only occurs if the second thread initiates a split-phase assignment that is local to the processor on which it executes.

We choose to have a processor suspend if a `sync` is executed when there are outstanding stores. Additionally we choose to have a unique counter per thread for split-phase operations.

[1] `store_synch` can still be used.

6.5 Other Examples

Here we present two additional examples of Split-C+threads programs. The first shows how lazy threads can affect the style of programming. The second points out a deficiency in Split-C+threads.

6.5.1 Lazy Thread Style

Lazy Threads reduce the overhead of the thread call, but do not in and of themselves eliminate parallel overhead even if each thread runs to completion. We define *parallel overhead* as the overhead introduced into an algorithm which enables it to run in parallel. As we shall now see there are ways to reduce parallel overhead.

In this example we show how to implement the nqueens program without parallel overhead. The nqueens program finds all the placements of n queens on an nxn chess board. It recursively places a queen in a row and then forks off a thread to evaluate the placements in the remaining portion of the chess board. The key function in a sequential version is below:

```
1       int
2       try_row (int* path, int n, int i)
3       {
4         int j, sum;
5
6         if (i>=n) return 1;              // one successful placement
7
8         sum = 0;
9         for (j=0; j<n; j++) {
10          if (safe (path, i, j)) {      // can we put a queen in square i,j?
11            path[i] = j;
12            sum += try_row (path, n, i+1);
13          }
14        }
15        return sum;
16      }
```

This loop tries to place a queen in every row `j` for the column `i` that this thread is responsible for. `path[i]` is the row number of the queen in column `i`. `safe(path, i, j)` checks to see if a queen may be placed in square `i,j` given the previous placements in `path`.

This code cannot be turned into a correct parallel program simply by making the call in Line 12 into a `pcall`. If that were the only change, then different threads would overwrite inconsistent values into `path`. Each thread needs its own logical copy of `path`.

Furthermore, we would also need to make `path` a global pointer so that the `try_row` thread could be stolen by another processor.

The correct, but costly approach, is to make a copy of `path` before every `pcall`. We also make `path` a global pointer and if `path` points to a remote processor, then we use `bulk_get` to copy the data from the remote processor onto the processor running the function. However this is intolerably inefficient, particularly since we know that most of the `pcall`s will run to completion on the processor that invoked them.

Instead, at the start of the function we test to see if `MYPROC`, the processor in which the thread is running, is the same as the parent's processor. If they are the same, no copy is made. If they are not we need to get the data from the parent anyway, so we make a new copy.

```
forkable int
try_row (int* global path, int n, int i)
{
  int j, sum;
  int* lpath;

  if (i>=n) return 1;

  if (toproc (path) != MYPROC) {     // stolen thread–copy path
    int* new_path = (int*) malloc (n * sizeof (path[0]));
    bulk_get (new_path, path, i * sizeof (path[0]));
    lpath = new_path;
  } else
    lpath = (int*)path;

  sum = 0;
  forkset {
    for (j=0; j<n; j++) {
      if (safe (lpath, i, j)) {
        lpath[i] = j;
        pcall (1) sum += try_row ((int* global)lpath, n, i+1);
      }
    }
  }
  return sum;
}
```

We have reduced the parallel overhead by copying the shared data only when an independent thread is created by a remote fork operation. However, this example depends on the fact that neither `try_row` nor `safe` ever suspend. If either suspend, then an independent thread will be started on the same processor as its parent and the copy will never be performed. In fact, a thread can suspend when a child is stolen and does not return by the time the join is reached; the thread with the outstanding child will suspend to its parent. Since threads can suspend the code above will not work unless the compiler inserts a copy operation in the suspend routine.

The general strategy is for any disconnection to cause a copy of the local data to be made. We call this strategy copy-on-fork. In the future, we plan to explore language extensions and compiler analysis that will make this transformation automatic, even in the case of suspension.

6.5.2 I-Structures and Strands

I-structures, first introduced in Id90, are write-once data structures useful in a variety of parallel programs [6]. They provide synchronization between the producer and the consumer on an element by element basis. The two operations defined on I-structures are ifetch and istore, which respectively get and store a value to an istructure element. If an ifetch attempts to get the value of an empty element, the ifetch defers until an istore stores a value into the element.

We can implement I-structures using the suspension mechanism in Split-C+threads as shown in Figure 6.5. When an ifetch attempts to read the value of an empty element, the ifetch thread is enqueued on the deferred list of that element. Later when an istore stores a value into that element, it enqueues all the threads in the deferred list onto the ready queue. ifetch must be a forkable function since it can suspend. istore is not a forkable function since it cannot suspend and does not perform any `pcall`s. One nice thing about this implementation is that it uses the ifetch activation frame to construct the list of deferred elements.

Although I-structures are easily implemented in Split-C+threads, their full power cannot be exploited. For example in the code fragment below we would like to be able to execute the second set of ifetches even if one of the first two suspends.

```
x = ifetch(A, i);
y1 = ifetch(B1, i);
z1 = x * y1;
y2 = ifetch(B2, i);      // want to continue here
z2 = x * y2;
z3 = y1 * y2;
```

We can arrange for such behavior by invoking each ifetch with a `pcall` and putting the entire sequence in a fork-set.

```
typedef struct deferlist* DeferList;

struct deferlist                // list of readers waiting for a store
{
   Thread t;
   DeferList next;
};

typedef struct                  // the structure of an istructure element
{
   int value;
   int flag;                    // TRUE when full, FALSE when empty
   DeferList deferred;
} Element, *Istructure;

forkable int                    // return the value of i[index]
ifetch(Istructure i, int index)
{
   DeferList d;

   if (i[index].flag) return i[index].value;
    // the element is empty so defer
   d.t = ThreadId;
   d.next = i[index].deferred;
   i[index].deferred = &d;
   suspend;
   return i[index].value;
}

void istore(Istructure i, int index, int value)
{                               // i[index] ← value
   DeferList* d;
   DeferList* dnext;

   if (i[index].flag) abort();
   i[index].value = value;
   i[index].flag = TRUE;
   for (d = i[index].deferred; d != NULL; d=dnext)
   {
       dnext = d->next;
       enq(d->t);
   }
}
```

Figure 6.5: *Implementation of I-structures in Split-C+threads using* `suspend`*. When allocated the elements of an I-structure are initialized have their flags set to FALSE and the deferred field set to NULL.*

```
forkset {
  x = pcall ifetch(A, i);
  y1 = pcall ifetch(B1, i);
  y2 = pcall ifetch(B2, i);
}
z1 = x * y1;
z2 = x * y2;
z3 = y1 * y2;
```

Although this works for the simple example above, it requires that all three fetches complete before we can begin computing, even though we could start computing when any two of x, y1 or y2 were returned. If the function to compute z1, z2, or z3 were more complex, this could be a significant penalty.

A solution to this problem, borrowed from TAM [16], is to allow for logically parallel execution within a function. We extend Split-C+threads to allow a function to have multiple strands. A *strand* is simply a flow of control within a thread that cannot suspend and may depend on other strands.[2] Each strand is identified by a unique integer. Furthermore, a strand may have a condition specified which determines which other strands must have completed before it is allowed to run. The syntax of the strand statement follows.

$$\text{stmt} \rightarrow \textbf{strand}\ (\ \text{strand-id strand-condition}\ )\ \text{stmt} \tag{6.17}$$

$$\text{strand-id} \rightarrow \text{integer-constant} \tag{6.18}$$

$$\mid\ \epsilon \tag{6.19}$$

$$\text{strand-condition} \rightarrow \textbf{when}\ \text{logical-expr} \tag{6.20}$$

$$\mid\ \epsilon \tag{6.21}$$

$$\tag{6.22}$$

If strands were in Split-C+threads we could solve our ifetch problem as follows:

```
strand (1) x = ifetch(A1, i);
strand (2) y1 = ifetch(B1, i);
strand (3) y2 = ifetch(B2, i);
strand (when 1 && 2) z1 = x * y1;
strand (when 1 && 3) z2 = x * y2;
strand (when 2 && 3) z3 = y1 * y3;
```

[2]For those familiar with TAM, a strand is slightly more powerful than a TAM thread because it can branch, but is otherwise the same as a TAM thread.

6.6 Split-C+Threads Compiler

Split-C+threads is compiled using a modified GCC compiler. The compiler is based on one which was already modified in two ways: it could compile Split-C, and it produced abstract syntax trees for the entire function instead of for a single statement [64], making the compilation process much simpler.

Our compiler first analyzes the structure of the `pcall`s to determine what code needs to be duplicated and what optimizations can be applied to each fork-set. The optimizations we currently perform are centered around eliminating synchronization operations. After the AST pass is complete, RTL is generated. We added several new RTL expressions, the three most important being fork, seed, and reload expressions. The fork expression is used to represent the `pcall`s. The seed expression is used to construct the thread seeds that come after a fork. The reload expression indicates which registers need to be loaded when a thread is resumed. We also added new instruction templates in the machine definition files. Finally, the function prolog and epilog code is tailored to each storage model.

The synchronization optimization is aimed at removing the necessity for incrementing a synchronization counter before a `pcall` is made in the suspend or steal code-streams. During construction of the AST, it is recognized that the `pcall`s are consecutive, and thus we do not need to increment a synchronization counter for each `pcall` in the suspend stream. Instead, the code executed on entering the suspend code-stream sets the synchronization counter to the number of threads that remain to be completed, i.e., the number of `pcall`s to be executed plus one for the suspending `pcall`. If the fork-set contains conditional or loop statements, then the synchronization counter is incremented before each `pcall` in the suspend stream.

The most interesting challenge in implementing the compiler was to perform correctly the flow analysis and register allocation in the presence of forks, which can branch to one of three addresses: return, suspend, and work stealing. Our primary goal was to introduce no new operations in the sequential portion of the code. In order to achieve this, we ensure that during data flow analysis the non-sequential code-streams do not influence the sequential code-stream. To guarantee that liveness analysis is correct and minimal for the non-sequential code streams, we construct phantom flow links between basic blocks to represent the potential flow of control.

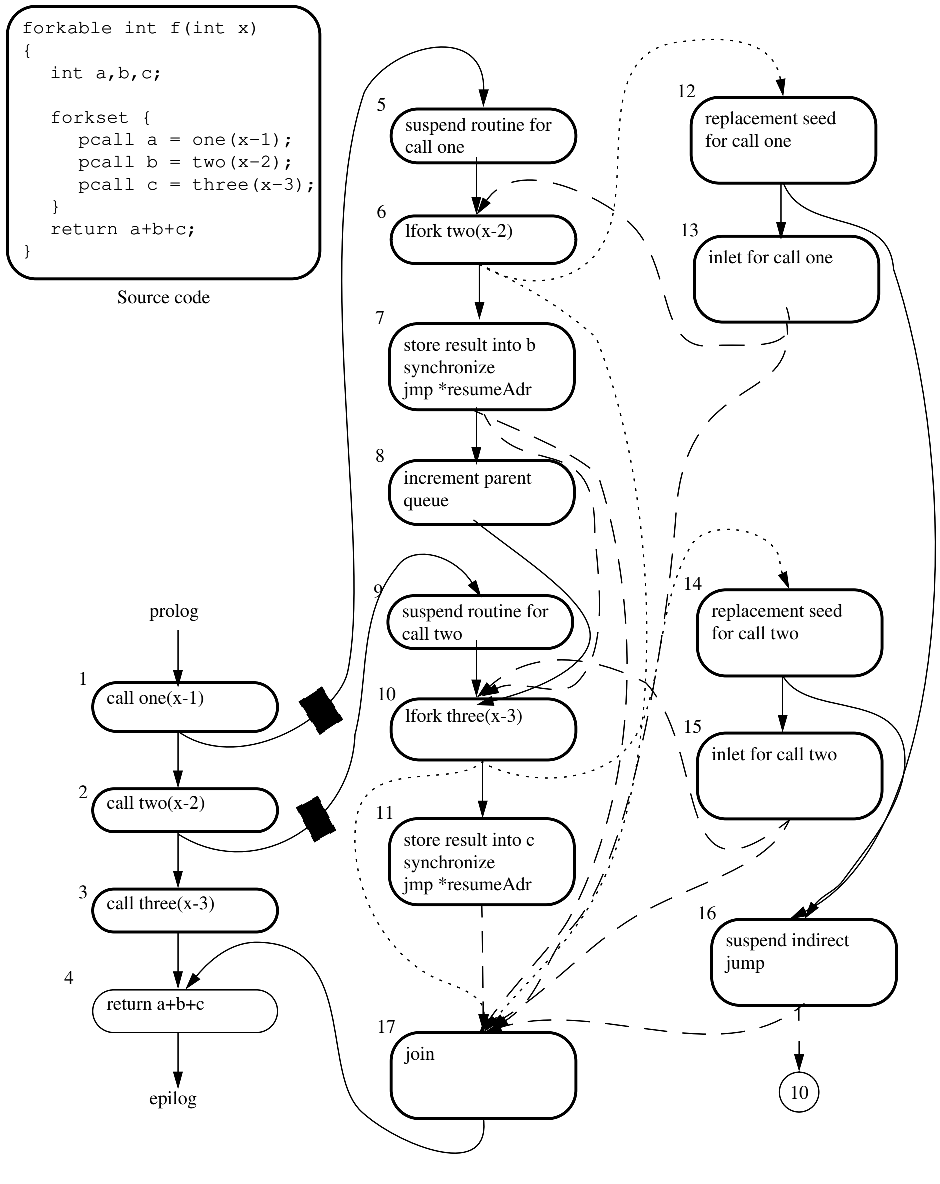


Figure 6.6: *The flow graph for the sequential and suspend code-streams for a forkable function compiled with an explicit parent queue, the stacklet storage model, lazy-disconnect, and thread seeds. The dashed lines indicate phantom control flow links.*

A flavor of the compilation process can be seen by examining Figure 6.6, which shows the control flow graph for a simple function with a fork-set consisting of three consecutive forks. Blocks 1–4 show the sequential code-stream, and Blocks 5–17 show the suspend code-stream. The steal code-stream is omitted so that the example fits on one page, but it follows an analogous structure to the suspend code-stream.

