An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors

A Dissertation

Presented to
The Faculty of the Graduate School of Arts and Sciences
Brandeis University
Department of Computer Science
James S. Miller, advisor

In Partial Fulfillment
of the Requirements of the Degree of
DOCTOR OF PHILOSOPHY

by

Marc Feeley

April 1993

This dissertation, directed and approved by the candidate's committee, has been accepted and approved by the Graduate Faculty of Brandeis University in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Dean, Graduate School of Arts and Sciences

Dissertation Committee

Dr. James S. Miller (chair)
(Digital Equipment Corporation)

Prof. Harry Mairson

Prof. Timothy Hickey

Prof. David Waltz

Dr. Robert H. Halstead, Jr.
(Digital Equipment Corporation)

Abstract

An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors

A dissertation presented to the Faculty of the Graduate School of
Arts and Sciences of Brandeis University, Waltham, Massachusetts

by Marc Feeley

This thesis describes a high-performance implementation technique for Multilisp's "future" parallelism construct. This method addresses the non-uniform memory access (NUMA) problem inherent in large scale shared-memory multiprocessors. The technique is based on lazy task creation (LTC), a dynamic task partitioning mechanism that dramatically reduces the cost of task creation and consequently makes it possible to exploit fine grain parallelism. In LTC, idle processors get work to do by "stealing" tasks from other processors. A previously proposed implementation of LTC is the shared-memory (SM) protocol. The main disadvantage of the SM protocol is that it requires the stack to be cached suboptimally on cache-incoherent machines. This thesis proposes a new implementation technique for LTC that allows full caching of the stack: the message-passing (MP) protocol. Idle processors ask for work by sending "work request" messages to other processors. After receiving such a message a processor checks its private stack and task queue and sends back a task if one is available. The message passing protocol has the added benefits of a lower task creation cost and simpler algorithms. Extensive experiments evaluate the performance of both protocols on large shared-memory multiprocessors: a 90 processor GP1000 and a 32 processor TC2000. The results show that the MP protocol is consistently better than the SM protocol. The difference in performance is as high as a factor of two when a cache is available and a factor of 1.2 when a cache is not available. In addition, the thesis shows that the semantics of the Multilisp language does not have to be impoverished to attain good performance. The laziness of LTC can be exploited to support at virtually no cost several programming features including: the Katz-Weise continuation semantics with legitimacy, dynamic scoping, and fairness.

Acknowledgements

Cette thèse est dédiée à mes grandparents Rose et Émile Monna pour l'amour que j'ai pour eux.

I wish to thank my family, my friends, and colleagues without whom this thesis would not have been possible.

Special thanks go to Jim Miller, my thesis advisor, for giving me the freedom to explore my ideas at my own pace. He has gone beyond the call of duty to see me through with my degree.

Bert Halstead's words of encouragement gave me the confidence that my ideas were interesting and worth writing about. Thank you Bert.

Sabine Bergler deserves special thanks for taking care of me.

To Chris, Mauricio, Harry, Emmanuel, Don, Shyam, Larry, Xiru, Mary and Paulo, thank you for making my stay at Brandeis so enjoyable.

Finally, I wish to thank the National Science and Engineering Research Council of Canada and the Université de Montréal for financial support, and Michigan State University, Argonne National Laboratory, Lawrence Livermore National Laboratory, and the MIT AI Laboratory for the use of their computers.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Why Multilisp? 3

1.3 Fundamental Issues 4

1.4 Architecture 4

1.4.1 Shared-Memory MIMD Computers 5

1.4.2 Non-Uniform Memory Access 6

1.4.3 Sharing Data 7

1.4.4 Caches 7

1.4.5 Memory Consistency 9

1.5 The GP1000 and TC2000 Computers 12

1.6 Memory Management 13

1.7 Dynamic Partitioning 15

1.7.1 Eager Task Creation 18

1.7.2 Lazy Task Creation 18

1.8 Overview 20

2 Background **21**

2.1 Scheme's Legacy 21

2.2 First-Class Continuations 25

2.2.1 Continuation Passing Style 25

2.2.2 Programming with Continuations 26

2.3 Multilisp's Model of Parallelism 28

2.3.1 `FUTURE` and `TOUCH` 28

2.3.2 Placeholders 30
2.3.3 Spawning Trees 30
2.4 Types of Parallelism 31
2.4.1 Pipeline Parallelism 31
2.4.2 Fork-Join Parallelism 33
2.4.3 Divide and Conquer Parallelism 34
2.5 Implementing Eager Task Creation 36
2.5.1 The Work Queue 37
2.5.2 `FUTURE` and `TOUCH` 37
2.5.3 Scheme Encoding 38
2.5.4 Chasing vs. No Chasing 41
2.5.5 Critical Sections 41
2.5.6 Centralized vs. Distributed Work Queue 42
2.6 Fairness of Scheduling 43
2.7 Dynamic Scoping 45
2.8 Continuation Semantics 49
2.8.1 Original Semantics 49
2.8.2 MultiScheme Semantics 49
2.8.3 Katz-Weise Continuations 51
2.8.4 Katz-Weise Continuations with Legitimacy 52
2.8.5 Implementing Legitimacy 54
2.8.6 Speculation Barriers 55
2.8.7 The Cost of Supporting Legitimacy 56
2.9 Benchmark Programs 59
2.9.1 `abisort` 60
2.9.2 `allpairs` 60
2.9.3 `fib` 61
2.9.4 `mm` 61
2.9.5 `mst` 61
2.9.6 `poly` 62
2.9.7 `qsort` 63
2.9.8 `queens` 63
2.9.9 `rantree` 64

2.9.10 `scan` . . . 64
2.9.11 `sum` . . . 65
2.9.12 `tridiag` . . . 65
2.10 The Performance of ETC . . . 66

3 Lazy Task Creation **73**
3.1 Overview of LTC Scheduling . . . 73
3.1.1 Task Stealing Behavior . . . 75
3.1.2 Task Suspension Behavior . . . 77
3.2 Continuations for Futures . . . 78
3.2.1 Procedure Calling Convention . . . 79
3.2.2 Unlimited Extent Continuations . . . 79
3.2.3 Continuation Heapification . . . 81
3.2.4 Parsing Continuations . . . 82
3.2.5 Implementing First-Class Continuations . . . 82
3.3 The LTC Mechanism . . . 84
3.3.1 The Lazy Task Queue . . . 85
3.3.2 Pushing and Popping Lazy Tasks . . . 86
3.3.3 Stealing Lazy Tasks . . . 88
3.3.4 The Dynamic Environment Queue . . . 92
3.3.5 The Problem of Overflow . . . 93
3.3.6 The Heavyweight Task Queue . . . 95
3.3.7 Supporting Weaker Continuation Semantics . . . 96
3.4 Synchronizing Access to the Task Stack . . . 98
3.5 The Shared-Memory Protocol . . . 99
3.5.1 Avoiding Hardware Locks . . . 101
3.5.2 Cost of a Future on GP1000 . . . 104
3.6 Impact of Memory Hierarchy on Performance . . . 107
3.7 The Message-Passing Protocol . . . 112
3.7.1 Really Lazy Task Creation . . . 114
3.7.2 Communicating Steal Requests . . . 114
3.7.3 Potential Problems with the MP Protocol . . . 116
3.8 Code Generated for SM and MP Protocols . . . 118

3.9 Summary 120

4 Polling Efficiently **123**

4.1 The Problem of Procedure Calls 125

4.1.1 Code Structure 125

4.1.2 Call-Return Polling 126

4.2 Short Lived Procedures 127

4.3 Balanced Polling 129

4.3.1 Subproblem Calls 129

4.3.2 Reduction Calls 131

4.3.3 Minimal Polling 132

4.4 Handling Join Points 135

4.5 Polling in Gambit 135

4.6 Results 136

4.7 Summary 139

5 Experiments **141**

5.1 Experimental Setting 142

5.2 Overhead of Exposing Parallelism 144

5.2.1 Overhead on GP1000 145

5.2.2 Overhead on TC2000 145

5.3 Speedup Characteristics 145

5.3.1 Speedup on GP1000 157

5.3.2 Speedup on TC2000 158

5.4 Effect of Interrupt Latency 159

5.5 Cost of Supporting Legitimacy 162

5.6 Summary 163

6 Conclusion **165**

6.1 Future Work 167

A Source Code for Parallel Benchmarks **169**

A.1 `abisort` 172

A.2 `allpairs` 175

A.3 `fib` 176
A.4 `mm` 177
A.5 `mst` 178
A.6 `poly` 181
A.7 `qsort` 182
A.8 `queens` 183
A.9 `rantree` 184
A.10 `scan` 185
A.11 `sum` 186
A.12 `tridiag` 187

B Execution Profiles for Parallel Benchmarks **191**
B.1 `abisort` 194
B.2 `allpairs` 195
B.3 `fib` 196
B.4 `mm` 197
B.5 `mst` 198
B.6 `poly` 199
B.7 `qsort` 200
B.8 `queens` 201
B.9 `rantree` 202
B.10 `scan` 203
B.11 `sum` 204
B.12 `tridiag` 205

List of Tables

1.1 Costs of memory hierarchy for the GP1000 and the TC2000. 13

2.1 Characteristics of parallel benchmark programs running on GP1000. . . 69

3.1 Size of closure for each future in the benchmark programs. 87

3.2 Cost of operations involved in task stealing. 106

3.3 Measurements of memory access behavior of benchmark programs. . . . 109

4.1 Overhead of polling methods on GP1000. 138

5.1 Performance of SM protocol on GP1000. 146

5.2 Performance of MP protocol on GP1000. 147

5.3 Performance of SM protocol on TC2000. 148

5.4 Performance of MP protocol on TC2000. 148

5.5 Performance of MP protocol on GP1000 with $I = 2$. 160

5.6 Performance of MP protocol on GP1000 with $I = 50$. 161

5.7 Overhead of supporting legitimacy, with and without speculation barrier on GP1000. 163

List of Figures

1.1 The shared-memory MIMD computer used in this thesis. 6

2.1 Non-local exit using `call/cc`. 26

2.2 Parallel `map` definition and spawning trees. 32

2.3 Parallel "vector" `map`. 35

2.4 Scheme encoding of Multilisp core. 39

2.5 Procedures needed to support Multilisp core. 40

2.6 Exception system based on dynamic scoping and `call/cc`. 46

2.7 Implementation of dynamic scoping with tail recursive `call/cc`. 48

2.8 MultiScheme's implementation of the future special form. 50

2.9 A sample use of futures and `call/cc`. 51

2.10 A future body's continuation called multiple times. 52

2.11 Exception processing with futures. 53

2.12 The Katz-Weise implementation of futures. 55

2.13 An application of speculation barriers. 56

2.14 Fork-join algorithms and their legitimacy chain in the absence of chain collapsing. 58

2.15 General case of legitimacy chain collapsing for fork-join algorithms. . . . 59

2.16 `Fib` and a poor variant obtained by unrolling the recursion. 70

3.1 The task stack. 75

3.2 Continuation representation and operations. 80

3.3 Underflow and heapification algorithms. 83

3.4 Resuming a heavyweight task. 88

3.5 The LTQ and the steal operation. 90
3.6 The task stealing mechanism. 91
3.7 The implementation of `dyn-bind`. 93
3.8 The DEQ and its use in recovering a stolen task's dynamic environment. 94
3.9 Code sequence for a future under the SM protocol. 101
3.10 Thief side of the SM protocol. 102
3.11 Victim side of the SM protocol. 102
3.12 Relative importance of stack and heap accesses of benchmark programs. 110
3.13 Thief side of the MP protocol. 115
3.14 Victim side of the MP protocol. 115
3.15 Assembly code generated for `fib`. 119

4.1 The `for-each` procedure and its corresponding code graph. 126
4.2 Two instances of short lived procedures. 127
4.3 The maximal delta method. 128
4.4 Procedure return invariants in balanced polling. 130
4.5 Compilation rules for balanced polling. 133
4.6 Minimal polling for the recursive procedure `sum` and a tail recursive variant.134

5.1 Speedup curves for `fib`, `queens`, `rantree` and `mm` on GP1000. 150
5.2 Speedup curves for `scan`, `sum`, `tridiag` and `allpairs` on GP1000. . . . 151
5.3 Speedup curves for `abisort`, `mst`, `qsort` and `poly` on GP1000. 152
5.4 Speedup curves for `fib`, `queens`, `rantree` and `mm` on TC2000. 153
5.5 Speedup curves for `scan`, `sum`, `tridiag` and `allpairs` on TC2000. . . . 154
5.6 Speedup curves for `abisort`, `mst`, `qsort` and `poly` on TC2000. 155
5.7 Task creation behavior of MP protocol on GP1000. 156
5.8 Task suspension behavior of MP protocol on GP1000. 156

Chapter 1

Introduction

This work is about the design of an efficient implementation strategy for Multilisp's "future" parallelism construct on large shared-memory multiprocessors. A strategy known as "lazy task creation" is used as a starting point for this work. Two implementations of lazy task creation, one based on a shared-memory paradigm and the other based on a message-passing paradigm, are explained and compared by extensive experiments with a large number of benchmarks. The result can be summarized as follows

> An implementation of lazy task creation based on a message-passing paradigm is superior to one based on a shared-memory paradigm because it is
>
> - simpler to implement,
> - more flexible and
> - more efficient in nearly all situations because it allows full caching of the stack on machines that lack coherent-caches (the difference in performance is as much as a factor of two on the TC2000 multiprocessor).
>
> In addition, this work shows how to efficiently implement two important language features in the presence of futures: dynamic scoping and first-class continuations. An efficient polling method designed to support message-passing is also described and evaluated.

This thesis provides a detailed account of this result.

1.1 Motivation

As applications become bigger and more demanding, it is hard to resist the seductive qualities associated with parallel processing. All too often however, application writers are disillusioned when they discover that their carefully rewritten application running on a parallel computer is barely faster, if not slower, than it was when running on a cheaper uniprocessor machine.

Poor performance can be caused by a combination of factors. The degree of parallelism in the algorithms is one of the most important factors because it puts a strict upper bound on the performance achievable by the program. Some algorithms have a limited amount of parallelism and thus it is not possible to increase performance beyond a certain size of machine. Moreover, even algorithms that scale up well with the size of the machine, i.e. yield a speedup roughly equal to the number of processors, may still have poor absolute performance if the parallel algorithm's "hidden constant" is large when compared to a sequential algorithm.

Another factor is the "technological lag" that the hardware of parallel machines often suffers. This is due to the smaller market and longer design times of parallel machines when compared to mainstream uniprocessor machines. This lag can be expected to decrease as parallel systems become more common.

The importance of these two factors can be minimized to some extent by careful algorithm design and coding and the use of state of the art hardware. However, there still remains another hurdle to overcome: the inherent inefficiency of the language implementation. Clearly, the language features needed to support parallelism must be implemented well to exploit the concurrency available in the application. It is just as important, however, for the sequential constructs to be efficient since they account for a high proportion of a program's code. There is little incentive to use a parallel machine with 10 processors if the implementation runs sequential programs on one processor 10 times slower than when a non-parallel language is used. This explains the lack of popularity of interpreter based implementations of Multilisp which run purely sequential code much slower than compiler based implementations of Lisp. Interestingly, the language implementations with poor absolute performance usually have excellent relative performance (i.e. self-relative speedup). This is because the aspects of the system that are critical to performance, such as memory latency and task spawning costs, are masked by the huge overhead of interpretation (usually a factor of 10 to 100 times slower than compiled code).

Absolute performance is a major concern in this thesis. For this reason, the Multilisp implementation techniques proposed here are evaluated in the context of a "production quality" implementation. To perform experiments, a highly efficient Scheme compiler called Gambit [Feeley and Miller, 1990] is used as a platform into which the implementation techniques are integrated and tested. This is to ensure that the setting is realistic and that performance-critical issues are not overlooked. Typically the code generated by Gambit for sequential programs is only about 20 percent slower (but sometimes faster) than code generated by optimizing C compilers for equivalent C programs. Multilisp is a sufficiently general programming language to be considered as a substitute for conventional languages for many sequential programming tasks. The results of this thesis will make it even more attractive to choose Multilisp over other languages since it also allows efficient parallel programming.

1.2 Why Multilisp?

Supercomputers have traditionally been employed for scientific purposes so it isn't surprising that numerical applications have been the focus of most of the parallel processing research. However, the need for high-performance is no longer bound exclusively to scientific applications as time-consuming symbolic applications become more widespread. These include applications such as expert systems, databases, simulation, typesetting, compilation, CAD systems and user interfaces.

The growing need for high-performance parallel symbolic processing systems is the initial motivation for this work. Multilisp suggests itself naturally since it is a member of the Lisp family of symbolic processing languages. It was designed by Halstead [Halstead, 1984] as an extension of Scheme with a few additional constructs to deal with parallelism. The most important of these is the *future* special form whose origin can be traced back to [Baker and Hewitt, 1978].

From its inception, the purpose of Multilisp has been to provide a testbed for experimentation in the design and implementation of parallel symbolic processing systems. Through the years it has evolved along several distinct paths to accommodate novel uses of the language. The first implementation of Multilisp was "Concert Multilisp" which ran on a custom designed multiprocessor [Halstead, 1987, Halstead *et al.*, 1986]. Multilisp's model of parallel computation has become increasingly popular and some of its features have now been adopted by other parallel Lisp systems. This includes both academic research systems such as QLisp [Gabriel and McCarthy, 1984, Goldman and Gabriel, 1988], MultiScheme [Miller, 1987, Miller, 1988], Mul-T [Kranz *et* *al.*, 1989], Gambit [Feeley and Miller, 1990], PaiLisp [Ito and Matsui, 1990], Spur Lisp [Zorn *et al.*, 1988], Butterfly portable standard lisp [Swanson *et al.*, 1988] and Concurrent Scheme [Kessler and Swanson, 1990, Kessler *et al.*, 1992] as well as commercially available systems such as BBN Lisp [Steinberg *et al.*, 1986], Allegro Common Lisp [Fra, 1990], and Top Level Common Lisp [Murray, 1990]. The future construct is actually quite general and it has been used in more conventional languages such as C [Callahan and Smith, 1989].

1.3 Fundamental Issues

Assuming that speed of computation is the main objective, the job of a Multilisp implementor can be seen as an optimization problem constrained by three factors

1. The semantics of the language.
2. The characteristics of the target machine.
3. The expected use of the system (i.e. applications).

Each instance of these factors defines a particular *implementation context*. It is the task of the designer to devise the most efficient implementation strategies that correctly realize the given language semantics on the target machine. It is also important to consider the target applications because it is through these that the features of the system that are most critical for high performance can be identified. They also form the ultimate measure of success of an implementation as a whole.

To explore the entire spectrum of implementation contexts for Multilisp would be a daunting task well beyond the scope of this work. Rather, contexts that are most likely to be useful in the present or the near future are examined. Emphasis is put on language features, multiprocessor architectures and programming styles that have acquired some popularity. The semantics of Multilisp and applications are discussed in greater depth in Chapter 2.

1.4 Architecture

Inherent limitations of the target machine are inevitable facts of life for the implementor of any language. To adequately address the issue of performance it is crucial to determine the salient features and weaknesses of the target architecture. This is especially true for parallel machines because of the vast disparity in parallel architectures.

1.4.1 Shared-Memory MIMD Computers

The multiple instruction stream, multiple data stream (MIMD) shared-memory multiprocessor computer is used as the target architecture for this work. This choice is fueled by on the one hand, the popularity and availability of these machines, and on the other, the similarity with the programming model adopted by Multilisp.

There are two major architectural requirements imposed by Multilisp. The first is the possibility for processors to act independently from one another. This is needed because Multilisp expresses parallelism through control parallelism, that is, it is possible to express concurrency between heterogeneous computations. Separate instruction streams operating on separate data are thus needed to execute these computations in parallel. The second requirement is the existence of a shared memory. In Multilisp, as in most other Lisps, all objects exist in a single address space that is visible to all parts of the program. There are no *a priori* restrictions on which procedure or tasks can access a given object.

The shared-memory architecture has been severely criticized by some. The most important objection is that the cost of accessing the shared memory must grow with the size of the machine. Thus, large machines will suffer from high latencies for references to shared memory.

This fact is duly acknowledged but must be put in perspective. Programs which offer a limited amount of parallelism only need to be run on machines whose size matches that parallelism. Secondly, the existence of a shared memory does not imply that the programs make an important use of it. Message-passing paradigms can easily and efficiently be implemented on top of a shared memory (for example, see [LeBlanc and Markatos, 1992]). However, implementing shared memory on conventional message-passing machines is impractical because shared-memory operations are usually fine grained whereas message-passing operations are typically optimized to manipulate large chunks of data. Programs with irregular and dynamically changing communication patterns have a legitimate need for shared memory. These programs are often found in symbolic processing applications which need to traverse linked data structures such as lists, trees, and graphs. Implementing these programs on a message-passing machine would be prohibitively expensive. Finally, it is expected that scalable caching techniques will hide the high latencies of large shared memory to some extent. Caching issues are explored

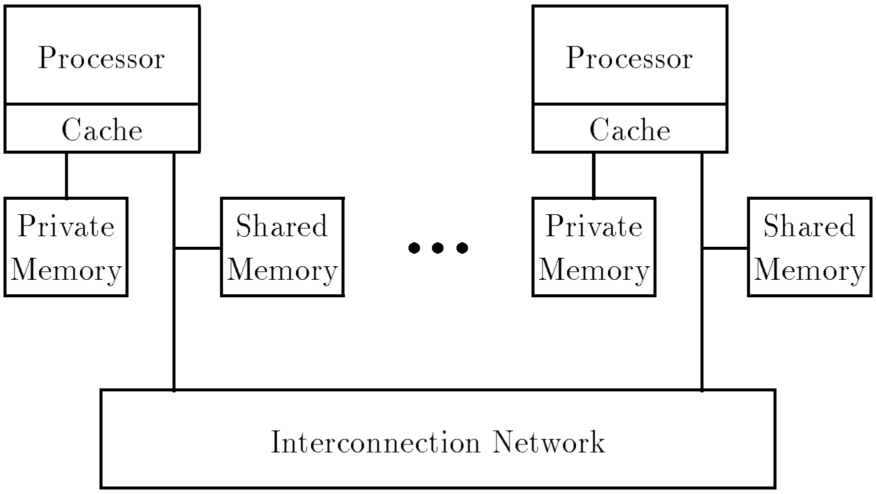


Figure 1.1: The shared-memory MIMD computer used in this thesis.

later in this chapter.

1.4.2 Non-Uniform Memory Access

The model of the shared-memory MIMD architecture used in this thesis is shown in Figure 1.1. A machine is composed of a number of processing nodes each of which has a processor and three forms of memory: cache memory, private memory and shared memory. Each processor has direct access to its own private and shared memory (i.e. *local* memory) and, through the use of the interconnection network, has access to the shared memory of other processors (i.e. *remote* memory). The shared memory is physically distributed across the machine while private memory is only visible to its associated processor.

This is a non-uniform memory access (NUMA) architecture because the cost of memory references is not constant. The cost depends on the type of memory being referenced and its distance from the processor. A reference to the cache is thus cheaper than a reference to local memory, which in turn is cheaper than a reference to remote memory. The NUMA model is interesting because it reflects realistic properties of the architecture as explained next.

1.4.3 Sharing Data

An important characteristic of data is the extent to which it must be shared. The following classification will be used for the different types of data

- **Private data** is data that does not need to be communicated to other processors. A simple example of private data is temporary values which are produced and used by the same program section.
- **Single writer shared data** is accessible to more than one processor but it is only mutated by a distinguished processor, the *owner* of the data.
- **Multiple writer shared data** is accessible to more than one processor and can be mutated by any of these processors.

These types of data have different storage requirements. Private data is the least restrictive (it could reside in the same storage as shared data) and multiple writer shared data is the most restrictive. These differences are a source of optimization for the architecture which can implement each type in a different way (and at a different cost). Thus, computers are often designed with various forms of private storage. Since a processor has exclusive access to this storage it can be implemented efficiently because there is no need for an arbitration mechanism or multiple data paths. The processor's registers are an extreme instance of private storage. Shared data is more expensive because it must be stored in a location that is accessible to all processors. Single writer and multiple writer shared data are distinguished because they offer different caching possibilities.

1.4.4 Caches

Caches are a well known mechanism to enhance the performance of memory. A property shared by almost all programs is that memory references are unevenly distributed. A large proportion of all references are to a small proportion of the data. This observation has lead to the design of multilevel memory systems. The idea is to place frequently accessed data in a fast memory, a *cache*, in order to reduce the average time needed for a reference. If the cache is large enough and the application's reference pattern is well behaved then the cache will service most of the references. A memory hierarchy can have several levels of caches but only a single one will be considered here.

Caches are quickly becoming a necessity to fully harness the power of modern processors. Current RISC processors have a cycle time that is much smaller than the fastest memory chips. Processors with a 1 nanosecond cycle time will soon be available but it is unlikely that the speed of large RAM chips will ever be close to that of the processor (for example DRAM chips currently have a 25 nanosecond cycle time at best). Cache memories are much faster than main memory because, due to their small size, they can be put on the same chip as the processor (or at least close to it) and it is permissible to use faster circuitry even if it is more expensive. The speed difference between these two types of memories varies from system to system but it is not uncommon for cache memory to be 5 to 20 times faster than main memory. Clearly, it is a good idea to design a system so that it maximizes cache usage. The benefits of caching on a range of programs is explored further in Chapter 3.

An important feature of caches is that they operate automatically. The programmer does not have to explicitly state where a particular piece of data should go. The accesses to memory are monitored and a copy of the frequently accessed data is kept in the cache. The first reference to a piece of data that is not in the cache (i.e. a cache *miss*) actually references the memory but subsequent references are potentially much faster because a copy has been put in the cache. When space is needed in the cache, older pieces of data are selectively purged from the cache according to a particular replacement policy (e.g. random or least-recently used (LRU)).

The performance of a cache depends on h, the probability of a cache hit (also called the *hit rate*), and L_{cache} and L_{main}, the latency of an access to the cache and to main memory respectively. The average access latency $\overline{L}_{mem}$ is given by

$$\overline{L}_{mem} = hL_{cache} + (1-h)L_{main}$$

Clearly, a high hit rate is advantageous since a value near one makes it appear as though the memory can respond at the speed of the cache. There are many ways to improve the hit rate. The size of the cache can be increased. Given the high cost of cache memory this may be a cost effective solution only up to a certain point. Another technique is to reorganize the program so that data references to a particular datum are closer in time. The probability of a datum being resident in the cache is higher if it has been referenced recently (and even more so if LRU replacement is used). Finally, it is sometimes preferable to disable the caching of data whose referencing pattern is such that it does not gain much by caching. Caching such data is detrimental because it causes the frequently used data to be purged from the cache, thus decreasing the hit rate.

Two caching strategies have been popular in uniprocessor computers: copy-back and write-through caching. These strategies differ in how writes to memory are handled.

- **Copy-back caching** handles a write by only modifying the copy in the cache. The memory will eventually receive the correct value when the datum is purged from the cache after a cache miss (this is called a *writeback*). The expense of writes is thus attributed to cache misses. If there are very few cache misses, writes to memory are essentially the same cost as reads.

- **Write-through caching** bypasses the cache and performs the write to main memory. However, the state of the cache is modified to reflect the new content of memory. If the address being written to is resident in the cache it is simply updated. Otherwise, the datum is added to the cache (most probably causing an entry to be purged)[1]. In addition to h, L_{cache} and L_{main}, the performance of write-through caching depends on the read ratio r (the proportion of all memory references which are reads). The average access latency for write-through caching is thus

$$\begin{aligned}\overline{L}_{mem} &= r(hL_{cache} + (1-h)L_{main}) + (1-r)L_{main} \\ &= rhL_{cache} + (1-rh)L_{main}\end{aligned}$$

  Note that here h is the hit rate for reads only. The two caching methods have the same performance when $r = 1$ but write-through caching quickly degrades as the number of writes increases.

1.4.5 Memory Consistency

The notion of a single monolithic shared memory is a convenient abstraction to write and reason about programs. However, caching if not done properly may violate this abstraction because *memory consistency* between processors is not preserved. For private data there is no consistency problem caused by caching since all references go through the cache. For single writer shared data it is possible to maintain consistency by using write-through caching. The processor owning the data uses write-through caching and the readers disable the caching of the data. Consistency is preserved because the memory always has the correct value for the datum and the readers always access the

[1] The datum could also be disregarded (i.e. not entered in the cache). This might be preferable for applications which rarely read the locations recently written to (such as when initializing or updating a large data structure).

memory when they reference the datum (of course, this means that only the owner of the data benefits from the cache). Unfortunately, write-through caching by itself is not sufficiently powerful to maintain consistency for multiple writer shared data. The problem is that the perception of the memory state can be different from processor to processor if each one has cached the same datum in its own cache and mutated it in a different way. For example, under copy-back and write-through caching, when two processors A and B read variable x, a copy of x will exist in A's cache and another in B's. If A then mutates x, B still believes that x has the original value.

There are two approaches to the memory consistency problem. The first is to put the responsibility of consistency on the programmer or compiler by providing a less rigid consistency model. At appropriate points in the program special operations must be added to flush or invalidate some of the entries in the caches. In the terminology of [Gharachorloo *et al.*, 1991], the strictest consistency model is *sequential consistency*. In this model, memory behaves as though only one access is serviced at a time (i.e. accesses are sequential). Thus any read request returns the last value written. In *processor consistency* writes can be delayed an arbitrary (but finite) amount of time as long as the writes from any given processor are performed in the same order as they were issued by that processor (there is no ordering restrictions between processors). This model can be implemented more efficiently than sequential consistency because it allows some form of pipelining and caching of the writes. Machines implementing processor consistency usually have a "write barrier" instruction which waits until the memory has processed all of that processor's writes. The *weak consistency* and *release consistency* models [Dubois and Scheurich, 1990] are still weaker and more efficient. They guarantee consistency only at synchronization points in the program. In other words, lock and unlock operations (or similar synchronization operations) are barriers which wait until the memory has processed all pending transactions. In these models, reads and writes can be buffered between synchronization operations.

An orthogonal approach to the consistency problem is to design specialized hardware that maintains consistency between the caches and memory. In the previous example, this would mean that when A mutates x, the new value for x is written to memory (as in write-through caching) and B's cache and any other cache holding a copy of x is notified to either invalidate or update the appropriate entry. This is relatively easy to perform on bus-based architectures because all caches and memory are immediately aware of all transactions (they are directly connected to the shared bus). So called snoopy-caches [Goodman, 1983] are based on this principle. Unfortunately, bus-based architectures do not scale well because the bus has a limited bandwidth. Typically, bus-based machines are designed with just enough processors to match the bandwidth of the bus. For example the bus in the Encore Multimax can support up to 20 fairly low-power processors[2].

Maintaining consistency on scalable architectures is much harder. Currently, most scalable cache designs are based on directories [Censier and Feautrier, 1978]. With each datum is kept a list of the caches that are holding a copy of the datum and that must be notified of any mutation. If n processors are holding a datum in their cache then a mutation by one processor will require at least $n-1$ messages to be sent to notify the caches. The moment at which these notifications are sent depends on the consistency model being used. Scalable cache designs usually do not implement strict consistency in order to exploit buffering and pipelining of writes. The main drawbacks of directory based methods are the added memory needed for the directory and the added inter-cache traffic which reduces the effective bandwidth of the interconnection network. Fortunately, it seems that in typical applications most of the shared data is shared by a very small number of processors [Lenoski *et al.*, 1992, O'Krafka and Newton, 1990]. Limited directory caching methods, such as [Chaiken *et al.*, 1991], take advantage of this fact to reduce the space for the directory by only allowing a small number of copies of a datum to exist at any given point in time.

However, there are certain forms of sharing that inevitably lead to poor cache performance. One such case is when two or more processors are very frequently writing to the same memory location (perhaps to implement some kind of fine-grain communication through shared memory). This causes thrashing in directory based methods because a substantial amount of time is spent sending messages between the caches. This poor performance is not surprising since caches are helpful only if there is locality of reference to exploit. If the goal is to exchange data as quickly as possible between the processors, caching is of little use since network latency will be unavoidable.

The moral here is that specialized hardware for memory consistency is not the solution to all data sharing problems. Specialized hardware can only help if the program has well behaved data usage patterns. When designing algorithms it is unreasonable to assume an efficient consistent shared memory simply because the machine supports it in hardware. The costs will vary according to how the data needs to be shared. As a general rule, algorithms should be designed to promote locality of reference and rely as little as possible on a strict consistency model and on multiple writer shared data.

---

[2]It is interesting to note that even though it uses snoopy-caches the Multimax only implements weak consistency.

1.5 The GP1000 and TC2000 Computers

Data sharing issues play a central role in this thesis. The multilevel memory system of the architectural model chosen here (i.e. Figure 1.1) reflects the importance of data sharing issues by making the costs of sharing explicit. In this model caches do not automatically preserve consistency. It is only by segregating the various types of data and using the appropriate caching policy that consistency is maintained. It is assumed that the caches can operate in copy-back and write-through caching on selected areas of memory. Because private memory always contains private data, it is cached with the most efficient caching policy: copy-back caching. Single writer shared data is cached using write-through caching by the owner of the data and is not cached by the other processors. Finally, multiple writer shared data is not cached in any way.

This model is attractive because building such a machine is relatively inexpensive using current technology yet it has a high potential performance. Each node in the architecture corresponds roughly to a modern uniprocessor computer. The only extra hardware needed to build a complete machine is that for the interconnect and its interface to the processing nodes. The TC2000 computer [BBN, 1990], manufactured by BBN Computers and introduced in 1989, matches this structure very closely. A scalable multistage butterfly network is used for the interconnection network. There is a single local memory per node that is partitioned into shared and private sections by system calls to the operating system. Other system calls allow the selection of the caching policy for each memory block allocated. The GP1000 computer [BBN, 1989], also by BBN, has a very similar architecture but uses older technology (the TC2000 uses M88000 processors rated at 20 MIPS whereas the GP1000 uses M68020 processors rated at roughly 3 MIPS). The GP1000 also suffers from a slower interconnection network (approximately half the bandwidth of the TC2000) and the lack of a data cache[3]. These two computers are used throughout the thesis to do measurements and to compare different implementation strategies. Because scalability is an important issue, large machines were used: a 94 processor GP1000 (at Michigan State University) and a 45 processor TC2000 (at Argonne National Laboratory). To serve as a guide, the costs of the memory hierarchy for these computers is given in Table 1.1. The timings correspond to the latency for referencing a single word for each level of the hierarchy[4]. Note that the

[3] However, each processor has a small instruction cache.

[4] These costs were measured with benchmarks specially designed to test the memory. As reported in [BBN, 1990], the timing depends on many parameters such as the caching policy in use, the type of access (read or write), the size of machine and the contention on the interconnection network. The timings in the table are the average time between reads and writes, caching was inhibited when measuring local and remote memory costs.

Machine	Latency in μsecs			Relative latency		
	Cache	Local	Remote	Cache	Local	Remote
GP1000		.475	5.750		1.0	12.1
TC2000	.150	.575	2.400	1.0	3.8	16.0

Table 1.1: Costs of memory hierarchy for the GP1000 and the TC2000.

cache on the TC2000 is faster than local memory by only a factor of 3.8. Many systems currently have caches that perform much better than this. Also note that the latency of a butterfly network grows logarithmically with the number of processors. Machines with several hundred processors would thus have roughly the same relative costs for the memory hierarchy.

1.6 Memory Management

The design of a high-performance Multilisp system is a complex task where many, often conflicting, issues have to be addressed. Clearly an implementor must worry about how to best implement the parallelism constructs themselves, but it is important to realize that the support of parallelism has an impact on the sequential parts of the language as well. High-performance techniques used in uniprocessor implementations of Lisp cannot always be carried over to Multilisp as is, either because they become inefficient in a multiprocessor environment or, even worse, they do not work at all due to the presence of concurrency.

As should be clear from the previous section, one of the most important problems to tackle for a NUMA architecture is that of memory management. Lisp, and symbolic processing in general, relies heavily on the manipulation of data structures and on their dynamic creation. The costs of allocating, referencing and deallocating objects are thus major components of the overall performance of the system. For a language like Multilisp where data is implicitly shared, memory management is tricky to implement efficiently because, in general, data must be accessible to all the processors and be mutable by all the processors.

In order to keep the reference costs low, a memory management policy for a NUMA architecture must strive to physically locate the shared data close to the processor that needs to access the data most frequently. For the TC2000, this means that data should reside in the cache or the local memory of the processor most frequently accessing the data. This is the *proximity* issue.