The first thing to note is that the three calls are not, as expected, in the same basic-block. If the calls had truly been sequential, then the control flow links to the suspend code-stream would not exist. However, since each call can potentially transfer control to the suspend code-stream, a `pcall` always ends a basic block. This is an artifact of our desire reuse the infrastructure of the GCC compiler as much as possible. We represent the possible control transfer out of the sequential code-stream by a thread seed RTL expression, which contains a jump table to three addresses: one for sequential return (which points to the following instruction), one for suspension (which points to the suspend routine for the call in question), and one for steal requests (which points to the steal routine).

The most important ramification of the control flow links from the sequential stream to the other streams is that liveness analysis causes all the variables used in the other streams to be marked as live at the beginning of the function. This is because GCC does not see how the variables are set due to the strange control flow out of a call expression. We eliminate this effect by blocking data flow information from the other code streams back to the sequential stream. This works for two reasons. First, we know that once another code stream is entered then the sequential stream is abandoned until the join. The dark rectangles in Figure 6.6 indicate the blocking of the data flow information. Second, we construct "phantom" flow links within the other streams to ensure that the minimum set of live variables required at the join point are correctly maintained.

There are two kinds of phantom flow links. The first, indicated by the dashed lines in Figure 6.6 are used to indicate the set of possible destinations of an indirect jump. The second, indicated by dotted lines, are used to indicate the set of potential destinations that can be reached during execution.

In GCC, if an indirect jump is encountered in the RTL, then every labeled instruction is considered to be a possible destination. This is unnecessarily conservative since we know the possible destinations of each of the indirect jumps (which are always jumps through the resume address). Thus we augmented the RTL expression for jump expressions to include a set of possible destinations. This improves code quality by limiting the number of variables considered live.

The second kind of phantom link is included to increase the number of variables considered to be live. Consider how thread seeds operate. Once a function has entered the sequential or steal code-streams, the destination of a jump through a resume address in an inlet (for example in block 7) can be any of the remaining `pcalls` or the join depending on which of the outstanding `pcalls` have been started (for example block 8 or 17). In order to make sure that the register allocator correctly saves the live variables and restores them on return, we must ensure that the compiler believes that at the time of the lfork (for example block 6) the next possible block to be executed will be any of the following calls or the join (for example blocks 8 or 17). For this reason we add phantom links from the lfork blocks to each of the other lforks and the inlets of the call that created them.

The compiler takes several options to allow us to compile code for the different memory models, the different parent queue implementations, the different work stealing models, and the different disconnection methods. In addition it can produce code for either sequential or parallel machines.

6.7 Id90

Id90 is an implicitly parallel language with synchronizing data structures. For the purposes of this dissertation its three salient features are that it is non-strict, it is executed leniently, and, it has synchronizing data structures. Non-strictness means that a function can be started before it has received all of its arguments. Lenient execution means that a function is evaluated in parallel, with its arguments evaluated as much as data dependencies permit [59]. Synchronizing data structures, like I-structures, require significant overhead. These three features combine to require a mechanism like strands.

Id90 was originally implemented on custom data flow machines [4, 51] and later on conventional processors by compiling to abstract machines like TAM [16] and P-RISC [47].

Our Id90 compiler translates Id90 into TL0, which is the instruction set for a Threaded Abstract Machine (TAM). TL0 is then translated to C with extensions to support lazy threads with strands.

Chapter 7

Empirical Results

One of the goals of this thesis is to compare fairly the ideas behind different multithreaded systems. To achieve this, we evaluate the points in the design space using executables produced by the same compiler and compilation techniques.

We first examine the benefits of integrating thread primitives into the compiler, as opposed to using a thread library. Compiler integration yields at least a twenty-five–fold improvement over thread libraries. We next examine the difference between eager and lazy threads, showing that lazy multithreading is always more efficient than eager threads. The rest of the chapter compares the different points in the design space of lazy multithreading.

We begin our analysis of the design space by showing how the different models perform when running code on a single processor. Even when running on a multiprocessor this is an important case because when the logical parallelism in the program is unnecessary the potentially parallel calls will be executed sequentially. A successful lazy threading system should execute such code without loss of efficiency relative to a sequentially compiled program. We examine every point in the design space: 36 different models composed from three storage models (linked frames, spaghetti stacks, and stacklets), two thread representations (continuations and thread seeds), two disconnection methods (lazy-disconnect and eager-disconnect), and three queue mechanisms (implicit, explicit and lazy). In our comparisons we focus on how the different policy decisions on each axis interact to form a complete system.

Each axis is assessed for its performance on multithreaded code that runs serially and then on multithreaded code that requires parallelism. We then examine the behavior of lazy threads on a distributed memory machine, the Berkeley NOW. Our last experiment looks at how lazy threads improve the performance of Id90 codes. A summary of the effectiveness of these models is shown in Figure 7.25 at the end of the chapter.

With the exception of the comparison between lazy threads and eager threads all performance numbers are comparisons between lazy threads as compiled by Split-C+threads and sequential programs compiled by GCC. Thus, uniprocessor results are reported as slowdowns and multiprocessor results as speedups. In the comparison between eager and lazy threads, we report speedups for lazy threads over eager threads.

7.1 Experimental Setup

To evaluate the different approaches to lazy threading we ran several micro-benchmarks compiled by the Split-C+threads compiler for both uni- and multiprocessors. The micro-benchmarks allow us to isolate the behavior of the different components of a lazy threading system. The uniprocessor machines are used to analyze the behavior of the individual primitives. The multiprocessor experiments are used to validate the quality of the complete system in a real parallel environment.

For the uniprocessor experiments we use the Sun UltraSparc Model 170 workstations (167 Mhz, 128 MB memory, 512 KB L2 cache) running Solaris 2.5. For our parallel experiments, we use the Berkeley NOW system [18] running the GAM active message layer [40]. The Berkeley NOW is a cluster of Sun UltraSparc 170 workstations (running Solaris 2.5) with Myricom "Lanai" network interface cards connected by a large Myricom network [11]. The round-trip message time using GAM is $24.6\mu s$. Since GAM does not support interrupts, the NIC must be polled to determine if a message has arrived. The cost of a poll operation when there is no message is $.7\mu s$, or the equivalent of 230 Sparc register-based instructions.

7.2 Compiler Integration

This work differs from previous work on multithreading in that thread operations are integrated into the compiler instead of being library calls. This improves performance, as shown in Table 7.1, at the cost of a slight reduction in portability. The fork+join time is the time it takes for the parent to fork and join on a single thread that does nothing but

	C-Threads (μs)	NewThreads (μs)	Split-C+threads(μs)
fork+join	129.0	86.0	1.09
context switch		30.0	1.15

Table 7.1: *The performance of basic thread operations made with library calls (using C-threads and NewThreads packages) versus having them integrated into the Split-C+threads compiler. Times are in microseconds on a 167Mhz UltraSparc.*

return. The context switch time is the time it takes for one thread to suspend and another to begin execution.

Performance improves mainly because every call to a library function in a package must handle the general case, whereas Split-C+threads can customize each call to the context in which it appears. For instance, a context switch in a thread with no other statements does not have any live registers at the time of suspension, so a compiler implementation does not have to save any registers. On the other hand, a library call has no knowledge of the context of a call, so it must store every register at the time of suspension. This effect is amplified on the Sparc architecture, since the entire register window set, not just the registers, must be saved.

When multithreading functionality is integrated into the compiler, every call performs exactly the work required by the operation. This results in a performance increase of at least 25 times over library implementations of multithreading.

7.3 Eager Threading

We now compare a compiler-integrated eager thread implementation to a lazy thread implementation within Split-C+threads. We use the micro-benchmark `grain`, shown in Figure 7.1 and first used by Mohr, Kranz, and Halstead [44]. The `grain` benchmark creates $2^{\texttt{depth}}$ threads. The leaf threads then perform $\texttt{grainsize} * 6 + 8$ instructions. We can vary `grainsize` to detect when the overhead due to multithreading becomes insignificant. The smaller the grain size is, the better the system performs on fine-grained programs.

Figure 7.2 shows the speedup of lazy threading relative to eager threading for `grain` with different grain sizes as the percentage of leaf threads suspending increases. We see that lazy threading is always more efficient than eager threading, even for large grain sizes, and even when all the threads suspend. For small grain sizes (<50 instructions), we

```
#include <threads.h>

forkable int grain(int depth, int grainsize)
{
    int i,j;

    if (n < 1)
    {
        int i;
        int sum = 0;
        for (i=1; i<grainsize; i++)
           sum = sum + 1;
        return sum;
    }

    forkset {
        pcall i = grain(n-1, grainsize);
        pcall j = grain(n-1, grainsize);
    }
    return i+j;
}
```

Figure 7.1: *The* `grain` *micro-benchmark.*

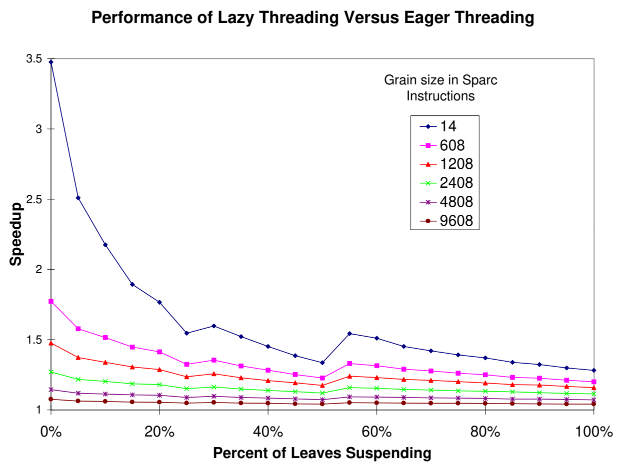


Figure 7.2: *Improvement shown by lazy multithreading (heap storage model, thread activation, lazy-disconnect, and explicit queue) over eager multithreading for the micro-benchmark* `grain` *for different grain sizes (in Sparc instructions) with different percentages of leaf threads suspending and resuming after all leaves have been created.*

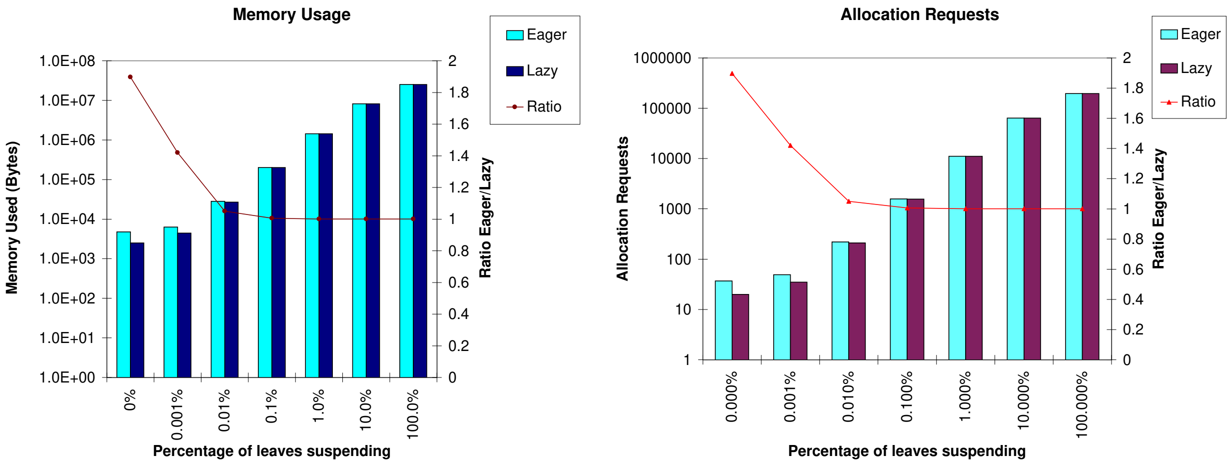


Figure 7.3: *Memory required and allocation requests not satisfied by the pool manager to store thread state for eager and lazy multithreading as a function of the number of leaves suspending.*

see improvements of over 300%. For large grain sizes (>6000 instructions), we see modest improvements of 4–7%. Even when the lazy threads are disconnected from their parents and have to run independently (as is the case when they suspend), we see improvements of between 4% and 28% depending upon the grain size. In no case is eager multithreading faster than lazy threading. The dips in speedup at 25% and 50% are an artifact of how the benchmark chooses threads to suspend. At the 50% point, every parent of a leaf thread suspends because one of its children always suspends. At 25%, every grandparent of a leaf thread suspends.

In this and in all the benchmarks, the ready queue is actually a LIFO stack, so the first thread put on the ready queue is the last scheduled. For this benchmark a leaf suspends after placing itself on the ready queue. Thus, the suspended leaves are not scheduled until the last leaf thread is created.

When parallelism is not needed (and thus none of the leaves suspend), the lazy threading system uses half the memory that the eager system uses, since the children are not created until they are run (See Figure 7.3). However, as soon as the children begin to suspend, we find, as expected, that the memory requirements are essentially the same in the lazy and the eager systems when heap-allocated frames are used.