Another important goal is to arrange the data so that *contention* is minimized. Contention occurs when more than one processor is trying to access the same shared resource (such as a memory bank or a path in the interconnection network). The resource becomes a bottleneck to performance because requests must be serviced sequentially. Contention can be inherent in the algorithm (when expressed explicitly as a critical section) but it can also appear insidiously because of some particularity of the language implementation or target machine. For example, a simple allocation strategy for vectors is to reserve the space for all elements in a given memory bank. In such a situation, the references to different elements of the vector are forced to be done sequentially even if they are all logically concurrent. The same problem occurs when unrelated data values are referenced simultaneously and they happen to have been allocated in the same memory bank. Certain shared-memory machines, such as the BBN Monarch [Rettberg *et al.*, 1990] and IBM RP3 [Pfister *et al.*, 1985], avoid some contention problems by using "combining" networks which combine similar requests to the same memory location (e.g. read, clear, add a constant). However, combining networks are ineffective for contention to unrelated data. A simple and general approach to minimize contention is to scatter the data among all the memory banks. If the referencing pattern is uniformly distributed, the probability that two references are to the same memory bank (out of n memory banks) is $\frac{1}{n}$. Unfortunately this strategy compromises proximity because the probability that a reference is to remote memory is $\frac{n-1}{n}$ which approaches 1 for a large machine.

There are basically two extreme ways in which the proximity and contention issues can be handled. The placement of objects in memory can be left to the user or be done automatically by the implementation. User controlled placement can be expressed in several ways including declarations and the use of specialized data manipulation operators. Automatic placement has the advantage of preserving the high-level nature of the language, that is, the user does not need to know the details of the target machine. However, there is just so much that can be expected of automatic techniques and, at least for special purpose applications, the user can have knowledge of the memory reference patterns that are next to impossible for the compiler to infer automatically.

It is important to distinguish two classes of data. *User data* is data explicitly created and referenced by the data manipulation procedures of the language (e.g. `cons`, `car` and `set-car!`). *Internal data* corresponds to data used internally by the implementation to support the language. Internal data includes

- Environment frames
- Continuation frames
- Closures
- Cells (for mutable variables)
- Global variables
- Tasks
- Constants
- Program code

Because these data structures are used in well defined ways under the control of the implementation, it is possible to design special purpose memory management policies for them. For instance, local, contention free accesses to the program code and constants are possible if they are copied to the private memory of each processor when the program is loaded.

Both user data and internal data are important to optimize in a system. However, this thesis concentrates on the management of internal data, and in particular the data structures that are involved in dynamic partitioning. The placement of user data is not considered here.

1.7 Dynamic Partitioning

One of the most fundamental operations performed by any parallel system is the distribution of work throughout the system. Each processor has to be aware of the computations it is required to do and at what time. The overall goal is to have the best usage of the processing resources, that is to have the greatest number of processors doing useful things. *Partitioning* consists of dividing the program's total workload into smaller tasks that can be assigned to the processors for concurrent execution. A prerequisite to partitioning is of course knowing which pieces of the program *can* be done concurrently. Since in Multilisp concurrency is stated explicitly by the user, it will be assumed here that the only source of concurrency is the future construct[5].

[5]Thus, in the expression `(+ (* x 2) (* y 2))`, the concurrency possible in the evaluation of the arguments to `+` will be disregarded because it is not expressed with a future.

Partitioning can be done once and for all before the program is run. This *static partitioning* has the advantage of being simple to conduct when the program naturally decomposes into a fixed number of equal sized tasks. It also permits some compilation optimizations because important information, such as the particular assignment of tasks to processors, the inter-task communication pattern, and the type of communication, can sometimes be known at compile time. Programs with a regular computational structure are good candidates for static partitioning.

Dynamic partitioning relegates the partitioning decisions to when the program is running. This approach is more general because it can be applied to programs with complex concurrency structures and also to programs whose concurrency is dependent on the input data set. This generality is needed for Multilisp because the arbitrary concurrency structures expressible with the future construct cannot be handled by static partitioning methods. Another advantage is that better partitioning decisions can be made because more information is available at run time. The size of the machine (number of processors and memory size) is an important parameter that may not be known at compile time. There are other equally important, but more subtle, partitioning parameters that are only available at run time. For example, the number of active tasks and idle processors at a given point in time are useful indicators of partitioning needs.

In a way, dynamic partitioning has the ability to adapt to its execution environment whereas static partitioning is stuck with irreversible compile time decisions that are based on predictions of what the execution environment will be. Adaptability is crucial to account for the varying computational nature of certain programs. Parallel sort is a good example to illustrate this point. The sort may have more or less concurrency depending on the data set size (i.e. the number of items to sort) and the cost of comparing two items. These parameters can vary in the same program if the sort is called multiple times. Concurrency can also be affected by the initial ordering of the items. The sort algorithm might degenerate to a sequential algorithm for some orderings and be perfectly parallel for others. Large programs add another dimension to the argument. Large programs are typically composed of several smaller independent modules. Concurrency can occur inside a module, between purely sequential modules, and also between internally concurrent modules. It is quite possible that an internally concurrent module, such as parallel sort, has to execute by itself at some point and concurrently with other modules at some other point. The partitioning requirements may vary greatly between these two cases. At one extreme, no partitioning is needed for the sort if the other modules are doing long sequential computations and there happen to be $n-1$ of them on an n processor machine.

The main inconvenience of dynamic partitioning is that it adds a run time overhead. Dynamic partitioning is "administrative" work that gets added to the operations strictly required by the program (i.e. the *mandatory* work). Tasks are created to enable concurrent execution, but each task created adds a cost, in time and space, because its state has to be maintained throughout its life (this includes task creation, activation, suspension and termination). A dynamic partitioning strategy must find some compromise between the benefit of added concurrency and the drawback of added overhead. Some have avoided this problem to some extent by relying on specialized hardware to reduce the cost of managing tasks. Dataflow machines [Srini, 1986, Arvind and Nikhil, 1990] and multithreaded architectures [Halstead and Fujita, 1988, Nikhil *et al.*, 1991, Agarwal, 1991] fall in this category. However, software methods are attractive because they offer portability and low hardware cost. This thesis explores software methods for lowering the cost of task management in the context of the Multilisp language.

In a strict sense, partitioning only refers to the way the program gets divided up into tasks. This definition is not very useful for Multilisp because each evaluation of a future leads to the creation of a new task; there are no partitioning decisions to be made. However, choices are available at another level. There can be several representations for tasks, each having its own set of features and management costs. The appropriate representation for a particular task will depend on many factors but as a general rule it will be best to select the one with the lowest cost that has all the required features. Partitioning has a broad sense in this thesis. It refers to the choice of representation that is used for the tasks in the program and the way that they are managed.

An important parameter affecting the performance of dynamic partitioning is the *granularity of parallelism* (G) of the program. G is defined as the average duration of a task

$$G = \frac{T_{seq}}{N_{task}}$$

Here N_{task} is the total number of tasks created by the program and T_{seq} is the duration of the program when all task operations are removed (i.e. T_{seq} is the mandatory work). When the task operations are present, the work required for the program is T_{seq} plus some task management overhead (T_{task}) for each task created

$$T_{par} = T_{seq} + N_{task} T_{task}$$

T_{task} contains the time to create, start and terminate a task. The total work required to run the program on an n processor machine, $T_{total}(n)$, will be T_{par} plus some amount that accounts for all other parallelism overheads including the costs of transferring tasks between processors, synchronizing tasks, sharing user data, and being idle. The run time on n processors is thus $\frac{T_{total}(n)}{n}$. The *efficiency* (E) of the processors is the proportion of the time they spend doing mandatory work. G and T_{task} are important parameters because they put an upper bound on efficiency

$$E = \frac{T_{seq}}{T_{total}(n)} \leq \frac{T_{seq}}{T_{seq} + N_{task}T_{task}} = \frac{1}{1 + \frac{T_{task}}{G}}$$

This equation suggests that efficiency is a function of the relative size of G with respect to T_{task}. Higher efficiency can be obtained either by increasing G or decreasing T_{task}.

1.7.1 Eager Task Creation

A well known dynamic partitioning method is *eager task creation* (ETC). Its main advantage is simplicity. Only a single representation for tasks exists in ETC: the heavyweight task object. Unfortunately, the task management cost for heavyweight tasks is relatively high (on the order of hundreds of machine instructions). A coarse granularity is thus required to get good performance. For example, the granularity must be at least in the hundreds of machine instructions to achieve better than 50% efficiency. This makes the programming task that much more difficult because granularity must be taken into account when designing programs. Moreover, coarse grain programs have less parallelism (fewer tasks) so there is a risk that they will only perform well on small machines. Finally, some programs are hard to express with coarse grain parallelism.

1.7.2 Lazy Task Creation

A more efficient partitioning method called *lazy task creation* (LTC) is explored in this thesis. In addition to the heavyweight task representation, LTC uses a much cheaper lightweight representation. The method is described in detail in Chapter 3 but a general description is given here to explain some of the issues.

LTC lowers the average task management cost by creating only as many heavyweight tasks as necessary to keep all processors working. To do this, each processor maintains a local data structure, the *lazy task queue* (LTQ), that indicates the availability of tasks on that processor. When the program asks for the creation of a task, the LTQ is updated to indicate the presence of this new task. This operation is efficient because a lightweight task representation is used. A lightweight task preserves enough information to recreate the heavyweight task later on, if needed. Each entry in the LTQ is a pointer into the stack, marking the boundary of that task's stack. The beauty of LTC is that, when the processor becomes idle it can get work from its own LTQ at a low cost and completely avoid the creation of a heavyweight task. When the LTQ is empty, the processor must instead find a task to resume from some other processor's LTQ. It is only in this case that a high cost is paid to create a heavyweight task and transfer it between processors.

Shared-Memory Protocol

But how exactly does this interaction take place? The protocol adopted in [Mohr, 1991] uses a shared-memory paradigm. The stack and LTQ of all processors are directly accessible to all processors (i.e. they are shared data). When processor A needs to get work from processor B, it directly manipulates B's LTQ and stack to extract a task. This approach has unfortunate consequences. First of all, access and mutation of the LTQ must be arbitrated because several processors may be competing for access. This means that the cost of lightweight task creation is higher than might have been expected because synchronization operations are needed to ensure that accesses to the LTQ are mutually exclusive. This may be tolerable in certain contexts since the overhead cost will be high only for parallel programs with fine grain parallelism. The second consequence is much more serious. The protocol assumes that the stack and LTQ are in consistent memory. Therefore, they cannot be cached as efficiently as private data. This can have a severe impact on performance because the stack is one of the most intensively used internal data structures. The cost is also unrelated to the use of parallelism; sequential programs will suffer just as much as parallel ones. It is preferable for the stack to be a private resource so that copy-back caching can be used (as is the case for sequential implementations of Lisp).

Message-Passing Protocol

The stack and LTQ can be made private by adopting a message-passing protocol for work distribution. When A needs to get work from B, it sends a request for work to B. Upon receiving this message, B checks its LTQ for an available task and, if one is available, sends it back to A. Since the LTQ and stack are only accessed locally there is no need for synchronization operations when updating them. Lightweight task creation is thus cheaper than with the shared-memory protocol. This allows very fine grain parallelism to be efficient. Sequential code also benefits because copy-back caching can now be used for the stack.

Although it is promising, the message-passing protocol introduces some new issues. How is the communication mechanism implemented and what is its cost? The latency of the communication is also a factor. Can the processor respond fast enough to minimize the idle time of the requesting processor?

1.8 Overview

The thesis is organized in 6 chapters. Chapter 2 gives a description of the Multilisp language and its traditional implementation using ETC. Some fine points of its semantics are discussed to clarify the constraints that must be met by the partitioning methods. Finally, the benchmarks used for later experiments are presented.

Chapter 3 provides a detailed description of the shared-memory and message-passing implementations of LTC. It is shown how support for dynamic scoping, continuations and fairness can be added to LTC. This chapter also examines the memory usage characteristics of the benchmark programs to evaluate the benefits of caching.

Chapter 4 concentrates on the communication mechanism required by the message-passing protocol. An efficient software implementation is described and evaluated.

Chapter 5 compares the two LTC protocols. The performance of both protocols is measured on several benchmarks and under numerous conditions.

The closing chapter summarizes the results of the thesis and suggests some future lines of research.

Chapter 2

Background

Before discussing the implementation of the future construct, it is necessary to establish the set of features that must be supported by the implementation. This is particularly important because there is no formal standard for the Multilisp language; nearly every implementation has its own peculiarities. This thesis takes the pragmatic view that Multilisp is defined by the set of features common to a number of implementations.

Choosing the set of supported features is a delicate process that is similar in many ways to language design itself. The set should not be limited to the features that are strictly common to all implementations as this would be ridiculously restrictive. Features that have acquired a certain level of acceptance in the field should also be included. On the other hand, it is wise to select a small set of features that interact in a coherent, well defined way in order to provide a programming model with few surprises.

The chapter starts off by giving a definition of the Multilisp semantics targetted by this work. This includes the future construct common to all Multilisp implementations and also two useful features of sequential Lisps which pose special problems in a parallel setting: dynamic scoping and first-class continuations. The ETC implementation of this semantics is then presented. The chapter ends with a description of some Multilisp programs later used to evaluate and compare various implementation strategies.

2.1 Scheme's Legacy

Multilisp inherits its sequential programming features from the Scheme dialect of Lisp [IEEE Std 1178-1990, 1991]. Scheme was designed to be a relatively small and simple language with exceptional expressive power. There are few rules and restrictions for forming expressions in Scheme, yet most of the major programming paradigms can conveniently be expressed with it. This is not surprising since the language is based on the theory of the lambda calculus.

There are six basic types of expressions in Scheme: constant, variable reference, assignment, conditional, procedure abstraction (`lambda`-expression) and procedure call. All the other types of expressions can be derived from the basic types and this is in fact how they are defined in the standard [IEEE Std 1178-1990, 1991, R4RS, 1991]. Being able to reduce a program to the basic expressions is helpful both as an implementation technique and as a means to understand programs and prove some of their properties. It is also a considerable advantage for any extension effort, such as Multilisp, because the interaction of the extensions with the language can be more carefully analyzed by limiting the study to the basic types of expressions.

Scheme offers a rich set of data types including numbers, symbols, lists, vectors, procedures, characters and strings. There are also several predefined primitives to operate on these data types including procedures to create, destructure and mutate data. Although Lisp-like languages have a historical inclination towards symbolic processing applications, the elaborate support of numerical types in Scheme makes it a candidate for numerical applications as well.

There has been an effort in Scheme to make the language as uniform as possible. All types of objects in Scheme share some basic properties that make them *first-class* values. Any object can be: used as an argument to procedures, returned as the result of procedures, stored in data structures, and assigned to variables. Departing from Lisp tradition, Scheme evaluates the operator position of procedure calls like any other expression and does not impose any particular ordering on the evaluation of arguments to procedures. The `let` and `let*` special forms are handy to force a particular ordering when it is needed (this is what is done in the examples).

Objects have unlimited extent. They conceptually exist forever after they have been created. In general this means that objects must be allocated in the heap. When there is no space left in the heap, the system automatically invokes the process of *garbage collection* to reclaim the heap space allocated to objects that are no longer needed for the rest of the computation. In certain circumstances it is possible at compile time to detect that an object is no longer needed past a certain point in the program. The compiler can then use a specialized allocation policy (such as a stack) and explicitly perform the deallocation. This reduces the frequency and cost of garbage collection.

Scheme relies solely on *static scoping* as a method to resolve variable names. An identifier refers to the variable with the same name in the innermost block that lexically contains the reference and declares the variable. If no such block exists, the identifier refers to a variable in the global environment. This naming rule corresponds to that of block structured languages such as Pascal and Algol 60. *Dynamic scoping* is an alternative method that has been traditionally used in other Lisps. The identity of variables is not based purely on the lexical characteristics of the program (available at compile time), but rather depends on the control path taken by the program at run time. Although dynamic scoping has its specialized uses (e.g. see Section 2.7), its pervasive use is not generally viewed as promoting modularity. In addition, efficient implementation of dynamic scoping is often based on shallow binding, a strategy that is not well suited for parallel execution. Static scoping permits the use of certain compilation techniques, such as data flow analysis, that are difficult or impossible to perform with dynamically scoped variables because the analysis would have to be done on the entire program.

In Scheme, procedures are viewed as first-class values and thus have the same basic properties as the other data types. With first-class procedures many programming techniques are easily implemented. Higher order functions, lazy evaluation, streams and object-oriented programming can all be done using first-class procedures (for example see [Adams and Rees, 1988, Friedman *et al.*, 1992]). Procedures created by `lambda`-expressions are usually called *closures* to distinguish them from predefined procedures. The static scoping rules require all closures to carry, at least conceptually, the set of variables to which they might refer (the *closed* variables). Consequently, variables have unlimited extent and cannot generally be allocated in a stack-like fashion as in more conventional languages. Closures pose additional problems in a parallel setting. Because closures are just another data structure, contention may happen if several processors are simultaneously calling the same closure. A typical situation would be the parallel application of a closure to a set of values. Some optimizations can avoid contention in some cases. For example, closures with no closed variables, such as globally defined procedures, are essentially constant so they can be created and copied to all processors when the program is loaded. Lambda-lifting can also eliminate the need to create closures by explicitly passing the closed variables between procedures. Both of these techniques are used in Gambit. However, the general case remains hard to solve as it is equivalent to the problem of data sharing. For this reason, true closures have been avoided as much as possible in the benchmarks.

In accord with the goal of simplicity, the only way to transfer control in Scheme is through the use of procedure calls. All types of recursion, whether they correspond to an iteration or not, are expressed as procedure calls. There are two types of calls. If the value returned by a call is immediately returned by the procedure containing the call it is a *reduction* call. Otherwise the call is a *subproblem* call. All implementations are required to be properly tail recursive. That is, they must guarantee that loops expressed recursively do not cause the program to run out of memory. In implementation terms, this means that reduction calls must not retain the current procedure's activation frame (the local variables and return address) past the actual transfer of control to the called procedure.

Scheme is a call by value or *applicative order* language. The evaluation of the program is forced to follow an ordering that evaluates all arguments to a procedure before the procedure is entered. The opposite policy, call by need or *normal order* evaluation, doesn't evaluate any of the arguments to a procedure when the procedure is called. Evaluation occurs when a *strict* operator, such as addition, needs the actual value. Data transfer operations such as parameter passing and creation of data-structures are not considered to be strict. Both policies have advantages. Programs using normal order evaluation sometimes terminate when their applicative order counterparts do not. On the other hand, applicative order is often more efficient. In Scheme, it is possible to get the equivalent of normal order evaluation by using the `delay` special form to delay evaluation and by redefining the primitive procedures so that they force the evaluation of the arguments in which they are strict[1]. The future construct is the dual of the `delay` special form giving eager evaluation instead of lazy evaluation.

Scheme supports various flavors of side-effects such as assignment, data structure mutation and input/output operations. Thus, it is considered to be an imperative programming language where sequencing of operations is a necessary concept. Nevertheless, Scheme contains a powerful functional subset which can be used for purely functional programming. Some algorithms are naturally expressed in a functional way, some others are expressed better with the use of side-effects. In Scheme, both paradigms can appear in the same program and the programmer can choose which best matches his needs at any given point. It is however a good idea to limit the scope of side-effects by hiding them through abstraction barriers. For example, a sorting procedure can have a functional specification even if it uses side-effects internally. In practice, it seems that Scheme favors a "mostly" functional style of programming where side-effects are used with discretion. This style of programming lends itself well to parallelism because subproblems are often independent and are thus possible targets for concurrent evaluation.

[1] `Delay` only exists in R^4RS [R4RS, 1991].

2.2 First-Class Continuations

Perhaps Scheme's most unusual features is the availability of first-class *continuation* objects. Continuations have been used in the past to express the denotational semantics of programming languages such as Algol60 and Scheme itself [R3RS, 1986, Clinger, 1984]. Most programming languages use continuations but they are usually hidden whereas in Scheme they can be manipulated explicitly. First-class continuations are useful to implement advanced control structures that would be hard to express otherwise.

Intuitively, a continuation represents the state of a suspended computation. The power of continuations stems from the ability to reinstate a computation at any moment and possibly multiple times. It is convenient to think of a continuation as a procedure that restores the corresponding computation when it is called. Often it is necessary to influence the computation that is being restored. This is done by passing parameters to the continuation. Continuations typically have a single parameter, the return value, but some continuations may take none or more than one parameter.

2.2.1 Continuation Passing Style

Continuations are best understood by examining the underlying mechanism of evaluation. Each expression in the program is the producer of a value that is to be consumed by some computation: the expression's continuation. For example, in `(f x)`, the procedure `f` is the consumer of the value produced by the expression `x`. Each expression can be viewed as being implemented by an "internal" procedure whose purpose is to compute the value of the expression and send it to the consumer computation. Thus, one of the parameters of this internal procedure is a continuation which takes a single argument: the value of the expression.

This model of evaluation gives rise to a programming style called *continuation passing style*, or CPS. CPS was originally used as a compilation technique for Scheme [Steele, 1978] but CPS is equally useful to explain how continuations work. The interest of CPS is that programs written in this style are expressed in a restricted variant of Scheme yet all Scheme programs can be converted to CPS. An important byproduct of CPS conversion is that procedure calls never have to return (they are always reductions) and can thus be viewed as jumps that pass arguments.

The CPS conversion process consists of adding a continuation as an extra argu-

```
(define (map-sqrt lst)
  (call-with-current-continuation
    (lambda (cont)
      (map (lambda (x) (if (negative? x) (cont #f) (sqrt x)))
           lst))))
```

Figure 2.1: Non-local exit using `call/cc`.

ment to each procedure call and adding a corresponding parameter to all procedures. Primitive procedures must also be redefined to obey this protocol. The continuation argument specifies the computation that will consume the result of the procedure being called. For subproblem calls, the continuation argument is a single argument closure representing the computation that remains to be done by the caller when the called procedure logically returns. For reduction calls, the continuation argument is the same as the caller's continuation (thus implementing proper tail recursion). Wherever a procedure would normally return a value other than by a reduction call, a jump to the continuation argument is performed instead.

In Scheme, access to the implicit continuation is provided by the predefined procedure `call-with-current-continuation`, abbreviated `call/cc`. A single argument procedure must be passed as the sole argument of `call/cc`. When it is called, `call/cc` takes its own implicit continuation, converts it into a Scheme procedure and passes it to its procedure argument. The CPS definition of `call/cc` is simply

```
CPS-call/cc ≡ (lambda (k proc) (proc k (lambda (dummy-k x) (k x))))
```

Note that there are two ways in which the captured continuation `k` can be invoked. Either `proc` calls the continuation it was passed as an argument or `proc` returns normally.

2.2.2 Programming with Continuations

Several control constructs can be built around `call/cc`. A typical application is for non-local exit and exception processing, which are normally done in Lisp using the special forms `catch` and `throw`. In Scheme, this can be done by saving the current continuation before entering a block of code. An exit from the block occurs either when the block terminates normally or when the saved continuation is called. An example of this is given in Figure 2.1. The procedure `map-sqrt` returns a list containing the square root of every item in a list but only if they are all non-negative. The value `#f` is returned if any item is negative. To do this, `map-sqrt` binds its continuation to `cont`. A call to `cont` thus corresponds to a return from `map-sqrt`. When a negative value is detected by `map-sqrt` the processing of the rest of the list is bypassed by the call `(cont #f)` which immediately causes `map-sqrt` to return `#f`.

`Call/cc` however is more versatile than Lisp's `catch` and `throw` because it does not restrict the transfer of control to a parent computation. Thus it is possible to directly transfer control between two different branches of the call tree. This characteristic can be exploited to implement specialized control structures such as backtracking [Haynes, 1986], coroutines [Haynes *et al.*, 1984] and multitasking [Wand, 1980]. A less frequent, but possible, use of continuations is to reenter a computation that has already completed (see [Rozas, 1987] for an application).

The generality of first-class continuations comes at a price: a more complex programming model. In many languages, including Lisp, procedure calls have dynamic extent. This means that every entry of a procedure is balanced by a corresponding exit (normal or not). This is not the case in Scheme because the computation performed in a procedure can be restarted multiple times, and thus a procedure can exit more than once even if it is called only once. Because the programmer's intuition often fails when dealing directly with continuations it is sometimes helpful to build abstraction barriers that offer restricted versions of `call/cc` (for example see [Friedman and Haynes, 1985]).

First-class continuations also cause an implementation problem. If procedures have dynamic extent, continuations can easily be represented by a single stack of control frames (i.e. return addresses). Control frames get allocated when procedures are called and deallocated when procedures return in a last-in first-out (LIFO) fashion. This form of garbage collection is possible because control frames cannot be referenced after the corresponding procedure returns. The unlimited extent of continuations in Scheme means that a more general garbage collection mechanism for control frames must be used because a procedure's control frame might still be needed after the procedure returns. At least in some cases, control frames must be allocated on the heap. A common implementation strategy is to allocate all control frames on the stack as though they had dynamic extent and to move them to the heap only when their extent is no longer known to be purely dynamic (usually at the moment a continuation is captured by a `call/cc`). This way, the efficiency of stack allocation is obtained for programs that do not make use of first-class continuations. This strategy is described in detail in Section 3.2.

The next section examines the problems that arise when continuations are used in a parallel setting.

2.3 Multilisp's Model of Parallelism

Parallel programming languages can be classified according to the level of awareness of parallelism required by the programmer when writing programs. At one end of the scale, there are languages with *implicit* parallelism that rely exclusively on the ability of the system to detect and exploit the parallelism available in programs. In these languages the compiler must analyze the program to determine what parts can and should be executed concurrently. In general this is a hard task for imperative languages because of the existence of side-effects. Even in the absence of side-effects, the compilation may be difficult if an algorithmic transformation is required to obtain a sufficiently parallel algorithm.

Multilisp is at the other end of the scale. Parallelism is explicitly introduced by the programmer through the use of the "future" construct. The future construct marks the parts of the program where concurrent evaluation is allowed. Of course this style has its price: the burden put on the programmer for specifying concurrency and the possibility of error (i.e. incorrectly specifying concurrency). The advantage of this approach is that it provides more control over the program's execution. The programmer can specify concurrency at places which might escape an automatic analysis and can choose to disregard some forms of concurrency if it is judged that the cost of exploiting the concurrency is greater than what is gained.

This level of control is useful for the programmer wanting to experiment with various ways of parallelizing a program. It is also appropriate when Multilisp is considered as the "object code" of a compiler for a higher level parallel language. Such a compiler could be aware of where parallelism is both possible and desirable and emit code with appropriately placed futures ([Gray, 1986] is a good example of this application).

2.3.1 `FUTURE` and `TOUCH`

Futures are expressed as `(FUTURE` *expr*`)` where *expr* is called the future's *body*. The future construct behaves like the identity function in the sense that its value is the value of its body. However, the body is conceptually evaluated concurrently with the future's continuation. The only restriction to this concurrency comes as a result of the ordering dependencies imposed by the strict operations in the program. When the value of a future is used in a strict operation, the operation can only be performed after the evaluation of the future's body. For example, in the expression

```
(let ((x (FUTURE (f 1))))
  (g (+ x (f 2))))
```

the evaluation of `(f 1)` is done concurrently with the evaluation of `(f 2)`[2]. Because `+` is a strict operation in both of its arguments, the addition and the call of the procedure `g` can only occur after the evaluation of `(f 1)` has completed.

As long as they respect the temporal ordering imposed by the strict operations, the operations required to compute the body of a future are subject to arbitrary interleaving with the operations performed by the future's continuation. Because Multilisp allows unrestricted side-effects, it is an indeterminate language. Separate runs of the same program can potentially generate different results. As a simple example consider the expression

```
(let ((x 0))
  (FUTURE (set! x 1))
  x)
```

The evaluation of this expression can either return `0` or `1` depending on whether the reference to `x` happens to be done before or after the assignment to `x`[3].

In certain circumstances a program needs to impose special control dependencies in addition to those given by the data dependencies of the program. Such control dependencies are only required in imperative parts of the program to enforce a certain ordering of side-effects. For example, it might be important to guarantee that some restructuring of a database has completed before some other processing of the database is performed. For this purpose, Multilisp provides the primitive procedure `TOUCH` that behaves like a strict identity function. `TOUCH` can be viewed as the fundamental "strictness" operation. All other strict operations use `TOUCH` internally.

In order to show clearly where the `TOUCH` operations are needed, the code examples and benchmark programs that follow include explicit calls to `TOUCH`.

---

[2] To be precise, the steps required to bind `x`, evaluate `g`, `+` and `x`, and enter the `+` procedure can also be done concurrently with the evaluation of `(f 1)`.

[3] Indeterminacy also exists in Scheme, but at a different level. In a procedure call, arguments and the operator position can be evaluated in any order, but sequentially (that is with no overlap in time). The following expression has 2 possible values: `0` and `1`.

```
(let ((x 0))
  (car (cons x (set! x 1))))
```

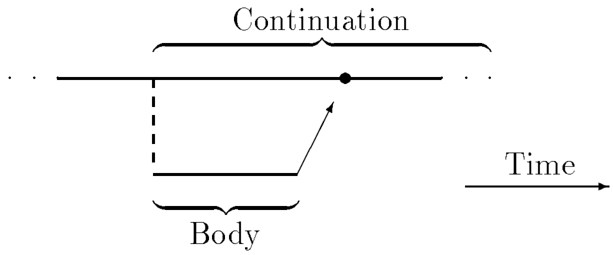
2.3.2 Placeholders

A more traditional description of futures consists of introducing a new type of object, the *placeholder*, that is used to synchronize the computation of a future's body with the touching of its value [Miller, 1987]. When a future is evaluated it returns a placeholder as a representative of the value of the body. A placeholder can be in one of two states. It is *undetermined* initially and for as long as the evaluation of the future's body has not completed. When the evaluation of the body is finished, the resulting value is stored in the placeholder object which is then said to be *determined*. Using placeholder objects, `TOUCH` has an obvious definition: if the argument is not a placeholder just return it, otherwise, wait until the placeholder is determined and then return its value.

It is important to understand that placeholders are used here as an artifice to explain how futures work. Although placeholders are commonly used in Multilisp systems, an implementation is free to choose any method that gives the same result. Even if placeholders are present in the system, the user can be totally unaware of their existence if the implementation does not provide constructs to manipulate them directly. This is the view adopted by Gambit.

2.3.3 Spawning Trees

It is sometimes useful to represent the effects of evaluating futures and touching placeholders by a diagram, the *spawning tree*, which shows the state of the concurrent computations as a function of time. A spawning tree resulting from the evaluation of a single future looks like



A computation is represented by a horizontal line whose extent corresponds to its duration. A dashed vertical line marks the evaluation of the future. At that point, a new computation corresponding to the body of the future is started. Arrows are used to express the data dependencies introduced by the `TOUCH` operation. An arrow links the computation that determined a placeholder with the computation(s) that touch(es) it (a computation can point to several others). The tail of an arrow indicates the point where a placeholder was determined whereas the head indicates the point where the

`TOUCH` was requested. If an undetermined placeholder was touched, the arrow will point backwards in time (indicating that the touching computation had to wait).

A second representation of spawning trees used here is as a rooted tree. Each node of the tree represents a future and the children of a node are the futures dynamically nested in the body of the corresponding future. The root of the tree corresponds to a virtual future in which the program is executed.

2.4 Types of Parallelism

Parallelism comes in many flavors. *Control parallelism* occurs when different parts of an algorithm can be done simultaneously. *Data parallelism* occurs when different data values can be processed concurrently. The advantage of data parallelism is that it scales well. Larger data sets will offer more parallelism and thus provide better opportunities for speedup. In control parallelism the degree of parallelism is in principle limited by the structure of the algorithm. For this reason data parallelism is more useful than control parallelism for large scale computations.

The future construct is appealing because it can be used to express several types of parallelism.

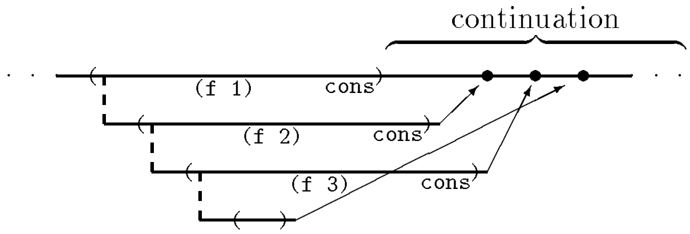
2.4.1 Pipeline Parallelism

Pipeline parallelism is a special case of control parallelism where the processing of data is overlapped with the processing of the result. Pipeline parallelism is the primitive form of parallelism provided by the future construct. It enables the production of a value by the future's body to be done concurrently with the consumption of the value by the future's continuation.

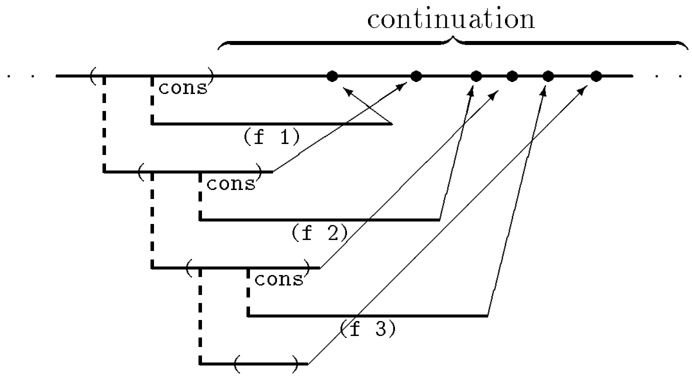
Pipeline parallelism is particularly useful when processing a data structure built incrementally (such as a list of values). At any given point in time, the part of the data structure that has been computed by the producer is available for processing by the consumer computation. An example of this is the procedure `pmap` as defined in

```
(define (pmap proc lst)
  (if (pair? lst)
    (let ((tail (FUTURE (pmap proc (cdr lst)))))
      (let ((val (proc (car lst))))
        (cons val tail)))
    '()))
```

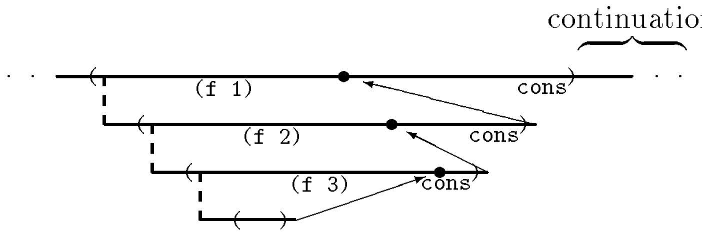
a) basic definition



b) spawning tree for basic definition



c) spawning tree for variant with `(FUTURE (proc (car lst)))`



d) spawning tree for variant with `(cons val (TOUCH tail))`

Figure 2.2: Parallel `map` definition and spawning trees.

Figure 2.2[4]. Pmap is a parallel version of map which applies a procedure to each element of a list and returns the list of results. Parallelism has been introduced by allowing the tail of the resulting list to be generated while the first element is computed and used by pmap's caller. Because cons is a non-strict operator, it immediately returns a pair with a placeholder as its tail (after proc has been called on the first element). The first element is thus immediately available for processing by the consumer. It is only when the consumer needs to access the tail that a synchronization must take place, possibly suspending the consumer until the next pair in the list is generated.

A variant of pmap with even more potential for parallelism is obtained by also wrapping a future around the call to proc. This allows the computation of the first element to overlap pmap's continuation. The difference in behavior is best visualized by examining the spawning tree for these two variants of pmap. Figure 2.2 shows the spawning trees for the call (pmap f '(1 2 3)). Parentheses have been added in these diagrams to indicate entry and exit of pmap. As is clear from the two upper spawning trees, the extra future allows more computations to overlap. Whether this added parallelism is actually beneficial will depend on the task granularity, the spawning cost, the number of processors and the way in which pmap's result is used by the continuation.

Pmap's parallelism is not easy to classify. At first glance it seems that it is an instance of control parallelism because it expresses concurrency between two different computations (the continuation and the application of the procedure to an element of the list). However, this control parallelism is not static. Pmap calls itself recursively so the parallelism varies with the length of the list. When viewed globally, pmap exhibits data parallelism because it expresses the parallel application of a procedure to a set of values. If the task granularity is large enough, the processing of longer lists will offer more parallelism.

2.4.2 Fork-Join Parallelism

The above variants of pmap are said to *export* concurrency because some of the work logically started "inside" pmap may be in progress after the procedure has returned.

[4] The shorter definition

```
(define (pmap proc lst)
  (if (pair? lst)
    (cons (proc (car lst)) (FUTURE (pmap proc (cdr lst))))
    '()))
```

is not equivalent because the two possible orderings of the evaluation of the arguments to cons do not give the same parallelism behavior.

Exported concurrency is a nuisance for some programming styles. If `proc` performs some side-effects on a global state, the computation following `pmap` cannot assume that they have all been done. Some explicit synchronization is needed to guarantee that all of `pmap`'s futures are done. In the simple case where `proc` does not itself export any concurrency, this synchronization can be done by walking the resulting list and touching all values that are the result of a future. A more elegant solution is to include the required synchronization inside `pmap`. This is easily achieved by having the future's extent match that of the procedure's body. In other words, the procedure is written so that each future (the fork) is balanced with a corresponding `TOUCH` (the join) executed before the procedure returns. This is a trivial change to `pmap`: a `TOUCH` is added around the second argument to `cons` (i.e. `(cons val (TOUCH tail))`). The spawning tree resulting from this variant of `pmap` is shown in Figure 2.2 (d).

2.4.3 Divide and Conquer Parallelism

An unfortunate characteristic of `pmap` is that it scales poorly due to the inherently sequential nature of lists. The processing of an n element list requires at least n sequential steps just to traverse the list. No matter how quickly each element can be processed, the time required to process n elements will be $\Omega(n)$. This may be of little consequence when task granularity is large and lists are short but massively parallel applications are bound to suffer more.

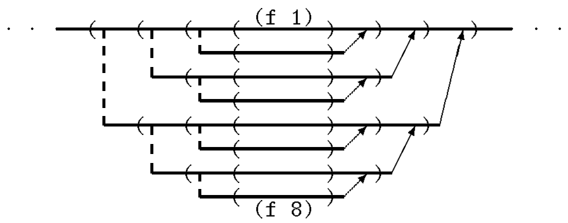
For this reason, it is preferable to use scalable data structures such as trees and arrays when lists would create a bottleneck. But this is not the only step to take. As long as futures are started sequentially, such as in a loop, a bottleneck will be present. A divide and conquer paradigm (DAC) can be used to start futures faster, allowing n futures to be started in $\Omega(\log n)$ time. This is actually the best that can be expected of the future construct because each future splits a thread of computation into two.

`Pvmap!`, shown in Figure 2.3, is a DAC version of `pmap` that works on vectors. The input elements are stored in a vector which is mutated to construct the result. The vector is divided in two and the mapping is performed recursively on both parts. When a single element is obtained, the mapped procedure is applied to the value and the result is stored back in the vector. To avoid allocating new vectors, subvectors are represented by two indices, `lo` and `hi`, which denote the subvector's extent. Because it uses a fork-join paradigm, all side-effects will be finished when `pvmap!` returns. Note also that the `TOUCH` is used only for synchronization. The actual value of `sync` is irrelevant.

Multilisp programs are frequently organized around DAC parallelism. Not only is it

```
(define (pvmap! proc vect)

  (define (map-range! proc lo hi)
    (if (= lo hi)

      (vector-set! vect lo (proc (vector-ref vect lo)))

      (let ((mid (quotient (+ lo hi) 2)))
        (let ((sync (FUTURE (map-range! proc (+ mid 1) hi))))
          (map-range! proc lo mid)
          (TOUCH sync)))))

  (map-range! proc 0 (- (vector-length vect) 1))
  vect)
```

a) definition



b) spawning tree for `(pvmap! f v)` with `v = #(1 2 3 4 5 6 7 8)`

Figure 2.3: Parallel "vector" `map`.

a fundamental technique for constructing parallel algorithms [Mou, 1990], it also blends naturally with the recursive algorithms and data structures commonly found in Lisp and symbolic processing. Several of the parallel benchmarks used in this thesis (see Section 2.9) are based on DAC parallelism.

2.5 Implementing Eager Task Creation

This section describes the eager task creation (ETC) implementation of futures. It will serve both as a reference implementation and as a basis on which lazy task creation is built. A few implementation details have been omitted for the sake of clarity. A more elaborate description can be found in [Miller, 1987].

As might be expected, the implementation of a Multilisp system is in many ways similar to that of a multitasking operating system. At the heart of both are utilities to support the management of various processing resources. For the management of the processors, an important concept is that of the *task* which is an abstract representation of a computation in progress. A program first starts out with a single *root* task in charge of performing the computation required by the program. Tasks are created and terminated dynamically as the computation progresses, possibly causing the number of tasks to exceed the number of processors in the machine.

The task abstraction is supported by the *scheduler* whose job is to run tasks by assigning them to processors. A task can be in one of three states. It is *running* when it is being executed by some processor. It is *ready* or *runnable* if it is only waiting for the scheduler to assign it to a processor. Finally, it is *blocked* if some event must occur before it is allowed to run.

Eager task creation (ETC) is a straightforward dynamic partitioning method that has been used in several implementations of Multilisp [Halstead, 1984, Miller, 1988, Swanson *et al.*, 1988, Kranz *et al.*, 1989]. With ETC there is a single representation for tasks: the *heavyweight* task object[5]. This is a heap allocated object with a number of fields that describe the state of the computation associated with the task. When the task needs to be started or resumed its state is restored by reading the fields of the corresponding task object. When a task needs to be suspended, the task object is updated to reflect the current state of the task. The most important information

[5]The definition of heavyweight tasks used here is not the same as the common meaning in operating systems (i.e. a process with its own address space). Here heavyweight task simply means a representation that is more expensive than the one used for lazy task creation.

retained in a task object is the continuation. It indicates where control must return when the task is resumed. Task continuations differ from first-class continuations in that they do not need to be given a "result" to continue with. They are zero argument procedures. Also, the full generality of first-class continuations is not necessary for task continuations since they are invoked at most once. Other fields can be added to task objects to support special language features but they are not strictly required for implementing futures. In fact, an implementation could simply use continuations to represent tasks. Nevertheless, task objects will be used here to make the algorithms more general.

2.5.1 The Work Queue

ETC lends itself well to *self scheduling*, where each processor is responsible for scheduling tasks to itself. All processors share a global queue, the *work queue*, that contains the set of runnable tasks. When a processor becomes idle, typically after a task blocks or terminates, it removes a task from the work queue and starts running it. If there are none available, the processor just keeps on trying until one is added to the work queue by some other processor. Self scheduling has the advantage of automatically balancing the load across the processors. As explained in Section 2.5.6, the work queue can be distributed but for now it is assumed to be a single centralized queue.

2.5.2 `FUTURE` and `TOUCH`

Tasks are created through the evaluation of futures. When a task, the *parent*, evaluates (`FUTURE` *expr*), it creates a placeholder object to represent the value of *expr* and then creates a *child* task whose role is to compute *expr* and determine the placeholder with the resulting value. The child task is added to the work queue to make it runnable and the placeholder is returned as the result of the future. Thus, the parent task immediately starts working on the continuation using the placeholder as a substitute for the value of *expr* while the child task waits in the work queue until it can be started by an idle processor.

Placeholder objects can be represented by a structure containing three slots: the state, the value and the waiting queue. The meaning of the state and value slots is obvious. The waiting queue is used to record the tasks that have become blocked because they need to wait until the placeholder has a value. When the placeholder gets determined, the tasks that are in the waiting queue are transferred to the work queue because they are now ready to run. When a task touches an undetermined placeholder it is suspended and added to the placeholder's waiting queue. The processor is now idle and must find a new task to run from the work queue. When the blocked task later resumes (inside the `TOUCH`), the placeholder's value is fetched and returned.

2.5.3 Scheme Encoding

A Scheme encoding of these algorithms is given in Figure 2.4 and the definition of the support procedures is given in Figure 2.5. Note that the code in Figure 2.4 is schematic and does not address all atomicity issues.

`Idle` is the procedure that is run by processors in need of work. When the program starts up, all processors call `idle`, except for the single processor that is running the root task. `Idle` continually tries to remove a ready task from the work queue. To implement `TOUCH`, each processor must keep track of its currently running task. When a task is found, `resume-task` is called. The task becomes the "current task" of that processor and it is restarted by calling its associated continuation. It is assumed that each processor has a private storage area to store the currently running task. The procedures `current-task` and `current-task-set!` access this storage.

The future special form can be thought of as a derived form that expands into a call to `make-FUTURE`. Its only argument is a nullary procedure (a *thunk*) that contains the future's body. The expression `(FUTURE` *expr*`)` is really an abbreviation for the procedure call `(make-FUTURE (lambda ()` *expr*`))`. `Make-FUTURE` first creates an undetermined placeholder to represent the body's value and then creates a child task. The child task is set up so that its continuation, when called by `resume-task`, will compute the value of the body by calling the thunk. The procedure `end-body` contains the work to be done after the body is computed. `End-body` calls `test-and-determine!` to determine the result placeholder with the body's value. Control then goes back to `idle`. Note that `end-body` signals an error when a placeholder is determined more than once. This might happen if a continuation captured by a `call/cc` in the body is invoked after the body has already returned.

`Test-and-determine!` is an atomic operation similar in spirit to the traditional "test-and-set" operation. It tests if a placeholder is determined and if it isn't, the placeholder gets determined to the second parameter and true is returned to indicate success. Otherwise the placeholder remains as is and false is returned. When a placeholder is determined, the tasks on its waiting queue are transferred to the work queue, thus making them runnable.

```
(define (idle)
  (if (queue-empty? (work-queue))
    (idle)
    (resume-task (queue-get! (work-queue)))))

(define (resume-task task)
  (current-task-set! task)
  ((task-continuation task)))

(define (make-FUTURE thunk)
  (let ((res-ph (make-ph)))
    (let ((child (make-task
                   (lambda () (end-body res-ph (thunk))))))
      (queue-put! (work-queue) child)
      res-ph)))

(define (end-body res-ph result)
  (if (test-and-determine! res-ph (TOUCH result)) ; ①
    (idle)
    (error "placeholder previously determined")))

(define (test-and-determine! ph val)
  (if (ph-determined? ph)
    #f
    (begin
      (determine! ph val)
      #t)))

(define (determine! ph val)
  (ph-value-set! ph val)
  (ph-determined?-set! ph #t)
  (queue-append! (work-queue) (ph-queue ph)))

(define (TOUCH x)
  (if (ph? x)
    (if (ph-determined? x) (ph-value x) (TOUCH-undet x)) ; ②
    x))

(define (TOUCH-undet ph)
  (call-with-current-continuation
    (lambda (cont)
      (let ((task (current-task)))
        (task-continuation-set! task
          (lambda ()
            (cont
              (if (ph? ph) (ph-value ph) ph)))) ; ③
        (queue-put! (ph-queue ph) task)
        (idle)))))
```

Figure 2.4: Scheme encoding of Multilisp core.

Operations on queues:	
`(queue-empty?` q`)`	Tests if q is empty.
`(queue-get!` q`)`	Removes and returns the item at q's head.
`(queue-put!` q x`)`	Adds x to q's tail.
`(queue-append!` $q1$ $q2$`)`	Transfers all items from $q2$ to $q1$'s tail.
Operations on placeholders:	
`(make-ph)`	Creates and returns an undetermined placeholder.
`(ph?` x`)`	Tests if x is a placeholder.
`(ph-determined?` ph`)`	Tests the state of ph.
`(ph-determined?-set!` ph x`)`	Sets the state of ph.
`(ph-value` ph`)`	Returns the value of ph.
`(ph-value-set!` ph x`)`	Sets the value of ph.
`(ph-queue` ph`)`	Returns the waiting queue of ph.
Operations on tasks:	
`(make-task` c`)`	Creates and returns a task whose continuation is c.
`(task-continuation` t`)`	Returns t's continuation.
`(task-continuation-set!` t c`)`	Sets t's continuation to c.
Operations on the processor's local state:	
`(current-task)`	Returns the task currently running on the processor.
`(current-task-set!` t`)`	Sets the task currently running on the processor to t.
Other operations:	
`(work-queue)`	Returns the work queue.

Figure 2.5: Procedures needed to support Multilisp core.

Touching is implemented by `TOUCH` and `TOUCH-undet`. `TOUCH-undet` handles the case where the value to be touched is an undetermined placeholder. When an undetermined placeholder is being touched, the current task must be suspended and put on the placeholder's waiting queue. This is done by a call to `call/cc` which captures `TOUCH`'s continuation. Note that since this continuation is guaranteed to be called at most once, a less general but more efficient version of `call/cc` could be used. The task is then put on the placeholder's waiting queue so that it can later be made runnable by `test-and-determine!`. As the current task is now blocked, control is transferred to `idle` to move on to some other piece of work. When the task is resumed, the placeholder's value will be returned to `TOUCH`'s continuation.

2.5.4 Chasing vs. No Chasing

An interesting issue is whether placeholders should be allowed to be determined with other placeholders. If this is permitted, the touching of a placeholder must perform the recursive touching of its value. This *chasing* process can be expensive if the chain of placeholders is long. This happens in programs where the future bodies often return placeholders and placeholders are touched multiple times.

The alternative *strict* method requires that placeholders be only determined with non-placeholders. The code in Figure 2.4 implements the strict method. A chasing implementation is obtained by removing the `TOUCH` on line ①, adding a `TOUCH` around line ② and replacing line ③ by `ph`. The drawback of the strict method is that the number of blocked tasks will increase in the cases where chasing would be required. It may also restrict concurrency because it has an additional control dependency. None of these methods is clearly superior to the other in all contexts. Fortunately, both methods can coexist in the same system as long as the two types of placeholders are distinguished and the appropriate touching and determining mechanisms are called. Having two types of placeholders is useful to implement legitimacy (see Section 2.8.4).

Unless otherwise noted, the strict method will be assumed because it is conceptually simpler (i.e. determined placeholders are guaranteed to have a non-placeholder value) and it gives a shorter code sequence for inline calls to `TOUCH`.

2.5.5 Critical Sections

Various implementation details have been omitted from the above description. One problem that must be addressed is the possible race conditions in these algorithms.

Several processors may simultaneously attempt to mutate the work queue or a placeholder. To preserve the integrity of these data structures, some operations must appear to be mutually exclusive. This is usually done by introducing locks in the data structures to control access to them. Spin locks are sufficient because the critical sections consist of only a few instructions. The operations that must be protected are

1. Testing and removing a task from the work queue (when a processor is idle).
2. Adding a task to the work queue (when a future is evaluated).
3. Checking the state of a placeholder and adding a task to a placeholder's waiting queue (when an undetermined placeholder is touched).
4. Changing the state and value of a placeholder (when a placeholder gets determined).

Garbage collection adds another complication. If the value of placeholders is assumed to be immutable, it is perfectly valid to replace any reference to a determined placeholder by the placeholder's value. This optimization, called *splicing*, can in principle be done at any moment but usually it is performed by the garbage collector. The advantage of splicing is that subsequent calls to `TOUCH` will be faster because the dereferencing of the placeholder is avoided (this is particularly helpful to reduce the cost of chasing). Consequently, the implementation must prevent the splicing of the placeholder currently being manipulated. Several techniques are possible such as temporarily disabling the garbage collector or temporarily marking the placeholder as non-spliceable. The test at line ③ in `TOUCH-undet` is needed to account for the splicing of the touched placeholder. Aside from this test, the code in Figure 2.4 does not include the operations required to prevent splicing.

2.5.6 Centralized vs. Distributed Work Queue

A potential source of inefficiency in the scheduler is caused by the centralized work queue accessed by all processors. The contention for the work queue may become an important bottleneck as the number of processors is increased. Each access to the work queue is mutually exclusive so all operations on the work queue get sequentialized. The time it takes to add and remove a task from the work queue puts an upper bound on the rate at which tasks can be created and resumed. Clearly, it would be preferable if this rate scaled up with the number of processors.

A common solution is to distribute the work queue. Each processor has its own work queue which it uses to make tasks runnable. These work queues are accessible from all processors. When a processor is looking for work, it first looks for runnable tasks in its own work queue and goes on to search the work queue of other processors only if its work queue is empty. This reduces contention and remote memory traffic and also improves locality since tasks restarted from the local work queue are likely to have been created locally.

2.6 Fairness of Scheduling

Another important consideration is fairness of scheduling. In a fair system, a task's computation is guaranteed to progress as long as the task is runnable. In other words, there is a finite amount of time between a task becoming runnable and it actually running on a processor.

Fairness can be implemented by preventing a task from running longer than a certain stretch of time (*quantum*) without giving all other runnable tasks a chance to run as well. The scheduler effectively cycles through all runnable tasks giving each of them a quantum of time to advance their computation. At regular time intervals all processors receive a *preemption* interrupt to signal that the quantum has expired. Upon receiving this interrupt, a processor suspends the currently running task, puts it at the tail of the work queue and then resumes the task at the head.

In a system with a centralized work queue at least $\min(n, r)$ tasks are resumed every quantum (where n is the number of processors and r is the number of runnable tasks)[6]. It follows that a task will start running in no more than $\lfloor r'/n \rfloor$ quantums, where r' is the number of runnable tasks at the time the task was made runnable. If r' does not vary much, the tasks will get an even share of the processors (roughly the power of n/r' processor per task if $r' > n$).

In a system with a distributed work queue at least one task is resumed from every work queue every quantum. A task will thus start running in no more than $q + 1$ quantums, where q is the length of the local work queue at the time the task was made runnable. Thus, the processing power given to tasks residing on a processor is evenly distributed but the processing power of tasks residing on different processors may be substantially different.

[6] It is assumed that the quantum is large enough so that the effects of contention on the work queue are negligible.

The original Multilisp semantics [Halstead, 1985] had a scheduling policy that was fair as long as all tasks were of finite duration. The only guarantee made by the scheduler was that a runnable task would run if there were no other runnable tasks. Under the finite task assumption, this implies that all tasks will eventually run. Finiteness is a reasonable assumption for Multilisp programs since it is common to design parallel programs by annotating terminating sequential programs with futures. In sequential programs, all expressions evaluated correspond to *mandatory* work that needs to be done to compute the result of the program. Any execution order for the tasks will compute the correct result as long as it respects the basic ordering imposed by the strict operations. However, there are special situations where true fairness is useful.

Programs are sometimes organized around tasks that conceptually never terminate. One example is the client/server model where each task implements a particular service for some clients. Server tasks receive requests from the clients and send back a reply for each request serviced. Each server task is in an infinite receive-compute-respond loop. Without a fair scheduler, a set of server tasks could monopolize all the processors if they continually have requests to service. Other server tasks would never get a chance to run. A multi-user Multilisp system can be viewed as an instance of this model (the clients are the users and the server tasks are the read-eval-print loops).

Another application of fairness is to support *speculative* computation. A computation is speculative if it is not yet known to contribute to the program's result. Speculative computation arises naturally in search problems where multiple solutions may exist but only one is needed. Several search paths can be explored in parallel and as soon as a solution is found the search can be stopped. This form of computation, which Osborne [Osborne, 1989] calls *multiple approach* speculative computation, is known in parallel logic programming as *OR-parallel*. If the likelihood of finding a solution in any given path is fairly similar, then it is reasonable to spend an equal effort searching each path. This is easily approximated by a fair scheduler which timeslices tasks from a centralized work queue.

However the solutions are typically not distributed equally among the search paths. The paths that are likely to lead quickly to a solution should be searched more eagerly than others. Thus a system aimed at general speculative computation should provide some finer level of control over the scheduler (such as a mechanism to assign priorities to the speculative tasks). Because there is currently no consensus as to which level of control is best, this thesis does not investigate the implementation of such priority mechanisms. Fairness of scheduling plays a minor role in this thesis. Chapter 3 shows that lazy task creation can support fairness.

2.7 Dynamic Scoping

Multilisp uses static scoping as its primary variable management discipline. Static scoping has the advantage of clarity because the identity of a variable only depends on the program's local structure, not its runtime behavior. With the exception of global variables, a variable can only be accessed by an expression textually contained in the binding form that declares the variable.

Static scoping is not well suited for certain applications. Sometimes it is necessary to pass an argument to one or several procedures far down in the call tree (such as the default output port or the exception handler). Such arguments must either be passed in global variables or be passed as explicit arguments from each procedure to the next in the call chain. The first solution is not appropriate in a parallel system because of the possible conflict between tasks. The second solution clearly lacks modularity because each procedure must be aware of the arguments passed from parent procedures to all its descendants.

Dynamic scoping offers an elegant solution. A dynamically scoped variable can be accessed by any computation performed during the evaluation of the body of the binding form that declares the variable. In a sense, dynamic variables are implicit parameters to all procedures. The set of bindings (the *dynamic environment*) is passed implicitly by each procedure to its children in the call tree. A given binding is thus only visible in the call tree that stems from the binding form with the exception of the subtrees where the binding is shadowed by a new binding to the same variable.

There are several possible constructs to express dynamic scoping. For the sake of simplicity two special forms are used here[7]. The form `(dyn-bind` *id val body*`)` introduces a new binding of the dynamic variable *id* to the value *val* for the duration of the body. The form `(dyn-ref` *id*`)` returns the value of the dynamic variable *id* in the current dynamic environment. Note that *id* is not evaluated and that lexically scoped variables and dynamic variables exist in separate namespaces. Figure 2.6 shows a typical use of dynamic scoping to implement a simple exception system. The dynamic variable `EXCEPTION-HANDLER` contains a single argument procedure that is called with an error message when an error is detected. The procedure `catch-exceptions` takes a thunk as argument and calls it in a dynamic environment where `EXCEPTION-HANDLER` is bound to the continuation of `catch-exceptions`. Thus, the call to the exception handler in `raise-exception` will immediately exit from `catch-exceptions` with the error message as its result (for example, the call `(map-sqrt '(1 -2 5))` returns the

[7] An obvious extension would be an assignment construct.

```
(define (catch-exceptions thunk)
  (call-with-current-continuation
    (lambda (abort)
      (dyn-bind EXCEPTION-HANDLER abort (thunk)))))

(define (raise-exception msg)
  ((dyn-ref EXCEPTION-HANDLER) msg))

(define (square-root x)
  (if (negative? x)
    (raise-exception "domain error")
    (sqrt x)))

(define (map-sqrt lst)
  (catch-exceptions
    (lambda () (map square-root lst))))
```

Figure 2.6: Exception system based on dynamic scoping and `call/cc`.

string `"domain error"`).

An implication of the above semantics is that dynamic environments are associated with continuations. All continuations carry with them the dynamic environment that was in effect when they were created (i.e. due to the evaluation of some subproblem call). When a continuation is invoked, the captured dynamic environment becomes the current dynamic environment. `Dyn-bind` creates a new dynamic environment for the evaluation of the body simply by adding a new binding to the current dynamic environment. This new binding remains in effect only for the duration of the body because the continuation invoked to exit the body (normally `dyn-bind`'s continuation but possibly some continuation captured with `call/cc` outside the body) will restore the dynamic environment to the appropriate value. In implementation terms, this implies that each subproblem call must save the dynamic environment on the stack prior to the call and restore it upon return.

Because the save/restore pair is added to all subproblem calls, this may result in an unacceptably high overhead. Notice that in normal situations the dynamic environment does not actually change when a continuation is invoked. Only `dyn-bind`'s continuation and continuations captured by `call/cc` might be invoked from a different dynamic environment. An alternative approach is thus to put the save/restore pair only around the evaluation of `dyn-bind`'s body and around calls to `call/cc`. This approach offers more efficient subproblem calls but also has the unfortunate consequence that `call/cc` and `dyn-bind` are no longer properly tail-recursive. `Call/cc`'s procedure argument and `dyn-bind`'s body are not reductions because their continuation contains a new continuation frame[8]. The loss of proper tail recursion for `dyn-bind` is probably not very troublesome (most Lisp systems implement the dynamic binding construct with similar save/restore pairs). However it is harder to justify for `call/cc`.

To preserve `call/cc`'s tail recursive property, `call/cc` can be redefined as shown in Figure 2.7. It is assumed that the state of the dynamic environment is maintained in a global data structure accessible through the procedures `current-dyn-env` and `current-dyn-env-set!`. The implementation exploits the invariant that procedures always invoke their implicit continuation with the same dynamic environment that existed when they were called. Thus a normal return from the call to `proc` in `call/cc` invokes the captured continuation with the correct dynamic environment. An abnormal return to `cont` is only possible by calling the closure passed to `proc`. This closure explicitly restores the correct dynamic environment before invoking the captured continuation.

Parallel processing raises additional implementation issues. In order for the future construct's semantics to be as non-intrusive as possible, the dynamic environment used for the evaluation of the future's body should be the same as the one in effect when the future itself was evaluated. Consequently, the parent task must save the dynamic environment into the child task and the child task must restore this environment when it starts running. This adds an overhead to task creation, suspension and resumption.

Another issue is the representation of dynamic environments. A popular approach in uniprocessor Lisps is *shallow binding*. The environment is represented as a table of cells. Each cell holds the current value of a dynamic variable. A new binding is introduced by saving the current value of the cell on a stack and assigning the new value to the cell. Upon exit from the binding construct, the previous binding is restored by popping the old value off the stack. Thus `dyn-bind` and `dyn-ref` are constant time operations. However, saving the entire dynamic environment (i.e. the operation `current-dyn-env`) is expensive because it implies a copy of the binding table. An alternative approach (shown in Figure 2.7) is *deep binding*. The dynamic environment is represented as a stack of bindings (i.e. an association list). `Dyn-bind` simply adds a new binding at the head of the list and `dyn-ref` searches the list for the most recent binding of the variable. Unfortunately the cost of `dyn-ref` is $O(b)$ where b is the number of bindings in the environment. This may be expensive if b is large and the variables looked up are those

[8] The following procedure will thus run out of memory when it is called

```
(define (loop) (call-with-current-continuation (lambda (k) (loop))))
```

```
(define (call-with-current-continuation proc)
  (primitive-call-with-current-continuation
    (lambda (cont)
      (proc (let ((env (current-dyn-env)))
              (lambda (val)
                (current-dyn-env-set! env)
                (cont val)))))))
```

The special forms `dyn-ref` and `dyn-bind` expand into:

`(dyn-ref` *id*`)` ⟶ `(current-dyn-env-lookup '`*id*`)`

`(dyn-bind` *id val body*`)` ⟶ `(begin`
`(current-dyn-env-push! '`*id val*`)`
`(let ((result` *body*`))`
`(current-dyn-env-pop!)`
`result))`

Definitions for deep binding:

```
(define (current-dyn-env-lookup id)
  (cdr (assq id (current-dyn-env))))

(define (current-dyn-env-push! id val)
  (current-dyn-env-set! (cons (cons id val) (current-dyn-env))))

(define (current-dyn-env-pop!)
  (current-dyn-env-set! (cdr (current-dyn-env))))
```

Figure 2.7: Implementation of dynamic scoping with tail recursive `call/cc`.

that were bound early[9]. On the other hand, `current-dyn-env` only requires a single pointer copy so the overhead for `call/cc` and task operations is minimal. Deep binding is adequate when dynamic variables are referenced infrequently, for example if their main purpose is to support the exception processing system. Yet another approach is to represent environments with 2-3 or AVL search trees, thus permitting $O(\log n)$ cost for `dyn-bind` and `dyn-ref`, where n is the number of variables bound in the environment, and constant cost for `current-dyn-env` and `current-dyn-env-set!`. It isn't clear which of these last two representations is most efficient in practice. The deep binding approach has been used in this work for simplicity but the implementation strategies explained in the next chapter are equally applicable to the search tree representation.

2.8 Continuation Semantics

Continuations also present special problems in a parallel setting. It isn't clear what the *terminal* continuation of a child task should be. This continuation is the one that is passed to the body of the future. In other words, what should be done with the value returned by the body? This is an important question because the approach chosen will specify the behavior of first-class continuations in the presence of futures.

2.8.1 Original Semantics

Several approaches have been proposed. In the original Multilisp definition [Halstead, 1985] the body's value was used to determine the placeholder created for the future and the task was simply terminated. This is the semantics implemented by the code in Figure 2.4[10].

2.8.2 MultiScheme Semantics

MultiScheme adopted a subtly different model for continuations. The child task and placeholder created by a future are conceptually linked. The placeholder is called the *goal* of the task and the task is the placeholder's *owner*[11]. This linkage was introduced

[9] Efficiency can be improved somewhat by adding a cache to hold the value of recently accessed variables (for example see [Rozas and Miller, 1991]).

[10] Multilisp was not designed to support first-class continuations so it isn't surprising that the original semantics does not interact well with them.

[11] The term "motivated task" was used in [Miller, 1987].

```
(define (make-FUTURE thunk)
  (let ((res-ph (make-ph)))
    (let ((child (make-task
                   (lambda () (end-body (thunk)))
                   res-ph)))
      (queue-put! (work-queue) child)
      res-ph)))

(define (end-body result)
  (let ((res-ph (task-goal-ph (current-task))))
    (if (test-and-determine! res-ph (TOUCH result))
      (idle)
      (error "placeholder previously determined"))))
```

Figure 2.8: MultiScheme's implementation of the future special form.

to permit the garbage collection of tasks. Finding the value of the future's body is seen as the task's sole reason of existence. Since the goal placeholder is the representative of this value, the owner task can safely be terminated if the placeholder is known to be unnecessary for the rest of the computation.

The implementation of this semantics is given in Figure 2.8. Note that the procedure `make-task` now takes two arguments: the continuation and the goal placeholder. Also note that `end-body` takes only one argument because the placeholder to determine implicitly comes from the task executing `end-body` (i.e. the current task). The goal placeholder is now embeded in the child task instead of the terminal continuation (as is done in the original semantics). This is an important distinction because a task can replace its current continuation by a completely different one by calling a continuation created by `call/cc`. However, the goal placeholder never changes. Interestingly, the original and MultiScheme implementations are equivalent in the absence of `call/cc`. This is because in such a case the only task that can execute a given continuation is the task created with that continuation. Taking the placeholder to determine from the continuation (as in the original semantics) or from the task object (as in MultiScheme) will give the same placeholder because of the one-to-one correspondence between continuations and tasks.

Figure 2.9 gives an example where the two implementations differ. Here two tasks (T_1 and T_2) are involved in addition to the root task. The corresponding placeholders are Ph_1 and Ph_2. The call to `call/cc` binds `k` to T_1's continuation. Thus, `k` corresponds to a call to `end-body`. With the original implementation of futures, `k` contains an implicit

```
(define x
  (TOUCH (FUTURE₁
           (call-with-current-continuation
             (lambda (k)
               (+ 1 (TOUCH (FUTURE₂ (k 0)))))))))
```

Figure 2.9: A sample use of futures and `call/cc`.

reference to Ph_1. When T_2 calls `k`, Ph_1 gets determined to 0. Following this, the root task can return from the first `TOUCH` and consequently `x` gets bound to 0. Note that T_1 is suspended indefinitely on the second `TOUCH` because Ph_2 never gets determined.

With MultiScheme's implementation of futures, a call to `k` determines the goal placeholder of the current task. Since it is T_2 that is calling `k`, Ph_2 gets determined to 0. T_1 then proceeds from the second `TOUCH`, adds 1 and calls `k` with 1 (the `lambda`-expression's body implicitly calls `k`). This time, it is T_1 that is calling `k`, so Ph_1 gets determined to 1. Finally, the root task can return from the first `TOUCH`, binding `x` to 1.

2.8.3 Katz-Weise Continuations

A nice feature of futures is that, in typical purely functional programs, they can be added around any expression without changing the result of the program. In other words, futures are equivalent to an identity operator when only the result of the computation is considered. Futures only affect the order of evaluation. This suggests an attractive mode of programming: first write a correct functional program without any futures and then explore various placements of futures to turn the program into an efficient parallel one.

Unfortunately the original and MultiScheme semantics for continuations do not permit this for all purely functional programs because inserting futures in a program that uses `call/cc` can alter the result computed. For MultiScheme, this should be clear from the previous example. For the original semantics all is fine as long as the future body's continuation is invoked at most once, including the normal return from the body. To explain what happens when the continuation is called multiple times, consider the contrived expression in Figure 2.10. In this expression, the continuation created by `call/cc` is called exactly twice. Assume for the moment that the `TOUCH` and `FUTURE` operations are not present. `Y` will get bound to the continuation created by `call/cc`; the continuation that takes a value and binds `y` to it. Since at this point `y` is not a

```
(define x
  (let ((y (TOUCH (FUTURE
                    (call-with-current-continuation
                      (lambda (k) k))))))
    (if (number? y)
      y
      (y 123))))
```

Figure 2.10: A future body's continuation called multiple times.

number, the continuation is restarted with 123 thus binding `y` to 123. Since `y` is now a number it is returned and `x` gets defined to 123.

When `TOUCH` and `FUTURE` are present, an undetermined placeholder will be created and a child task created to evaluate the `call/cc`. The continuation captured here (i.e. `k`) corresponds to the task's continuation, that is a call to `end-body`. The placeholder will get determined to this continuation and, through the `TOUCH`, `y` gets bound to it. However, when this continuation is called an attempt is made to determine the placeholder a second time (this time with 123) and then to terminate the current task. This is clearly an error because a placeholder cannot represent more than one value and deadlock would occur (since all tasks would have terminated).

An interesting implementation of futures that solves this problem was proposed by Katz and Weise [Katz and Weise, 1990]. The idea is to preserve the link between the future body's continuation and the future's continuation. On the first return to the body's continuation, the placeholder gets determined and the task is terminated (as in the original semantics). However, on every other return the body's continuation acts exactly like the future's continuation, as if the future had never existed.

2.8.4 Katz-Weise Continuations with Legitimacy

Unfortunately, this approach does not solve all interaction problems between first-class continuations and futures. It is still possible to write purely functional programs that do not return the same value when futures are added. Consider the program in Figure 2.11 which is a simplified form of exception processing. If the future special form is not present, a value of 0 is returned because the call `(abort 0)` is done first, bypassing the body of the `let` and the binding of `dummy`. With the future, a child task is created to evaluate `(abort 0)` and the parent task implicitly returns 1 to `abort`. Each task

```
(call-with-current-continuation
  (lambda (abort)
    (let ((dummy (FUTURE (abort 0))))
      1)))
```

Figure 2.11: Exception processing with futures.

exits the `call/cc` with its own belief of the result: the parent task with 1 and the child task with 0. In general, this means that multiple tasks may return to the program's root continuation. One of these tasks has the right result (i.e. the same result as a sequential version of the program) but which task? Choosing the first task to arrive at the program's root continuation is not a valid technique because of the race condition involved.

The solution proposed in [Katz and Weise, 1990] introduces the concept of *legitimacy*. A particular sequence of evaluation steps (a *thread*) is legitimate if and only if it is executed by the sequential version of the program. Legitimacy is thus a characteristic that depends on the control flow of the program. It can be derived from 1) the fact that the root thread is legitimate and 2) the causality rules inherent in the sequential subset of the language. In particular, if a thread is legitimate and it returns from *expr* with the value v, then the thread corresponding to the execution of *expr*'s continuation with the value v is also legitimate. This rule naturally extends to the future special form by attaching legitimacy to tasks: after a child task is spawned by `(FUTURE` *expr*`)`, the parent task is legitimate if and only if the corresponding placeholder gets determined by a legitimate task. The parent task's legitimacy is thus equal to the legitimacy of the task that gets to determine the placeholder. Note that the child task inherits the legitimacy of its parent at the moment of the task spawn. As an example, consider the following program which involves three tasks (T_1, T_2, and the root task T_{root})

```
(let* ((x (FUTURE₁ expr₁))
       (y (FUTURE₂ expr₂)))
  expr₃)
```

After spawning the tasks T_1 and T_2 the root task will evaluate $expr_3$. The root task is legitimate if and only if the first task to return from $expr_2$ is legitimate. This fact can be expressed by the constraint

$$Legit(T_{root}) = Legit(Det(Ph_{T_2}))$$

That is, the legitimacy of the root task is equal to the legitimacy of the task that determines the placeholder created for T_2. Similarly, task T_2 is legitimate if and only if the first task to return from $expr_1$ is legitimate

$$Legit(T_2) = Legit(Det(Ph_{T_1}))$$

In the event that it is T_2 that returns first from $expr_2$ (i.e. $Det(Ph_{T_2}) = T_2$), the root task's legitimacy will become equal to the legitimacy of the first task returning from $expr_1$. That is

$$Legit(T_{root}) = Legit(T_2) = Legit(Det(Ph_{T_1}))$$

This illustrates that a task's legitimacy at a given point in time is represented by a chain that models the legitimacy dependencies inferred up to that point. Initially the links between tasks are unknown and, as tasks terminate (and determine placeholders), the links get filled in. The gaps in the chain correspond to future bodies that have not yet returned normally. Abnormal exits from the body of a future can create independent chains that never get connected to the legitimate chain. Note that there is at all times exactly one legitimate task in the system. All other tasks can be viewed as being speculative tasks because there is no guarantee that they actually contribute to the computation at hand. At the moment of its death, the legitimate task will turn one of the speculative tasks into the legitimate task.

2.8.5 Implementing Legitimacy

An implementation of the Katz-Weise semantics with legitimacy is shown in Figure 2.12. The legitimacy chain is conveniently implemented with placeholders. Each task has a legitimacy flag represented by a placeholder. The root task is initially legitimate so its legitimacy flag is a non-placeholder. When a child task is created its legitimacy flag is taken from the parent task. Since the parent task is going to invoke the future's continuation, its legitimacy flag is replaced by a newly created undetermined placeholder, `leg-ph`, which represents the as of yet unknown legitimacy of the first task to return from the future's body (which might not be the child). `Leg-ph` must also be embeded in the body's continuation. When this continuation is returned to, which corresponds to a call to `end-body`, the result placeholder gets determined and the legitimacy chain is extended by unifying `leg-ph` with the current task's legitimacy flag.

```
(define (make-FUTURE thunk)
  (call-with-current-continuation
    (lambda (k)
      (let ((res-ph (make-ph))
            (leg-ph (make-ph))
            (parent (current-task)))
        (let ((child (make-task
                       (lambda () (end-body k res-ph leg-ph (thunk)))
                       (task-legitimacy parent))))
          (task-legitimacy-set! parent leg-ph)
          (queue-put! (work-queue) child)
          res-ph)))))

(define (end-body k res-ph leg-ph result)
  (if (test-and-determine! res-ph (TOUCH result))
    (begin
      (determine! leg-ph (task-legitimacy (current-task))) ; ①
      (idle))
    (k result)))

(define (speculation-barrier)
  (TOUCH (task-legitimacy (current-task))))
```

Figure 2.12: The Katz-Weise implementation of futures.