When this benchmark is run on a single processor, the lazy system never uses more memory than the eager system. However, there are cases when the eager system uses

Grain Size	Register Windows	Flat	slowdown
8	82.7	115.7	1.40
14	95.9	131.9	1.38
68	172.5	192.8	1.12
608	645.6	667.2	1.02
6008	5384.0	5403.5	1.00

Table 7.2: *Comparison of* `grain` *compiled by GCC for register windows and for a flat register model. Grain size is in Sparc instructions. Times are in microseconds.*

less memory than the lazy system, even though both systems schedule threads depth-first. Since the eager threading system always schedules children from the ready queue, suspended frames that are enabled by a remote processor are scheduled before all the lazy children are created. In this benchmark, this would result in the enabled children finishing before all the leaves were created, thus reclaiming the memory they used. In all cases, the lazy threading system creates all the leaves before any of the enabled leaves on the ready queue are run. Thus, under certain conditions, some programs can consume less memory under an eager threading system.

7.4 Comparison to Sequential Code

Perhaps the most important measure of success for any lazy threading system is the speed of a program when all of the potentially parallel calls run serially. If a lazy threading system performs well on this test, then programmers or language implementors can parallelize wherever possible without concern about threading overhead. Further, when all the potentially parallel calls run serially they could have been written as sequential calls, and we can compare the program to a similar program compiled by a sequential compiler. In this section, we compare `grain` compiled with different options by our lazy thread compiler to `grain` compiled with GCC.

7.4.1 Register Windows vs. Flat Register Model

Before comparing programs generated by the lazy threads compiler to those generated by GCC, we need to isolate the effects of register windows on the Sparc. The lazy threads compiler does not use register windows, so we make our comparisons to code produced by GCC for the flat register model (the -mflat option of GCC). As shown in

Grain	Heap Model		Spaghetti Model		Stacklet Model	
8	Seed-Copy-Lazy	1.96	Seed-Link-Lazy	1.20	Cont-Copy-Lazy	1.15
14	Seed-Copy-Lazy	1.81	Seed-Link-Lazy	1.14	Seed-Link-Impl	1.10
68	Cont-Copy-Impl	1.60	Seed-Link-Lazy	1.10	Cont-Copy-Lazy	1.10
608	Seed-Link-Impl	1.17	Seed-Link-Lazy	1.03	Cont-Copy-Lazy	1.03
6008	Seed-Copy-Impl	1.02	Seed-Link-Lazy	1.00	Cont-Copy-Lazy	1.00

Table 7.3: *The minimum slowdowns relative to GCC-compiled executables for the different memory models and where in the design space they occur for a grain size of 8 instructions.*

Table 7.2, the code produced for the flat register model is less efficient than the register window model. There are two reasons for this inefficiency: the flat model uses callee save and the prolog/epilog code requires more memory operations. Since callee save is in effect, more register save/restore operations occur in the flat register model. The flat register model also requires an additional two stores and two loads per function invocation than the equivalent code compiled for register windows. In retrospect, we should have optimized the code produced for GCC's flat model before introducing the lazy thread modifications. In the rest of this chapter we compare the lazy thread code to the flat register model.

7.4.2 Overall Performance Comparisons

Figure 7.4 shows the slowdown of each of the models for different grain sizes. As expected, we see little difference between the models at large grain sizes (> 6000 instructions) and large differences at small grain sizes (< 100 instructions). When all the threads run to completion, as in this case, the two biggest factors are the memory model and the queueing method. The linked-frame memory model is significantly slower than the other memory models, running at half their speed for small grain sizes. The other major factor is the use of the explicit queueing method, which increases overhead by 5–50% independently of the other choices in the design space.

7.4.3 Comparing Memory Models

The most important improvement introduced by lazy threads is the use of stack-like structures to store thread state. As Figure 7.5 shows, the overhead of the linked-frames model, when averaged over the other twelve points in the design space, is a factor of two to fives times more than that of the other memory models. This occurs for two reasons: frame

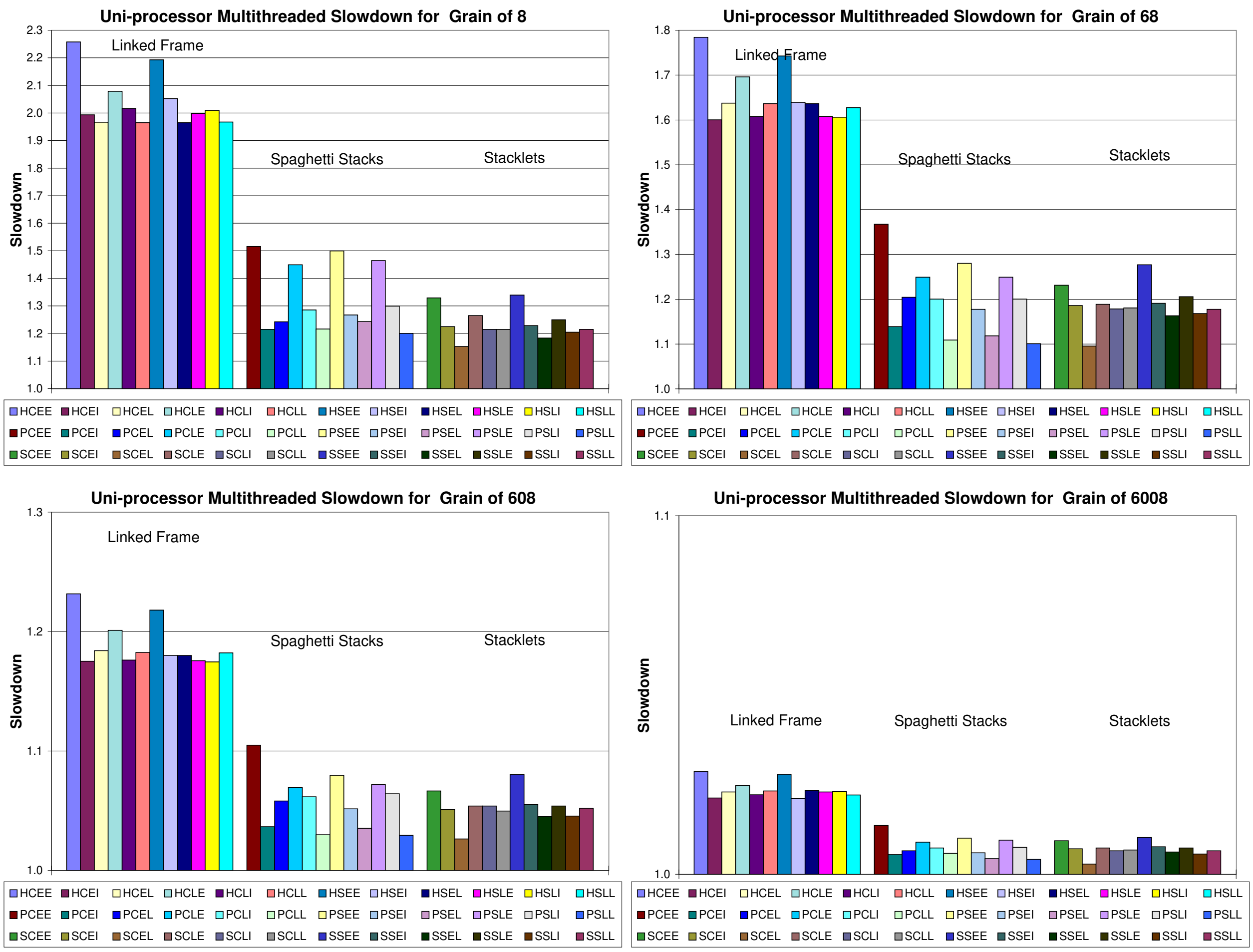


Figure 7.4: *Slowdown of uniprocessor multithreaded code running serially over GCC code. The four letter abbreviations specify the memory model (H–Linked frame, P–Spaghetti, S–Stacklets), the thread representation (C–Continuations, S–Thread Seeds), the disconnection model (E–Eager, L–Lazy), and the queue model (E–Explicit, I–Implicit, and L–Lazy). Note the different scales of the Y axis on each of the four graphs.*

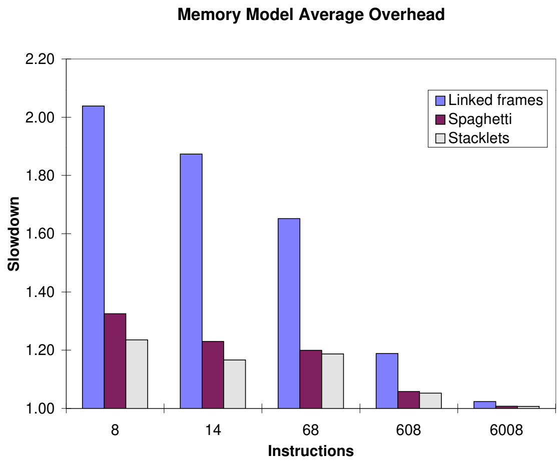


Figure 7.5: *Average slowdown of multithreaded code running sequentially on one processor over GCC code. The slowdown is the average of all points in the design space for each memory model.*

allocation is more expensive, and the child must perform a memory operation in order to find its parent.

The difference between spaghetti stacks and stacklets is less pronounced. Stacklets are slightly more efficient than spaghetti stacks because the overflow check is less expensive than saving and restoring a link between the child and the parent. However, stacklets may be more expensive than spaghetti stacks if many overflows occur. By adjusting the depth argument to the `grain` we can cause an overflow to occur for every leaf. A worst-case scenario is when all the allocations occur at the boundary. A function that recursively calls itself and finally calls another function, named `null()`, in a loop will take 219% longer if overflows occur when `null` is invoked and does nothing but return. In other words, the allocation for an overflow doubles the cost of a function call, which is expected since it is similar to a linked-frame allocation.

To evaluate the effect of the memory model in isolation, we also look at a sequential version of `grain` compiled by the lazy thread compiler. Figure 7.6 shows the slowdown of the multithreaded version of `grain` relative to GCC, of the sequential version of `grain` relative to GCC, and of the multithreaded relative to the sequential version of `grain` when both are compiled by the lazy threads compiler. The last data line shows the overhead of potentially parallel calls versus sequential calls within the lazy threading framework.

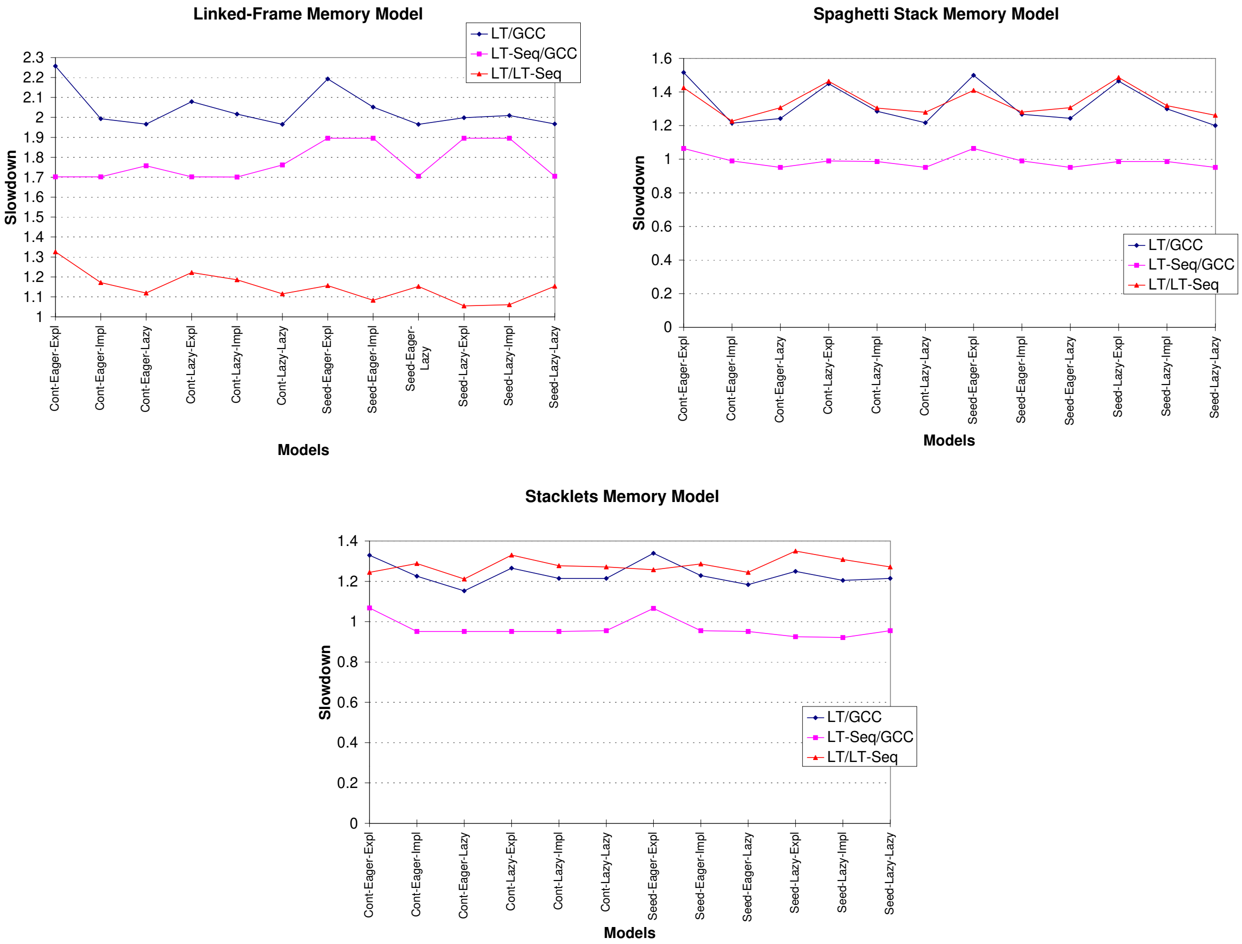


Figure 7.6: *Comparison of the three memory models for a grain size of 8 instructions. LT/GCC shows the slowdown of the multithreaded* `grain` *relative to a sequential version compiled using GCC. LT-SEQ/GCC compares a sequential version of* `grain` *compiled with Split-C+threads relative to GCC. LT/LT-Seq shows the slowdown introduced by turning sequential calls into* `pcalls` *when both are compiled by Split-C+threads.*

As expected, the linked-frame memory model introduces most of its overhead not in handling potentially parallel calls, but just in managing the activation frames. On the other hand, both the spaghetti stack and stacklet models show almost no overhead in the sequential version; all the overhead is introduced by the potentially parallel calls.