2.8.6 Speculation Barriers

A straightforward use of legitimacy is to prevent speculative tasks from terminating the program and only allowing the legitimate task to do this. This *speculation barrier* can be accomplished simply by touching the task's legitimacy flag at the program's terminal continuation. Conceptually, this touch walks down as far as it can in the task's legitimacy chain and blocks until the task is known to be legitimate. Only the legitimate task is allowed to proceed beyond the touch, the other tasks are suspended indefinitely.

Using a speculation barrier at the very tail of a program guarantees that the correct result will be returned but it does little to prevent speculative tasks from consuming processing resources. It is possible to add speculation barriers at well chosen places in the program to limit the extent of speculative parallelism. Even though this reduces the amount of parallelism in the program, it may yield a more efficient program because a higher proportion of the time will be spent doing mandatory work. A case where this might be useful is given in Figure 2.13. For simplicity, it is assumed that `map` processes

```
(define (map-sqrt lst)
  (call-with-current-continuation
    (lambda (abort)
      (map (lambda (x)
             (FUTURE
               (if (negative? x) (abort x) (sqrt x))))
           lst))))

(define (map-sqrt-with-barrier lst)
  (let ((result (map-sqrt lst)))
    (speculation-barrier)
    result))
```

Figure 2.13: An application of speculation barriers.

the values from head to tail[12]. For each value in the list, `map-sqrt` spawns a task to compute the square root of the value and returns a list of the results. In a sequential version of the program (i.e. if the future is absent), the first negative value is returned by `map-sqrt`. In the parallel version, the root task and all tasks processing negative values will return from `map-sqrt`. `Map-sqrt-with-barrier` obtains the same result as the sequential version by using a speculation barrier after the call to `map-sqrt`. Only the task processing the first negative value will be legitimate and will cross the barrier. Since this task bypasses the determining of its result placeholder, its parent's legitimacy flag will remain undetermined forever. All the tasks spawned by the parent and its children after the legitimate task will have undetermined legitimacy flags. Consequently, these tasks will get suspended when they reach the barrier.

2.8.7 The Cost of Supporting Legitimacy

The cost of supporting legitimacy is an important issue. Speculation barriers are certainly useful to express some programs, but many programs have no need for them, in particular those that only contain mandatory tasks. Consequently, it is important to evaluate the cost of supporting legitimacy in both contexts.

For programs which contain speculation barriers, one concern is the space occupied by tasks suspended at barriers. A careful study of Figure 2.12 reveals that these tasks are only retained if they might become legitimate. These tasks are suspended on `leg-ph`

[12] The Scheme language does not impose a particular ordering.

which is only accessible through the child's terminal continuation. In the previous example (Figure 2.13) this continuation was discarded when `abort` was called by the child. Since `leg-ph` is unreachable it will eventually get garbage collected along with the tasks suspended on it. On the other hand, if the child's continuation had been saved prior to the call to `abort` (by calling `call/cc` and saving the continuation away), it would not be possible to garbage collect the suspended tasks because `leg-ph` would still be reachable. This is clearly the correct behavior since any number of the suspended tasks could still become legitimate (for example, if the saved continuation is invoked by the legitimate task).

Two other costs are legitimacy testing and propagation. The cost of legitimacy propagation is particularly important because it is paid even by programs that do not use legitimacy (or that use it infrequently). In Figure 2.12, the current task's legitimacy placeholder is propagated directly to the next task in the chain (line ① in `end-body`). Legitimacy propagation is thus constant cost but legitimacy testing can be expensive. A program which spawns n mandatory tasks, thus creating a legitimacy chain with n placeholders, will require $O(n)$ time to test legitimacy at the program's termination (the task spawning strategy, whether it is a sequential loop or DAC loop, is irrelevant).

Another approach is to touch the current task's legitimacy (on line ①) before propagating it to the next task. In other words the task waits to be legitimate before marking the next task as legitimate. Legitimacy testing is then constant cost but legitimacy propagation is expensive for two reasons: it is inherently sequential and it produces frequent task switches. Because of the touch, a particular legitimacy placeholder in the chain can only be determined after the previous legitimacy placeholder has been determined. This implies that the last task will at best be marked as legitimate $\Omega(n)$ time after the first task. Also, any task terminating before its predecessor in the chain will have to be suspended and eventually resumed, just to set the next legitimacy placeholder.

A better strategy is to shrink the legitimacy chain as the computation progresses. All the links in the chain will have to be followed but this can be done in parallel. The method uses a "collapse" operation that walks a chain of placeholders and returns its tail element (i.e. either an undetermined placeholder or a non-placeholder). This operation is added to line ① so that the current task propagates its collapsed legitimacy chain to the next task. Nothing is gained if a task terminates before its predecessor but if it terminates afterwards, one or more links in the chain will get removed for the benefit of the successor tasks. But how frequently will it be possible to collapse the chain?

Clearly, the order of task termination has a direct influence on the collapsing of the

```
(define (fj1 n)
  (if (= n 0)
    1
    (let* ((l (FUTURE (fj1 (- n 1))))
           (r (fj1 (- n 1))))
      (+ (TOUCH l) r))))
```

```
(define (fj2 n)
  (if (= n 0)
    1
    (let* ((l (FUTURE (fj2 (- n 1))))
           (r (FUTURE (fj2 (- n 1)))))
      (+ (TOUCH l) (TOUCH r)))))
```

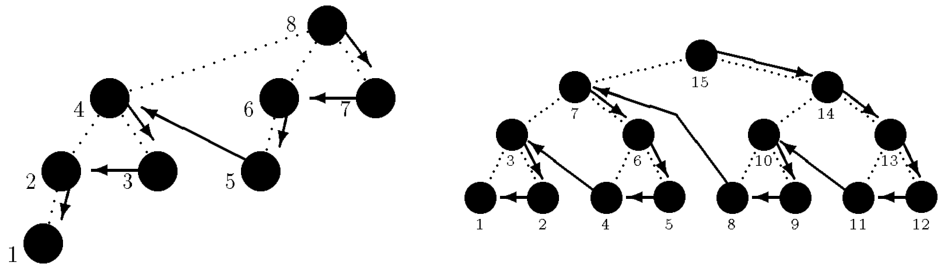


Figure 2.14: Fork-join algorithms and their legitimacy chain in the absence of chain collapsing.

chain. An important case to consider is fork-join parallel algorithms which impose a strict termination order on tasks. In fork-join algorithms, a parent task P sequentially spawns a certain number of children (C_1 to C_k) and later touches the result of the children before terminating. In the absence of collapsing, the legitimacy chain corresponds to a postfix walk of the spawning tree. Figure 2.14 illustrates this for two fork-join procedures (`fj1` and `fj2`). Each node corresponds to a task in the spawning tree. The nodes are numbered according to a postfix walk of the tree (the left child is spawned first) and the arrows represent links of the legitimacy chain (e.g. task 8 is legitimate if task 7 is legitimate). Note that the link coming out of task $i+1$ is only filled in when task i terminates. Due to the fork-join nature of the program, all tasks in the spawning tree rooted at task i will have terminated when task i terminates. This implies that when task i terminates, all links of the legitimacy chain enclosed in task i's spawning tree are known and can be collapsed. In the worst case, this collapsing will stop at L_i, the leftmost task in task i's spawning tree. In other words, task i will set task $i+1$'s legitimacy link to L_i. But, as shown in Figure 2.15, if $i = C_j$ (i.e. i is the j^{th} child of P), then either $i+1 = P$ or $i+1 = L_{C_{j+1}}$. It follows that the collapsing of the links in the legitimacy chain between P and L_P takes at most k sequential steps after all children are done. Given that the spawning of the children by P takes $\Omega(k)$ time anyway, the cost of propagating legitimacy does not change the complexity of the program. There is only a constant overhead per task created. This overhead is rather low since it amounts

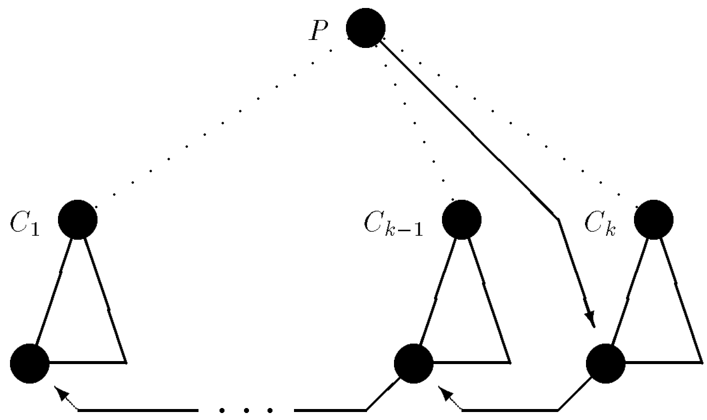


Figure 2.15: General case of legitimacy chain collapsing for fork-join algorithms.

to following one link of the legitimacy chain per task spawned. This result holds for any fork-join algorithm regardless of how well balanced the spawning tree is (including the fork-join DAC procedures `fj1` and `fj2` above as well as the "linear" fork-join procedure `pmap` in Section 2.4.2).

2.9 Benchmark Programs

In order to guide the design process and provide a basis for evaluating and comparing the performance of the implementation strategies, it is important to identify the salient characteristics of the target applications. Following common practice, a set of benchmark programs were selected as representives of "typical" applications of Multilisp. These benchmark programs are used throughout the thesis for various evaluation purposes.

The biggest flaw of these benchmarks is their small size. Real applications will probably be much longer and more complex. Characteristics such as locality of reference, paging, task granularity and available parallelism may be substantially different. Small programs are no substitute for the real thing. They can only serve as rough models of real applications. The main advantage of small programs is that they usually stress a well defined part of the system, so the measurement can be interpreted more readily.

Both sequential and parallel benchmarks were used. The sequential benchmarks are mostly taken from the Gabriel suite [Gabriel, 1985] which has traditionally been used to evaluate implementations of Lisp. To these benchmarks were added four sequential benchmarks: `compiler` (the Gambit compiler), `conform` (a type checker), `earley` (a parser) and `peval` (a partial evaluator). These are sizeable programs that achieve some useful purpose (`compiler` contains more than 15,000 lines of Scheme code). Note that for some measurements it was not possible to run `compiler` due to lack of memory.

There are twelve parallel benchmarks. Half of these were originally written in Mul-T by Eric Mohr as part of his PhD thesis work [Mohr, 1991]. To these were added a few classical parallel programs (matrix multiplication, parallel prefix and parallel reduction) and programs based on pipeline parallelism (polynomial multiplication and quicksort). A general description of the parallel benchmarks is given next. None of the benchmarks require the Katz-Weise continuation semantics or legitimacy (Chapter 5 evaluates their cost in another way). Appendix A contains some additional details including the source code and compilation options. Appendix B contains execution profiles for the benchmarks. These indicate the activity of the processors as a function of time, thus allowing a better visualization of the program's behavior.

2.9.1 `abisort`

This program sorts $n = 16384$ integers using the adaptive bitonic sort algorithm [Bilardi and Nicolau, 1989]. This algorithm is optimal in the sense that, on the PRAM-EREW[13] theoretical model, it runs in $O(\frac{n \log n}{p})$ time, where p is the number of processors and $1 \leq p \leq \frac{n}{2^{\lfloor \log \log n \rfloor}}$. To achieve this performance, `abisort` stores the sequence of elements in a *bitonic tree* which is a full binary tree with the property that many elements can be logically exchanged by a small number of pointer exchanges. To sort a tree, both subtrees are first sorted recursively in parallel and then they are merged. The advantage of this algorithm over mergesort is that the merging of bitonic trees can be done in parallel. Both the recursive sorting phase and the merging phase are based on parallel fork-join DAC algorithms. `Abisort` puts high demands on the memory interconnect because it frequently references and mutates the shared bitonic tree data structure.

2.9.2 `allpairs`

This program computes the shortest paths between all pairs of $n = 117$ nodes using a parallel version of Floyd's algorithm. The input is a square distance matrix D where D_{ij} is the length of the edge between nodes i and j. The algorithm goes through n

[13] Parallel Random Access Machine with Exclusive Read Exclusive Write memory.

steps, each of which updates D in place based on its current state. At the beginning of the k^{th} step, D_{ij} represents the length of the shortest path from i to j that does not go through any node greater or equal to k. The update operation consists of replacing for each possible i and j, D_{ij} by $D_{ik} + D_{kj}$ if that value is smaller. Since D_{kk} is always 0, neither row k or column k of D will change during the k^{th} step. Consequently, all update operations of a given step can be done concurrently. Parallelizing both the loop on i and j would have resulted in an unnecessarily fine task granularity so only the outermost of the two loops was done in parallel (by a parallel fork-join DAC loop). The computation thus consists of a sequence of 117 steps, each of which contains 117 tasks. The execution profile for this program looks like a "comb" where each "tooth" corresponds to one step of the outer loop. `Allpairs` has the coarsest task granularity and the highest run time of all the benchmarks.

2.9.3 `fib`

This program computes F_{25}, the 25^{th} fibonacci number, using the straightforward (but obviously inefficient) doubly recursive algorithm. It is a very compute intensive benchmark which does not reference any heap allocated data. `Fib` is interesting to examine because it can serve as a model for fine grain fork-join DAC algorithms. `Fib` has the finest task granularity of all the benchmarks. The spawning tree is fairly bushy but is not perfectly balanced. The imbalance follows the golden ratio: each subtree has roughly 62% more tasks on the fat branch than on the other branch.

2.9.4 `mm`

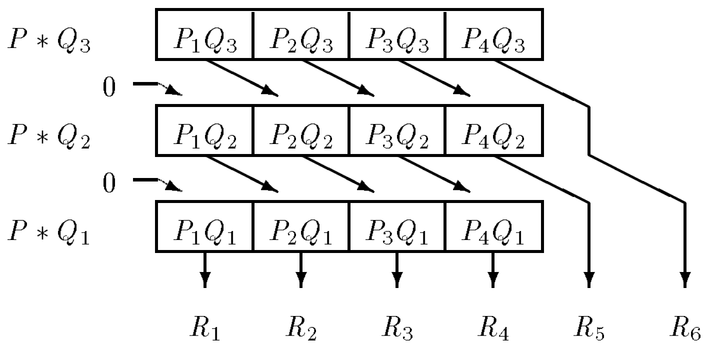
This program multiplies two matrices of integers (50 by 50). The standard algorithm with three nested loops is used. All these loops can be parallelized but only the two outermost loops were turned into parallel fork-join DAC loops. The program thus involves 2500 fairly coarse grain tasks, each of which is in charge of computing one of the entries in the result matrix.

2.9.5 `mst`

This program computes the minimum spanning tree of an $n = 1000$ node graph. A parallel version of Prim's algorithm is used. The input is a symmetric distance matrix D where D_{ij} is the length of the edge between node i and node j. The algorithm constructs the minimum spanning tree incrementally in $n-1$ steps. It starts with a set of nodes containing a single node and at each step it adds to this set the node not yet in the set that is closest to one of the nodes in the set. In order to find the closest node quickly, each node not yet in the set remembers the shortest edge that connects it to the set. This "shortest connecting edge" must be recomputed when a new node is added to the set. The k^{th} step is a loop over $n-k$ nodes that first recomputes each node's shortest connecting edge based on the last node added to the set and then finds the shortest of these edges. `Mst` performs this loop in parallel using a parallel fork-join DAC loop. Note that the degree of parallelism decreases with time (this is clearly visible in the execution profile). The k^{th} step involves $n-k$ tasks.

2.9.6 `poly`

This program computes the square of a 200 term polynomial of x (with integer coefficients). The resulting polynomial is then evaluated for a certain value of x. This ensures that the computation of all coefficients has finished. Polynomials are represented as a list of coefficients. The product of two polynomials P and Q with coefficients $(P_1, \ldots, P_n)$ and $(Q_1, \ldots, Q_m)$ is obtained by first computing the product of P and $Q' = (Q_2, \ldots, Q_m)$ and then adding the result shifted by one position to P scaled by Q_1. The following diagram shows the unfolded recursion for computing $R = PQ$ when $n = 4$ and $m = 3$



This algorithm is coded with two loops. The inner loop does the operations corresponding to a row in the above diagram. It combines the scaling and summing operations in a single multiply-and-add step. The result of the inner loop is the list of coefficients to be added by the next row. `Poly` exploits the parallelism available in the inner loop in a way similar to the procedure `pmap` of Figure 2.2. The multiply-and-add step corresponding to P_iQ_j is done after spawning a task to process the rest of row j. Consequently, there is one task per multiply-and-add step. Moreover, the processing of the rows is pipelined (the processing of row $j+1$ can start before the processing of row j is finished). An alternative algorithm is to spawn a task for each coefficient of R. Task k computes

$$R_k = \sum_{j=\max(1,k-m)}^{\min(k,m)} P_{k-j+1} Q_j$$

Because it spawns fewer tasks ($O(n+m)$ instead of $O(nm)$), this algorithm is probably more efficient. However, the first algorithm was chosen because it is more representative of applications with fine grain pipeline parallelism.

2.9.7 `qsort`

This program sorts a list of 1000 randomly ordered integers using a parallel version of the Quicksort algorithm. The list's head element is used to construct two sublists with the remaining elements: a list of the smaller values and a list of the not smaller values. The two partitions are then sorted in parallel. The partitioning procedure uses a pipeline parallelism technique similar to the procedure `pmap`. The beginning of the partition is available to the continuation before the rest of the list has been partitioned. This means that the sorting of the partition can start as soon as the first element of the partition is generated. Although there are more efficient parallel sorting algorithms (e.g. `abisort`), `qsort` is interesting to consider because it combines pipeline parallelism and DAC parallelism.

2.9.8 `queens`

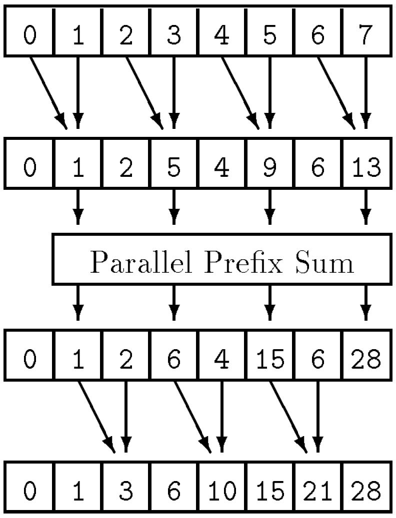
This program computes the number of solutions to the n-queens problem, with $n = 10$. It is based on a recursive procedure which, given a placement of k queens on the first k rows, computes the number of legal ways the remaining $n-k$ queens can be placed (a queen must not be on the same row, column or diagonal as another queen). For each valid position of a queen on row $k+1$, the procedure spawns a task that calls the procedure recursively with the new placement. The number of solutions in each branch is finally summed up. Bit vectors are used to efficiently encode the current placement of queens. As a consequence, `queens` does not access any heap allocated data structure. The call tree is not well balanced. Most branches of the search tree lead to dead ends quickly. `Queens` is a good model for combinatorial search problems such as the traveling salesman problem and the searching of game trees.

2.9.9 `rantree`

This program models the traversal of a random binary tree with on the order of 32768 nodes. The branching factor is 50%. This means that the subnodes of a node are uniformly distributed in the left and right branches. The average length of the paths from the root is 36. Path length roughly follows a normal curve distribution extending from a length of 1 to a length of 73. Like `queens`, `rantree` uses fork-join DAC parallelism, it does not access any heap allocated data and the call tree is not well balanced.

2.9.10 `scan`

This program computes the parallel prefix sum of a vector of 32768 integers. The vector is modified in place. A given element is replaced by the sum of itself and all preceding elements in the vector. `Scan` is based on the "odd-even" parallel prefix algorithm illustrated by the following diagram



The first step is to sum every element at an odd index with its immediate predecessor. The parallel prefix algorithm is then applied recursively to the subvector consisting of the elements with an odd index. Finally every element with an even index is summed with the preceding element (if it exists). When the recursion is unfolded, this algorithm consists of two passes over the vector using tree-like reference patterns. In the Multilisp encoding, the first pass is performed by the combining phase of a parallel fork-join DAC loop whereas the second pass is performed by the dividing phase of a second parallel fork-join DAC loop. These two passes are clearly visible on the execution profile.

2.9.11 sum

This program computes the reduction (using `+`) of 32768 integers stored in a vector. A parallel fork-join DAC algorithm is used. The vector is logically subdivided in two, both halves are then processed recursively in parallel and finally the two resulting sums are added. `Sum` is the finest grain program that accesses heap allocated data. It serves as a model for fine grain data parallel computations such as the reduction of a set of values or the mapping of a function on a set of values.

2.9.12 tridiag

This program solves a tridiagonal system of 32767 equations. The computation proceeds in two sequential phases: the reduction of the system by the method of cyclic reduction [Hockney and Jesshope, 1988] followed by backsubstitution. Cyclic reduction takes a tridiagonal system of order $n = 2^k - 1$ (i.e. n equations over the variables x_0 to x_{n-1}) and produces a reduced tridiagonal system of order $(n+1)/2 - 1$. For each odd numbered equation i, the equations $i-1$, i and $i+1$ are combined in such a way as to eliminate variables x_{i-1} and x_{i+1}. The resulting equation only contains variables x_{i-2}, x_i and x_{i+2} as shown here

$$
\begin{array}{ccccccccccccccc}
\multicolumn{7}{c}{\text{Tridiagonal system}} & & \multicolumn{7}{c}{\text{Reduced system}} \\
0 & + & B_0 x_0 & + & C_0 x_1 & = & Y_0 & & & & & & & & \\
A_1 x_0 & + & B_1 x_1 & + & C_1 x_2 & = & Y_1 & \Longrightarrow & 0 & + & B'_1 x_1 & + & C'_1 x_3 & = & Y'_1 \\
A_2 x_1 & + & B_2 x_2 & + & C_2 x_3 & = & Y_2 & & & & & & & & \\
A_3 x_2 & + & B_3 x_3 & + & C_3 x_4 & = & Y_3 & \Longrightarrow & A'_3 x_1 & + & B'_3 x_3 & + & C'_3 x_5 & = & Y'_3 \\
A_4 x_3 & + & B_4 x_4 & + & C_4 x_5 & = & Y_4 & & & & & & & & \\
 & & \vdots & & & & & & & & \vdots & & & & \\
A_{n-2} x_{n-3} & + & B_{n-2} x_{n-2} & + & C_{n-2} x_{n-1} & = & Y_{n-2} & \Longrightarrow & A'_{n-2} x_{n-4} & + & B'_{n-2} x_{n-2} & + & 0 & = & Y'_{n-2} \\
A_{n-1} x_{n-2} & + & B_{n-1} x_{n-1} & + & 0 & = & Y_{n-1} & & & & & & & &
\end{array}
$$

The reduction process is applied to the reduced system until a single equation of the form $bx_{(n+1)/2-1} = y$ is obtained (this takes $k-1$ reductions). Note that because equation i will not be needed later it can be replaced by the new equation (in other words, the $k-1$ reductions produce an equivalent set of $n-1$ equations). The solution to $x_{(n+1)/2-1}$ is then backsubstituted to find the value of $x_{(n+1)/4-1}$ and $x_{3(n+1)/4-1}$ and so on recursively. After k backsubtitutions, the value of all variables is obtained.

The backsubstitution is implemented with a single tree-like DAC method. The reductions could be directly parallelized by performing a sequence of $k-1$ parallel fork-join DAC loop, but `tridiag` uses a clever tree-like method that has fewer synchronization constraints.

2.10 The Performance of ETC

The main problem with ETC is the high cost of manipulating heavyweight tasks. This section evaluates the best performance that can be expected of ETC for typical programs.

The total work performed by a Multilisp program when run on an n processor machine (i.e. the product of the run time and n) is

$$T_{total}(n) = T_{seq}\, O_{expose}\, O_{exploit}(n)$$

T_{seq}, O_{expose} and $O_{exploit}(n)$ all depend on the program. T_{seq} corresponds to the run time of a sequential version of the program (the parallel program with futures and touches removed). The overhead of parallelism is split into two components[14]. O_{expose} represents the overhead of exposing the parallelism to the system. It reflects the extra work performed by the futures and touches in the program with respect to the sequential version. The product $T_{seq}\, O_{expose}$ is thus the run time of the parallel program on one processor (i.e. T_{par}). The extra work is the sum of the costs for each future and touch executed by the program

$$O_{expose} = 1 + \frac{\sum_{i=1}^{N_{future}} T_{future\,i} + \sum_{i=1}^{N_{touch}} T_{touch\,i}}{T_{seq}}$$

N_{future} and N_{touch} are respectively the number of futures and touches evaluated by the program. $T_{future\,i}$ and $T_{touch\,i}$ are respectively the cost of the i^{th} future and touch operations when only one processor is being used. In general, the costs of these operations are not constant because they depend on several factors including the task scheduling order (which might vary from one run to the next), the compiler's ability to

[14] Overheads are expressed as multipliers. An overhead of x indicates that the amount of work (or other measure) is larger by a factor of x. Consequently, an overhead below 1 indicates a decrease. The term "an overhead of $x\%$" is used to denote small overheads. It means an overhead of $1 + \frac{x}{100}$.

generate special case code for the operation given its particular location in the program, and the complexity of the task to be created, suspended or resumed. For evaluating best case performance, it is useful to define a minimum cost for futures and touches: T_{future_min} and T_{touch_min} respectively. This leads to the following lower bound on O_{expose} (expressed as a function of T_{future_min} and the program's granularity)

$$O_{expose} \geq 1 + \frac{N_{future}\,T_{future\_min} + N_{touch}\,T_{touch\_min}}{T_{seq}} \geq 1 + \frac{T_{future\_min}}{G}$$

G is a measure of the program's granularity. It is the average amount of computation performed by each task ($G = \frac{T_{seq}}{N_{future}}$).

The second part of the parallelism overhead, $O_{exploit}(n)$, indicates how well the program's parallelism is exploited by the system. It corresponds to the additional work performed when running the parallel program on an n processor machine. $O_{exploit}(n)$ contains the following costs not present in O_{expose}: memory interconnect contention and processor starvation (i.e. lack of tasks to run). Processor starvation is both dependent on the program's degree of parallelism and on the scheduler's speed at assigning runnable tasks to idle processors. In addition, $O_{exploit}(n)$ reflects the variation in scheduling order which might cause an increase or decrease in the number of tasks suspended and resumed. By definition, $O_{exploit}(1) = 1$.

In ETC, T_{future_min} is relatively high. If it is assumed that all tasks created eventually run and terminate, T_{future_min} is the cost of creating, starting and terminating a heavyweight task[15]. The bare minimum work caused by the evaluation of a future corresponds to the following sequence

(1) Creating a closure for the future's body.

In `make-FUTURE`:

(2) Creating the result placeholder, associated lock and waiting queue.
(3) Creating the child's initial continuation.
(4) Creating the child task object.
(5) Locking the work queue.
(6) Enqueuing the child on the work queue.
(7) Unlocking the work queue.

[15] All tasks terminate in programs with mandatory tasks (those that perform all the work of their sequential counterpart). This is the case for all the parallel benchmarks.

In `idle`:

(8) Locking the work queue.

(9) Dequeuing the child from the work queue.

(10) Unlocking the work queue.

(11) Restoring the child's continuation.

In `determine!`:

(12) Locking the result placeholder.

(13) Setting the placeholder's value and `determined?` flag.

(14) Checking for suspended tasks to reactivate.

(15) Unlocking the placeholder.

This sequence does not include the operations for dynamic scoping, Katz-Weise continuation semantics and legitimacy. A few tricks can be used to improve the efficiency of this sequence. The heap allocations of steps 1 through 4 can be combined to reduce the cost of checking for heap overflow. In fact, nothing prevents the closure, placeholder, task object and initial continuation to be the same physical object. This reduces the effectiveness of garbage collection (all objects are retained for as long as any of them is reachable) but it does lessen the object formatting overhead. The use of local work queues also permits some optimization of the locking and unlocking of the work queue. To simplify step 2, 13 and the touch operation, a special value can be assigned to the placeholder's value slot to indicate that it is undetermined.

Even with all these optimizations, the sequence and the associated control flow instructions will translate into a moderate number of instructions, probably around 50 to 100 machine instructions. The performance of previous implementations of ETC seem to confirm this lower bound. The Mul-T system was carefully designed to minimize the cost of ETC [Kranz *et al.*, 1989]. When run on an Encore Multimax, Mul-T requires roughly 130 machine instructions to implement the sequence (the actual cost depends on the number of closed variables, their location, etc.). Other compiler based systems require even more instructions. Portable Standard Lisp on the GP1000 [Swanson *et al.*, 1988] takes 480 μsecs (about 1440 instructions given that each processor gives out 3 MIPS) and QLisp on an Alliant FX/8 [Goldman and Gabriel, 1988] takes 1400 instructions.

With this lower bound on T_{future_min} it is possible to get a lower bound on O_{expose} from the value of G. The left part of Table 2.1 gives the value of G, T_{seq}, N_{future} and N_{touch} measured for the benchmark programs when run on the GP1000 with a single processor. The benchmarks have been ordered by increasing granularity. Note that the

Program	G in μsecs	T_{seq}	N_{future}	N_{touch}	Lower bound on O_{expose} when T_{future_min} in μsec is				
					1	5	25	125	625
`fib`	7	.819	121392	121392	1.15	1.74	4.71	19.54	93.69
`sum`	12	.392	32767	32767	1.08	1.42	3.09	11.45	53.24
`qsort`	16	.210	13318	27637	1.06	1.32	2.59	8.94	40.71
`scan`	16	1.061	65534	65534	1.06	1.31	2.54	8.72	39.60
`queens`	31	1.092	34814	34814	1.03	1.16	1.80	4.99	20.93
`rantree`	42	.394	9487	9487	1.02	1.12	1.60	4.01	16.05
`abisort`	44	4.734	106496	106496	1.02	1.11	1.56	3.81	15.06
`poly`	58	2.308	39801	40200	1.02	1.09	1.43	3.16	11.82
`mst`	94	23.414	249001	249001	1.01	1.05	1.27	2.33	7.65
`tridiag`	161	3.958	24574	24574	1.01	1.03	1.16	1.78	4.88
`mm`	624	1.558	2499	2499	1.00	1.01	1.04	1.20	2.00
`allpairs`	1831	24.852	13572	13572	1.00	1.00	1.01	1.07	1.34

Table 2.1: Characteristics of parallel benchmark programs running on GP1000.

number of futures is equal to the number of touches for all benchmarks based on fork-join parallelism (all benchmarks except `qsort` and `poly`). The right part of the table gives the lower bound on O_{expose} computed from G and various values of T_{future_min}.

According to this table, an optimized version of ETC (i.e. one with T_{future_min} = 25μsecs = 75 machine instructions) will have an overhead that spans a range from essentially nonexistent to fairly sizeable. As the granularity decreases, the overhead increases and almost reaches a factor of 5 for fine grain programs. This overhead is a conservative estimate. Mul-T's implementation of ETC gives a measured value of O_{expose} = 8.9 for `fib` [Mohr, 1991]. Whether this is an acceptable overhead or not for "typical" programs is of course a subjective matter. However, it is clear that a high overhead for fine grain programs will have an impact on the style of programming adopted by users.

There will be a high incentive to design programs with coarse grain parallelism even if there exists a natural fine grain solution. Frequently it is possible to manually transform a fine grain program into a coarser grain program by grouping several small tasks into a single one that executes them sequentially (this is akin to unrolling loops by hand in sequential languages to reduce the loop management overhead). This type of transformation has several drawbacks. If the task grouping is artificial, the program

```
(define (fib n)
  (if (< n 2)
    n
    (let* ((x (FUTURE (fib (- n 1))))
           (y (fib (- n 2))))
      (+ (TOUCH x) y))))
```

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib2 (- n 1)) (fib2 (- n 2)))))

(define (fib2 n)
  (if (< n 2)
    n
    (let* ((x (FUTURE (fib (- n 1))))
           (y (fib (- n 2))))
      (+ (TOUCH x) y))))
```

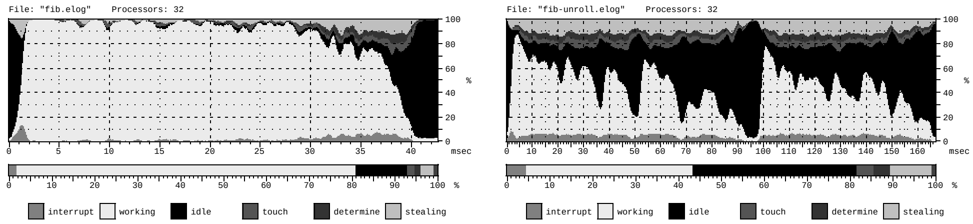


Figure 2.16: `Fib` and a poor variant obtained by unrolling the recursion.

becomes more complex and harder to maintain. An overhead cost must also be expected if task grouping is managed dynamically by user code (as is the case for the depth and height cutoff methods proposed for tree-like computations by Weening [Weening, 1989]). The transformation is also error prone. Logical bugs as well as performance problems can be introduced by the user. For example, the recursion of `fib` can be unrolled once as shown in Figure 2.16 to double the task granularity. One might expect the program to be more efficient because of the lower task management overhead but in reality it performs poorly because a sequential dependency has been introduced (this can be seen clearly in the execution profiles). Finally, the program will be less portable because the selection of an appropriate granularity depends on several parameters of the run time environment (number of processors, task operation costs, shared memory costs, etc).

The problem with a high task management cost is not so much that it prevents the user from attaining good performance. The problem is that the language cannot realistically be viewed as a high-level language because the user must program at a low-level to attain good performance. Selecting the right granularity for a program can quickly become the user's overriding concern.

The next chapter explores a more efficient approach to task management called *lazy task creation*. The cost of evaluating a future with this approach is very small (T_{future_min} on the order of 1 μsec on the GP1000). Table 2.1 can be used to approximate the overhead of this approach. The finest grain program (i.e. `fib`) should have a value of O_{expose} close to 15%. Note that the table gives a lower bound and that the actual overhead will be somewhat larger. Chapter 5 contains the measured value of O_{expose} for the benchmarks. With such a small overhead, the user has virtually no incentive to avoid fine grain tasks and thus has added liberty in the programming styles that can be used.

Chapter 3

Lazy Task Creation

Several plausible semantics for Multilisp were compared in the preceding chapter. The Katz-Weise semantics with legitimacy is attractive because it provides an elegant interaction between futures and continuations. In addition, dynamic scoping and fairness of scheduling are desirable features. Unfortunately, ETC is not an adequate implementation of futures because its performance is poor on fine grain programs.

This chapter explores *lazy task creation* (LTC), an alternative task creation mechanism that is more efficient than ETC; especially for fine grain programs. The LTC mechanism described here supports the Multilisp semantics given above. Two variants of LTC are examined: one that assumes an efficient shared memory and one that does not. As confirmed in Chapter 5, both variants have roughly the same performance when consistent shared memory is efficient but when this is not the case, for example on large scale multiprocessors, the later variant permits a more efficient execution (faster by as much as a factor of 2 on the TC2000).

In this chapter, algorithms are given in pseudo-C. Assembly code is also used to explain the details of the code sequences generated by the compiler.

3.1 Overview of LTC Scheduling

This section explains the scheduling policy adopted by LTC and its benefits.

Task execution order has a direct impact on performance. The implementation must choose an ordering that minimizes the task management overheads. There are four places where an implementation has liberty as to which task to run next

- Task spawning.
- Task termination.
- Task suspension.
- Preemption interruption.

Only the first two situations are examined here (the last two are discussed in later sections). Any runnable task can be run next in these four situations. However, only the subsets of runnable tasks that are most promising are considered in the following discussion. In particular, the task to run next is preferentially selected from the local work queue because this will promote locality and reduce contention. When the local work queue is empty a task must be *stolen* from another processor's work queue. Task stealing is the only way for work to get distributed between processors. The two processors involved in a task steal are the *thief* processor and the *victim* processor.

When a task is spawned, one of two tasks can be run next by the spawning processor: the child task or the parent task. The ETC implementation described in the preceding chapter uses *parent first* scheduling. When a future is evaluated the child task is made to wait for an available processor whereas the parent task immediately starts executing the future's continuation. LTC uses the reverse scheduling order, *child first* scheduling. The child's execution is started immediately by the spawning processor and the parent is delayed until a processor is ready to run it.

The use of child first scheduling in Multilisp has important advantages. First, it tends to reduce the number of task suspensions caused by touches. The child is computing a value that is used by the future's continuation. Since the parent gets delayed with respect to the child there is a higher likelihood that the child will have completed when its result is first touched by the parent or one of its other descendants.

When a task terminates however, there is no incentive to delay its parent any further. In fact, now that the task's result is known, it makes sense to execute the parent next. Since the parent consumes the value just computed, it is less likely that it will get suspended. This policy will be called *parent next* scheduling.

Child first scheduling combines naturally with parent next scheduling to give an efficient stack-like scheduling policy: *LIFO* scheduling. The set of runnable tasks on a processor is kept in a stack, the *task stack*, associated with that processor (see Figure 3.1). The main operations available on the task stack are: task push, task pop, and

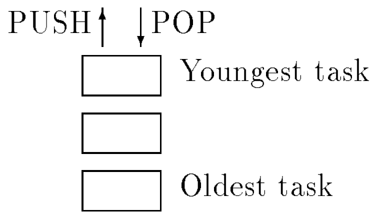


Figure 3.1: The task stack.

task steal. When a task is spawned, the parent is simply pushed onto the task stack and control goes to the child. When a task terminates, the parent is necessarily on top of the task stack if it hasn't been run yet (this assumes that processors can steal but cannot push a task onto another processor's task stack). If the parent is still there, it gets popped from the task stack and executed by the same processor that pushed it.

LIFO scheduling yields a task execution order very similar to that of the program with futures removed. In fact, the execution order is identical when no task is ever stolen from the task stack. This happens for example when the machine has a single processor or when all processors have enough local work to keep them busy. In this situation, there are no task suspensions because the only computation that might touch the task's placeholder (i.e. the continuation) necessarily follows the termination of the task.

3.1.1 Task Stealing Behavior

Under LIFO scheduling, tasks could be stolen from either end of the task stack. Tasks are always stolen from the task stack's bottom in LTC. It is interesting to see why this *bottom stealing* is preferable to *top stealing*. Top stealing might seem better for the same reason as child first scheduling. Favoring the execution of younger tasks should reduce the likelihood of suspension in older tasks.

However, this analysis does not take into account that older tasks generally run longer before termination or suspension than younger tasks. For DAC programs with balanced spawning trees, the task size will decrease geometrically with the task stack depth. When a child task is pushed onto the task stack, the amount of work it contains is a fraction (f) of the amount remaining in the parent[1]. Thus, in a DAC program, the

[1] The amount of work remaining in a task is all the work remaining before its termination including the work contained in the tasks that it will spawn. In a well balanced binary DAC program, such as

i^{th} removed child from a task has f^i times the work of that task and collectively, a task and the d descendants below it on the task stack have $\sum_{i=0}^{d} f^i = \frac{1-f^{d+1}}{1-f}$ the amount of work. This means that the amount of work in the oldest task is approximately equal to that of its youngest $d' = d + 1 - \log_f(1-f)$ descendants[2]. Consequently, the amount of work T_{oldest} remaining in the oldest task is equal to the work in all other tasks on the task stack except a constant number of the oldest tasks. The task stealing overhead will be higher for top stealing because it requires at least d' times more task steals than bottom stealing to distribute T_{oldest} units of work. In reality, the number of steals will be higher than d' because the victim is continuously replenishing the task stack with small tasks as the thief is stealing them. The probability of stealing a task close to the leaves of the spawning tree is relatively high.

Individual task steals are also faster with bottom stealing because there are two nearly independent ways to access the task stack. A processor can push or pop a task from its local task stack while some other processor is simultaneously stealing a task. This parallelism, which is no more than a degree of 2, enables tasks to be created and started faster. In addition, better caching of the task stack top is possible because it is single writer shared data (as opposed to multiple writer shared data for top stealing).

Mohr [Mohr, 1991] has analyzed the task stealing behavior of bottom stealing for tree-like DAC parallel programs. He has derived an upper bound of p^2h task steals for programs with binary spawning trees of height h running on a machine with p processors. This upper bound relies on the use of *polite stealing*. In polite stealing a processor whose last steal was from victim V must try to steal from all other processors before stealing again from V. An outline of Mohr's proof follows.

At any given point in time, a processor i is either idle (and is trying to steal a task) or is in charge of running the tasks in some subtree of the spawning tree. Call h_i the height of processor i's subtree ($h_i = 0$ when it is idle) and H the maximum height of all subtrees ($H = \max_{i=1}^{p} h_i$). After a task is stolen from processor i, both the victim and the thief will be in charge of subtrees of height $h_i - 1$. Note that to decrease H by one it is necessary to steal a task from all processors i with $h_i = H$. Polite stealing guarantees that all these processors will have been tried by a given processor in no more than p steals (or steal attempts). Because up to p processors might be attempting to steal tasks, it will take no more than p^2 steals to steal at least one task from each processor with $h_i = H$. When H reaches zero no tasks are left to steal. Consequently, no more

---

`sum`, f will be close to $\frac{1}{2}$. For `fib`, which has an imbalanced spawning tree, f is about .618. An f close to 1 approximates loop based parallel algorithms such as `pmap`.

[2] This result is obtained by solving for $d' = d - k$ in $1 = \sum_{i=k+1}^{d} f^i = \frac{1-f^{d+1}}{1-f} - \frac{1-f^{k+1}}{1-f} = \frac{f^{k+1}-f^{d+1}}{1-f}$.

than p^2h steals can occur.

In the absence of polite stealing $O(2^h)$ steals can occur (potentially all tasks are stolen). Although polite stealing insures the upper bound of p^2h steals it isn't clear that this makes a difference in practice. Mohr ran programs with and without polite stealing for a wide range of values of h and p. The number of steals was comparable (usually within 10% to 30%) and only in extreme cases was there a noticeable advantage to use polite stealing (a factor of 2 to 3 for high h and p). Gambit uses polite stealing with the particularity that each processor has a probing order generated randomly when the system is loaded. This was done in an effort to reduce interference between competing thief processors. With a sequential probing order there is a potential loss of parallelism because several thieves might become synchronized, following each other in lockstep.

3.1.2 Task Suspension Behavior

Bottom stealing also leads to fewer task suspensions. To simplify the analysis, it is assumed that tasks touch the value of their children just before termination and that there are only two processors.

When bottom stealing, T_{oldest} time units will elapse before the first touch that might cause a suspension. The d' youngest tasks are not affected by the steal so in this time period they will have a suspension-free execution. When $f \leq \frac{1}{2}$ there is necessarily no task suspension because all the descendants have terminated when the touch is performed. A single suspension occurs when $f > \frac{1}{2}$ and the steal happened not too late after the first descendant was spawned.

When top stealing, there are d' tasks (at least) that might suspend in the same time period. The likelihood of suspension increases with the depth of the task due to a combination of two factors. First, deeper tasks have less work and second, it is faster to remove tasks from the local task stack than to steal them from other processors (the costs are respectively T_{local} and T_{steal}). Let T_{task} be the amount of work remaining in the stolen task and T_{child} the work remaining in its currently running child. The stolen task will terminate (or get suspended) in $T_{steal} + T_{task}$ time whereas its parent will touch its value in $T_{child} + T_{local} + \frac{T_{task}}{f}$ time (the processor will finish executing the child and then locally resume the stolen task's parent). A suspension occurs in either of the following cases

1. $T_{steal} + T_{task} < T_{child}$ $\Rightarrow$ stolen task gets suspended
2. $T_{steal} + T_{task} > T_{child} + T_{local} + \frac{T_{task}}{f}$ $\Rightarrow$ stolen task's parent gets suspended

The second case is highly likely for fine grain DAC programs because, as the depth of the task increases, T_{task} and T_{child} become negligible when compared to T_{steal} and it is always the task closest to the leaves of the spawning tree that is being stolen.

3.2 Continuations for Futures

Continuations play a central role in the implementation of futures. A task's state is mostly composed of a continuation. In addition, the Katz-Weise semantics as defined in Figure 2.12 requires that the future's continuation be captured and shared between the child and parent tasks. Consequently, the efficiency of continuation operations and futures are intimately tied. This section describes the implementation of continuations on top of which LTC will be implemented.

Conceptually, a continuation is a chain of frames. Each frame corresponds to some subproblem call that is currently pending completion. A frame contains the context required to perform the computation that follows the corresponding subproblem call. The frame includes temporary values and variables (or alternatively an environment pointer) and also contains a *parent* continuation. The parent continuation is used when the procedure containing the subproblem call exits (by a normal return or a reduction call). This link is what gives the stack structure to continuations. Note that in some situations the parent continuation is never used and could be removed from the frame by a smart compiler[3]. For simplicity, it is assumed that the parent continuation is always present in the frame. The oldest frame's parent is the *root* continuation which is special in that it has no parent. The root continuation symbolizes the end of the program.

Several strategies for implementing continuations have been described and compared by [Clinger *et al.*, 1988]. Their results suggest that the *incremental stack/heap* strategy is more efficient than the other strategies in most cases and not noticeably slower than the other strategies in extreme cases. With the exception of a few details, this is the strategy used by Gambit.

[3] This is permissible if the subproblem call is done inside an infinite loop. For example, in the following definition, the frame for the subproblem call to `g` need not contain `f`'s continuation because `f` never returns.

```
(define (f)
  (g)
  (f))
```

3.2.1 Procedure Calling Convention

Since continuations are manipulated at every procedure call and return, it is important to have efficient support for these common operations. The incremental stack/heap strategy puts very few constraints on procedure calling conventions. This means that the presence of unlimited extent continuations in the language does not impose a special runtime overhead[4].

Parameters can be passed in any location (typically in registers and/or on the stack) and a procedure can return simply by jumping to the return address passed to the procedure by the caller. Within a procedure, the stack can be used freely to allocate temporary values and local variable bindings.

Continuation frames, created at subproblem calls, are always allocated from the run time stack (as is normally done for other languages). The procedure that allocated a frame is responsible for its deallocation from the stack. Deallocation occurs at some point before the procedure is exited (by a normal return or a reduction call). This insures that at the subproblem call's return point, the continuation frame created for the call is still topmost on the stack. A procedure's continuation is thus a combination of two values: the return address and the value of the stack pointer. Note that the return address passed to a procedure is always contained in any continuation frame it creates.

3.2.2 Unlimited Extent Continuations

This implementation can be extended to support unlimited extent continuations. The continuation is split into two parts. The most recently created frames of a continuation are on the stack and the oldest frames reside in the heap. This situation is depicted in Figure 3.2 (where frame i is created by procedure `p`i and `ret`$_i$ is the return address into `p`i). The implicit continuation passed to a procedure is represented by a triplet: (`SP`,`RET`,`UNDERFLOW_CONT`). The stack pointer `SP` points to the topmost frame on the stack and the return register `RET` contains the return address[5]. `UNDERFLOW_CONT` corresponds to the heap continuation and it contains two fields: `link` (a pointer to the

[4] Note that the semantics of continuations in Scheme require that there be only one instance of any variable allocated. To support this, it is common to create a cell in the heap for each mutable variable. The extra dereference needed to access mutable variables adds an overhead whose importance will depend on the program. However, there is no overhead for functional programs.

[5] `RET` could also be passed on the stack but it is simpler to think of it as being contained in a dedicated register. Gambit actually dedicates a register for the return address.

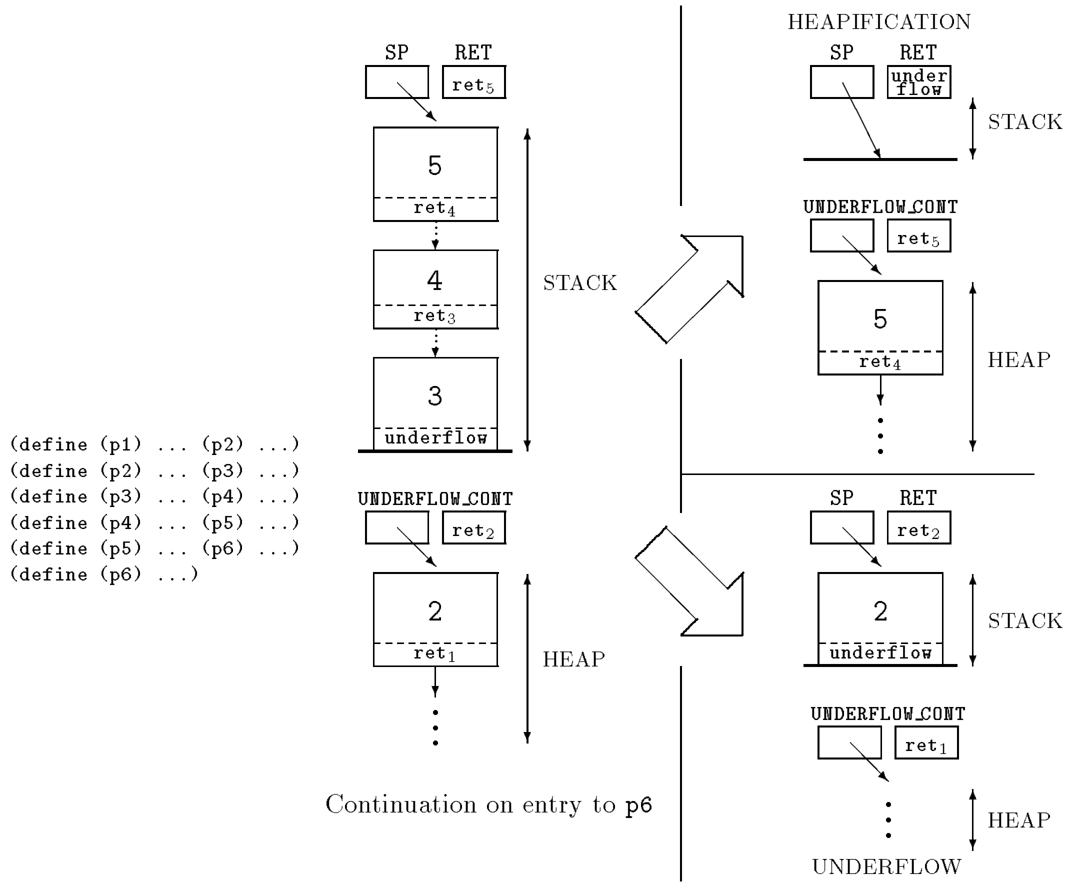


Figure 3.2: Continuation representation and operations.

topmost heap frame) and `ret` (the return address for the topmost heap frame). Note that the stack frames are only linked conceptually; in reality they are allocated contiguously on the stack. On the other hand, heap frames are independent objects in a format suitable for garbage collection and explicit links between them are maintained.

The link between the stack frames and the heap frames is preserved in a special way. This link is traversed when a procedure returns to its continuation and the stack is empty. This is called a stack *underflow*. When the stack underflows, the topmost heap frame must be copied back to the stack so that the return point can access the content of the continuation frame in a normal manner. This is the only frame that is immediately needed. The older heap frames get restored one at a time by subsequent underflows.

A special mechanism is used to avoid having to check explicitly for stack underflow at every procedure return. The return address logically attached to the oldest stack frame is stored in `UNDERFLOW_CONT.ret`. In its place, the continuation frame contains a pointer to the *underflow handler*. This handler consequently gets called by the normal procedure return mechanism when the stack underflows. The handler performs the following sequence of steps: the correct return address is extracted from `UNDERFLOW_CONT.ret`, the topmost heap frame is copied to the stack, `UNDERFLOW_CONT` is updated to represent the parent heap frame, the return address in the stack frame is replaced by the underflow handler to prepare it for underflow, and finally control is returned to the correct return address. The cost for an underflow is thus dependent on the frame size which in typical cases is fairly small. For example, the largest frame size for the parallel benchmarks is 10 slots and the average, measured statically, is just below 4. An underflow should thus be fairly cheap for these programs (between 10 and 20 instructions if the underflow handler and heap frame format are chosen carefully).

3.2.3 Continuation Heapification

Heap continuations are created by the process of *heapification*. Heapification transforms the current continuation into one that only contains heap frames. The stack frames are transferred one by one to the heap with the appropriate links between them. The oldest stack frame must be handled specially. When it is copied, its return address is first recovered from `UNDERFLOW_CONT.ret` and its parent link is obtained from `UNDERFLOW_CONT.link`. Finally, the stack is cleared by resetting `SP` to the bottom of stack, and `RET` and `UNDERFLOW_CONT` are updated to reflect the new location of the continuation. The current continuation before and after heapification are logically equivalent; only the representation changes.

3.2.4 Parsing Continuations

One complication with the underflow and heapification mechanisms is that it must be possible to parse the stack to know where each frame begins and ends, and also which frame slot contains the return address[6]. One way to achieve this is to associate the description of a frame's layout (length and return address location) with the return address of the subproblem call that created the frame. The frame descriptor can for example be stored just before the return point, as is done in [Hieb *et al.*, 1990]. `RET` can then be used to get the size of the topmost stack frame and the location of its return address. The return address in this frame in turn gives the size of the next frame and so on.

The heapification and underflow mechanisms can now be described in detail. The algorithms are given in Figure 3.3. In these algorithms two functions are used to parse the continuation: `frame_size(`r`)` and `ret_adr_offs(`r`)` return respectively the size and return address offset of the continuation frame associated with return address r. It is assumed that all data structures grow towards higher addresses and that, in all drawings, addresses grow towards the top of the page.

3.2.5 Implementing First-Class Continuations

First-class continuations can easily be implemented with the heapification mechanism. `Call/cc` first heapifies its implicit continuation and then packages up `UNDERFLOW_CONT` in a new closure. When called, this closure discards the current continuation by resetting `SP` to the bottom of stack, restores the new continuation by setting `UNDERFLOW_CONT` to the saved value, and then jumps to the underflow handler to transfer control to the return point. Support for dynamic scoping is a simple addition to this mechanism. The current dynamic environment is saved in the closure at the moment of the `call/cc` and is restored just before jumping to the underflow handler.

Heapification might seem to be doing more work than strictly required by `call/cc`. By leaving the stack in its original state after its content is copied to the heap, some returns would become cheaper because the restoration of the frames by the underflow mechanism would be avoided. However, new costs in space and time would be introduced

[6] The ability to parse the stack is also useful to implement introspective tools such as debuggers and profilers.

```
typedef struct frm                  /* heap frame format      */
{ struct frm *link;                 /* parent frame pointer   */
  value slots[];                    /* content of frame       */
} frame;

value *SP;
instr *RET;
struct { frame *link; instr *ret; } UNDERFLOW_CONT;


underflow()
{
  frame *f = UNDERFLOW_CONT.link;   /* get topmost heap frame */
  instr *r = UNDERFLOW_CONT.ret;    /* get return address     */

  for (i=0; i<frame_size(r); i++)   /* copy frame to stack    */
    SP[i] = f->slots[i];

  UNDERFLOW_CONT.link = f->link;    /* prepare for underflow  */
  UNDERFLOW_CONT.ret  = SP[ret_adr_offs(r)];
  SP[ret_adr_offs(r)] = underflow;

  SP += frame_size(r);              /* update stack pointer   */

  jump_to( r );                     /* jump to return point   */
}


heapification()
{
  if (RET != underflow)             /* check for empty stack  */
    heapify_frame( SP, RET );

  SP  = bottom_of_stack;            /* clear stack            */
  RET = underflow;
}


heapify_frame( s, r )
value *s;
instr *r;
{
  value *b = s - frame_size(r);       /* compute frame's base  */
  frame *f = alloc( frame_size(r) );  /* allocate heap frame   */
  instr *p = b[ret_adr_offs(r)];      /* get parent ret adr    */

  if (p == underflow)                 /* oldest frame?         */
    b[ret_adr_offs(r)] = UNDERFLOW_CONT.ret;
  else
    heapify_frame( b, p );

  for (i=0; i<frame_size(r); i++)     /* copy frame content    */
    f->slots[i] = b[i];
  f->link = UNDERFLOW_CONT.link;      /* link frame to parent  */

  UNDERFLOW_CONT.link = f;            /* update UNDERFLOW_CONT */
  UNDERFLOW_CONT.ret  = r;
}
```

Figure 3.3: Underflow and heapification algorithms.

since there could now be multiple copies of the same stack frame. This occurs when multiple continuations which share the same tail are captured. Programs with nested calls to `call/cc`, such as those typically found in backtracking algorithms and exception processing, exhibit this behavior. As an example, consider this definition for `f`

```
(define (f n)
  (if (zero? n)
    0
    (+ 1 (call-with-current-continuation
           (lambda (cont)
             (f (- n 1)))))))
```

Note that the call `(f` n`)` calls `call/cc` n times. If there are k stack frames in the continuation for the call `(f` n`)`, $n(k + \frac{n+1}{2})$ heap frames will be created. The sharing properties of heapification are much better because there is at most one heap copy of any continuation frame. In the example, only $k + n$ heap frames will be created (a savings of a factor of $O(n)$). The same reasoning holds for nested futures when they are implemented with `call/cc` (as is the case for the implementation of the Katz-Weise semantics shown in Figure 2.12).

3.3 The LTC Mechanism

An important benefit of combining LIFO scheduling and bottom stealing is that it promotes stack-like execution. For fork-join DAC programs, entire subtrees of the spawning tree get executed in an uninterrupted stack-like fashion because it is the older tasks that get stolen (those closer to the spawning tree's root). Since the tasks in these subtrees are exactly those that are not stolen, they will be called *non-stolen* tasks. Stack-like execution stops only when the oldest non-stolen task terminates (the one at the non-stolen subtree's root).

LTC presupposes that this stack-like execution is the predominant execution order. In other words, LTC speculates that most tasks are not stolen. Several task spawning steps are only required if the task is stolen. Referring to Figure 2.12, these steps include: the heapification of the parent continuation (the call to `call/cc`), and the creation and manipulation of the task's result and legitimacy placeholders (the calls to `make-ph`). LTC postpones these steps until it is known that the task is stolen (this explains the name "lazy task creation"). In summary, non-stolen tasks completely avoid these steps whereas stolen tasks perform these steps when the task is stolen.

To achieve this, LTC uses a lightweight task representation. When a future is evaluated, a lightweight task representation of the parent task is pushed on the task stack. The task stack push and pop operations, which are the only operations needed for a purely stack-like execution, can be implemented at a very low cost with this representation. Moreover, there is enough information in a lightweight task to recreate the corresponding heavyweight task object if the task is ever stolen from the task stack. The rest of this section is a more detailed description of the LTC mechanism. The important issue of synchronization between the thief and victim is discussed in the section that follows.

3.3.1 The Lazy Task Queue

The task stack is represented by a group of three stack-like data structures: the run time stack, the *lazy task queue* (LTQ), and the *dynamic environment queue* (DEQ). The same terminology as [Mohr, 1991] has been used when possible for consistency. The term *lazy task* refers to a task in the lightweight representation (i.e. a task contained in the task stack). These three data structures are really double ended queues which are mostly used as stacks. Items can be pushed and popped from the tail of these queues. Items can also be removed from the head. For efficiency, the entries are laid out contiguously in memory. For the LTQ and DEQ, two pointers indicate the extent of the queue (the head and tail).

The run time stack contains the continuation frames of all the tasks in the task stack. The LTQ and DEQ contain pointers to continuation frames in the run time stack. The DEQ, which is only needed to support dynamic scoping, is explained in Section 3.3.4. The purpose of the LTQ is to keep track of each lazy task's continuation. For each lazy task in the task stack there is exactly one pointer on the LTQ. Each pointer points to the first continuation frame of the corresponding future's continuation. The "before" part of Figure 3.5 shows a possible state of the LTQ and run time stack on entry to procedure `p9` after a call to procedure `p1`

```
(define (p1) ... (p2) ...)
(define (p2) ... (p3) ...)
(define (p3) ... (FUTURE (p4)) ...)
(define (p4) ... (p5) ...)
(define (p5) ... (FUTURE (p6)) ...)
(define (p6) ... (p7) ...)
(define (p7) ... (FUTURE (p8)) ...)
(define (p8) ... (p9) ...)
(define (p9) ...)
```

The LTQ's `TAIL` points to the youngest entry on the LTQ whereas `HEAD` points just below the oldest entry. Thus, the LTQ is non-empty if and only if `HEAD` < `TAIL`. Otherwise, the LTQ is empty and HEAD = TAIL. The same is true for the DEQ with the pointers DEQHEAD and DEQTAIL.

3.3.2 Pushing and Popping Lazy Tasks

The task stack's push and pop operations translate into a small number of steps. When a future is evaluated, the thunk representing the future's body is called as a subproblem. The continuation frame created on the run time stack for this call corresponds to the first frame of the parent task's continuation. To indicate the presence of the parent task on the task stack, a pointer to the continuation frame (i.e. SP) is pushed on the LTQ (thereby incrementing TAIL) upon entering the thunk. This pointer is used by the steal operation to recreate the parent task. The processor has effectively queued the parent on the task stack and is now running the child. When the thunk returns, the LTQ is either empty (indicating that the parent was stolen), or not (indicating that the parent is still on the LTQ). If the LTQ is not empty, the parent task gets resumed in parent next fashion. Note that at this point both SP and the topmost pointer on the LTQ point to the parent's continuation frame. To pop the parent task it is sufficient to place an instruction that decrements TAIL at the subproblem call's return point. After decrementing TAIL, the processor has effectively terminated the child and resumed the parent. The body's result has been transferred from the child to the parent without having to create a placeholder. Moreover, legitimacy propagation cost nothing because the parent task's legitimacy before and after executing the child are identical. A single legitimacy flag, CURRENT_LEGITIMACY, is needed per processor. It logically corresponds to the legitimacy of the task currently running on that processor. Similarly, each processor has a CURRENT_DYNAMIC_ENV variable that is always bound to the dynamic environment of the currently running task. There is no need to change this variable when a lazy task is pushed or popped from the task stack. The handling of a stolen parent is explained in the next section.

It would seem that most of the work to push a task on the task stack goes into two operations: the creation of the closure for the body and the creation of the continuation frame. However, these operations do not really constitute an important overhead with respect to a purely sequential execution of the program.

Firstly, it isn't necessary to heap allocate the closure because its single call site is known. It is more efficient to lambda-lift the closure so that the closed variables are passed to the body as parameters. Frequently, these variables are already in registers so they can be left as is for the body to use. As shown in Table 3.1, most of the

Program	Number of closed variables for each future and number copied
abisort	3 (0), 1 (0)
allpairs	7 (3)
fib	1 (0)
mm	7 (3), 6 (2)
mst	5 (1)
poly	3 (0)
qsort	3 (0)
queens	6 (2)
rantree	3 (0)
scan	5 (1), 4 (0)
sum	4 (0)
tridiag	4 (0), 3 (0)

Table 3.1: Size of closure for each future in the benchmark programs.

benchmarks require little or no work to setup the closed variables for the body because they are already in registers (Gambit does a good job at allocating variables to registers). A system could be designed to avoid any copying by directly accessing the closed variables in the parent continuation frame. However, this would create dependencies between frames which are hard to manage (in particular, heapification would become more complex and expensive because the frames can't be separated).

Secondly, the continuation frame created by the future can be reused by the future's body. Futures are typically subproblems and have a procedure call as their body (all the futures in the benchmarks are like this). A sequential version of the program would create a continuation frame for the call, just before the procedure is invoked. The same continuation frame is created by the future but there is no need to create another frame for the call in the body since it is now a reduction call. The only difference is that the frame is created before the arguments to the procedure are evaluated rather than afterwards but the cost will be the same.

```
resume_task( t )
task *t;
{
  CURRENT_TASK = t;
  UNDERFLOW_CONT.link = CURRENT_TASK->cont_link;
  UNDERFLOW_CONT.ret  = CURRENT_TASK->cont_ret;
  CURRENT_DYNAMIC_ENV = CURRENT_TASK->cont_denv;
  result_location     = CURRENT_TASK->cont_val;
  CURRENT_LEGITIMACY  = CURRENT_TASK->leg_flag;
  SP = bottom_of_stack;
  TAIL = bottom_of_LTQ;
  HEAD = bottom_of_LTQ;
  DEQTAIL = bottom_of_DEQ;
  DEQHEAD = bottom_of_DEQ;
  underflow();
}
```

Figure 3.4: Resuming a heavyweight task.

3.3.3 Stealing Lazy Tasks

When a thief processor steals a lazy task from a victim processor's task stack, it removes the oldest entry on the LTQ (thereby incrementing `HEAD`) and then must do three things: recreate the parent task as a heavyweight task object, notify the victim so that it knows the oldest lazy task is no longer on the task stack, and finally resume the parent task.

A heavyweight task is represented with a structure containing five fields

- `cont_link`
- `cont_ret`
- `cont_denv`
- `cont_val`
- `leg_flag`

The first four fields describe the task's continuation. `Cont_link` is a pointer to the continuation frames in the heap, `cont_ret` is the continuation's return address, `cont_denv` is the continuation's dynamic environment, and `cont_val` is the value passed to the continuation when the task is resumed. The fifth field, `leg_flag`, is the task's legitimacy flag. Resuming a heavyweight task is performed by the steps in Figure 3.4. Note that variables are local to the processor unless explicitly marked otherwise (the notation P`->`v, where P is a processor, will be used to denote P's local variable v). Thus, `resume_task` first sets the processor's current task and, after initializing the task stack, uses the underflow mechanism to restore the task's continuation. The value in `cont_val` is passed to the continuation by setting `result_location`. It is assumed that all continuations, including those for futures, receive their result in this location (`result_location` is a machine register in Gambit). This restriction could be lifted by parameterizing the result location by the return point, that is `UNDERFLOW_CONT.ret` (this would require adding a field to the frame descriptor).

Figure 3.5 will help illustrate the effect of a steal on the LTQ and run time stack. The pointer p removed from the victim's LTQ points to the first continuation frame of the corresponding task (frame 3 in the figure). To ease its manipulation, the task's continuation is first heapified from this continuation frame down to the next frame having the underflow handler as its return address. This is achieved by the call

`heapify_frame(` p, r `)`

where r corresponds to the return address associated with frame p (i.e. $\mathtt{ret}_3$ in the example). In addition, r must be replaced by a pointer to the underflow handler so that the child invokes `UNDERFLOW_CONT` when it is done. An important issue is how to locate r from p but for now this operation will be hidden in the procedure `swap_child_ret_adr_with_underflow(`p`)` that sets r to `underflow` and returns its previous value. The victim's current continuation is now logically the same as before; only the representation has changed.

After being heapified, the future body's continuation is in `UNDERFLOW_CONT`. Note that `UNDERFLOW_CONT.ret` contains the address of the subproblem's return point. The first instruction at this address is the one which decrements `TAIL`. The only purpose of this instruction is to pop the parent task on a "parent next" transition and it shouldn't be executed in any other case. The future's continuation is reconstructed by adjusting `UNDERFLOW_CONT.ret` so that it points to the following instruction[7] (i.e. $\mathtt{ret}'_3$ in the example). At this point `UNDERFLOW_CONT` corresponds to the parent task's continuation (`k` in Figure 2.12). The thief can now use this continuation to create a heavyweight task representation of the parent. The `cont_link` and `cont_ret` fields are initialized directly from `UNDERFLOW_CONT`. An undetermined placeholder, `res_ph`, is also created to represent the result of the future. `Res_ph` is stored in the field `cont_val` so that it will get passed to the parent's continuation. To represent the parent task's legitimacy, another undetermined placeholder, `leg_ph`, is created and stored in the field `leg_flag`. The field `cont_denv` is initialized to the dynamic environment in effect when the task was pushed on the task stack (the next section explains how this is done).

[7]This may not be this simple because all return addresses must be parsable. Gambit always generates a secondary return point along with each future body return point (at a constant distance from it). The secondary return point contains a jump to the instruction that follows the popping of the parent task.

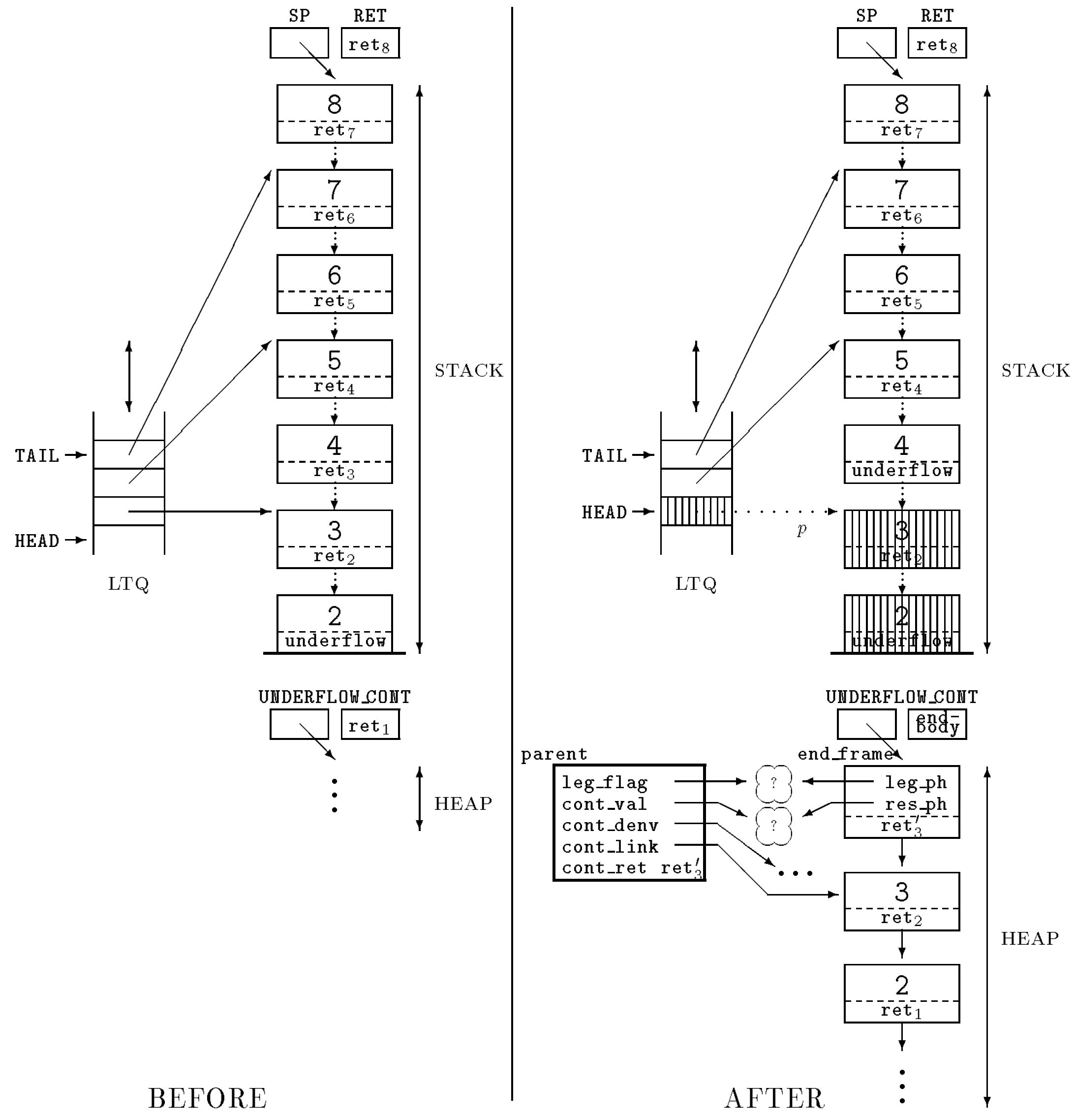


Figure 3.5: The LTQ and the steal operation.