The interaction between the different axes of the design space is seen in the extra overhead of the explicit queue models for both the spaghetti stacks and stacklets memory models. In both of these memory models, the stack must be managed specially when a thread suspends. When explicit queueing is used, the parent of a suspending thread may not be notified that its child has suspended. Thus, to prepare for the possible suspension of a child, the compiler must initialize some frame slots on every call, resulting in extra overhead.

Having shown that the linked-frame memory model is always significantly slower than the other memory models, we eliminate it from the rest of the comparisons in this section.

7.4.4 Thread Representations

In this section we investigate the effect of the thread representation on performance. We begin by looking at `grain` running sequentially. We then turn our attention to how the choice of thread representation interacts with the compilation process. This interaction varies with the number of `pcall`s in a fork-set and the data flow in the fork-set. Finally, we show the effect of running threads in parallel on `grain` by varying the number of leaf threads that suspend.

For any choice of memory model, disconnection method, or queueing mechanism, there is no significant difference between thread seeds and continuations when all the potentially parallel calls run to completion. (See Figure 7.7.) This is expected since we use the same techniques to implement both. When the potentially parallel call runs to completion, the continuation and seed are both enqueued in the same way, and the `pcall` is implemented as a sequential call.

When a `pcall` in a fork-set does not run to completion, the different thread representations can perform differently. If threads are represented with continuations, then when a child suspends, the continuation to the rest of the thread is stolen, and when the suspending child returns, it only tests the join to see if it is the last returning child. If it is

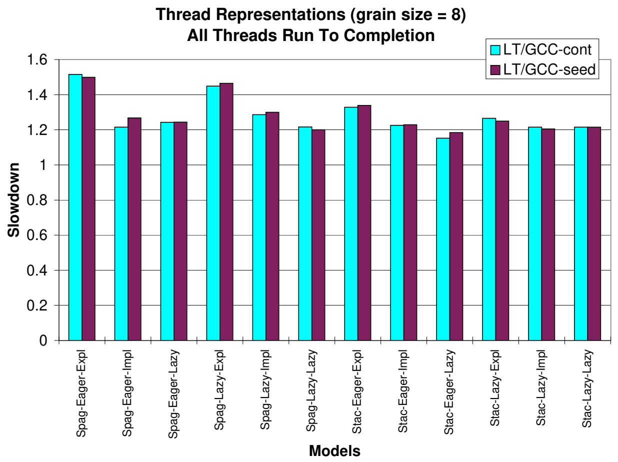


Figure 7.7: *Comparison of continuation stealing and seed activation when all threads run to completion across the design space for* `grain` *with a grain size of 8 instructions. The slowdown is relative to a GCC-compiled sequential* `grain`.

not, the join fails, and the general scheduler is entered. Thus the control flow of the suspend and steal streams is identical to that of the sequential stream. When thread seeds are used to represent nascent threads, children that previously suspended will, when they complete and return to their parents, activate a thread seed if it exists. Thus the control flow in the suspend and steal streams is from every `pcall` to every other `pcall` in the fork-set. The additional control paths present when thread seeds are used require that every inlet end with an indirect jump. Furthermore, every register that is live before all the `pcall`s must be restored before the indirect jump is executed.

To evaluate the difference in control flow we compare three programs with fork-sets which differ in whether or not results are returned by the children threads. Each program consists of a loop that repeatedly calls a routine, `caller`, which consists entirely of one fork-set with `pcall`s to a another routine, `test`. The `test` routine either suspends and later returns or returns immediately, depending on its argument. In the first program, `voidvoid`, both `caller` and `test` are void functions and no values need to be restored or saved in any code stream. In the second program, `void`, `test` is a void function and `caller` returns a value, so one value needs to be restored and saved in both models. In the last program, `return`, both `test` and `caller` return values; thus every `pcall` in the suspend stream to save and restore all the variables in the thread seed model, while in the continuation-stealing model the n^{th} call needs to save and restore n registers. As we see in Figure 7.8, the

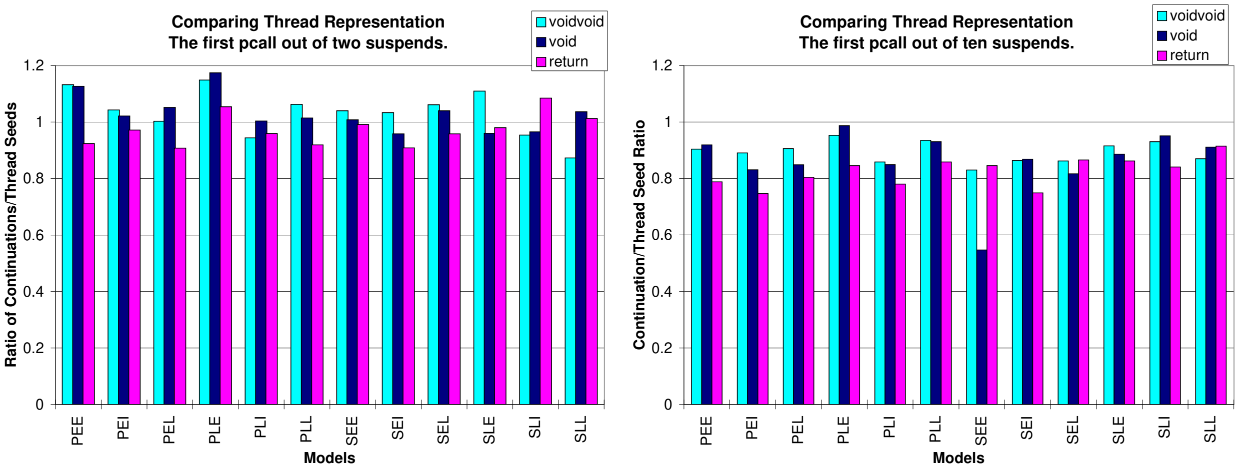


Figure 7.8: *The execution time ratio between the continuation-stealing model and the thread-seed model when the points on the other axes are fixed and the first* `pcall` *of the fork-set suspends. Lower ratios indicate that continuations perform better than thread seeds.*

continuation-stealing model is more efficient for large fork-sets, and for small fork-sets there is no real difference between the two models. This data is consistent with the fact that the thread representation does not significantly affect the other axes of the design space. Interestingly, even though more save/restore operations are performed in the last program (`return`) in the thread seed model, the difference in performance is lower because there is slightly more work per thread.

The Figures 7.7 and 7.8 show the worst case for thread seeds, in that the first `pcall` in the fork-set suspends and all the remaining calls must run in the suspend stream. As Figure 7.9 shows, if a later `pcall` suspends, the difference between the two models disappears.

When we look at the behavior of `grain` in Figure 7.10 we see that the thread representation has little effect on the performance of the program even as the number of threads suspending changes. The small difference is explained by two factors. First, there are only two `pcall`s in the fork-set, so there is no difference in the control flow in the suspend stream. Second, the suspending threads are evenly distributed between the first and last `pcall`s in the fork-set.

As more leaves suspend, continuations perform better than thread seeds by only a small margin. This is particularly true when the memory model has low overhead (e.g,

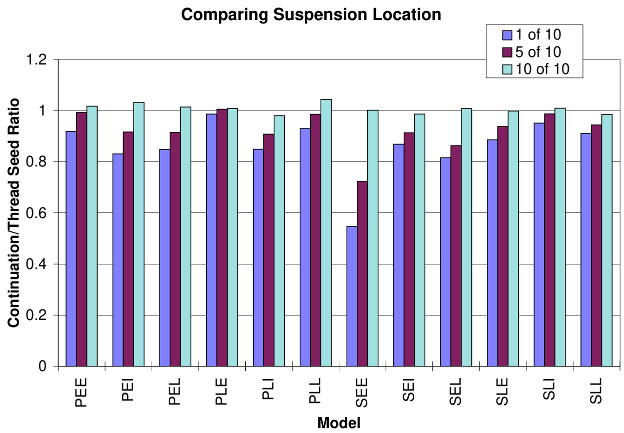


Figure 7.9: *Comparing the effect of which of the ten* `pcalls` *in a fork-set suspends.*

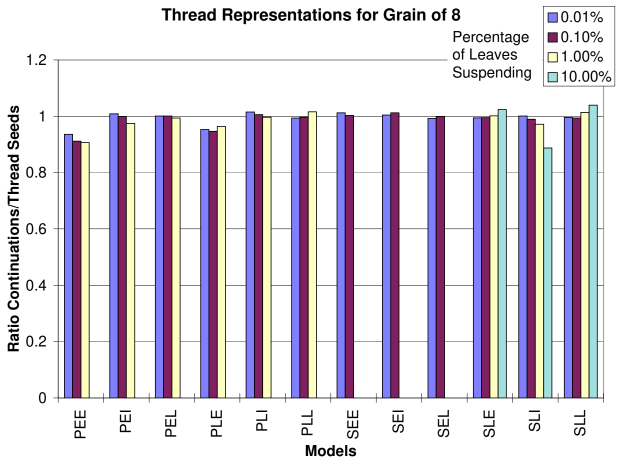


Figure 7.10: *Comparing thread representations for a grain size of 8 as the percentage of leaves that suspend increases. Missing bars indicate that the program was unable to run because it ran out of memory or took more than three orders of magnitude longer than the other programs.*

stacklets with lazy disconnection) and the implicit queue is used. When implicit queueing is used, the suspending thread always returns to its parent. When continuation stealing is in effect, the parent immediately suspends to its parent if it has no work; with thread seeds, in contrast, registers might be restored needlessly, and a jump through the resume address would be performed before the parent suspends.

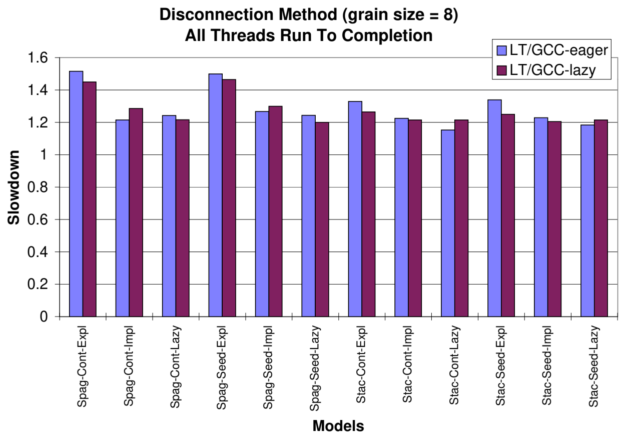


Figure 7.11: *Comparison of eager and lazy disconnection for* `grain` *with a grain size of 8 and all the threads running to completion.*

Continuations and thread seeds perform equally well when all the threads in a fork-set run to completion or, when a thread does suspend, if there are few remaining `pcalls` left in the fork-set. Since the total number of `pcalls` in a fork-set is often very small (in most cases two) both methods of representing threads should work equally well when executing multithreaded code on a uniprocessor.

7.4.5 Disconnection

In this section we compare eager-disconnect and lazy-disconnect. We first look at the cases when all threads run to completion in `grain` and then when threads suspend in `grain`. Next we look at how the number of `pcalls` in a fork-set affects the relative performance of the two disconnection methods.

As we see in Figure 7.11, the different disconnection models have little effect on the sequential performance of potentially parallel calls. When eager-disconnect is used, the parent must be able to modify the child's link back to its parent. In both the linked-frame and stacklets models, this requires an extra store in the function prolog of every child. There is no difference in the spaghetti model because when explicit queueing is used, extra overhead is always required.

The real difference between the methods of disconnection shows up when the potentially parallel calls run in parallel. In this case we see an overwhelming disadvantage

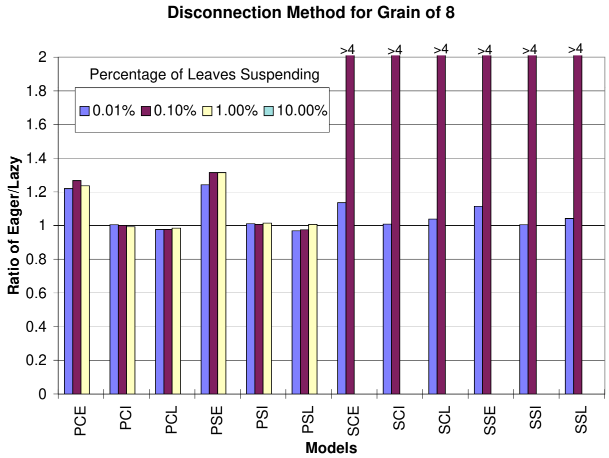


Figure 7.12: *Comparison of eager and lazy disconnection for* `grain` *of 8 as the number of leaves suspending changes. Higher bars indicate lazy-disconnect is better.*

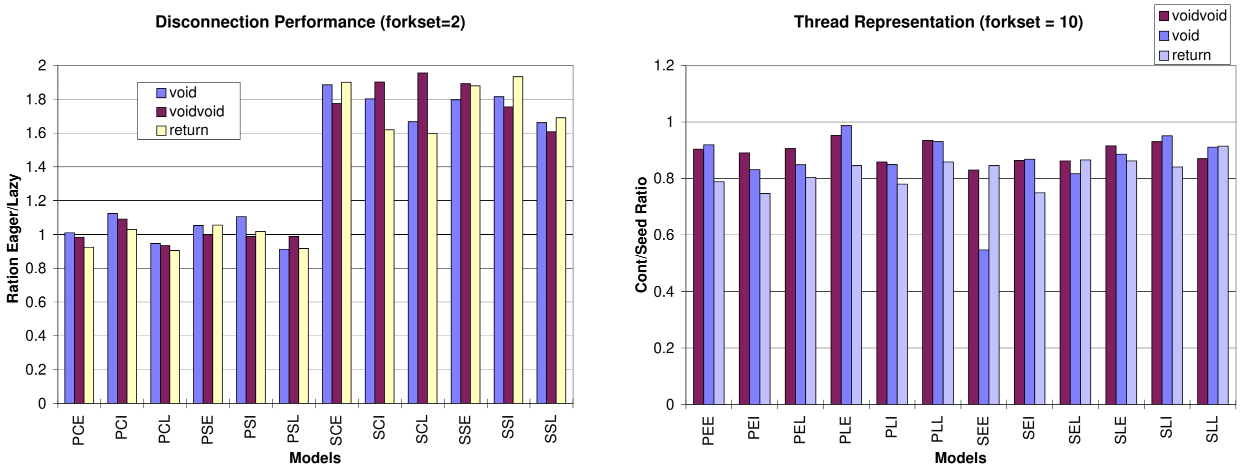


Figure 7.13: *Comparison of eager and lazy disconnection when the first* `pcall` *in a fork-set with two (or ten)* `pcalls` *suspends.*

to the eager-disconnect method for stacklets, since entire activation frames must be copied. Figure 7.12 compares the two methods as the number of leaves suspending changes. Eager-disconnect is significantly worse for the stacklet models even when as few as one out of 1000 threads suspends. When more threads suspend the programs run for so long (hours as opposed to seconds) that we omit the data. The only other significant difference between disconnection methods is when spaghetti stacks are used with the explicit queue. In this case the eager-disconnect method requires extra setup on every call, whether or not it suspends.