```
task *steal_task( p )
value *p;
{
  instr *r = swap_child_ret_adr_with_underflow(p);   /* update child's ret adr    */
  heapify_frame( p, r );                             /* heapify parent's cont     */

{ task  *parent       = alloc_task();                /* allocate heavyweight task */
  frame *end_frame    = alloc_frame(3);              /* allocate end_frame        */

  parent->cont_link   = UNDERFLOW_CONT.link;         /* setup parent's cont       */
  parent->cont_ret    = future_secondary_ret_adr(r); /* (using secondary ret adr) */
  parent->cont_denv   = recover_dyn_env(p);          /* setup task's dynamic env  */
  parent->cont_val    = alloc_ph();                  /* allocate result ph        */
  parent->leg_flag    = alloc_ph();                  /* allocate legitimacy ph    */

  end_frame->link     = parent->cont_link;           /* setup end_frame           */
  end_frame->slots[0] = parent->cont_ret;
  end_frame->slots[1] = parent->cont_val;
  end_frame->slots[2] = parent->leg_flag;

  UNDERFLOW_CONT.link = end_frame;                   /* setup UNDERFLOW_CONT      */
  UNDERFLOW_CONT.ret  = end_body;

  return parent;
}}
```

Figure 3.6: The task stealing mechanism.

The thief will resume the parent task by a call to `resume_task`. Before doing this however, the victim's underflow continuation must be changed so that it will take the appropriate action when it returns from the child. Note that this new continuation will be invoked with the result of the future's body. Consequently, this continuation must logically correspond to procedure `end-body` of Figure 2.12. The first time it is called, `end-body` uses the result it is passed to determine the placeholder `res_ph` and the task is terminated after propagating the task's legitimacy (i.e. `CURRENT_LEGITIMACY`) to `leg_ph`. Subsequently, the result is simply passed on to the parent continuation.

This functionality is obtained by pushing a new continuation frame, `end_frame`, to the front of the continuation in `UNDERFLOW_CONT`. `End_frame` corresponds to the continuation frame created for the call to `thunk` in Figure 2.12. Thus, `UNDERFLOW_CONT.ret` is set to that call's return address (which is essentially a call to procedure `end-body`). `End_frame` contains the following values needed by `end-body`: the parent task's continuation and the placeholders `res_ph` and `leg_ph`. The "after" part of Figure 3.5 shows the system's state just before the thief resumes the parent task. Figure 3.6 gives the complete task stealing mechanism (except for removing p from the LTQ).

3.3.4 The Dynamic Environment Queue

For every task that is stolen, it is necessary to know what the dynamic environment was when the task was pushed on the task stack. When the recreated task is resumed by the thief, `CURRENT_DYNAMIC_ENV` will be set to that dynamic environment, thus restoring it to its previous state.

A straightforward solution is to store the value of the dynamic environment in the future's continuation frame. In other words, `CURRENT_DYNAMIC_ENV` is pushed on the stack on entry to the future body's thunk. Unfortunately, this adds an overhead to all futures independently of how heavily dynamic scoping is actually used, if at all.

It would be preferable if the cost of supporting dynamic scoping was only related to how heavily it is used. This can be achieved by a lazy mechanism that recreates a task's dynamic environment when it is stolen. It is assumed that the dynamic binding construct, `dyn-bind`, creates a new continuation for the evaluation of its body (as in Figure 2.7). The continuation frame contains `prev_env`, the dynamic environment that was in effect when `dyn-bind`'s evaluation was started. Since a change of the dynamic environment is always indicated by one of these frames, the following invariants will hold

- The dynamic environment E_f associated with a continuation frame f is equal to the `prev_env` field of the first dynamic binding continuation frame above f on the stack.
- If there is no dynamic binding continuation frame above f then E_f is equal to `CURRENT_DYNAMIC_ENV`.

The DEQ provides an efficient mechanism to find the first dynamic binding continuation frame above the stolen task's continuation frame. For each dynamic binding continuation frame on the stack there is exactly one entry in the DEQ; a pointer to the frame. The pointer is pushed onto the DEQ just before evaluating the body and is popped after the body as shown in Figure 3.7 (this code uses the association list representation of dynamic environments but the search tree representation could also be used).

A stolen task's dynamic environment is easily recovered with the DEQ. If the frame pointer removed from the LTQ is p, a linear or binary search can locate the lowest pointer on the DEQ that is larger than p. Figure 3.8 shows how this is done. Note that a linear search, as shown, is acceptable because its cost is of the same order as the cost

```
dyn_bind( id, val, body )
value id, val;
instr *body;
{
  *++SP = RET;                          /* create continuation frame   */
  *++SP = CURRENT_DYNAMIC_ENV;          /* setup prev_env              */
  *++DEQTAIL = SP;                      /* push frame pointer onto DEQ */
  CURRENT_DYNAMIC_ENV =                 /* install new dynamic env     */
    cons( cons( id, val ), CURRENT_DYNAMIC_ENV );
  RET = env_restore;                    /* execute body                */
  jump_to( body );
}

env_restore()
{
  if (DEQTAIL > DEQHEAD) DEQTAIL--;     /* pop frame pointer from DEQ  */
  CURRENT_DYNAMIC_ENV = *SP--;          /* restore dyn env to prev_env */
  RET = *SP--;                          /* return from dyn_bind        */
  jump_to( RET );
}
```

Figure 3.7: The implementation of `dyn-bind`.

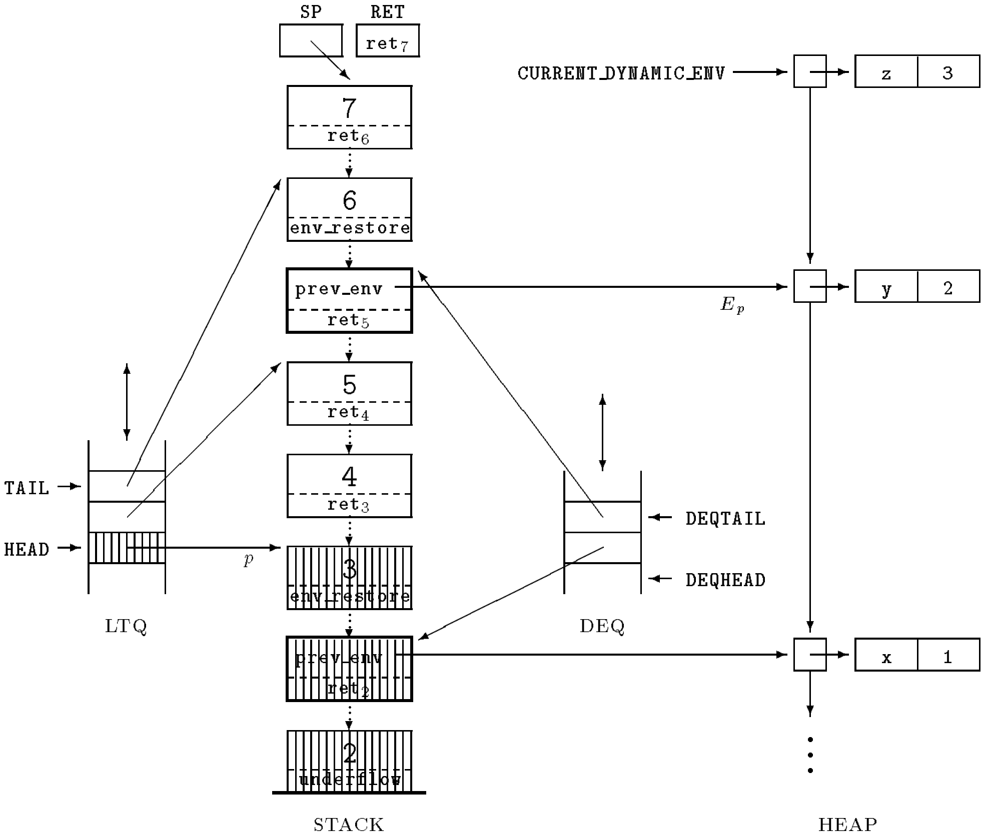
of heapifying the stolen task's continuation (i.e. there are no more entries skipped on the DEQ as there are frames heapified).

The cost of supporting dynamic scoping can be attributed entirely to the use of `dyn-bind` (i.e. the cost is $O(n)$ where n is the number of `dyn-bind`'s evaluated). For each `dyn-bind` evaluated, a few instructions in `dyn-bind` are needed to maintain the DEQ and a few more instructions are needed in `recover_dyn_env` to skip its entry on the DEQ if it is part of a stolen task's continuation (a DEQ entry is never skipped more than once).

3.3.5 The Problem of Overflow

Because the LTQ, DEQ, and run time stack are of finite size, an important concern is the detection and handling of overflows. A useful invariant of these structures is that the combined number of entries in the LTQ and DEQ is never more than the number of frames in the stack. Since each frame contains at least one slot for the return address, the space occupied by the LTQ and DEQ is never more than the space occupied by the stack. If these structures are allocated in two equal sized areas, one for the LTQ and DEQ growing towards each other and one for the stack, then the stack will always overflow before the LTQ and DEQ. Thus, it is only necessary to check for stack overflow. Chapter 4 explains how stack overflows can be detected efficiently.

```
(define (p2) ... (dyn-bind y 2 (p3)) ...)
(define (p3) ... (FUTURE (p4)) ...)
(define (p4) ... (p5) ...)
(define (p5) ... (FUTURE (dyn-bind z 3 (p5))) ...)
(define (p6) ... (FUTURE (p7)) ...)
(define (p7) ... (p8) ...)
(define (p8) ...)
```




```
value recover_dyn_env( p )
value *p;
{
  while ((DEQHEAD < DEQTAIL) && (DEQHEAD[1] < p)) DEQHEAD++;

  if (DEQHEAD == DEQTAIL)
    return CURRENT_DYNAMIC_ENV;
  else
    return DEQHEAD[1][0];  /* get frame's prev_env */
}
```

Figure 3.8: The DEQ and its use in recovering a stolen task's dynamic environment.

A stack overflow could simply cause the program to signal an error (or to terminate). This approach puts a strict limit on the depth of the call chain so it is inappropriate for a language like Lisp where recursion is used liberally. A more elegant approach that removes this restriction is to heapify the current continuation and then clear the stack, LTQ, and DEQ. Note that because the stack might contain lazy tasks this heapification is special (as discussed in the next section). Subsequent computation will reuse the stack and possibly cause some other stack overflows. The continuation thus migrates to the heap incrementally and it is only when there is no space left in the heap that an error is signalled.

3.3.6 The Heavyweight Task Queue

In general, the current continuation might contain lazy tasks when it is heapified. The four situations where this happens are

1. Task suspension (for touching an undetermined placeholder)
2. Task switch (caused by a preemption interrupt)
3. Stack overflow
4. `call/cc`

In these situations, something has to be done with the lazy tasks currently on the stack so that they remain runnable and independent. Since the lightweight representation is no longer adequate for these tasks, they are converted to the heavyweight representation and added to the processor's *heavyweight task queue* (HTQ). This queue contains all the heavyweight tasks runnable on that processor. It is in this queue that suspended tasks are put when the placeholder they are waiting on gets determined. Before heapifying the current continuation, the processor will in essence steal all lazy tasks on its own task stack (by calling `steal_task( *++HEAD )` while `HEAD<TAIL`) and add the resulting tasks to its HTQ.

But is this the best thing to do in the case of a task suspension? The only task that has to be suspended is the currently running task so it seems wasteful to remove all lazy tasks. The topmost lazy task could simply be recreated and resumed (i.e. popped from the task stack) after adding the current task on the placeholder's waiting queue. Mohr's system [Mohr, 1991] uses this approach (which he calls *tail-biting*) even though he concedes that

> "... it goes against our preference for oldest-first scheduling, since we have effectively

> created a task at the newest potential fork point. Performance can suffer because this task is more likely to have small granularity; also, further blocking may result, possibly leading to the dismantling of the entire lazy task queue."

Tail-biting offers no savings when supporting the Katz-Weise semantics because the parent continuation must be saved in the suspended task. Thus, the whole stack needs to be heapified anyway. In addition, by immediately moving all lazy tasks to the HTQ on a task suspension and by managing the HTQ as a FIFO structure, the same scheduling order as bottom stealing is obtained (oldest task first). There is also greater liberty as to which task to run next after the suspension. Gambit uses the following heuristic for choosing the next task: if x is the placeholder that caused the suspension, then the child task associated with x (i.e. x's owner task) is resumed if it is runnable; otherwise the processor goes idle[8]. Conversely, when a task terminates after determining placeholder y, one of the tasks waiting on y will be resumed if there is one; otherwise the parent task associated with y is resumed if it is runnable[9]. These heuristics promote an execution order close to the program's data dependencies so it tends to reduce the number of task suspensions.

Since there are two sources of runnable tasks per processor, the HTQ and the task stack, idle processors could obtain a runnable task from either source. Gambit however checks the HTQ first and then the task stack because this promotes the LIFO scheduling order, it avoids allocating new heavyweight tasks, and it is faster because the heavyweight tasks can be resumed immediately.

Another advantage of managing the HTQ as a FIFO structure is that scheduling will be fair because all runnable tasks, including the lazy tasks on the task stack, are guaranteed to start running in a finite amount of time. On every preemption interrupt, all lazy tasks and the current task are transferred to the HTQ and the first task on the HTQ is resumed. Consequently, if there are m tasks in the task stack and n tasks in the HTQ at the moment of the preemption interrupt, then these $m+n$ tasks will get at least one quantum out of the next $m+n$ quantums.

3.3.7 Supporting Weaker Continuation Semantics

The task stealing algorithm can be modified to accommodate any of the other continuation semantics described in Section 2.8. These weaker semantics offer a lower cost for task stealing because they avoid some steps.

[8] The link to the owner task is recorded in x when the parent task is stolen.

[9] A link to the parent task is recorded in `end_frame` when the parent task is stolen.

Firstly, since these semantics do not support legitimacy, they do not need to create the legitimacy placeholder (and of course the parent task and `end_frame` need not contain the `leg_flag` and `leg_ph` fields). Also, legitimacy propagation in `end-body` is not needed.

Secondly, the parent task's continuation is not needed in `end_frame`. In fact, `end_frame`, just like the root continuation frame, has no parent continuation[10]. For the original Multilisp semantics, `end_frame` will only contain the result placeholder `res_ph`. It is the only parameter passed to the procedure `end-body` apart from the body's result.

For the MultiScheme semantics, `end-body` only takes the body's result as a parameter. Consequently, `end_frame` contains no pertinent information and can simply be preallocated once and for all at program startup. Nevertheless, the result placeholder is needed by the child task so an extra field, `goal_ph`, must be added to heavyweight task objects. At the time of the steal, the parent task's goal placeholder is initialized from the child's goal placeholder and the result placeholder becomes the new goal placeholder of the child, i.e.

```
parent->goal_ph = CURRENT_TASK->goal_ph;
CURRENT_TASK->goal_ph = res_ph;
```

The steps avoided by the weaker continuation semantics do not amount to much; perhaps a saving of the order of 10 to 20 machine instructions per steal. A more promising source of saving is the handling of the parent continuation. Since only the parent task needs this continuation and it is immediately going to be restored by the thief, it seems useless to heapify the continuation. The steal operation could transfer the continuation frames from the victim's stack to the thief's stack in a single block (with a "block transfer" or similar operation). When heapifying the continuation, two copies of the frames are done: once to the heap (for heapification) and once to the stack (because of underflow). Moreover, these copies are more complex to perform than a block transfer of the stack because of the frame formatting and underflow handler overheads.

Upon closer examination, neither method is clearly superior to the other. Firstly, communication between the thief and victim processors is more important than the complexity of the algorithms. Assuming the thief actually returns through all the continuation frames, the frames only need to be transferred once between the processors in either method. When using heapification, one of the transfers will be between processors

[10] To preserve the format of frames and avoid a special case in the underflow handler, it is best if these frames contain a dummy parent continuation.

and one between local memory and the cache (assuming the stack lives mostly in the cache). Since interprocessor communication is an order of magnitude more expensive than local memory accesses, both methods will have roughly similar performance.

Secondly, the thief might not use all of the parent continuation frames. In such a case a block transfer will do more work than strictly required. When using heapification, only the frames which are needed are transferred (since frames are restored on demand). This can make a big difference in some programs, in particular when a given task spawns several children deep in some recursion. To explain this case, consider the following variant of `pmap`

```
(define (pmap proc lst)
  (if (pair? lst)
    (let ((val (FUTURE (proc (car lst)))))
      (let ((tail (pmap proc (cdr lst))))
        (cons (TOUCH val) tail)))
    '()))
```

Assume the root task calls `pmap` with a continuation containing k stack frames. Note that the continuation of the i^{th} evaluation of the future contains $k+i$ frames. Also note that the only task that ever gets stolen with LTC is the root task. If the list is of length n and there are n steals, a total of $\sum_{i=1}^{n} k+i = n(k+\frac{n+1}{2})$ frames are transferred between processors when using the block transfer method. The cost is lower by a factor of $O(n)$ when the parent continuation is heapified on every steal. On the first steal, $k+1$ frames are heapified and the topmost is transferred and restored by the thief. Subsequent steals will heapify two frames (one for the recursive call to `pmap` and one for the call to the future's thunk) and a single frame will be transferred and restored. Finally, in the unwinding of the recursive calls to `pmap`, n frames will be transferred and restored. The total is: $2n+k+1$ heapified frames, $2n$ restored frames, and $2n$ frames transferred between processors.

3.4 Synchronizing Access to the Task Stack

In the above description of LTC a critical issue was not addressed: the synchronization of the processors. This is an issue because multiple processors, including the victim, might try to simultaneously remove the same task from the task stack. Some synchronization is needed to resolve this race condition.

The case of multiple thieves can be prevented by associating a “steal” lock with every processor. A processor wanting to steal from a victim first acquires the victim's steal lock before attempting to steal a task. The lock is released when the attempt is finished so there in never more than one thief trying to steal from a given victim.

The only remaining race condition occurs when the victim's task stack contains a single task and the thief tries to steal the task while the victim is trying to pop the task. The term *protocol* refers to how the thief and victim processors interact to avoid conflicts when accessing the task stack. Two protocols are explored here: the *shared-memory* (SM) protocol and the *message-passing* (MP) protocol.

3.5 The Shared-Memory Protocol

The SM protocol tries to maximize concurrency between the thief and victim by minimizing the interference of the thief on the victim's current execution. The victim does not cooperate with the thief but rather the responsibility of stealing falls entirely on the thief (a cute analogy is that the thief is behaving like a pickpocket trying to stay unnoticed by its victim). Thus, it is the thief that executes the steps in Figure 3.6. The problems with this approach are explained throughout the description of the SM protocol that follows.

The first problem is that, at the moment of a steal, the thief has no way of knowing where the child's return address r is because the victim could be in any of several states (this problem shows up in `swap_child_ret_adr_with_underflow(`p`)`). The return address is only on the victim's stack if the child is in the process of executing a subproblem call. Even if the procedure calling convention required that r be passed on the stack in a predetermined slot (e.g. the first), there would be a problem because when r is invoked to return from the future's body, r will first get popped from the stack before the parent task is popped. This race condition between the thief mutating r and the victim invoking r can be handled in the following way. Instead of having the thief mutate r to bring the victim to call `underflow` when it returns from the child, the detection of a stolen parent task is done explicitly by the victim at the future's return point. The test at the return point will cause a branch to the underflow handler if the parent was stolen. Nevertheless, the thief must still know the value of r to reconstruct the parent's continuation. A simple solution is to save the value of r inside the future's continuation frame (just before pushing the lazy task on the LTQ). Thus, the thief can get the value of r by indirecting p.

Before stealing a task, the thief must first verify that one is present, that is check if `HEAD<TAIL`. However, this only tests the instantaneous presence of a task because

nothing prevents the victim from immediately decrementing TAIL as part of the popping of a lazy task. To prevent this from happening, each LTQ entry could be augmented with a "popping" lock that controls the popping of the corresponding task. The victim acquires the popping lock under TAIL before decrementing TAIL and the thief acquires the popping lock under HEAD+1 before testing for the presence of a task. If a task is present, i.e. HEAD<TAIL, the thief is certain that this condition will remain true until the popping lock is released because the victim cannot decrement TAIL from HEAD+1 to HEAD. Note that locking is not needed for pushing a lazy task since this can't cause a race with the thief (as long as TAIL is updated *after* the entry is written to the LTQ). To complete the stealing of the task, the thief increments HEAD, recreates the task by calling `steal_task( *HEAD )`, and releases the popping lock under HEAD. Unfortunately, the cost of lock operations on some machines is an order of magnitude more expensive than typical instructions. For example, the aquisition of a lock on the GP1000 is done through a system call that takes 6 μsecs (the equivalent of roughly 20 instructions). Accessing the locks would constitute the dominant cost of a future because it is needed on every task pop. The next section explains how hardware locks can be avoided.

A major problem with the SM protocol is that the task stack and related data structures must be accessible to all processors. This includes the following data structures

- the runtime stack and UNDERFLOW_CONT,
- the LTQ and its HEAD and TAIL pointers, and
- CURRENT_DYNAMIC_ENV, the DEQ and its DEQHEAD and DEQTAIL pointers.

The problem is that these data structures must be in shared memory and can't be cached optimally. The victim processor would have faster access to these data structures if they were private data. This is the prime motivation for the MP protocol described in Section 3.7. Two of these data structures can nevertheless be private even with the SM protocol: the TAIL and DEQTAIL pointers. Since this is achieved in a similar way for both pointers it will only be explained for TAIL. The idea is to maintain the following invariant: all LTQ entries above TAIL contain a special marker, for example a NULL pointer (all LTQ entries are initialized with this value). This means that, for all X>HEAD, X>TAIL if and only if X[0]=NULL. The thief can thus replace the test HEAD<TAIL by HEAD[1]$\neq$NULL. The victim can keep TAIL in the most convenient place (Gambit dedicates one of the processor registers). Pushing and popping an entry on the LTQ each require a single memory write to the LTQ (SP and NULL respectively) and an adjustment of TAIL. The code sequences for this method are given in the next section.

```
  .
  .
  .
  RET     = ret_point; /* setup future body's return address  */
  *++SP   = RET;       /* save ret adr in continuation frame  */
  *++TAIL = SP;        /* push parent task on LTQ             */
  future's body
ret_point:
  SM_attempt_pop();    /* pop parent task if still there      */
secondary_ret_point:
  SP--;                /* pop ret adr from continuation frame */
  .
  .
  .
```

Figure 3.9: Code sequence for a future under the SM protocol.

3.5.1 Avoiding Hardware Locks

Hardware locks can be avoided in the task popping operation by implementing the popping locks with any of several "software lock" algorithms based on shared variables (such as Dekker's algorithm [Dijkstra, 1968] and Peterson's algorithm [Peterson, 1981]). The same basic principles used by these algorithms can be adapted to design a special purpose synchronization mechanism for LTC as described next. With the exception of the previously mentioned method to make `TAIL` private, this algorithm is similar to the one described in [Mohr, 1991]. The only atomic operations in these algorithms are the memory references and lock operations (increments and decrements do not have to be atomic).

The mechanism arbitrates access to the task stack during task steal and task pop operations using only the pointers `HEAD` and `TAIL`, and a lock governing mutation of `HEAD` (i.e. `HEAD_LOCK`). Note that `HEAD_LOCK` can be either a hardware or software lock but because it is used infrequently in the popping operation it doesn't really matter which type it is. The task stealing and popping operations are implemented by the procedures `SM_attempt_steal` and `SM_attempt_pop` respectively (the code is given in Figures 3.10 and 3.11). These procedures attempt to remove a task from the task stack and indicate if the attempt was successful. `SM_attempt_steal` indicates failure by returning `NULL`; otherwise it returns a heavyweight task object corresponding to the stolen task. `SM_attempt_pop` indicates failure by calling the underflow handler directly; otherwise control returns to the caller. The code sequence generated for a future calls `SM_attempt_pop` at the future's return point, as shown in Figure 3.9. The performance of the popping operation can be improved by inlining the instructions of procedure `SM_attempt_pop` at the return point (or at least the two first instruction; which are the

```
  task *SM_attempt_steal( V )             /* V is victim processor        */
  processor *V;
  {
    value *p;                             /* entry obtained from V's LTQ   */

①   if (V->HEAD[1] == NULL) return NULL;  /* nothing to steal if LTQ empty */

    acquire_lock( V->HEAD_LOCK );         /* get right to increment HEAD   */
②   V->HEAD++;                            /* increment HEAD                */
③   p = *V->HEAD;                         /* get entry from LTQ            */
    if (p != NULL)                        /* check for conflict            */
    {
      task *parent = steal_task( V, p );  /* won race... recreate parent   */
      release_lock( V->HEAD_LOCK );       /* done with HEAD                */
      return parent;                      /* indicate success              */
    }
④   V->HEAD--;                            /* lost race... undo increment   */
    release_lock( V->HEAD_LOCK );         /* done with HEAD                */
    return NULL;                          /* indicate failure              */
  }
```

Figure 3.10: Thief side of the SM protocol.

```
  SM_attempt_pop()
  {
⑤   *TAIL-- = NULL;                    /* remove topmost LTQ entry        */
⑥   if (HEAD > TAIL)                   /* check for possible conflict     */
    {
      boolean thief_won;

⑦     acquire_lock( HEAD_LOCK );       /* prevent thief from mutating HEAD */
⑧     thief_won = (HEAD > TAIL);       /* definitive conflict check       */
      release_lock( HEAD_LOCK );

      if (thief_won)                   /* if thief won race...            */
      {
        *TAIL++ = SP;                  /* restore LTQ top                 */
        underflow();                   /* jump to end-body                */
      }
    }
  }
```

Figure 3.11: Victim side of the SM protocol.

most frequently executed instructions). In `SM_attempt_steal`, `steal_task` needs to know which task stack to access so it is called with the victim processor as an extra argument. Also note that the operation `swap_child_ret_adr_with_underflow(p)` used by `steal_task` is equivalent to `*p` (the child's return address is not mutated).

Clearly there is no possible conflict between the thief and victim when the task stack contains more than one task. The thief can increment `HEAD` and take the lowest entry on the LTQ at the same time that the victim voids the topmost entry (by writing `NULL`) and decrements `TAIL`. A conflict can only occur if calls to `SM_attempt_steal` and `SM_attempt_pop` overlap in time and the task stack contains a single task, that is `HEAD`=`TAIL`−1. The idea is to let the thief and victim blindly access the LTQ as though there was no conflict (thereby adjusting `HEAD` and `TAIL`) and only then check to see if there is a conflict (that is check if `HEAD`=`TAIL`+1 or equivalently `HEAD`>`TAIL`). When a conflict is detected, one of the two processors is selected as the "winner" of the race for the task and it returns success. The other processor undoes its mutation of the LTQ and returns failure. The thief detects success very simply: it is the winner if and only if the entry it reads from the LTQ at line ③ is not `NULL`. This entry can only become `NULL` if the victim voids it by executing line ⑤. The two possible orderings of these lines are considered next.

1. **Thief executes line ③ before the victim executes line ⑤**

   The thief has won the race. It will recreate the parent task and returns it from `SM_attempt_steal`. Note that from this point on, `HEAD` will never point lower than the entry that was removed (`HEAD` can only increase). When the victim eventually executes line ⑤ with `TAIL` pointing to the removed entry, it will decrement `TAIL` to below `HEAD` and consequently line ⑥ will detect the conflict. Line ⑧ will find the same result so the victim will conclude that the parent was stolen and will jump to `end-body`.

2. **Victim executes line ⑤ before thief executes line ③**

   The thief will lose the race because it will read `NULL` at line ③. Consequently, the thief will restore `HEAD` to its previous value (at line ④). There are two subcases depending on what the thief is doing when the victim executes line ⑥.

   (a) **Thief is <u>not</u> between lines ② and ④ when victim executes line ⑥**
   The thief has either not yet tried to remove the entry or has restored `HEAD` to the value it had just before line ②. Thus, `HEAD`=`TAIL` when line ⑥ is executed. The victim sees no conflict and declares success by returning from `SM_attempt_pop`.

(b) **Thief is between lines ② and ④ when victim executes line ⑥**

The thief has not yet restored HEAD to its original value so HEAD=TAIL+1. The victim thus detects a possible conflict at line ⑥. The reason for acquiring HEAD_LOCK at line ⑧ is to make sure that the thief is not between lines ② and ④ when the test at line ⑧ is executed. At that point the thief will have restored HEAD and will not mutate HEAD again (because HEAD_LOCK is locked). Line ⑧ thus sees HEAD=TAIL, causing SM_attempt_pop to return successfully.

The role of line ① is to ensure that the victim eventually acquires the lock at line ⑦ in systems where locks are not fair. It prevents new thieves from crossing line ①, so eventually the victim will be the only processor trying to lock HEAD_LOCK. It also avoids the overhead of attempting to steal from a processor with an empty task stack.

Thus, the SM protocol satisfies the following correctness criteria

- **Safety** — Either the thief or the victim, but not both, will remove a given entry from the LTQ.
- **Liveness** — An attempt to remove an entry will eventually indicate failure or success (i.e. deadlock and livelock are impossible).

3.5.2 Cost of a Future on GP1000

This section describes the details of the GP1000 implementation of the SM protocol and evaluates the costs related to the evaluation of a future on that machine. As explained above, the cost of a future depends on many parameters but mostly on whether the corresponding parent task is stolen or not.

Parent Task is not Stolen

If the parent is not stolen, the cost is simply that of pushing and popping a lazy task. Pushing a lazy task requires four steps: setting up the body's return address, setting up the arguments to the body (the closed variables), pushing the return address to the stack, and pushing the stack pointer to the LTQ. The first step typically replaces the same step that would be required in a sequential version of the program to evaluate the body (assuming it is a procedure call) so it won't be counted as overhead. Often the second step requires no instructions because the arguments are already in a location accessible to the body (e.g. in the registers). Only the last two steps are necessary extra work with respect to a sequential version of the program. Popping a lazy task takes two steps: popping and voiding the topmost entry on the LTQ, and checking for a conflict. The popping of the return address from the stack has no cost because it can be combined with the deallocation of the continuation frame by the future's continuation.

To get a concise code sequence on the GP1000, some of the special addressing modes of the M68020 processor were used, in particular predecrement and postincrement indirect addressing. `TAIL`, `SP`, and `RET` are all kept in address registers (`a4`, `sp`, and `a0` respectively). The two required steps in the lazy task push translate into two instructions and a lazy task pop translates into three instructions as shown below.

```
   movl  a0,sp@-     ; push return address to stack
   movl  sp,a4@+     ; push stack pointer to LTQ
   .
   .  code for future's body
   .