As Figure 7.13 shows, when the fork-set has only two `pcall`s the overhead of copying the frames overwhelms the advantage of being able to run the next `pcall` in the same stacklet as the parent. For the other memory models there is no real difference between the two disconnection methods when the fork-set has few `pcall`s.

We conclude that eager-disconnect should not be used with stacklets, or with spaghetti stacks if explicit queueing is used. Further, there is no advantage to eager-disconnect unless the majority of fork-sets have many `pcall`s in them.

7.4.6 Queueing

We now examine the behavior of the different queueing mechanisms when all the threads run to completion and when they suspend. As expected, Figure 7.14 shows that the explicit queue is more expensive than either the lazy or the implicit queue for threads that run to completion. The differences between the implicit and lazy queues is small when the threads run to completion.

When threads suspend, the explicit model becomes more attractive and the implicit and lazy models diverge in performance. Figure 7.15 shows the improvement of the explicit queue model as more threads suspend. It also shows the destructive interaction between spaghetti stacks and the explicit queue. As discussed above, this arises because there is no guarantee that threads are notified when they must reclaim storage, and every activation frame must set and check some variables in the prolog and epilog.

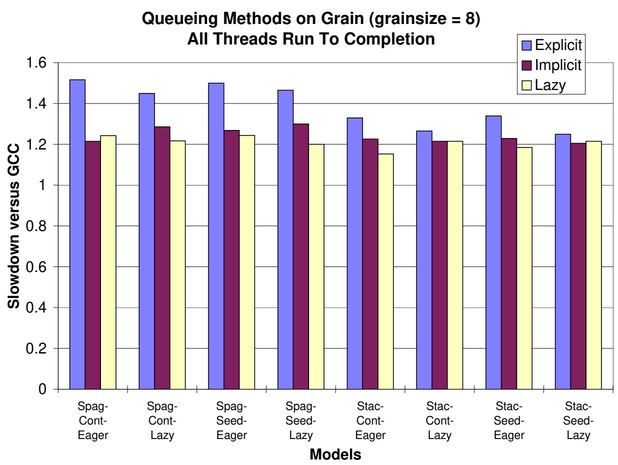
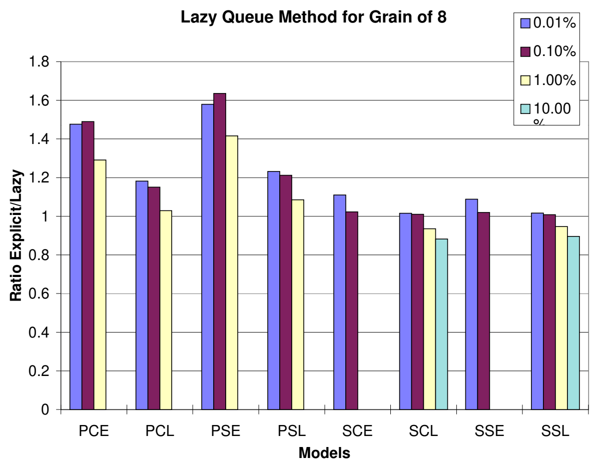


Figure 7.14: *Comparison of the three queueing methods—explicit, implicit, and lazy—for* `grain` *when all threads run to completion.*



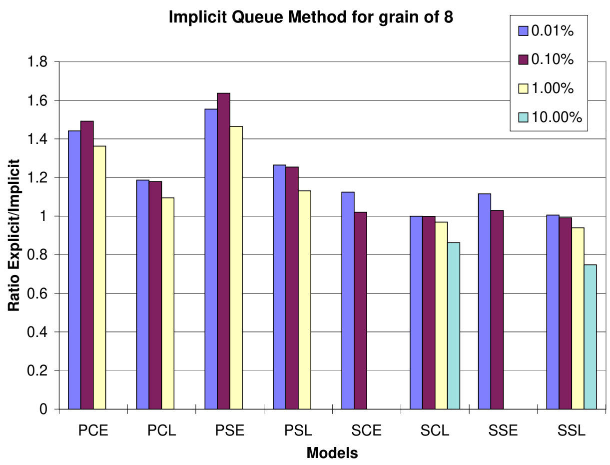


Figure 7.15: *Comparison of the queueing mechanisms as the number of threads suspending increases for* `grain` *with a grain size of 8 instructions. Lower bars indicate that explicit queueing is preferred.*

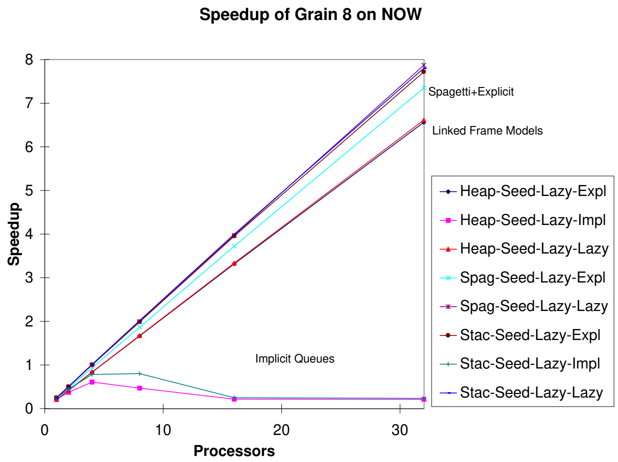


Figure 7.16: *Speedup of* `grain` *(grain size 8 instructions) on the NOW relative to a version compiled with GCC on a single processor for the different models.*

7.5 Running on the NOW

Having analyzed the behavior of the different points in the design space on a uniprocessor we now investigate the performance of lazy thread programs on a distributed memory multiprocessor. We limit our attention to those models that can be reasonably supported on a distributed memory machines. Thus we do not look at continuation stealing or eager-disconnect. Continuation stealing requires migration of instantiated threads, which is costly and very difficult to implement.[1] Eager-disconnect is clearly too costly, even on a single processor, to have any positive benefit on multiprocessors. We look at the implicit queueing method only briefly because it promotes the distribution of the finest-grained work and thus does not scale well on multiprocessors.

Figure 7.16 shows the speedup of `grain` with a grain size of 8 instructions when the network is polled on every function call. Clearly, the work distribution method implemented by the implicit queue is not efficient, as the programs compiled with implicit queue run very poorly. As expected, the linked-frame storage model performs poorly. Independently of the number of processors, it is at least 15% slower than the other storage models. The other

[1] Continuation stealing can be implemented without frame migration, but in this case, when a continuation is stolen the parent must fork off its next child onto the remote processor, just as in seed activation. However, with continuation stealing the parent cannot be continued until its remote child returns, and then it will also have to fork off its next child onto a remote machine.

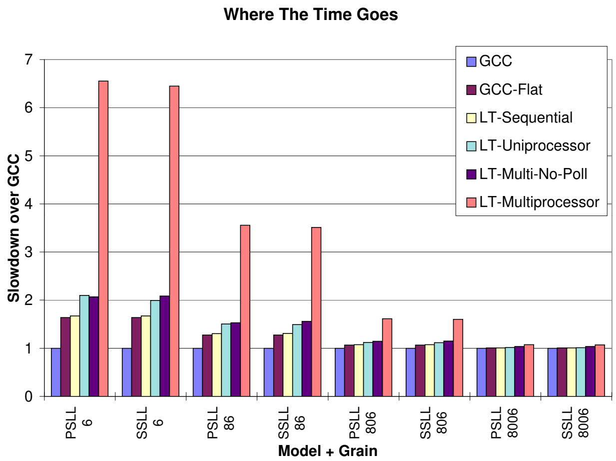


Figure 7.17: *A breakdown of the overhead when running on one processor of NOW. LT-Sequential is the serial version of* `grain`*. LT-Uniprocessor is a multithreaded version of* `grain` *compiled for one processor. LT-Multi-No-Poll is compiled for an MPP, but does not include any network polling. LT-Multiprocessor is an MPP version with polling.*

models all perform well, getting linear speedup relative to themselves. However, the lazy-threads programs run four times slower on a single multiprocessor than on a uniprocessor.

Figure 7.17 shows where the performance is lost. Focusing on the smallest grain size, we pay an immediate penalty of 64% for using the -mflat option of GCC. Ignoring the linked-frames model there is no additional penalty for using Split-C+threads for compiling a sequential version of `grain`. However, turning the function calls into `pcalls` incurs an additional loss of performance of about 50%. Compiling the multithreaded version for a multiprocessor adds no cost, but adding the instructions to poll the network interface more than triples the running time.

If we compare the performance of lazy threads on the NOW with that of the Thinking Machines CM-5 [58] we see how the less expensive poll operation on the CM-5 boosts performance. Figure 7.18 shows the speedup of `grain` compiled by an earlier compiler which used stacklets, thread seeds, lazy-disconnect, and explicit queueing. Even the finest grain sizes have efficiencies of over 50%. In the case of NOW, the more expensive poll operation reduces the overall performance of even the large grain sizes.

When we reduce the number of poll operations, efficiency increases dramatically. If we change the poll operation into a conditional poll which checks the network only every n polls, we increase the efficiency of the program without implicitly increasing the grain

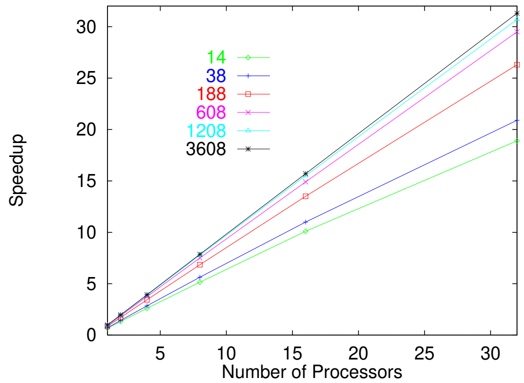


Figure 7.18: *Speedup of lazy threads (stacklets, thread seeds, lazy disconnection, and explicit queue) on the CM-5 compared to the sequential C implementation as a function of granularity.*

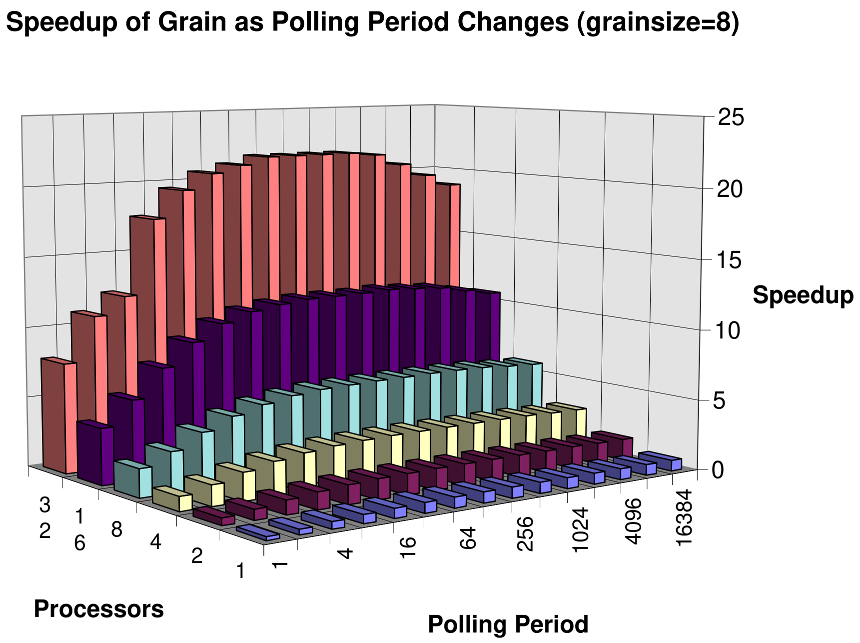


Figure 7.19: *The effect of reducing the polling frequency on* `grain` *with a grain size of 8 instructions using stacklets, thread seeds, lazy-disconnect, and lazy queueing.*

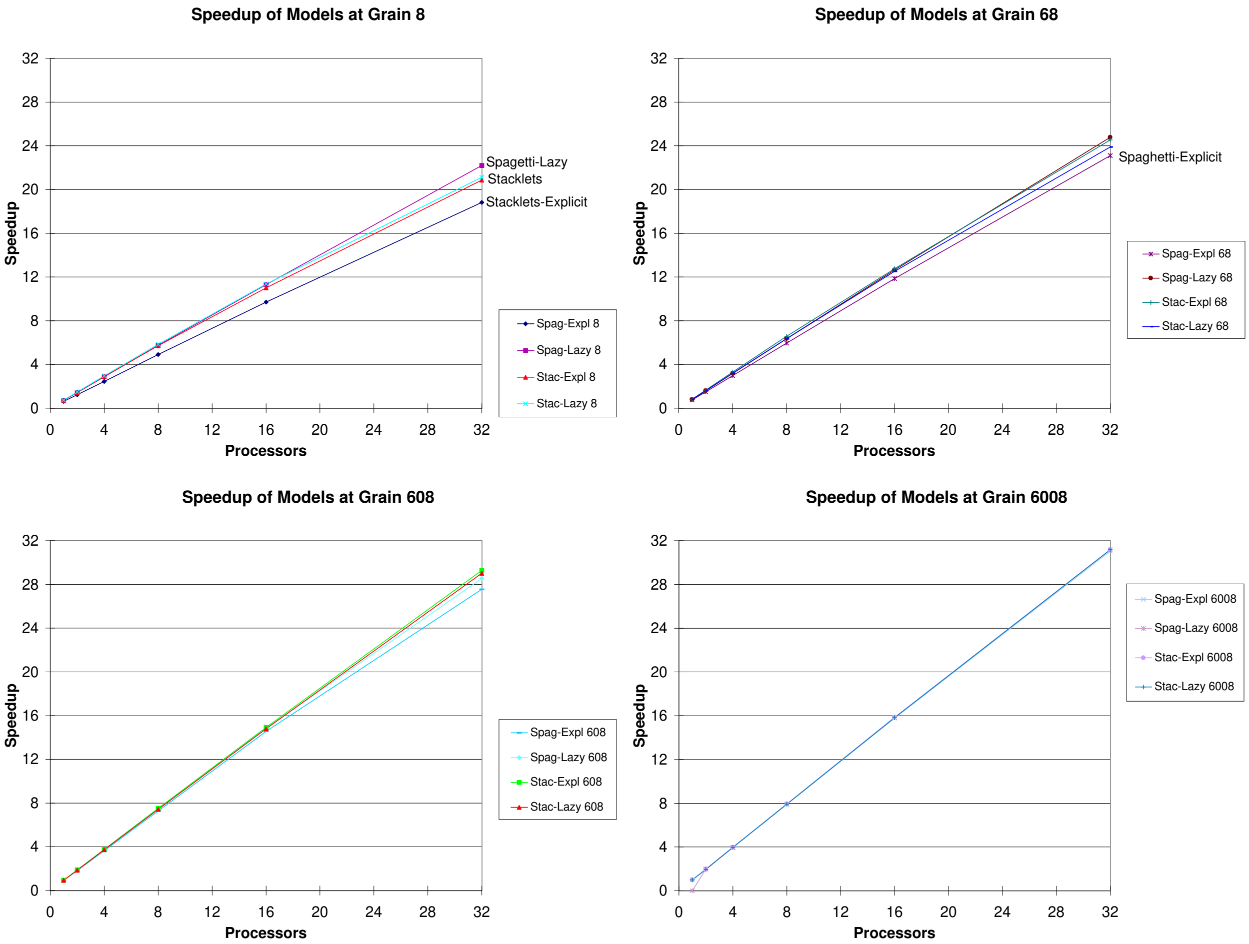


Figure 7.20: *Speedup of* `grain` *at different grain sizes for the explicit and lazy queue models. In all cases, the polling frequency is 1/64.*

size of work that can be stolen. Figure 7.19 shows how the speedup of `grain` changes as we increase the period between polls. Performance increases as we decrease the number of polls until a point is reached when the long periods of ignoring the network interfere with load balancing. For the rest of our experiments we set the polling period to 64.

In Figure 7.20 we show the speedup of `grain` on the NOW with a polling period of 64. In all cases, the combination of spaghetti stacks and the explicit queue performs poorly. The other combinations are roughly equivalent when the grain size exceeds 60 instructions. When run on 32 processors the efficiency ranges from 65% (for a grain size of 8 instructions)

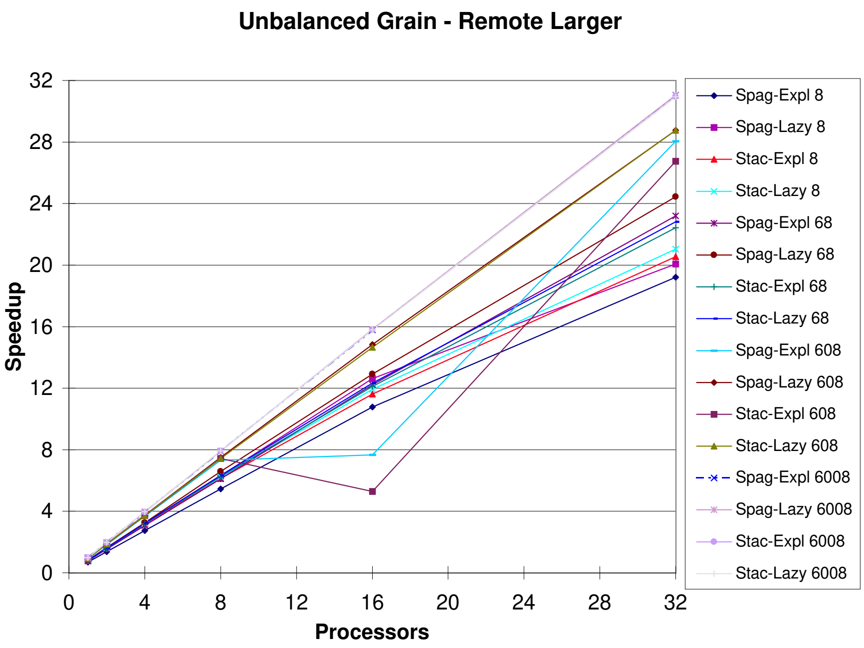


Figure 7.21: *Speedup of an unbalanced* `grain` *where there is more work in the second* `pcall` *of the fork-set.*

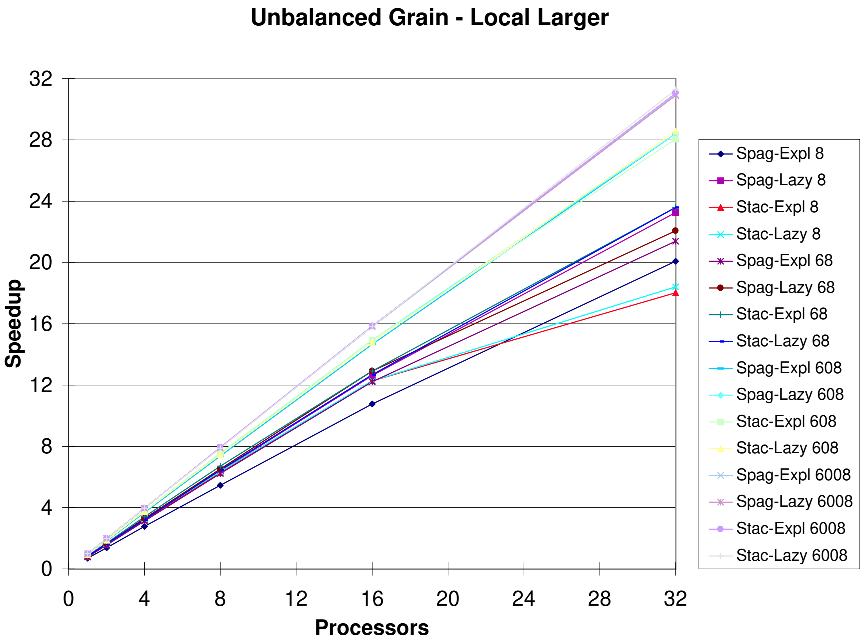


Figure 7.22: *Speedup of an unbalanced* `grain` *where there is more work in the first* `pcall` *of the fork-set.*

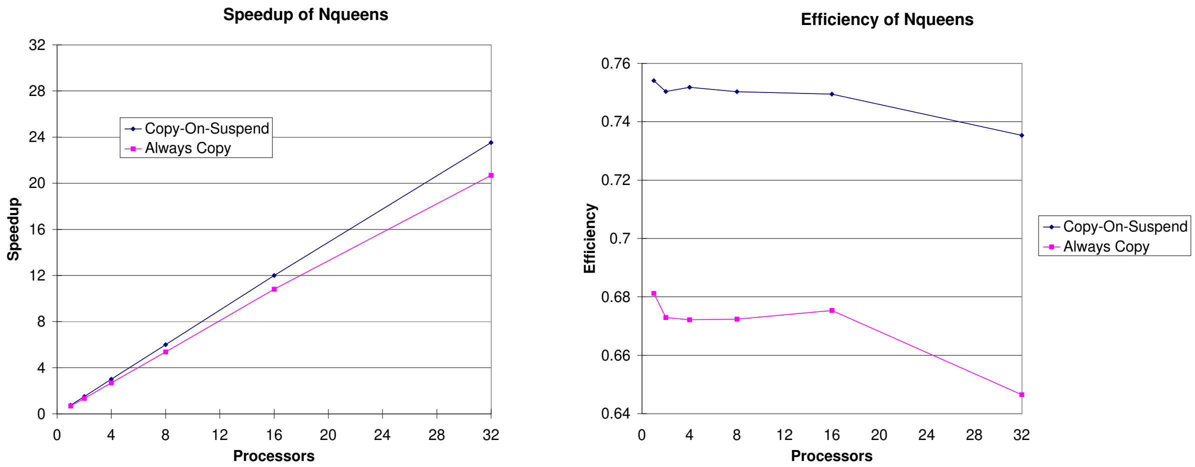


Figure 7.23: *Speedup and efficiency achieved with and without the copy-on-suspend optimization for* `nqueens` *using stacklets, thread seeds, lazy-disconnect, and the lazy queue.*

to over 90% (for a grain size of 600 or more) showing that lazy threading is effective, when implemented correctly, even on distributed memory machines.

Lazy threads also work well when the amount of work performed by the different threads is not uniform. Figures 7.21 and 7.22 show the speedup of a modified `grain` in which the amount of work performed by a parent's children differs by a factor of two. In Figure 7.22 the first `pcall` performs twice as much work as the second. Thus work that is stolen is always smaller than work performed locally. In Figure 7.21, the second `pcall` performs twice as much work as the first, leading to larger-grained work being stolen. When grain size is small, we see a slight performance degradation when the work that can be stolen is smaller than the work available locally. While there is only a small difference between these two programs, the difference shows the asymmetry in the overall approach taken by lazy threads.

7.6 Using Copy-on-Suspend

In our final experiment using Split-C+threads we examine the effect of the copy-on-suspend optimization as applied to the `nqueens` program. As Figure 7.23 shows, the copy-on-suspend optimization yields up to a 20% improvement in performance. If the copy-on-suspend optimization is not applied to `nqueens` there is a loss in performance for using lazy threads even on a single processor because of the need to copy the shared data (See

Section 6.5.1). The copy-on-suspend optimization eliminates the need to copy the shared data except when parallelism is actually needed.

When running on a distributed multiprocessor there is the additional penalty of sending the data across the network. This imposes two costs: the explicit cost of sending the data and the implicit cost of having to poll the network often enough to service the data requests. We compiled the program using a polling period of 64. When compiled for a multiprocessor and run on one processor, the program gets 75% efficiency, indicating that the cost of polling is relatively low. Since `nqueens` achieves a speedup of 23.5 on 32 processors, the poll rate is sufficient to handle both load balancing and requests for data.

7.7 Suspend and Steal Stream Entry Points

As discussed in Section 5.1.2, for ease of experimentation we chose to implement the suspend and steal entry points associated with a seed or continuation as jump instructions placed after the call to which they are associated. This has a larger than expected impact on the performance of a `pcall` due to the difference in the offset used by a return instruction. If we modify the assembly code by changing the jumps to nops and have the function return like an ordinary sequential call, we see no effect on the runtime even though six extra instructions are executed on every return. If we then remove the nops so the same instructions are executed, we see an improvement in performance of about 10%.

7.8 Comparing Code Size

Our compilation techniques incur additional overhead in the form of an increase in code size. Every call is duplicated (for single-processor concurrency) or triplicated (for a multiprocessor). If there is code between `pcall`s, it too must be copied. Furthermore, each call has associated with it the suspend and steal stream entry points, setup code to handle disconnection, etc. Figure 7.24 compares the size of an object file for the `grain` function compiled by Split-C+threads for a uniprocessor and multiprocessor to the size of the same function compiled by GCC. The multithreaded versions are approximately 2.5 and 4.5 times larger than the sequential code as compiled by GCC for the uniprocessor and multiprocessor version respectively. The largest effect on code size comes from the queueing method chosen. The lazy queue has extra code to construct the explicit queue and the implicit queue has

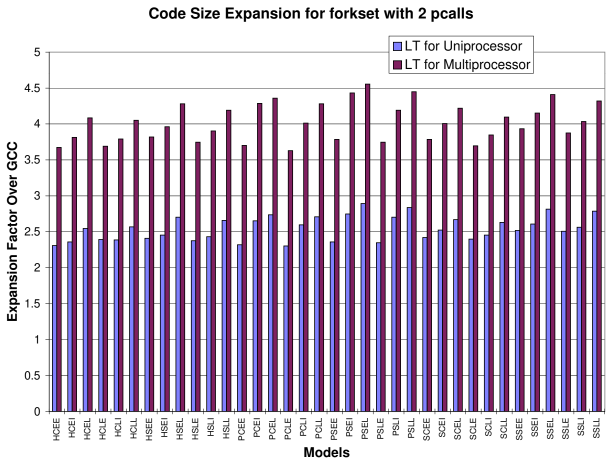


Figure 7.24: *The amount of code dilation for each point in the design space.*

extra code to transfer control to the parent. In effect we have traded off time for space as compared to the library implementation of eager multithreading.