ret_point:
   clrl  a4@-        ; pop and void entry on LTQ
   cmpl  HEAD,a4     ; compare head and tail
   bcs   conflict    ; jump to handler if conflict
secondary_ret_point:
   .
   .  code for future's continuation
   .
```

Note that the stack grows downward on the M68020. Of the five instructions, three are writes to shared memory. The sequence accounts for a run time of roughly 2 μsecs. The assembly code generated for the SM protocol when compiling the `fib` benchmark is given in Section 3.8.

Parent Task is Stolen

To the above cost must be added the extra work performed as a consequence of the steal. Assuming that there is always a single return from the future's body, the thief and victim will perform the following operations

Thief

(1) **Heapify the parent continuation**

(2) **Find the parent's dynamic environment**

(3) **Allocate new objects** — This includes the allocation and initialization of the parent task, result and legitimacy placeholders and `end_frame`.

Operation	Instruction count	
`steal_task` (excluding `heapify_frame` and `recover_dyn_env`)	75	
`heapify_frame`	$6 + 34f + 2s$	
`recover_dyn_env`	$8 + 2b$	
`resume_task` (excluding `underflow`)	10	
`underflow`	$50 + 2s'$	
`determine!`	28	($w = 0$)
	$37 + 6w$	(otherwise)
`idle` (only accounts for search)	15	($n = 0$)
	$34 + 8n$	(otherwise)

Table 3.2: Cost of operations involved in task stealing.

(4) **Resume the parent task** — Note that only the first continuation frame needs to be restored.

Victim

(5) **Invoke** `end-body` — This is performed by the underflow handler.

(6) **Terminate the child** — The result and legitimacy placeholders get determined and then control goes to `idle`.

(7) **Find new work** — The victim must find a runnable task to resume. The task either comes from the victim's HTQ or is stolen from another processor.

In addition, there is a cost for restoring the other frames of the parent continuation heapified in (1). This is done at least in part by the thief but maybe also by some other processors (if the parent task migrates to other processors).

Table 3.2 gives the cost of the operations involved in task stealing (the costs correspond to the number of machine instructions executed in Gambit's encoding of the algorithms). In this table, f is the number of frames heapified (which is the number of frames separating the future from the enclosing future), s is the number of values on the stack, b is the number of dynamic variable bindings that were added to the dynamic environment since the enclosing future, s' is the size of the continuation frame to restore, w is the number of tasks on the placeholder's waiting queue, and n is the number of processors that were considered in the search for a runnable task ($n = 0$ when the task is found in the local HTQ). Note that these costs do not account for the location (i.e. local vs. remote memory) of the data being accessed.

From the table can be derived the approximate costs associated with the victim (T_{victim}), the thief (T_{thief}), and the processors that restore the parent's continuation ($T_{underflow}$).

$$\begin{array}{llllll} T_{victim} & = & (50 + 2 \times 3) + 28 + 28 + 15 & = & 127 \\ T_{thief} & = & 75 + (6 + 34f + 2s) + (8 + 2b) + 10 & = & 99 + 34f + 2s + 2b \\ T_{underflow} & & & = & \underline{50f + 2s} \\ & & & & 226 + 84f + 4s + 2b \end{array}$$

The minimal cost corresponds to $f = 1$, $s = 2$, $b = 0$, $w = 0$, and $n = 0$. This gives a total cost of 318 instructions (106 μsecs). In a more realistic situation, the frames will be larger and more numerous so the cost of heapification and underflow will increase. Assuming $s = 8$ and $f = 2$, the total cost will be 426 instructions (142 μsecs).

3.6 Impact of Memory Hierarchy on Performance

An unfortunate requirement of the SM protocol is that all processors must have access to the task stack's data structures; in particular the runtime stack and LTQ. Making these structures accessible to all processors has a cost because it precludes the use of the more efficient caching policies. The runtime stack and the LTQ are read and written by the victim but are only read by thief processors; thus, they are single writer shared data and can be cached by the victim using the write-through caching policy (as explained in Section 1.4.5). This however is not as efficient as the copy-back caching policy normally used in single processor implementations of Lisp. For typical Lisp programs, caching of the stack will likely be an important factor since the stack is one of the most intensely accessed data structures. Caching of the LTQ will also be an important factor for parallel programs with small task granularity because each evaluation of a future causes a few memory writes to the LTQ and stack (three in the SM protocol). Although this may not seem like much at first sight, the cost of a memory write to a write-through cached location on modern processors (such as the M88000 processors in the TC2000) is 5 to 20 times larger than the cost of a non-memory instruction or a cache hit (read or write) to a copy-back cached location. Note that this is not an issue on the GP1000 which lacks a data cache.

But how large is the performance loss due to a suboptimal caching policy? To better understand the importance of caching on performance, it is useful to analyze the memory access behavior of typical programs. The run time of a Lisp program can be broken down into the time spent accessing data in memory and the time spent on "pure computation". Memory accesses can further be broken down into two categories: accesses to the stack and accesses to the heap. Thus, a program is described by the three parameters S (stack), H (heap), and C (pure computation) which represent the proportion of total run time spent on each category of instructions ($S + H + C = 1$). For reference purposes, these parameters are defined with respect to an implementation where the stack and heap are not cached (i.e. all accesses go to local memory).

Some experiments were conducted to measure the value of S, H, and C for each benchmark program on both the GP1000 and TC2000. All these programs were run on a single processor as sequential programs (futures and touches were removed from the parallel benchmarks). The run time of each program was measured in three different settings. The first run was with the stack and heap located in non-cached local memory. The second run was with the stack located in remote memory (on another processor) so that each access to the stack would cost more. The final run was with the heap in remote memory. The three run times are respectively T, T_S, and T_H. Now since the relative cost R of a remote access with respect to a local access is known ($R = 12.1$ on the GP1000 and $R = 4.2$ on the TC2000), a system of three linear equations is obtained

$$\begin{aligned} S \quad + H \quad + C &= 1 \\ SR + H \quad + C &= T_S/T \\ S \quad + HR + C &= T_H/T \end{aligned}$$

This system can easily be solved to find the value of S, H, and C. Note that this model does not take into account factors such as the pipelining of instructions by the processor and the difference in costs between reads and writes. Also note that the values are dependent on the quality of the code generated by the compiler, but because an optimizing compiler was used, the measurements are representative of a high-performance system. As a sanity check, the values of S, H, and C obtained on the TC2000 were used to predict the run time of the program when the stack is cached with the copy-back policy. Assuming that the cache hit ratio for the stack is close to 1 (which is reasonable due to the high locality of stack accesses), the run time should be $T(\frac{S}{K} + H + C)$ where $K = 3.8$ is the relative cost of a local memory access with respect to a cache access. For most programs (21 out of 27), the prediction was within 5% of the actual run time. Only 3 programs had a difference above 10%: `fib` with 12%, `mm` with 13%, and `sum` with 15%. This suggests that the values obtained for S, H, and C are reasonably close to reality.

	GP1000				TC2000					
									Stack Caching	
Program	S	H	C	$O_{RemHeap}$	S	H	C	$O_{RemHeap}$	O_{None}	O_{WT}
`boyer`	.08	.08	.84	1.84	.32	.16	.53	1.64	1.29	1.15
`browse`	.15	.09	.75	2.05	.25	.15	.60	1.58	1.20	1.10
`cpstak`	.11	.25	.64	3.79	.24	.49	.27	2.95	1.24	1.10
`dderiv`	.14	.06	.80	1.70	.28	.13	.59	1.51	1.23	1.11
`deriv`	.14	.06	.80	1.64	.26	.12	.61	1.49	1.23	1.11
`destruct`	.01	.14	.85	2.52	.03	.41	.56	2.32	1.02	1.01
`div`	.08	.25	.67	3.82	.19	.47	.34	2.71	1.16	1.07
`puzzle`	.09	.21	.70	3.37	.17	.35	.48	2.23	1.13	1.04
`tak`	.53	.00	.47	1.00	.83	.00	.17	1.00	2.45	1.55
`takl`	.16	.27	.57	4.01	.32	.45	.23	2.82	1.28	1.13
`traverse`	.35	.14	.52	2.53	.56	.17	.27	1.90	1.63	1.33
`triangle`	.20	.13	.67	2.45	.38	.19	.43	1.81	1.29	1.16
`compiler`	.17	.11	.72	2.20	—	—	—	—	—	—
`conform`	.15	.10	.74	2.16	.25	.10	.65	1.40	1.20	1.09
`earley`	.25	.06	.69	1.67	.58	.10	.32	1.58	1.88	1.43
`peval`	.17	.14	.69	2.54	.35	.26	.38	2.13	1.34	1.14
`abisort`	.19	.32	.49	4.56	.31	.49	.20	3.03	1.31	1.13
`allpairs`	.56	.12	.33	2.30	.73	.14	.13	2.01	2.27	1.54
`fib`	.41	.00	.59	1.00	.70	.00	.30	1.00	1.81	1.34
`mm`	.43	.10	.47	2.08	.71	.14	.15	1.99	2.36	1.58
`mst`	.29	.13	.58	2.44	.59	.23	.19	2.27	1.76	1.30
`poly`	.09	.03	.88	1.32	.33	.12	.55	1.49	1.28	1.14
`qsort`	.25	.26	.49	3.94	.37	.26	.37	2.14	1.38	1.18
`queens`	.41	.00	.59	1.00	.74	.00	.26	1.00	2.15	1.48
`rantree`	.15	.00	.85	1.00	.40	.00	.60	1.00	1.36	1.17
`scan`	.49	.05	.46	1.54	.70	.09	.21	1.55	1.96	1.35
`sum`	.38	.05	.57	1.56	.73	.08	.19	1.44	1.85	1.34
`tridiag`	.35	.10	.55	2.13	.65	.18	.17	2.13	1.99	1.48

Table 3.3: Measurements of memory access behavior of benchmark programs.

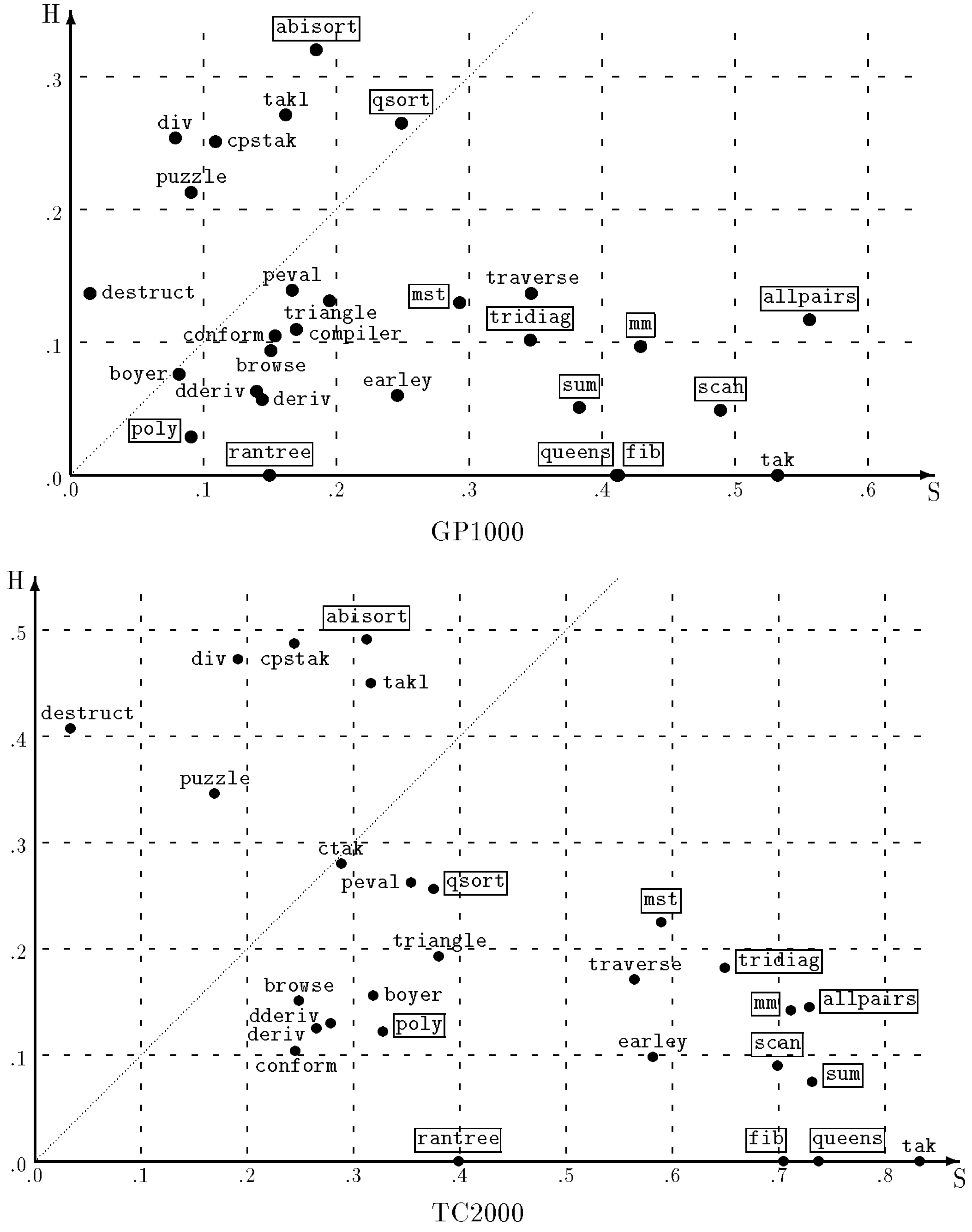


Figure 3.12: Relative importance of stack and heap accesses of benchmark programs.

These additional measurements were also taken

- $O_{RemHeap}$, the overhead of locating the heap in remote memory rather than local memory when the stack is cached optimally (i.e. no caching on GP1000 and copy-back caching on TC2000). This value is a good indicator of the overhead that will appear due to the sharing of user data if the program is run in parallel (assuming user data gets distributed uniformly to all processors, the number of processors is large, and there is little contention).
- O_{None} (TC2000 only), the overhead of not caching the stack rather than using copy-back caching.
- O_{WT} (TC2000 only), the overhead of caching the stack with write-through caching rather than with copy-back caching.

The measurements are given in Table 3.3 and Figure 3.12 presents this data in a more readable form (plots in S-H space).

A few observations can be made from Figure 3.12. Firstly, most of the programs access the stack more often than the heap (i.e. all the programs below the $S = H$ line). This tendency is even more pronounced for the parallel benchmarks (i.e. the boxed names in the plots). This is to be expected since the majority of the parallel benchmarks are based on recursive (DAC) algorithms.

Secondly, the importance of memory accesses is greater on the TC2000 than on the GP1000 (i.e. the position of a given program on the S-H plane is further from the origin). This is in agreement with the well known fact that modern processors need caches and a high hit rate to keep them going at peak speed. Most of the programs actually spend more time accessing memory than doing pure computation when run on the TC2000 (C is below $\frac{1}{2}$). As indicated by column O_{None} of Table 3.3, copy-back caching the stack provides an important performance gain. This gain is in some cases higher than a factor of 2. However, the median gain is 1.34 and the average is 1.56.

The last column in the table, O_{WT}, is of special interest because it reflects the cost of suboptimally caching the stack to support the SM protocol. The overhead of using write-through caching rather than copy-back caching is as high as 1.58. The sequential benchmarks have a median overhead of 1.11 (average of 1.17) whereas the median overhead for the parallel benchmarks is 1.34 (average of 1.34). Note also that the cache on the TC2000 is not very fast (only a factor of 3.8 faster than local memory). Some machines have caches which operate several times faster, with a corresponding increase in O_{WT}. The objective of the MP protocol is to avoid this overhead altogether.

3.7 The Message-Passing Protocol

If the role of the thief in the SM protocol is analogous to a pickpocket, in the MP protocol stealing a task is analogous to a holdup because the victim actively cooperates with the thief. To initiate a task steal, the thief sends a *steal request* message to the victim and starts waiting for a reply. The victim eventually interrupts its current execution and calls a *steal request handler* routine to process the message. This handler checks the task stack and, if a lazy task is available, recreates the oldest task and sends it back to the thief. Otherwise a failure message is sent back to the thief which must then try stealing from some other processor. The victim then resumes the interrupted computation.

There are several advantages to this protocol. Firstly, it relies less on an efficient shared memory. All the data structures comprising the task stack are private to each processor. The stack, LTQ, DEQ, and associated pointers can all be cached with copy-back caching. All programs which use the stack and/or dynamic scoping will thus benefit, whether they are sequential or parallel. Parallel programs will in addition benefit from the caching of the LTQ which reduces the cost of pushing and popping lazy tasks.

Secondly, it is possible to handle the race condition more efficiently than the SM protocol because all task removals from the task stack are performed by its owner. Preventing the race condition between task steals and task pops is as simple as inhibiting interrupts for the duration of the task pop. This can be achieved by adding a pair of instructions around the task popping sequence to disable and then reenable interrupts to the processor. The method used by Gambit is to detect interrupts via polling and never check for interrupts inside the popping sequence (efficient polling is explained in Chapter 4). There are other methods that have no direct overhead. For example, in the *instruction interpretation* method [Appel, 1989] the hardware interrupt handler checks to see if the interrupted instruction is in an "uninterruptible" section (i.e. a popping sequence). If it is, the rest of the section is interpreted by the interrupt handler before the interrupt is serviced. Other zero cost techniques are described in [Feeley, 1993].

Thirdly, the operation `swap_child_ret_adr_with_underflow(`p`)` can be implemented according to its original specification (i.e. an actual mutation of the child's return address), thus avoiding the push of the body's return address to the stack and the explicit check for underflow at the future's return point. The sequence generated for a future only has to push an entry to the LTQ before evaluating the body and to decrement `TAIL` at the future's return point. Doing this in the SM protocol was not possible because the thief could not know where the victim had stored the return address r. In the MP protocol r can be located in several ways.

- **Scanning the stack downward from the top** — The system can be designed so that the steal request handler is always called in the same way as a subproblem call. This is fairly easy to do when the system detects interrupts through polling because the call to the handler *is* a subproblem call. For a system that uses hardware interrupts it is more complex but still possible[11]. Thus, when the handler is executed, `SP` and `RET` can be used to parse the content of the stack. The handler can walk back through the frames until the frame directly above p is found. At this point the format of this frame is known, so r can be accessed directly. This approach may be expensive since there can be an arbitrary number of frames above p at the moment the steal request is received.

- **Scanning the stack upward from** p — Assuming the handler is always called as a subproblem, either r has been saved to the stack by the child's outermost subproblem call or it has been saved in the continuation frame for the call to the handler. Thus, when the handler is executed, r will necessarily be the first return address above p on the stack (i.e. the return address in the frame directly above p). An upward search of the stack starting from p and stopping at the first return address will locate r. It is assumed here that the values on the stack are tagged, at least to the extent of allowing return addresses to be distinguished from other values. It is also assumed that return addresses are not first-class objects and that return addresses are never saved to more than one location. Achieving this might require a close coupling of the steal request handler, interrupt system, and compiler. The cost of finding r with this method is $O(n)$ where n is the size of the frame above p. This method is used by Gambit. Gambit makes an effort to lessen the cost of the search by using heuristics that favor the saving of the return address in the lower end of continuation frames.

Finally, in the MP protocol it is the victim that is in charge of creating the parent task, its continuation, and related structures. By allocating these structures in the victim's local memory `steal_task` avoids remote memory accesses and thus completes faster than in the SM protocol. Remote memory accesses are performed by the thief when it resumes the task but strictly on demand. The parent task may actually start

[11] For example, a table could be setup with a description of the register allocation for every instruction in the program. This description indicates among other things where the parent return address is located when the instruction is executed. This table is used by the handler to build a correctly formatted continuation frame for the return to the interrupted code.

executing sooner than with the SM protocol because only the parent task object and its first continuation frame need to be transferred from victim to thief. The total number of remote memory accesses may also be smaller if the parent's continuation is not used fully by the thief (for example if the parent task migrates to another processor).

The disadvantages of the MP protocol are explained in Section 3.7.3.

3.7.1 Really Lazy Task Creation

The basic idea of LTC is to defer the creation of heavyweight tasks to the moment they are known to be required, that is when they are stolen. This usually saves a lot of work because non-stolen tasks are handled at very low cost and the cost of stealing a task is roughly the same as creating a heavyweight task in the first place. In the MP protocol, the cost of a non-stolen task is two instructions. This cost can actually be removed completely by doing more work when the task is stolen. Notice that the only purpose of the LTQ is to facilitate the reverse parsing of the stack (i.e. from bottom to top) to find the task continuation boundary of the lowest task. Finding the task continuation boundaries can however be done by parsing the stack from top to bottom and checking for return addresses to future return points. As explained previously, this parsing can be done by the steal request handler. The problem with this method is that the cost of stealing is not bounded since all the stack must be parsed. Fine grain programs with shallow recursions may nevertheless perform better with this method if most tasks are not stolen. Due to its worst-case behavior and the fact that it saves only two inexpensive instructions, this method is not very appealing for general use.

3.7.2 Communicating Steal Requests

The algorithms for the thief and victim sides of the MP protocol are shown in Figures 3.13 and 3.14 respectively. Even though they are based on a message-passing paradigm, these algorithms implement the communication using shared variables: `THIEF` and `REPLY`. In addition, the parent task is also communicated through shared memory. The victim's `THIEF` variable is set by the thief so that the victim can tell which processor has sent the steal request. It is also used to indicate the presence of a steal request (when there is a steal request `THIEF`$\neq$`NULL`). A thief's `REPLY` variable is set by the victim in response to a steal request. After the thief has sent a request, it busy-waits until the victim responds by setting the `REPLY` variable to the task that was stolen or

```
   task *MP_attempt_steal( V )      /* V is victim processor             */
   processor *V;
   {
     REPLY    = NONE_YET;           /* initialize with special marker     */
①    V->THIEF = CURRENT_PROCESSOR;  /* tell victim who the thief is       */
②    raise_interrupt( V );          /* get victim to process the request  */
③    while (REPLY == NONE_YET) ;    /* busy-wait until victim replies     */
     return REPLY;
   }
```

Figure 3.13: Thief side of the MP protocol.

```
   interrupt_handler()
   {
     if (THIEF != NULL)                        /* check for a steal request   */
     {
       /* the steal request handler: */

④      processor *T = THIEF;                   /* get pointer to thief        */
⑤      THIEF = NULL;                           /* set it up for next request  */
       if (HEAD < TAIL)                        /* anything on the task stack? */
         T->REPLY = steal_task( *++HEAD );     /* send oldest task to thief   */
       else
         T->REPLY = NULL;                      /* indicate failure to thief   */
     }
     .
     . /* check other sources of interrupts */
     .
   }
```

Figure 3.14: Victim side of the MP protocol.

to `NULL` if the victim had an empty task stack[12]. Note that the interrupt handler can get invoked for other reasons than the call to `raise_interrupt` at line ② (assuming all types of interrupts go through `interrupt_handler`). This means that the victim might detect the steal request at line ④ as soon as line ① is executed. Consequently, it is important for the thief to initialize `REPLY` before line ①. `THIEF` must also be reset (line ⑤) before the reply is sent back. In the reverse order a deadlock might occur if a second steal attempt executes line ① before `THIEF` is reset. The victim would be unaware of the second request and would never send a reply back to the thief (the thief would thus busy-wait forever).

The implementation of `raise_interrupt` will depend on the interrupt handling mechanism. If polling is used, then `raise_interrupt` can simply raise the victim's interrupt flag (the cost is that of a remote memory access). Sometime after this, the

[12]The advantage of having `REPLY` in the thief's local memory is that the busy-waiting does not create any traffic on the memory interconnect.

victim will detect the interrupt and call `interrupt_handler`. Note that this requires the interrupt flag to be multiple writer shared data so it can't be cached by the victim (or any other processor). Other systems send interrupts to other processors through dedicated hardware in the interconnect (the CM-5 for example). Sending an interrupt on these systems might require a system call. Clearly the cost will vary according to the features of the machine and operating system.

3.7.3 Potential Problems with the MP Protocol

The MP protocol has a number of characteristics that enhance performance but also some others that degrade it. This section examines the detrimental aspects and briefly discusses their severity. An important question is whether the performance gains are more important than the losses. This question will not be answered fully here because there are too many performance related parameters to consider. Chapter 5 will instead evaluate the performance of the MP and SM protocols experimentally.

Busy-Waiting

The most obvious problem with the MP protocol is that the busy-wait for the reply wastes processing resources. The total time wasted by the thief is the time it takes before the victim sends back the reply. This is the *steal latency*. The steal latency is the sum of the time needed by the victim to detect the steal request (T_{detect}) and the time to process the request ($T_{process}$). If the request is successful, $T_{process}$ is roughly the time required to call `steal_task` ($T_{\texttt{steal\_task}}$); otherwise $T_{process} = 0$.

The time wasted by the busy-wait must be put in context. If the steal is successful, the thief receives a task after wasting $T_{detect} + T_{\texttt{steal\_task}}$ of its time and taking $T_{\texttt{steal\_task}}$ time away from the victim, so the total amount of work expended to get the task is $T_{detect} + 2T_{\texttt{steal\_task}}$. If T_{work} is the time the thief spends running the stolen task before another task needs to be stolen, the overhead costs for stealing the task in the MP and SM protocols are

$$\begin{aligned} O_{MP} &= 1 + \frac{T_{detect} + 2T_{\texttt{steal\_task}}}{T_{work}} \\ O_{SM} &= 1 + \frac{T_{\texttt{steal\_task}}}{T_{work}} \end{aligned}$$

O_{SM} and O_{MP} are hard to compare because $T_{\texttt{steal\_task}}$ for the SM protocol is larger than for the MP protocol due to the additional remote memory accesses. If the penalty of a remote memory access is sufficiently low, O_{SM} will be lower than O_{MP}. However, the difference will be small when T_{work} is large relative to $T_{\texttt{steal\_task}}$ and T_{detect}. This is helped by the fact that LTC tends to increase the effective granularity of programs (i.e. the granularity of heavyweight tasks) and T_{work} is directly related to the effective granularity. However, an increase in the number of processors tends to decrease the effective granularity, thus increasing the importance of O_{MP} relative to O_{SM}.

Speed of Work Distribution

The speed at which work gets distributed to the processors is dependent on the steal latency. Distributing work quickly is crucial to fully exploit the machine's parallelism. It is especially important at the beginning of the program[13] because all processors are idle except one. Reducing the steal latency not only gets processors working sooner but also allows these processors to generate new tasks sooner for other processors. The MP protocol has a potentially smaller steal latency than the SM protocol, but only if T_{detect} is kept small. Unfortunately, minimizing T_{detect} may increase the cost of other parts of the system thus creating a trade-off situation. As explained in the next chapter, polling will become more expensive because interrupts need to be checked more frequently.

Interrupt Overhead

Finally, the cost of failed steal requests is a concern because the victim pays a high price for getting interrupted but this serves no useful purpose. The victim might get requests at such a high rate that it does nothing else but process steal requests. For example, a continuous stream of steal requests will be received by the victim if it is executing sequential code and all other processors are idle. The problem here is that processors are too "secretive". No information about the task stack is shared with other processors so the only way for a thief to know if the victim has some work is to send it a steal request.

A simple solution is to have each processor regularly save out `HEAD` and `TAIL` in a predetermined shared-memory location. Before attempting a steal, the thief checks the copy of `HEAD` and `TAIL` in shared memory to see if a task might be available. For thief processors this snapshot only reflects a previous state of the task stack but, if it is updated frequently enough, its correlation to the current state will be high. If the snapshot indicates a non-empty task stack it is thus likely that the steal attempt will be successful. Gambit always keeps `HEAD` in shared memory so it does not need to be saved

[13] Or more precisely a transition from sequential to parallel execution.

out (this does not affect performance because the victim accesses `HEAD` infrequently). `TAIL` is saved out on every interrupt check.

Unfortunately, this strategy reduces the speed of work distribution because thieves can only become aware of a task's presence at the next interrupt check. Performance is not affected if the task stack was not empty at the last interrupt check. However, if the task stack was empty, the newly created task can at best be stolen at the second following interrupt check. The first interrupt check will announce the task's presence to the thieves and the steal request will be handled at best at the second interrupt check. Since a processor's task stack is empty immediately after it has stolen a task, it is important to have a low interrupt check latency so that work can spread quickly to idle processors.

3.8 Code Generated for SM and MP Protocols

This section compares the code generated for a small program when using the SM and MP protocols on the GP1000. The program used here is the benchmark `fib`. Figure 3.15 shows the M68020 assembly code generated for `fib` for each protocol.

The following information will be useful to understand the code. Integer objects are 8 times their value because the three lower bits are used for the type tag. `Fib`'s entry point is label `L1`. When `fib` is called, the return address is passed in register `a0` and parameter `n` is passed in register `d1`. Register `d1` is also used to return `fib`'s result. The following registers have a dedicated role: `a4` contains `TAIL`, `a5` is a pointer to the interrupt flag and processor local data, `d6` is a mask to test for placeholder objects, `d5` is a private counter to perform interrupt checks intermittently (this counter is explained in the next chapter).

The boxed parts contain the instructions that relate to polling and the parallelization of `fib`. The rest of the code is identical in both protocols[14]. A sequential version of `fib` is obtained by removing the boxed parts from the code. One parallelization cost common to both protocols is the touch operation. Of its three instructions, only the first two are executed when a non-placeholder is touched (the run time for this case was measured at roughly .7 μsecs). The most important difference between the protocols is in the lazy task push and pop operations. These operations take two instructions in the MP protocol. The run time for these instructions was measured at roughly .7 μsecs

[14] Except for the instruction at `L7` which is different due to one of the compiler's stack allocation optimizations.

```
(define (fib n)
  (if (< n 2)
    n
    (let* ((f1 (FUTURE (fib (- n 1))))
           (f2 (fib (- n 2))))
      (+ (TOUCH f1) f2))))
```

Shared-Memory Protocol

```
L1:
   moveq #16,d0
   cmpl  d1,d0
   ble   L3
   bra   L8
L2:
   movl  a0,sp@-
   movl  sp,a4@+        lazy task push
   subql #8,d1
   moveq #16,d0
   cmpl  d1,d0
   bgt   L8
L3:
   movl  a0,sp@-
   movl  d1,sp@-
   lea   L6,a0
   dbra  d5,L2
   moveq #9,d5
   cmpl  a5@,sp         Interrupt check
   bcc   L2
   jsr   intr_handler

   bra   L2
L4:
   jsr   conflict       conflict handler
L5:
   bra   L7
L6:
   clrl  a4@-           lazy task pop
   cmpl  a5@(8),a4
   bcs   L4
L7:
   movl  d1,sp@
   movl  sp@(4),d1
   moveq #-16,d0
   addl  d0,d1
   lea   L9,a0
   moveq #16,d0
   cmpl  d1,d0
   ble   L3
L8:
   jmp   a0@
L9:
   movl  sp@+,d2
   btst  d2,d6
   bne   L10            TOUCH
   jsr   touch_undet
L10:
   addl  d2,d1
   dbra  d5,L11
   moveq #9,d5
   cmpl  a5@,sp         Interrupt check
   bcc   L11
   jsr   intr_handler

L11:
   addql #4,sp
   rts
```

Message-Passing Protocol

```
L1:
   moveq #16,d0
   cmpl  d1,d0
   ble   L3
   bra   L8
L2:

   movl  sp,a4@+        lazy task push
   subql #8,d1
   moveq #16,d0
   cmpl  d1,d0
   bgt   L8
L3:
   movl  a0,sp@-
   movl  d1,sp@-
   lea   L6,a0
   dbra  d5,L2
   movl  a4,a5@(4)
   moveq #9,d5
   cmpl  a5@,sp         Interrupt check
   bcc   L2
   jsr   intr_handler
   bra   L2


L5:
   bra   L7
L6:                     lazy task pop
   subql #4,a4


L7:
   movl  d1,sp@-
   movl  sp@(4),d1
   moveq #-16,d0
   addl  d0,d1
   lea   L9,a0
   moveq #16,d0
   cmpl  d1,d0
   ble   L3
L8:
   jmp   a0@
L9:
   movl  sp@+,d2
   btst  d2,d6
   bne   L10            TOUCH
   jsr   touch_undet
L10:
   addl  d2,d1
   dbra  d5,L11
   movl  a4,a5@(4)
   moveq #9,d5
   cmpl  a5@,sp         Interrupt check
   bcc   L11
   jsr   intr_handler
L11:
   addql #4,sp
   rts
```

Figure 3.15: Assembly code generated for `fib`.

(compared to 2 μsecs for the five instructions required in the SM protocol). Notice that in both protocols, label `L6` is the future's return point and `L5` is the secondary return point (which jumps past the popping sequence). The frame description information has been removed from the code for clarity. The other difference is in the interrupt check sequence. The code for the MP protocol has one more instruction to save out `TAIL`. However, this instruction is in the body of the interrupt check sequence which is executed once out of 10 times. The only accesses to shared memory in the MP protocol are in the body of the interrupt check sequence (a test of the interrupt flag and the saving of `TAIL`).

3.9 Summary

ETC is not an adequate implementation of futures because the overhead of creating a heavyweight task for each future is too high for fine grain programs. LTC postpones the creation of the heavyweight task until it is known to be required. This only happens when another processor needs work (or there is a task suspension, a preemption interrupt, a stack overflow, or a call to `call/cc`). To do this, LTC uses a lightweight task representation that contains enough information to recreate the corresponding heavyweight task. Lightweight tasks are put in a local task stack that is accessed by three operations: push, pop, and steal. A future translates to pushing the parent task onto the task stack, evaluating the future's body, and then popping the parent task to resume it (assuming it is still on the task stack). Since a task is essentially a continuation, a future is nothing more than a special procedure call. The task stack is the runtime stack and a table (LTQ) that indicates the extent of each continuation on the stack. In principle, the push and pop operations are only one instruction apiece. The Katz-Weise continuation semantics and dynamic scoping have no cost for non-stolen tasks because the associated support operations (i.e. copying the future's continuation and the dynamic environment) can also be postponed to the time of the steal.

Thief processors access the task stack from the bottom (the older task is stolen first). In divide-and-conquer algorithms this has the advantage of reducing the number of task steals required because the task containing the most work is transferred between processors.

A critical issue is which processor extracts the task from the task stack at the time of a steal. In the shared-memory (SM) protocol, the thief accesses the victim's stack and LTQ directly to steal the task. Careful synchronization between the thief and victim is needed to avoid a steal and pop of the same task. An unfortunate consequence of the SM protocol is that the stack and LTQ must be accessible to all processors, so they can't be cached optimally on a machine such as the TC2000. This suboptimal caching of the stack causes a sizeable overhead because the stack is one of the most frequently accessed data structures. In the message-passing (MP) protocol, the stack and LTQ are only accessed by the owner processor so they can be fully cached. The thief sends a work request message to the victim which sends back a task from its task stack if one is available. One of the important issues for the MP protocol is the interrupt latency. If it is too large then the thief will lose precious time busy-waiting and it will hinder the exploitation of the machine's parallelism because work distribution will be slow.

Chapter 4

Polling Efficiently

The message-passing implementation of LTC relies on a mechanism to communicate messages asynchronously from one processor to another. Such a mechanism must have the ability to interrupt a processor at any time. Conceivably, this could be done using some special feature of the hardware (e.g. interrupt lines of the processor) or the operating system (e.g. the Unix "signal" system). Unfortunately, these solutions are not very portable and a suitable performance cannot be guaranteed across a range of machines. Instead, it is better to consider software methods that are portable and provide a finer control of performance.

The idea behind software methods is rather simple. Each processor has a flag in shared memory that indicates whether or not that particular processor has a pending interrupt. The processor periodically checks (i.e. *polls*) this flag and traps to an interrupt handling procedure when it discovers that the flag has been raised. The interrupt check code necessary for polling the flag is added by the compiler to the normal stream of instructions required for the program. This unfortunately means that there is an overhead cost for any program, even if interrupts never occur. Minimizing this overhead is thus an important goal.

In theory, the compiler could arbitrarily reduce the polling overhead (O_{poll}) by decreasing the proportion of executed interrupt checks with respect to the normal instructions executed by the program. If all instructions take unit time then $O_{poll} = N_{poll}/N_{instr}$, where N_{poll} is the number of interrupt checks executed and N_{instr} is the number of non interrupt check instructions executed. This strategy lowers the frequency of interrupt checking and consequently increases the time between an interrupt request and the actual acknowledgement by the processor. Average latency ($\overline{L}$) and polling overhead are inversely related by $\overline{L} = \frac{N_{poll}+N_{instr}}{N_{poll}} = 1 + \frac{1}{O_{poll}}$. Note that interrupt latency here refers to the time interval between interrupt checks and not the time between an interrupt request and its acknowledgement. Here latency is expressed in number of instructions. To account for non-unit time instructions, latency can be expressed in units of time or number of machine cycles. This leads to the definitions $O_{poll} = T_{poll}/T_{instr}$ and $\overline{L} = \frac{T_{poll}+T_{instr}}{N_{poll}}$ where T_{poll} is the total time spent on interrupt checks and T_{instr} the time spent on other instructions. If an interrupt check takes k units of time on average then $\overline{L} = k(1 + \frac{1}{O_{poll}})$. To simplify the discussion, all instructions will be assumed to take unit time.

As explained in the previous chapter, increasing the interrupt latency is detrimental to parallel programs because it will take longer to respond to steal requests. This limits the rate at which work can get distributed to other processors. Thus, there is a trade-off between overhead and latency. High latency is preferable for sequential code because the polling overhead is low and low latency is best for parallel code because parallelism can be exploited better. The importance of latency is actually more subtle than this simple statement suggests. A high latency may be appropriate for applications where tasks often suspend on undetermined placeholders. Tasks that become ready following a `determine!` are made available to other processors by placing them on the HTQ. The HTQ is conveniently accessed through shared memory making it impervious to interrupt latency. If most of the tasks migrate in this fashion to the HTQ, a low latency may not significantly improve the rate of work distribution.

An "optimal" latency for all programs does not exist because the ratio of sequential to parallel code differs from program to program. The compiler could select a latency that suits the needs of the particular program, or procedure, being compiled. Even if the compiler had enough information to make such a decision, this strategy is still questionable. Latency requirements vary at runtime as the program switches back and forth between a sequential and parallel mode of execution. A procedure might be called both when latency requirements are low and high, and so a fixed polling frequency will give suboptimal performance. One could imagine having multiple versions of each procedure with varying polling frequencies, but this introduces new problems.

Instead of further exploring such ad hoc strategies, this chapter addresses the problem of efficiently achieving a particular latency with the use of polling. It will be assumed that code duplication is not permitted. The next chapter explores the effect of interrupt latency on the performance of the parallel benchmark programs. The results indicate that a particular choice of latency performs well for a wide range of programs.

4.1 The Problem of Procedure Calls

Although polling seems simple enough to implement, there is a complication. Normally, programs are not composed of a single stream of instructions. If this were the case the compiler could simply count the instructions it emits and insert an interrupt check after every so many instructions. Branches and procedure calls can alter the flow of control in unpredictable ways and so, it isn't clear how the compiler can achieve a constant number of instructions between interrupt checks. A reasonable compromise is to ask of the compiler to emit interrupt checks such that a given latency (L_{max}) is never exceeded.

4.1.1 Code Structure

To explore the problem further, it is convenient to introduce a formalism to describe the structure of a procedure's code. In general, the code of a procedure can be viewed as a graph of basic blocks of instructions. There are two special types of basic blocks: *entry points* and *return points*. There is a single entry point per procedure and one return point for each procedure call in subproblem position.

The only place where branches are allowed is as the last instruction of a basic block. There are four types of branches: local branches (possibly conditional) to other basic blocks of the same procedure, tail calls to procedures (i.e. reductions), non-tail calls to procedures (i.e. subproblems) and returns from procedures. Local branches and non-tail calls are not allowed to form cycles and thus they impose a DAG structure to the code. Loops can only be expressed with tail calls.

Note that subproblem and reduction calls always jump to entry points and that procedure returns always jump to return points. These restrictions are important because they simplify the analysis of a program's control flow.

Figure 4.1 gives the graph for the procedure `for-each` which contains all four types of branches. Returns and tail calls have been represented with dotted lines because they do not correspond to DAG edges. Solid lines are used for subproblem calls to highlight the fact that, just like direct branches, it is known where control continues after the procedure returns (if it returns at all). The generality of the DAG is only needed to express the sharing of code. For the moment, it is sufficient to make the simplifying assumption that the DAG has been converted into a tree by duplicating each shared branch. The handling of shared code is described in Section 4.4.

A necessary condition for any polling strategy is that an inline sequence of more

```
(define (for-each f l)
  (if (null? l)
    #f
    (begin
      (f (car l))
      (for-each f (cdr l)))))
```

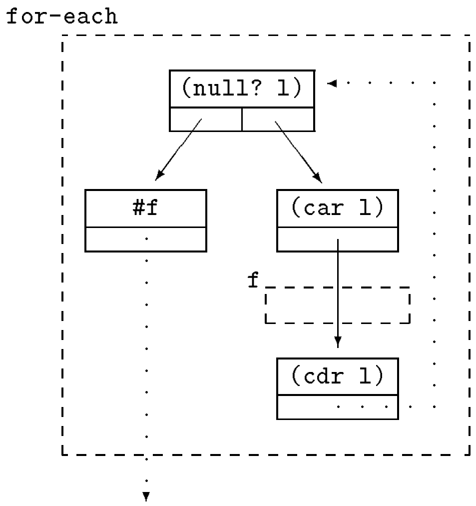


Figure 4.1: The `for-each` procedure and its corresponding code graph.

than L_{max} instructions is never generated without an intervening interrupt check. The compiler can exploit the code structure for this purpose. A *locally connected section* is any subset of the basic blocks that is connected by local branches only (for example, the three basic blocks at the top of Figure 4.1 or the bottom one). For any instruction i in a locally connected section, it is easy to determine what instructions are on the path to i from the section's root. These instructions are exactly those that are executed at runtime before i. Thus, for any instruction in a locally connected section, the compiler can tell how far back the last interrupt check occurred (assuming there is one on the same path from that section's root). The number of instructions that separate an instruction from the previous interrupt check is called the instruction's *delta*[1]. When the delta is L_{max}, an interrupt check is inserted by the compiler before the instruction.

4.1.2 Call-Return Polling

Polling strategies differ in how the transition between locally connected sections is handled. *Call-return polling* is a simple polling strategy that consists of putting an interrupt check as the very first instruction of each section's root. Since the root of a section is either the entry point of the procedure or the return point of a subproblem call, this corresponds to polling on procedure call and return.

[1] For instructions that are not preceded by an interrupt check in the same section, the definition of delta will vary according to the polling strategy.