7.9 Id90 Comparison

Our earlier work on lazy threading was motivated by previous work on efficiently implementing Id90 using a threaded abstract machine (TAM). The compilation process involved compiling Id90 to TL0, an assembly language for TAM, and then compiling TL0 to C. In TL0, every function call is implemented as a parallel fork with the associated overhead of scheduling, synchronization, frame allocation, and use of memory to transfer arguments and results. To improve the performance of Id90 executables, we developed a new assembly language, TL1, that would support lazy threading. We developed a lazy thread based compiler for TL1 that used stacklets, thread seeds, lazy-disconnect, and explicit queueing. In addition, TL1 supports strands. This compiler compiled TL1 code to C using techniques that became the basis for the Split-C+threads compiler. In Table 7.4 we show the results of this early work. We can see that even with the overhead of compiling to C, the need to support strands, and no control over register usage, we achieved a significant speedup over TL0.

Program	Short Description	Input Size	TAM	Stacklets, Seeds, Lazy-disconnect, Explicit Queue
Gamteb	Monte Carlo neutron transport	40,000	220.8	139.0
Paraffins	Enumerate isomers of paraffins	19	6.6	2.4
Simple	Hydrodynamics and heat conduction	1 1 100	5.0	3.3
MMT	Matrix multiply test	500	70.5	66.5

Table 7.4: *Dynamic runtime in seconds on a SparcStation 10 for the Id90 benchmark programs under the TAM model and lazy threads with multiple strands. The programs are described in [16].*

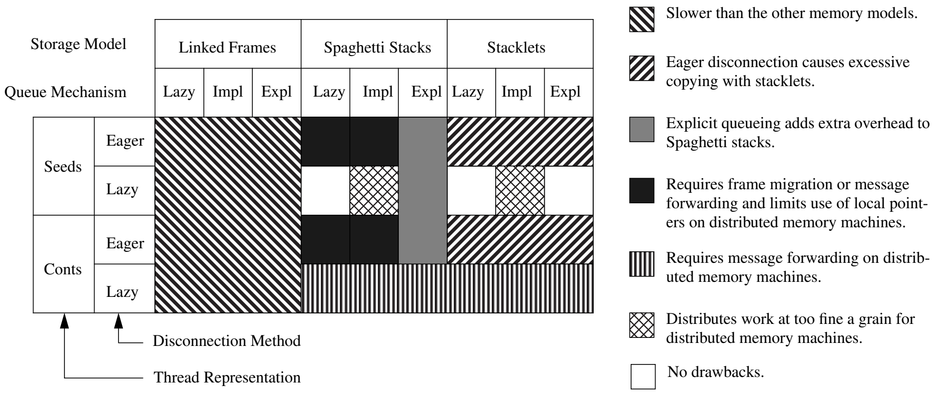


Figure 7.25: *Which points in the design space are effective and why. The white squares are the effective models.*

7.10 Summary

In each of the preceding sections we examined the axes of the design space and analyzed the performance of the points on that axis with respect to the other axes. Figure 7.25 summarizes our conclusions. Each of the shaded regions indicates a point in the space that we have rejected either because the intersection of models lead to poor performance or because a particular point was poor compared to the other possible points on that axis. For example, eager-disconnect and stacklets clearly work against each other and any implementation that uses both of these will perform poorly.

The figure also includes three additional reasons for rejecting particular models for use on distributed memory machines. Eager-disconnect requires either frame migration or message forwarding on distributed memory machines. It also excludes the use of pointers to automatic variables. Continuation stealing is excluded since it requires frame migration. Finally, implicit queueing distributes work at too fine a grain.

All of the three remaining models perform well even on very fine-grained programs. Stacklets, thread seeds, lazy-disconnect, and lazy queueing work particularly well together, requiring almost no preparation overhead before a potentially parallel call is initiated and little work when parallelism is actually needed.

Chapter 8

Related Work

Attempts to accommodate logical parallelism include thread packages [20, 54, 14, 28], compiler techniques and clever runtime representations [16, 49, 44, 63, 61, 53, 30], and direct hardware support for fine-grained parallel execution [34, 3]. These approaches have been used to implement many parallel languages, e.g. Mul-T [39], Id90 [16, 49], CC++ [13], Charm [35], Opus [43], Cilk [7], Olden [12], and Cid [48]. The common goal is to reduce the overhead associated with managing the logical parallelism. While much of this work overlaps ours, none has combined all of the techniques described in this thesis. More importantly, none has started from the premise that all calls, parallel or sequential, can be initiated in exactly the same manner.

Our work grew out of previous efforts to implement the non-strict functional language Id90 for commodity parallel machines. Our earlier work developed a Threaded Abstract Machine (TAM) which serves as an intermediate compilation target [16]. The two key differences between this work and TAM are that under TAM calls are always parallel, and due to TAM's scheduling hierarchy, calling another function does not immediately transfer control.

Our lazy thread fork allows all calls to begin in the same way and creates only the required amount of concurrency. In the framework of previous work it allows excess parallelism to degrade efficiently into a sequential call. Many other researchers have proposed schemes which deal lazily with excess parallelism. One of the simplest schemes is *load based inlining*, which uses load characteristics of the parallel machine to decide at the time a potentially parallel call is encountered whether to execute it sequentially (inline it) or execute it in parallel [39]. This has the advantage of dynamically increasing the granularity of the program. However, these decisions are irrevocable, which can lead to serious load imbalances or deadlock. Our approach builds on lazy task creation (LTC), which introduced the idea of lazy threads. LTC is based on linked frames, continuation stealing, eager-disconnect, and an explicit queue. [44]. LTC also uses work stealing to perform dynamic load balancing. These ideas were studied for Mul-T running on shared-memory machines. Since work-stealing causes the parent to migrate to a new thread, LTC depends on the ability of the system to migrate activation frames. This either requires shared-memory hardware capabilities or restricts the kind of pointers allowed in the language. Finally, LTC also depends on a garbage collector, which hides many of the costs of stack management.

Another proposed technique for improving LTC is leapfrogging, which uses multiple stacks, a limited form of continuation stealing, eager-disconnect, and explicit queues [63]. Unlike the techniques we use, it restricts the use of continuation stealing in an attempt to reduce the cost of futures. Leapfrogging has been implemented using Cthreads, a light-weight thread package.

StackThreads [57] uses both a stack and the heap for storing activation frames in an attempt to reduce overhead for fine-grained programs running on a single processor. Activation frames are initially placed on the stack, and if a thread blocks, its activation frame is moved onto the heap, in effect implementing eager-disconnect on a hybrid memory model. Since the sequential call invariants are not enforced, StackThreads does not take advantage of passing control and data at the same time, reducing register usage and increasing synchronization overhead. They take a point of view diametrically opposed to ours in that all calls, sequential or parallel, use the same representation. This triples the direct function call/return overhead and prevents the use of registers.

A much simpler thread model is advocated in Shared Filaments [19] and Distributed Filaments [23]. A filament is an extremely lightweight thread which does not have a stack associated with it. This works well when a thread does not fork other threads. More general threads are supported with a single stack because language restrictions make it impossible for a parent to be scheduled when any of its children are waiting for a synchronization event. Distributed filaments are combined with a distributed shared memory system. In the case of a remote page fault, communication can be overlapped with computation because each processor runs multiple scheduling threads.

Olden [12] is based on spaghetti stacks, continuation stealing, eager-disconnect, and an explicit queue. In Olden, as well as in previous uses of spaghetti stacks [10, 31], a garbage collector is used to reclaim freed frames. Olden's thread model is more powerful than ours, since in Olden threads can migrate. The idea is that a thread computation that follows references to unstructured heap-allocated data might increase locality if migration occurs [52]. On the other hand, this model requires migrating the current call frame as well as disallowing local pointers to other frames on the stack.

We use stacklets for efficient stack-based frame allocation in parallel programs. Previous work by Hieb et al. [30] describes similar ideas for handling continuations efficiently, but it uses a garbage collector.

Our implementation technique for encoding the suspend and steal entry points for a thread seed or continuation builds on the use of multiple offsets from a single return address to handle special cases. This technique was used in SOAR [60]. It was also applied to Self, which uses parent controlled return continuations to handle debugging [32].

Finally, many lightweight thread packages have been developed. Cthreads is a runtime library which provides multiple threads of control and synchronization primitives for parallel programming at the level of the C language [14]. Scheduler activations reduce the overhead by moving fine-grained threads completely to the user level and relying on the kernel only for infrequent cases [1]. Synthesis is an operating systems kernel for a parallel and distributed computational environment that integrates dynamic load balancing and dynamic compilation techniques [41]. Chant [28] is a lightweight threads package which is used in the implementation of an HPF extension called Opus [43]. Chant provides an interface for lightweight, user-level threads that have the capability of communication and synchronization across separate address spaces. While user-level thread packages eliminate much of the overhead encountered in traditional operating-systems thread packages, they are still not as lightweight as many of the systems mentioned above that use special runtime representations supported by the compiler. Since the primitives of thread packages are exposed at the library level, compiler optimizations presented in this paper are not possible for such systems.

Chapter 9

Conclusions

Many modern parallel languages support dynamic creation of threads or require multithreading in their implementations. In these languages it is desirable for the logical parallelism expressed by the threads to exceed the physical parallelism of the machine. In practice most logical threads need not be independent threads. Instead, they can run as sequential calls, which are inherently cheaper than independent threads. The challenge to the language implementor is that one cannot generally predict which logical threads can be implemented as sequential calls. In lazy multithreading systems each logical thread begins execution sequentially, with the attendant efficient stack management and direct transfer of control and data. Only if a thread must execute in parallel does it get its own thread of control.

This dissertation presents a design space for implementing multithreaded systems without preemption. We introduce a sufficient set of primitives so that any point in the design space may be implemented using our new primitives. The primitives and implementation techniques that we propose reduce the cost of a potentially parallel call to nearly that of a sequential call, without restricting the parallelism inherent in the program. Using our primitives we implement a system (in some cases a previously proposed system) at each point in the design space.

We have shown that, using appropriate compiler techniques, we can provide a fast parallel call with nearly the full efficiency of a sequential call when the child thread executes locally and runs to completion without suspension. Such child threads occur frequently with aggressively parallel languages such as Id90, and with more conservative languages such as C with parallel calls (e.g., Split-C+threads).

The central idea behind a fast parallel call is to pay only for what is used. Thus a local fork is performed essentially as a sequential call, with the attendant efficient stack management and direct transfer of control and data. If the child actually suspends before completion, control is returned to the parent so that it can take appropriate action. Similarly, remote work is generated lazily.

Our compilation techniques, and the new points in the lazy thread design space we introduce, exploit the one bit of flexibility in the sequential call: the indirect jump on return. First, we exploit this flexibility by associating with each return address multiple return entry points. The extra entry points are linked to compiler-generated code streams that handle any necessary parallelism without advance preparation. Second, we exploit the indirect jump by using parent-controlled return continuations to eliminate the need for synchronization until the child and parent need to run in parallel. The new design points introduced (stacklets on the memory axis, thread seeds on the thread representation axis, lazy-disconnect on the disconnection axis, and implicit and lazy queueing) also arise from exploiting the flexibility of the indirect jump on return.

We find that performance is best in implementations that strike a balance between preparation before the potentially parallel call and extra work when parallelism is actually needed. Our implementation of spaghetti stacks illustrates this point perfectly. If a lazy or implicit queue is used, then when a thread runs to completion, no unnecessary check is made to see if reclamation is necessary. However, if a thread suspends, we use PCRCs to ensure that the check is performed when the thread eventually completes. This strikes a proper balance by avoiding runtime preparation and limiting the work necessary when parallelism is needed. However, when spaghetti stacks are used in conjunction with an explicit queue, we prepare for the worst case by performing the check on every return, making this an unattractive combination. Stacklets also achieve balance for storing thread state, while linked frames require too much preparation. Continuations and thread seeds, when implemented using multiple return entry points and PCRCs, both achieve this balance. Eager-disconnect, when combined with stacklets or spaghetti stacks and explicit queueing, requires too much work when parallelism is needed. Finally, of the three queueing methods, lazy queueing works well while explicit queueing requires preparation on every parallel call, and implicit queueing interacts badly with work stealing when parallelism is actually needed.

Our empirical studies, using Split-C+threads on the Berkeley NOW, show that the combination of stacklets or spaghetti stacks, thread seeds, lazy-disconnect, and lazy queueing offers very good parallel performance and supports fine-grained parallelism even on a distributed memory machine. In contrast, the linked-frame storage model, which is the most commonly used storage model for compiler-integrated multithreading, performs poorly. When concurrency is needed on a single processor, we find that continuations are also a reasonable choice for representing threads. The trade-off between thread seeds and continuations on shared memory machines remains unexplored; on such machines migration across address spaces is not necessary, removing the obstacle to using continuations.

This dissertation opens several areas for exploration. The most obvious is comparing the design space on shared memory machines. Three other interesting areas are generalizing the use of thread seeds, the use of multiple code streams to support copy-on-suspend in parallel programs, and the efficient placement of network polling operations.

- One way to view thread seeds is as an efficient way to handle exceptions without advance preparation. Broadly speaking, an "exception" is an infrequently encountered condition that must nonetheless be handled as part of normal program execution. In our case, the exception is the need to run a child in parallel. Perhaps thread seeds can be applied to a more broad class of exceptions in languages that promote the use of explicit exception handling.

- Our results depend heavily on generating multiple code streams for each fork-set. We also see how multiple code streams enable the copy-on-suspend optimization, which eliminates unnecessary copying of shared data structures. Perhaps copy-on-suspend can be implemented automatically, and if so, multiple code streams may allow other optimizations.

- Because our compilation strategy eliminates most of the overhead due to multithreading, the dominating overhead on a distributed memory multiprocessor is the network poll. We currently take a very conservative approach and insert many more polling operations than necessary. An important optimization to investigate is how to reduce the number of polling operations while still guaranteeing correct network behavior.

In conclusion, we have shown that fine-grained parallelism can be efficiently supported, thus allowing programmers (and back-end compilers) the freedom to specify all the parallelism in a program without concern about excessive overhead.

Appendix A

Glossary

cactus stack Page 10
The global tree of thread frames.

`is-child` Page 31
The state of a frame that has been invoked with an `lfork`, but has not been disconnected.

child thread Page 20
The thread created by a one of the fork operations (`fork`, `dfork`, or `lfork`). The thread executing the fork operation is the parent thread.

closure Page 40
A data structure that represents a nascent thread, containing the arguments and address of the thread's codeblock. The closure is self-contained and can be turned into an independent thread by the general scheduler.

codeblock Page 18
The set of instructions that comprise the body of a function or a thread.

continuation stealing Page 33
An operation that allows a parent thread to disconnect from its lazy child thread. It disconnects the parent by restarting the parent at its current continuation.

disconnection Page 28
An operation that elevates a lazy thread into an independent thread. This may involve disconnecting the control link between the lazy thread and its parent, the storage used by the lazy thread and its parent, or both.

daemon thread Page 12
A thread that remains alive until the end of the program.

`dexit` Page 32
: The instruction in MAM/DF that terminates a thread that was invoked by `dfork`.

`dfork` Page 31
: An instruction in MAM/DF that creates and directly schedules a child thread.

downward thread Page 12
: A thread that terminates before its parent terminates.

eager-disconnect Page 51
: The act of disconnecting a child from its parent by changing the parent frame so that future lazy threads may be invoked in the same manner as before disconnection. (Compare to lazy-disconnect.)

eager fork Page 30
: A fork that always creates an independent thread.

`enable` Page 23
: The instruction that ensures that a thread is on the ready queue and in the `ready` state.

`exit` Page 20
: The instruction that terminates a thread and idles the processor on which it was running.

`fork` Page 20
: The instruction that creates a new independent thread.

fork-set Page 11
: A set of forks that are related by a common join.

frame area Page 62
: The area of a stack or stacklet that holds the activation frames for one independent thread (the base frame), its lazy thread descendants, and their sequential call frames.

`has-child` Page 29
: The state a parent thread enters when it `lfork`s a lazy child thread.

idle thread Page 24
: A thread waiting on an event. It is always an independent thread.

independent thread Page 28
: A thread that is not lazy, i.e., one with its own logical stack.

indirect inlet address Page 44
: An address of a location in a thread's frame which contains the address of an inlet. The address is used by `lreturn` to find the inlet to run when the child terminates.

inlet Page 22
A code fragment that processes data received from another thread. All inter-thread communication is handled by transferring data with `send` to an inlet.

`ireturn` Page 23
The instruction used to return control from an inlet back to the sending thread.

`is-child` Page 31
The state a lazy thread enters when it is created.

join counter Page 22
A counter in the parent thread used in the synthesis of the join operation for threads.

lazy thread Page 28
A thread that has been started by an `lfork` and is using its parent's stack.

`lfork` Page 44
An instruction in MAM/DS and LMAM that creates and directly schedules a child thread.

lazy-disconnect Page 51
The act of disconnecting a child from its parent without changing the parent's frame. It causes future lazy threads to be invoked differently than before disconnection. (Compare to eager-disconnect.)

linked frames Page 59
A thread storage model where every activation frame is fixed-sized and heap-allocated.

LMAM Page 50
A Lazy Multithreaded Abstract Machine. A lazy thread is directly scheduled and can share the same stack as its parent thread.

`lreturn` Page 44
An instruction in MAM/DS and LMAM that simultaneously returns control and data from a child thread to its parent.

nascent thread Page 35
A thread that has not yet started, but will start the next time its parent is resumed.

MAM Page 15
The basic multithreaded abstract machine which supports only eager thread creation.

MAM/DF Page 27
A multithreaded abstract machine with direct fork.

MAM/DS Page 41

A multithreaded abstract machine with direct scheduling for both thread creation and termination. `lfork` directly calls its lazy child, and `lreturn` directly schedules its parent.

multiple stacks Page 62

A thread storage model in which each thread is assigned its own stack. `lfork`s and `call`s are allocated on the same stacks as their parents.

parallel overhead Page 120

The overhead introduced by a parallel algorithm over its sequential algorithm.

parent queue Page 29

The set of parent threads that have given control of their processors to their children. These threads are all in the `has-child` state.

parent thread Page 20

The thread that executes a fork operation (`fork`, `lfork`, or `dfork`). The thread created by the fork is the child thread.

ready thread Page 24

A thread in the `ready` state. It is always an independent thread.

`recv` Page 23

The first instruction of an inlet. It describes the data expected by the inlet.

ready queue Page 17

The set of independent threads that are in the `ready` state, i.e., that are ready to execute.

return register Page 16

Contains the return continuation used by a child to return results to its parent. In MAM the return continuation is an inlet pointer and a thread pointer. It is redefined in MAM/DS (on Page 44) to be the indirect inlet address.

running thread Page 24

A thread in the running state. It is always an independent thread.

`send` Page 23

The instruction used to transfer data from one thread to the inlet of another thread.

spaghetti stack Page 71

A memory region that is unbounded in one direction, but unlike a stack, has no free operation. A spaghetti stack requires the use of a `top` pointer which points to the next free location in the stack.

stacklet Page 76

A fixed-sized heap-allocated chunk of memory that behaves like a stack.

strands Page 124
: A flow of control within a function that may not suspend. Strands allow parallelism within a function.

stub Page 62
: A frame at the bottom of a stack or stacklet that is used to maintain the links between independent threads and the cactus stack.

`suspend` Page 20
: The instruction that causes a thread to give up its processor and enter the `idle` state. If the thread was lazy, this may also involve disconnecting the thread from its parent.

thread seed Page 36
: A representation for a nascent thread. It is a pointer to a context and a set of related code pointers where each of the code pointers can be derived at link time from a single code pointer.

thread Page 16
: A locus of control which can perform calls to arbitrary nesting depth, suspend at any point, and fork additional threads. Threads are scheduled independently and are non-preemptive. Threads can be either independent or lazy.

upward thread Page 12
: A thread that terminates after its parent terminates.

`yield` Page 20
: The instruction that causes the thread executing it to be placed on the ready queue.

work stealing Page 15
: The act of assigning a thread that comes from the parent queue to an idle processor. Work stealing is accomplished through either continuation stealing or seed activation.

Bibliography

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[2] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, Jan 87.

[3] Arvind and D. E. Culler. Dataflow architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986. Reprinted in *Dataflow and Reduction Architectures*, S. S. Thakkar, editor, IEEE Computer Society Press, 1987.

[4] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Technical Report FLA memo, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, August 1983.

[5] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proc. of the Fourth Int. Symp. on Biological and Artificial Intelligence Systems*, pages 255–286. ESCOM (Leider), Trento, Italy, September 1988.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, P. Lisiecki, K. H. Randall, A. Shaw, and Y. Zhou. *Cilk 1.1 reference manual.* MIT Lab for Comp. Sci., 545 Technology Square, Cambridge, MA 02139, September 1994.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the Thirty-Fifth Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 256–368, Sante Fe, NM, November 1994.

[10] Daniel G. Bobrow and Ben Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16:591–602, 1973.

[11] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Sietz, J. Seizovic, and W. Su. Myrinet — A gigabit-per-second local-area network. In *IEEE Micro*, pages 29–36, February 1995.

[12] M.C. Carlisle, A. Rogers, J.H. Reppy, and L.J. Hendren. Early experiences with Olden (parallel programming). In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*, pages 1–20. Springer-Verlag, 1994.

[13] K.M. Chandy and C. Kesselman. Compositional C++: compositional parallel programming. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 124–44. Springer-Verlag, 1993.

[14] E. C. Cooper and R. P. Draves. C-Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, February 1988.

[15] D. Culler, A. Dusseau, S. Goldstein, S. Lumetta, T. von eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, November 1993.

[16] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[17] D. E. Culler, K. E. Schauser, and T. von Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, FL.* North-Holland, January 1993.

[18] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel computing on the Berkeley NOW. In *To appear in Ninth Joint Symposium on Parallel Processing*, Kobe, Japan, 1997.

[19] D. R. Engler, D. K. Lowenthal, and Andrews G. R. Shared Filaments: efficient fine-grain parallelism on shared-memory multiprocessors. Technical Report TR 93-13a, University of Arizona, April 1993.

[20] J.E. Faust and H.M. Levy. The performance of an object-oriented threads package. In *SIGPLAN Notices*, pages 278–88, Oct. 1990.

[21] Message Passing Interface Forum. Mpi: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, vol.8,(no.3-4):169–416, Fall-Winter 1994.

[22] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[23] V.W. Freeh, D.K. Lowenthal, and G.R. Andrews. Distributed Filaments: efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–13. USENIX Assoc, 1994.

[24] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[25] Seth Copen Goldstein. The implementation of a threaded abstract machine. Master's thesis, University of California at Berkeley, Computer Science Division, University of California, Berkeley, Ca 94720, May 1994. Technical Report UCB94-818.

[26] James Gosling, Bill Joy, and Steele Guy. *The Java Language Specification*. Sun Microsystems, version 1.0 edition, August 1996.

[27] D. Grunwald, B. Calder, S. Vajracharya, and H. Srinivasan. Heaps o'Stacks: combined heap-based activation allocation for parallel programs. URL: http://www.cs.colorado.edu/~ grunwald/, 1994.

[28] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: a talking threads package. In *Proceedings Supercomputing '94 (Cat. No.94CH34819)*, pages 350–9. IEEE Comput. Soc. Press, 1994.

[29] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[30] R. Hieb, R. Kent Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *SIGPLAN Notices*, pages 66–77, June 1990.

[31] Guido Hogen and Rita Loogen. A New Stack Technique for the Management of Runtime Structures in Distributed Implementations. Aachener Informatik-Berichte 93-3, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany, 1993.

[32] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN Notices*, pages 32–43, July 1992.

[33] IEEE. *IEEE Standard for Threads Interface to POSIX*, ieee draft std p1003.1c/d10 edition.

[34] H. F. Jordan. Performance measurement on HEP — a pipelined MIMD computer. In *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[35] L.V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *SIGPLAN Notices*, pages 91–108, Oct. 1993.

[36] V. Karamcheti and A. Chien. Concert: efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings SUPERCOMPUTING '93*, pages 598–607. IEEE Comput. Soc. Press, Nov. 1993.

[37] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[38] C. koelbel, D. Loveman, R. Schreiber, G. Steel Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[39] D.A. Kranz, Jr. Halstead, R.H., and E. Mohr. Mul-T: a high-performance parallel Lisp. In *SIGPLAN Notices*, pages 81–90, July 1989.

[40] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proc. of the 24th Int'l Symposium on Computer Architecture*, June 1997.

[41] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[42] Dylan McNamee. Newthreads. http://www.cs.washington.edu/research/compiler/papers.d/nt.html, 1994.

[43] P. Mehrotra and M. Haines. An overview of the Opus language and runtime system. In *Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings*, pages 346–60. Springer-Verlag, 1995. AN4917658.

[44] E. Mohr, D.A. Kranz, and Jr. Halstead, R.H. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, vol.2,(no.3):264–80, July 1991.

[45] R. S. Nikhil. Id (version 88.0) reference manual. Technical Report CSG Memo 284, MIT Lab for Comp. Sci., March 1988.

[46] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo, to appear, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, 1990.

[47] R. S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proc. Sixth Ann. Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[48] R. S. Nikhil. Cid: A parallel, "shared memory" C for distributed-memory machines. In *Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings*. Springer-Verlag, 1995.

[49] R.S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*, pages 390–405. Springer-Verlag, 1994.

[50] M.D. Noakes, D.A. Wallach, and W.J. Dally. The J-Machine multicomputer: an architectural evaluation. In *Computer Architecture News*, pages 224–35, May 1993.

[51] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, Seattle, Washington, May 1990.

[52] A. Rogers, M.C. Carlisle, J.H. Reppy, and L.J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, vol.17,(no.2):233–63, March 1995.

[53] A. Rogers, J. Reppy, and L. Hendren. Supporting SPMD execution for dynamic data structures. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 192–207. Springer-Verlag, 1993.

[54] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69. USENIX Assoc, 1992.

[55] Anurag Sah. Parallel language support on shared memory multiprocessors. Master's thesis, University of California — Berkeley, May 1991.

[56] Sun Microsystems, Inc. *Solaris Threads.*

[57] K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: an abstract machine for scheduling fine-grain threads on stock CPUs. In *Theory and Practice of Parallel Programming. International Workshop TPPP '94. Proceedings*, pages 121–36. Springer-Verlag, 1995. AN4986592.

[58] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 technical summary*, January 1992.

[59] K. R. Traub. *Implementation of Non-strict Functional Programming Languages.* MIT Press, 1991.

[60] D. M. Ungar. *The design and evaluation of a high performance Smalltalk system.* ACM distinguished dissertations. MIT Press, 1987.

[61] M.T. Vandevoorde and E.S. Roberts. WorkCrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, vol.17,(no.4):347–66, Aug. 1988.

[62] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[63] D.B. Wagner and B.G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *SIGPLAN Notices*, pages 208–17, July 1993.

[64] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, 1994.

[65] A. Yonezawa. *ABCL– an object-oriented concurrent system* . MIT Press series in computer systems. MIT Press, 1990.