```
(define (make-person name age gender) (vector name age gender))
(define (person-name   x) (vector-ref x 0))
(define (person-age    x) (vector-ref x 1))
(define (person-gender x) (vector-ref x 2))

(define (sum vect l h) ; sum vector from 'l' to 'h'
  (if (= l h)
    (vector-ref vect l)
    (let* ((mid (quotient (+ l h) 2))
           (lo  (sum vect l mid))
           (hi  (sum vect (+ mid 1) h)))
      (+ lo hi))))
```

Figure 4.2: Two instances of short lived procedures.

There are several variations on this theme. The interrupt check at the return point can be removed if checks are put on all return branches. Similarly, the interrupt check at the entry point can be replaced by checks on branches to procedures (both tail calls and non-tail calls). The four possible variations give equivalent dynamic behavior (i.e. same number of interrupt checks executed) but one may be preferable to the others if it yields more compact code. This depends on the particular code generation techniques used by the compiler and the programs being compiled. Compactness of code is not a big issue here so it won't be considered further.

4.2 Short Lived Procedures

Unfortunately, call-return polling can break down in certain circumstances. The worst case occurs when procedures are short lived, that is they return shortly after being called. At least two interrupt checks are performed per procedure call in subproblem position (once on entry and once on exit) and one if it is a reduction. This is a significant overhead if the procedure contains few instructions. This would not be a serious problem in languages that promote the use of large procedures, but in Lisp it is common to arrange programs into many short procedures.

Two instances of this style, typified in Figure 4.2, are the implementation of data abstractions and divide and conquer algorithms. This later situation is especially relevant because in Multilisp, parallelism is frequently expressed using divide and conquer algorithms. In binary divide and conquer algorithms, at least half of the recursive calls

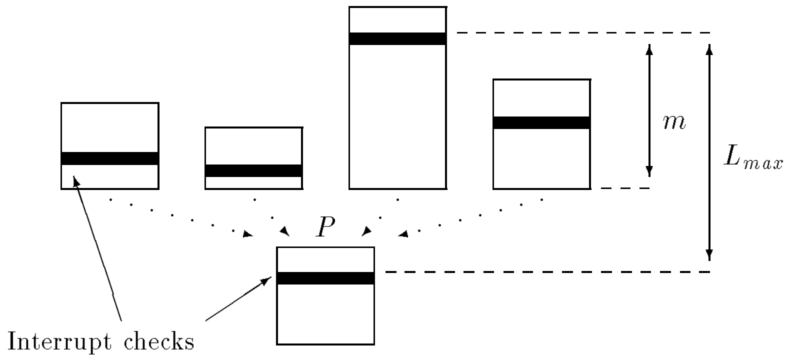


Figure 4.3: The maximal delta method.

correspond to the base case. If the algorithm is fine grained, such as the procedure `sum`, the overhead of polling will be noticeable because all the leaf calls are short lived.

Putting an interrupt check at every section's root is a very conservative method that doesn't take the structure of the program into account. If it is known that a procedure P is always called when delta is equal to $n-1$, then the compiler could infer that the first instruction in P has a delta of n. This would introduce a "grace period" of $L_{max} - n$ instructions at P's entry point during which interrupt checks are not needed. A similar statement holds for return points. Note that this yields a perfect placement of interrupt checks if it is carried out at all procedure entry and return points. Interrupt checks occur exactly every L_{max} instructions.

A more realistic solution is needed to handle the case where procedures and return points are called in different contexts (i.e. from call sites with different deltas). A simple extension to the previous method is to use m instead of n, where m is the maximum delta of all call sites to P (and similarly for return points). This *maximal delta* method is illustrated in Figure 4.3 where dark rectangles are used to represent interrupt check instructions. Note that delta now represents an upper bound on the number of non interrupt check instructions preceding an instruction. The maximal delta method is not an ideal solution for two reasons. First, it forces all control paths through P to have an early interrupt check (in P) if just one call site to P has a high delta. It would be much better if each procedure call "paid its own way", meaning that polling should be put on the call sites with high deltas. Not only would this improve P's grace period, it would put the interrupt check where it causes the least overhead (because a high delta at a call site is a sign of a high number of normal instructions preceding it)[2].

[2] For simplicity, it is assumed here that all paths to P are equiprobable.

A second shortcoming of this method is that the source and destination of procedure calls has to be known at compile time. In Scheme this information is not generally available, although one could reasonably argue that with the use of programmer annotations and/or control flow analysis the destination of most procedure calls could be inferred by the compiler for typical programs. However, the destination of returns is harder to determine because it would require a full dataflow analysis of the program and in general there are multiple return points for each procedure. The existence of higher order functions is another source of difficulty.

4.3 Balanced Polling

This section presents a general solution that does not rely on any knowledge of the control flow of the program. The method could be extended with appropriate rules, such as maximal delta, to better handle the cases where control flow information is available, but this is not considered here.

The idea is to define polling state invariants for procedure entry and exit. The polling strategy expects these invariants to be true at the entry and return points of all procedures and consequently must arrange for them to be true at procedure calls and returns.

Specifically, the invariant at procedure entry is that interrupts have been checked at most $L_{max} - E$ instructions ago. Here E is the grace period at entry points and is constant for all procedures. In other words, delta is defined to be $L_{max} - E$ at entry points. The invariant at procedure return is more complex. Either delta is less than E or, the path from the entry point to the return instruction is at most E instructions. These invariants are represented in Figure 4.4. Procedure P has two branches that illustrate the two cases for procedure return. Note that a procedure can be exited by a procedure return as well as a reduction call. For now, reduction calls will be ignored to simplify the discussion.

4.3.1 Subproblem Calls

These invariants have important implications. To begin with, short lived procedures are handled well because there is no need to check interrupts on any path that returns quickly without a call to another procedure (i.e. with less than E non-call instructions). This corresponds to the rightmost path in Figure 4.4.

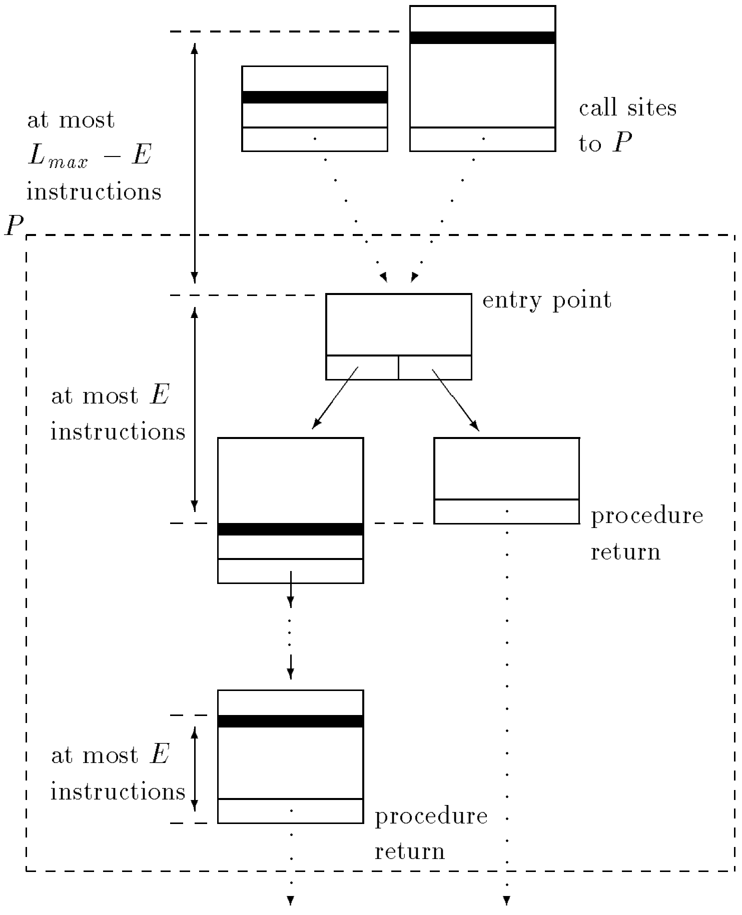


Figure 4.4: Procedure return invariants in balanced polling.

Moreover, the delta at return points can be defined as E plus the delta for the corresponding call point. This can be confirmed by considering the two possible cases. Assume procedure P_1 does a subproblem call to procedure P_2 which eventually returns back to P_1 via a procedure return in P_2, i.e.

$$P_1 \xrightarrow[\text{call}]{\text{subproblem}} P_2 \xrightarrow[\text{return}]{\text{procedure}} P_1$$

Either the last interrupt check was in P_2, so by definition delta at the return point (in P_1) is less than E. Alternatively, P_2 was short lived and didn't check interrupts, so there are at most E instructions that separate the call site (in P_1) from the return point (in P_1). As far as polling is concerned, a procedure called in subproblem position can be viewed as an interrupt check free sequence of E instructions. The compilation rule here is that if delta at a call point exceeds $L_{max} - E$ then an interrupt check is inserted at the call.

This rule means that up to $\lfloor L_{max}/E \rfloor$ subproblem procedure calls can be done in sequence without any interrupt checking. To see why, consider the scenario where the first call is immediately preceded by an interrupt check. At the return point, delta is equal to E. If the instructions for argument setup and branch are ignored, delta at the n^{th} return point is $n \times E$. Only when this reaches L_{max} is an interrupt check needed.

4.3.2 Reduction Calls

As described, the polling strategy does not handle reduction procedure calls (tail calls) very gracefully. The case to consider here is when a subproblem call is to a procedure which exits via a series of tail calls, finally ending in a procedure return, i.e.

$$P_1 \xrightarrow[\text{call}]{\text{subproblem}} P_2 \xrightarrow[\text{call}]{\text{reduction}} P_3 \cdots P_{n-1} \xrightarrow[\text{call}]{\text{reduction}} P_n \xrightarrow[\text{return}]{\text{procedure}} P_1$$

An interrupt check must always be put at a reduction call point to guard against the case where the called procedure returns quickly without checking interrupts (as in P_{n-1} calling P_n). Note that the return point in P_1 can have a delta as low as E. Note also that P_n might execute as many as E non interrupt check instructions before returning to the return point in P_1. Thus, it is not valid for P_{n-1} to jump to P_n with a delta greater than 0 because this would violate the polling invariant at the return point in P_1.

The treatment of reductions can be improved by introducing a new parameter (R) and consequently adjusting the polling invariants to support it. R is defined as the largest admissible delta at a reduction call. Thus, an interrupt check is put on any reduction call whose delta would otherwise be greater than R. Note that the same polling behavior as before is obtained by setting R to 0. The polling constraints for reduction calls can be relaxed by increasing the value of R. R can be as high as $L_{max} - E$ because a reduction call might be to a procedure that doesn't check interrupts for as many as E instructions.

A new invariant for return points has to be formulated to accomodate R. The delta at return points must now be at least $E+R$ to account for the case explained previously (a chain of reduction calls from P_2 to P_n ending in a procedure return to P_1). That is, on return to P_1 there could be up to E instructions in P_n plus as much as R instructions at the tail of P_{n-1} since the last interrupt check. When the compiler encounters a subproblem procedure call it sets the delta at the return points to E plus the largest value between R and the delta for the corresponding call point. If this value is greater than L_{max} an interrupt check is first put at the call site and the delta at the return point is set to $E+R$. The introduction of R also makes it possible to relax the invariant for procedure returns. Since the delta for return points is at least $E+R$, a delta as high as $E+R$ can be tolerated at procedure returns without requiring an interrupt check. With these new invariants, there can be up to $\lfloor (L_{max} - R)/E \rfloor$ subproblem procedure calls in sequence without interrupt checks. This polling strategy will be called *balanced polling*. A summary of the compilation rules for balanced polling is given in Figure 4.5.

The two constants E and R must be chosen carefully to achieve good performance. Small values for E and R increase the number of interrupt checks for short lived procedures and tail recursive procedures respectively. On the other hand, high values increase the number of interrupt checks in code with many subproblem procedure calls (e.g. recursive procedures). Choosing $E = R = \lfloor L_{max}/k \rfloor$ is a reasonable compromise and a value of $k = 6$ gives good performance in practice. This suggests that there are typically less than 6 subproblem procedure calls per procedure in the benchmark programs (see Section 4.6).

4.3.3 Minimal Polling

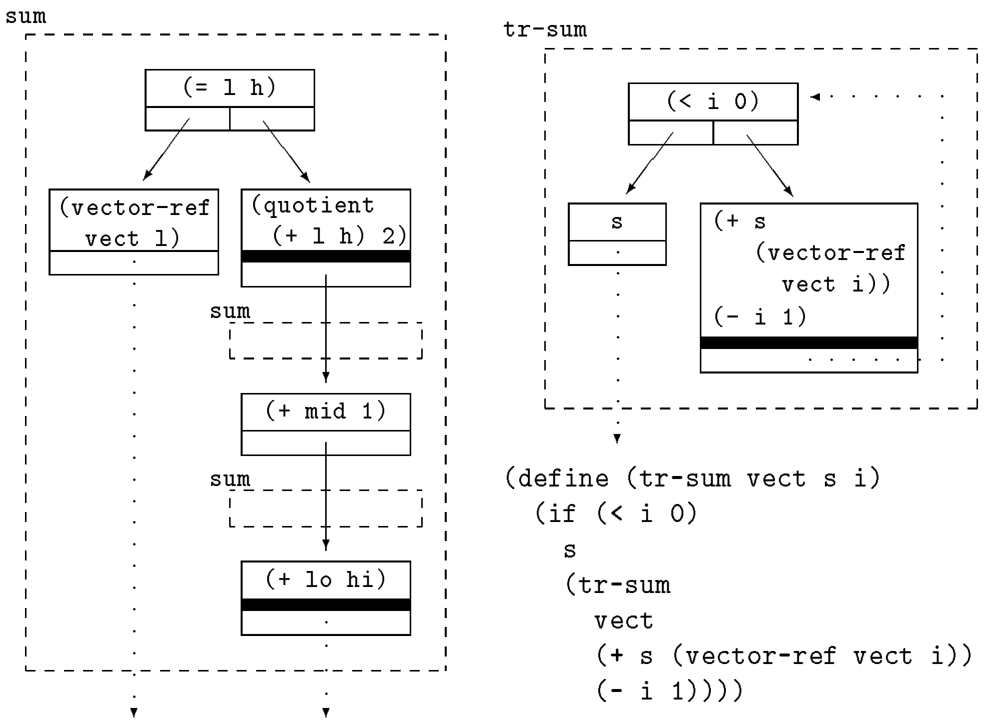
The choice of L_{max} is also an issue. A high L_{max} will give a low polling overhead. However, it is important to realize that there is a limit to how low the polling overhead can be made by increasing the value of L_{max}. This is due to the conservative nature of the strategy. Whatever the values of L_{max}, E and R are, at least one interrupt check

Location	Action by compiler
Entry point	$\Delta \leftarrow L_{max} - E$
Non-branch instruction	**if** $(\Delta \geq L_{max} - 1)$ **then** add interrupt check; $\Delta \leftarrow 0$ $\Delta \leftarrow \Delta + 1$ (for the next instruction)
Subproblem call	**if** $(\Delta \geq L_{max} - E)$ **then** add interrupt check; $\Delta \leftarrow 0$ $\Delta \leftarrow E + \max(R, \Delta)$ (for the return point)
Reduction call	**if** $(\Delta \geq R)$ **then** add interrupt check
Procedure return	**if** $(\Delta \geq E + R)$ **and** there is an interrupt check on the path from the procedure's entry point **then** add interrupt check

Figure 4.5: Compilation rules for balanced polling.

is generated between the entry point and the first procedure call. Delta is $L_{max} - E$ on entry to a procedure, so clearly the first call (reduction or subproblem) must be preceded by an interrupt check. Similarly, there is at least one interrupt check between any return point and the exit of the procedure (return or reduction call) because delta at any return point is at least $E + R$. These two types of paths are the only ones that are a necessary part of any unbounded length path. Thus, it is sufficient to have one interrupt check on each of these paths to guarantee that all possible control paths have a bounded number of instructions between interrupt checks. This *minimal polling* strategy is useful because its overhead is a lower bound that can be used to evaluate other techniques.

An example of minimal polling for the procedure `sum` and the tail recursive variant `tr-sum` is presented in Figure 4.6. For the call `(sum v` l h`)` there are exactly $2 \times (h - l)$ interrupt checks executed or nearly one interrupt check per procedure call (assuming $h - l + 1$ is a power of two). By comparison, checking interrupts at procedure entry and exit would execute twice as many interrupt checks (two per procedure call). However, for the tail recursive procedure `tr-sum` both methods are essentially equivalent with one interrupt check per iteration.




```
(define (tr-sum vect s i)
  (if (< i 0)
    s
    (tr-sum
      vect
      (+ s (vector-ref vect i))
      (- i 1))))
```

Figure 4.6: Minimal polling for the recursive procedure `sum` and a tail recursive variant.

It is interesting to note that balanced polling is more general than minimal polling and call-return polling. These can be emulated by judiciously choosing E, R and L_{max}. Minimal polling is obtained when $0 \ll E = R \ll L_{max}$ (i.e. E and R are arbitrarily large and L_{max} is arbitrarily larger). An interrupt check is put at the first call and another one is put at the return or reduction call that follows the last return point. Call-return polling occurs when $0 = E \ll R = L_{max}$. This places interrupt checks at all entry points and return points.

4.4 Handling Join Points

It has been assumed that the code of procedures is in the form of a tree. However, the compilation of conditionals (e.g. `and`, `or`, `if` and `cond`) in subproblem position introduces join points that give a DAG structure to the code. Certain optimization techniques, such as common code elimination, can also produce join points to express the sharing of identical code branches. A simple approach for join points is to use the maximal delta method. That is, the delta at the join point is the maximum delta of all branches to the join point. Although this is not an optimal strategy, its performance on the benchmark programs seems sufficiently good to be content with it.

4.5 Polling in Gambit

Polling is a general mechanism that can serve many purposes. In Gambit, polling is used for

- Stack overflow detection
- Inter-processor communication (for stealing work)
- Preemption interruption (for multitasking)
- Inter-task communication (for interrupting tasks)
- Barrier synchronization (e.g. for synchronizing all processors for a garbage collection and to copy objects to the private memory of every processor)

A special technique is used to check all these cases with a single test. The interrupt flag in shared memory is really a pointer that is normally set to point to the end of the area available for the stack. An interrupt check consists of comparing the flag to the current stack pointer, and to jumping to an out of line handler when the stack pointer exceeds this limit. A processor can be interrupted by setting the flag to a value that forces this situation (e.g. 0). The interrupt handler can then use some other flags to discriminate between the possible sources of interrupt.

Although it can be done with a single test, the interrupt check may still be relatively expensive due to the reference to shared memory. Increasing L_{max} is not a viable solution because the polling frequency can't be lowered beyond a certain point. To provide a finer level of control, interrupts can be checked intermittently. Polling instructions generated by the compiler represent "virtual" interrupt check points and an actual interrupt check occurs only every so many virtual checks. This new parameter is the *intermittency factor* and is called I. Intermittent checking is easily implemented by a private counter that is decremented at every virtual check. When it reaches zero it is reset to I and the interrupt check is performed. The average cost of an interrupt check will thus be the cost of updating and checking the counter plus $1/I^{\text{th}}$ the cost of checking the interrupt flag.

An interesting optimization occurs here. Balanced polling has a tendency to put the interrupt checks at branch points. An interrupt check itself involves a branch instruction so in many cases it is possible to combine the two branches into a single one. Moreover, several machines have a combined "decrement and branch" instruction that helps reduce the cost even further. All these ideas are implemented in Gambit.

4.6 Results

To have a better idea of the polling overhead that can be expected from these polling methods, it is important to measure the overhead on actual programs. Two situations are especially interesting to evaluate: the overhead on typical programs and on pathological programs that are meant to exhibit the best and worst performance.

Several programs and polling methods were tested. The programs were run on the GP1000 using a single processor. Each program was compiled in four different ways: with no interrupt checks, with minimal polling, with call-return polling and balanced polling. For balanced polling, L_{max} was set to values from 10 to 90 and E and R were set at $\lfloor L_{max}/6 \rfloor$. A value of $I = 10$ was used as the intermittency factor. The average run time on ten runs was taken for each situation. The polling overhead of minimal polling over the program compiled with no interrupt checks is reported in the first column of Table 4.1. The overhead for the other polling methods is expressed relatively to the overhead of minimal polling. Thus a relative overhead of 2 means that the overhead is twice that of minimal polling. Overheads lower than one can be explained by a combination of factors: timing inaccuracies and degradation of instruction cache performance (due to the different loading location of the programs). The table also gives the average latency obtained with minimal polling and balanced polling (at $L_{max} = 10$ and $L_{max} = 90$). The latency for `compiler` is not shown because the number of interrupt checks executed was not available (to measure it, the program must be compiled with a statistics gathering option which increases the size of the code so much that it can not fit anymore on the GP1000!).

The program `tight`, shown below, was designed to exhibit worst-case behavior.

```
(define (tight n)
  (if (> n 0)
    (tight (- n 1))))
```

It is a tight loop that doesn't do anything except update a loop counter. There are only two instructions executed on every iteration: an increment and a conditional branch. Interrupt checks will clearly add a high overhead to this. For most polling methods the overhead is about 80%. In the case of balanced polling with $L_{max} = 10$ the overhead is roughly twice that because two interrupt checks get added to every loop (because $E = R = 1$).

The program `unfolded` is the same loop as `tight` but unfolded 80 times. Thus, it is a long inline sequence of 80 decrements followed by one conditional branch instruction. The polling methods do well on this program (about 6% for minimal and call-return polling) because procedure calls are relatively infrequent and it is easy to handle the inline sequence of instructions. As expected for balanced polling, increasing L_{max} decreased the overhead, down to about 14%. L_{max} would have to be higher than 486 (i.e. 6×81) to reduce the overhead to that of minimal polling (at $L_{max} = 90$ there are two interrupt checks per loop).

The other programs are from the standard set of benchmarks. The parallel programs were compiled as sequential programs (i.e. with futures and touches removed) to factor out the overhead of supporting parallelism.

The results for these programs indicate that minimal polling outperforms call-return polling in nearly all cases. Sometimes by as much as a factor of four, but by a factor closer to 1.7 on average. The largest differences occur for fine grain recursive programs (e.g. `tak` and `fib`) and programs with a profusion of data abstraction procedures (e.g. `conform`). The performance of balanced polling is rather poor for small values of L_{max}, two to three times the overhead of minimal polling when $L_{max} = 10$. However,

	Minimal polling		Call-return polling	Balanced polling									$\overline{L}$ (μsecs)	
	O_{poll}	$\overline{L}$		Rel. ov. when $E = R = \lfloor L_{max}/6 \rfloor$ and L_{max} is									for L_{max} =	
Program	(%)	(μsecs)	Rel. ov.	10	20	30	40	50	60	70	80	90	10	90
`tight`	83.9	13	1.0	2.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	11	13
`unfolded`	6.1	154	0.9	10.8	6.5	4.2	3.5	3.7	3.9	2.3	2.3	2.3	22	83
`boyer`	21.5	58	1.4	1.7	1.1	1.0	1.0	1.1	1.0	0.9	0.9	0.9	36	57
`browse`	14.7	90	1.1	1.6	1.1	0.8	1.0	1.7	1.2	1.0	1.0	0.9	46	88
`cpstak`	10.9	108	1.2	1.9	1.5	1.2	1.0	1.1	1.0	1.0	1.0	1.1	59	110
`dderiv`	9.0	93	1.6	2.1	1.4	1.6	1.2	1.0	1.3	1.3	1.2	1.3	53	95
`deriv`	8.1	114	1.4	1.8	1.4	1.8	1.1	0.9	1.0	1.0	1.1	1.2	62	115
`destruct`	21.3	34	1.1	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	20	34
`div`	14.1	49	1.0	1.3	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	38	49
`puzzle`	14.5	58	0.9	2.1	1.7	1.2	1.0	1.0	1.0	0.9	0.9	0.9	28	57
`tak`	8.7	71	4.6	3.9	1.4	1.8	1.2	1.0	1.0	1.0	1.0	1.0	25	71
`takl`	29.3	38	0.9	1.5	1.0	1.1	1.0	0.9	0.9	0.9	0.9	0.9	21	36
`traverse`	16.9	36	1.5	2.5	1.3	0.9	0.9	0.9	0.9	0.9	0.9	0.9	27	35
`triangle`	3.9	63	3.7	6.0	6.0	3.2	3.8	2.4	2.1	2.3	1.0	2.0	38	65
`compiler`	14.4	—	1.8	2.3	1.3	1.1	1.0	1.0	1.0	1.1	1.0	1.0	—	—
`conform`	10.5	34	2.5	2.8	1.7	1.3	1.1	1.2	1.4	1.3	1.4	1.2	18	34
`earley`	6.4	120	1.5	2.3	1.6	1.5	1.0	1.1	2.1	0.8	1.1	1.2	59	122
`peval`	9.7	98	1.7	2.2	1.5	1.0	1.1	1.1	1.3	1.0	1.0	1.1	50	98
`abisort`	11.4	72	1.3	2.5	1.7	1.4	1.4	1.0	1.1	1.1	1.0	1.0	36	72
`allpairs`	4.4	149	1.0	3.9	2.6	2.0	2.0	2.0	0.5	1.8	1.0	1.0	56	149
`fib`	18.7	36	2.1	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	21	36
`mm`	4.7	115	1.1	3.0	2.7	3.0	1.6	2.2	2.2	0.8	0.9	0.9	61	114
`mst`	10.5	122	1.6	2.2	1.5	2.1	1.0	1.2	1.0	0.8	1.1	1.0	53	122
`poly`	23.4	101	0.6	1.2	0.7	0.7	0.3	0.4	0.8	0.6	0.4	0.6	39	93
`qsort`	12.3	64	1.3	1.9	1.3	1.0	1.3	1.0	1.0	1.0	1.0	1.0	44	63
`queens`	15.2	48	1.4	3.0	1.5	1.5	1.5	1.4	1.3	1.3	1.2	1.3	26	50
`rantree`	11.4	101	2.5	2.2	1.2	0.9	1.4	1.1	1.3	1.0	1.0	0.9	41	100
`scan`	6.6	84	2.4	3.5	2.0	0.8	0.8	1.2	1.0	1.0	1.0	1.0	39	84
`sum`	11.8	66	1.8	2.5	1.4	0.7	0.5	0.9	1.0	0.8	0.9	0.8	30	65
`tridiag`	1.6	364	2.7	7.9	4.5	4.3	4.2	3.4	3.7	3.9	3.0	3.6	103	259

Table 4.1: Overhead of polling methods on GP1000.

balanced polling gives performance close to minimal polling when L_{max} is high. With $L_{max} = 90$ the polling overhead ranges from 5% to 25%. The highest overheads are for fine grain recursive programs. The average overhead for balanced polling is about 12% for values of L_{max} higher than 50.

4.7 Summary

Interrupts can be detected by the processor's hardware interrupt system or by polling. Polling has the advantage of simplicity and portability. A common claim is that polling is not appropriate for a high-performance system because it has a high overhead. This chapter described the balanced polling method whose overhead is almost half that of the more straightforward call-return polling method. Balanced polling as implemented on the GP1000 has a 12% overhead on average. This overhead still seems rather high but this can be explained by the high quality of code generated by Gambit and the poor instruction set of the M68020 processors on the GP1000. Systems with a compiler that generates less tight code or with a processor that permits a lower cost code sequence for an interrupt check (for example, a fast "compare and trap on condition" instruction) would have a correspondingly lower overhead for polling.

Clearly, the processor's hardware interrupt system should be used to implement the MP protocol if the interrupt latency and overhead are low enough and the state of the processor at the time of interrupt can be recovered conveniently. If not, polling is at least a viable alternative.

Chapter 5

Experiments

Performance is the main design objective of the implementation strategies presented in this thesis. In most cases a purely theoretical performance analysis is not satisfying because it must abstract away many real issues to make the analysis manageable. The goal of this chapter is to evaluate performance using experiments. Concrete evidence for the following claims is given

1. Exposing parallelism with LTC is relatively inexpensive when the MP protocol is used. The worst-case overhead (when programs are very fine grain) is about 20%.
2. In the absence of a cache, the overhead of exposing parallelism with the SM protocol is about twice that of the MP protocol (i.e. the worst-case overhead is about 40%). When a cache is available, the overhead for the SM protocol can be higher than a factor of two.
3. LTC scales well to large shared-memory multiprocessors. The two protocols have very similar speedup characteristics when a cache is not present.
4. The MP protocol has speedup characteristics that are consistently better than the SM protocol on multiprocessors with caches. The difference in performance when using a large number of processors is as high as a factor of two on the TC2000.
5. The steal request latency can be relatively large without adversely affecting the MP protocol's performance.
6. Supporting the Katz-Weise semantics and legitimacy generally has a negligible impact on performance.

5.1 Experimental Setting

Several experiments were conducted to evaluate and compare the various implementation strategies. The experiments consisted of running each benchmark program in a particular context and measuring some of its characteristics. The context was dependent on the following parameters.

- **Machine and compiler** — The experiments were performed on the GP1000 and TC2000 multiprocessors. Each of the M68020 processors on the GP1000 delivers roughly 3 MIPS and each M88000 processor on the TC2000 delivers roughly 20 MIPS. Only the TC2000 has a data cache. Each machine has its own version of the compiler (but the front-ends are the same). The back-end for the GP1000 generates highly optimized native code, whereas the version for the TC2000 generates portable C code which must be subsequently compiled with a C compiler. The price to pay for this portability is a slowdown of a factor of 1.5 to 3 over native code depending on the program. The slowdown is a result of extra "pure computation" instructions. The number of memory accesses would however be the same in a native code implementation. This means that the importance of the TC2000's memory hierarchy is lower than it would be if the back-end generated native code. Consequently, the results obtained with the GP1000 are more representative of a high-performance compiler and the results obtained on the TC2000 are more representative of a modern multiprocessor with a low cost memory hierarchy.

  A severe handicap of these machines is the small size of physical memory. The local memory on each processor is only 4 Mbytes on the GP1000 and 8 Mbytes on the TC2000. Since this memory holds the operating system's code and data structures and the program's code, little space is left for the program's heap (only about 2 Mbytes on the GP1000). Allocating virtual memory is not a solution because it adversely affects the performance of garbage collection and also because it doesn't scale well (page faults are handled by a small set of processors dedicated for this purpose). To minimize these problems, the benchmarks were chosen so that the data they allocate fits in the heap without causing any garbage collection. In an effort to reduce the number of page faults, the benchmarks perform a few "dry runs" before the run actually measured. Nevertheless, some memory intensive programs, `allpairs` and `poly` in particular, consistently caused page faults due to their poor locality of reference.

- **Number of processors** — One of the goals of this thesis is to show that LTC scales well to large shared-memory multiprocessors. For this reason, the experiments were conducted on the largest machines that were accessible: a 94 processor GP1000 (at Michigan State University) and a 45 processor TC2000 (at Argonne National Laboratory). These are multi-user machines where processors are dynamically allocated into partitions at the time the program is launched by the user. The program is only aware of the processors in its partition but, because the memory interconnect is a butterfly network shared by all the partitions, the contention on the network depends on the other programs running on the machine. To minimize this effect, experiments were performed at "off-peak" hours and the average of several runs (usually 10) was taken. However, it was difficult to find times where large partitions could be allocated, so it was necessary to limit the number of experiments and number of runs for the larger partitions (this explains, at least in part, the greater variations in the results on large partitions). The largest partition used on the GP1000 was 90 processors; on the TC2000 it was 32 processors.

Another problem afflicts large partitions. Each processor on the GP1000 and TC2000 has a limited size TLB (translation lookaside buffer) for holding the mapping information that is used to translate virtual addresses to physical addresses. The TLB is managed like a cache and has roughly 60 entries. Each entry maps a page of the program's virtual address space. When a memory reference is to a page not currently mapped by the TLB, a translation fault occurs and the operating system must load the appropriate mapping information into the TLB from a table in memory. Translation faults must be avoided because they are handled in software and are relatively expensive. Programs with poor locality of reference and that have more than 60 or so pages in their working set will cause frequent translation faults. Unfortunately, several of the benchmarks have poor locality because they distribute user data evenly across the machine to reduce contention. The working set of these programs increases with the number of processors and thrashing occurs when the working set exceeds 60 pages (this typically starts happening somewhere between 32 and 64 processors but the exact point depends on the program). Moreover, poor locality is inherent in the search for a task to steal which possibly "flushes" several entries from the TLB that are part of the stolen task's working set. The importance of this factor will increase with the number of processors and the scarcity of tasks to steal.

- **Polling parameters** — Balanced polling with $E = R = 15$ and $L_{max} = 90$ was used for all experiments. The steal request latency was controled by changing the polling intermittency factor I. Unless otherwise indicated, I was set to 10 (the same value used in the previous chapter to evaluate the polling methods).

- **Stealing protocol** — Both the SM and MP protocols were tested.

- **Continuation semantics** — Two continuation semantics were used: the original Multilisp semantics and the Katz-Weise semantics. On the GP1000, the original semantics was used with the SM protocol and the Katz-Weise semantics was used with the MP protocol. The TC2000 used the original semantics for both protocols. For the original semantics the transfer of the stolen task's continuation was performed with a single block transfer operation. The Katz-Weise semantics was implemented with heapification.

- **Legitimacy** — Unless otherwise indicated, legitimacy was not supported.

5.2 Overhead of Exposing Parallelism

O_{expose} corresponds to the cost of exposing the parallelism to the system. Part of this cost comes from the futures and touches added to the sequential program to parallelize it. The other part of the cost is a consequence of the less efficient caching policy that is needed for the SM protocol. Recall that T_{seq} is the run time of a sequential version of the program (the parallel program with futures and touches removed) and T_{par} is the run time of the parallel program on one processor. T_{par}, T_{seq}, and O_{expose} are related by the equation

$$O_{expose} = \frac{T_{par}}{T_{seq}}$$

To evaluate O_{expose}, the run time was measured on a single processor partition with the program compiled with and without futures and touches (giving T_{par} and T_{seq} respectively). T_{par} and O_{expose} are given on the left side of Tables 5.1 through 5.4. The first two tables are for the SM and MP protocols on the GP1000 and the last two tables are for the SM and MP protocols on the TC2000. On the TC2000, the stack was write-through cached for measuring the SM protocol's T_{par} and the stack was copy-back cached for measuring T_{seq} and the MP protocol's T_{par}.

Notice that for nearly all programs, the SM protocol has an O_{expose} larger than the MP protocol. The only exceptions are the programs `mm` and `abisort` on the GP1000.

5.2.1 Overhead on GP1000

On the GP1000, O_{expose} is closely dependent on G, the task granularity, and n, the number of closed variables that must be copied for the future's body (Tables 2.1 and 3.1 give the value of G and n for each benchmark). O_{expose} is approximately equal to $1 + \frac{(2.7+1.6n)\mu\text{secs}}{G}$ when using the SM protocol and $1 + \frac{(1.4+1.6n)\mu\text{secs}}{G}$ when using the MP protocol. This is consistent with the costs measured in Chapter 3 for the lightweight task push and pop sequence, 2 μsecs for the SM protocol and .7 μsec for the MP protocol, and the .7 μsec cost for a touch (most programs have the same number of touches and futures). For the SM protocol, O_{expose} is at its lowest value (.3%) for `allpairs`, the program with the largest granularity. The highest overhead (37.5%) is for `fib`, the program with the smallest granularity. For the MP protocol, `allpairs` and `fib` also yield the lowest and highest overheads (.2% and 20.8%). This is about half the overhead of the SM protocol.

5.2.2 Overhead on TC2000

On the TC2000, O_{expose} for the MP protocol ranges from 2.3% to 20.9%, which is essentially the same range as for the GP1000. However, O_{expose} for the SM protocol is much larger, ranging from 27.1% to 127.8%. The highest overhead is for `fib`, which runs a factor of 2.278 slower than the sequential version of the program. For the MP protocol, the overhead for `fib` is only 15.7%. The large difference in overheads is mostly due to the SM protocol's use of write-through caching for the stack and LTQ. According to column O_{WT} of Table 3.3, write-through caching the stack accounts for an overhead of 1.34 on sequential `fib`. Thus, the additional overhead of the parallel version (to go from 1.34 to 2.278) is attributable mostly to the three stack and LTQ writes performed for each future. On the other hand, the overhead of coarse grain programs is closer to O_{WT}. For example, `allpairs` has an O_{WT} of 54% and an O_{expose} of 55%.

5.3 Speedup Characteristics

The right side of Tables 5.1 through 5.4 provides some information on the parallel behavior of the programs. The programs were run on increasingly large partitions (up to 90 processors on the GP1000 and 32 processors on the TC2000) to see how well they exploit parallelism. For the GP1000, three measurements were taken: the run time of the program, the number of heavyweight tasks created, and the number of task

Program	T_{par}	O_{expose}		Speedup, TC and TS when number of processors is						
				2	4	8	16	32	64	90
`fib`	1.1300	37.5%	S=	1.45	2.82	5.47	10.33	17.79	27.04	31.37
			TC=	.0000	.0002	.0005	.0008	.0019	.0039	.0042
			TS=	.0000	.0001	.0001	.0003	.0006	.0012	.0014
`queens`	1.3080	19.3%	S=	1.66	3.16	5.70	9.75	15.10	19.16	18.21
			TC=	.0003	.0015	.0042	.0083	.0152	.0305	.0404
			TS=	.0000	.0003	.0008	.0019	.0036	.0076	.0117
`rantree`	.4550	14.9%	S=	1.68	3.18	5.41	8.84	11.86	14.04	13.38
			TC=	.0012	.0025	.0085	.0190	.0346	.0593	.0722
			TS=	.0004	.0011	.0039	.0071	.0121	.0204	.0259
`mm`	1.5760	1.1%	S=	1.20	1.88	3.24	5.90	10.14	15.94	18.34
			TC=	.0005	.0091	.0238	.0491	.1048	.1830	.2408
			TS=	.0001	.0018	.0056	.0099	.0214	.0416	.0598
`scan`	1.2960	21.8%	S=	1.26	2.13	3.61	6.43	10.21	13.54	13.57
			TC=	.0001	.0009	.0022	.0045	.0083	.0155	.0201
			TS=	.0000	.0001	.0002	.0005	.0008	.0015	.0022
`sum`	.4820	22.6%	S=	1.22	2.09	3.72	6.55	10.23	11.77	12.20
			TC=	.0001	.0009	.0019	.0041	.0075	.0133	.0171
			TS=	.0000	.0001	.0002	.0004	.0008	.0013	.0021
`tridiag`	4.0320	1.7%	S=	1.20	1.78	2.93	5.18	8.60	12.58	16.51
			TC=	.0004	.0021	.0055	.0126	.0238	.0454	.0631
			TS=	.0001	.0001	.0005	.0014	.0026	.0055	.0073
`allpairs`	24.9530	.3%	S=	—	—	—	—	—	—	—
			TC=	—	—	—	—	—	—	—
			TS=	—	—	—	—	—	—	—
`abisort`	5.0710	6.9%	S=	.62	.76	1.20	2.16	3.60	5.63	6.95
			TC=	.0001	.0013	.0030	.0072	.0167	.0386	.0563
			TS=	.0000	.0001	.0003	.0008	.0019	.0046	.0064
`mst`	25.1160	7.0%	S=	—	—	—	—	—	—	—
			TC=	—	—	—	—	—	—	—
			TS=	—	—	—	—	—	—	—
`qsort`	.2630	25.8%	S=	—	—	—	—	—	—	—
			TC=	—	—	—	—	—	—	—
			TS=	—	—	—	—	—	—	—
`poly`	2.4340	6.3%	S=	—	—	—	—	—	—	—
			TC=	—	—	—	—	—	—	—
			TS=	—	—	—	—	—	—	—

Table 5.1: Performance of SM protocol on GP1000.

Program	T_{par}	O_{expose}		Speedup, TC and TS when number of processors is 2	4	8	16	32	64	90
`fib`	.9930	20.8%	S=	1.64	3.22	6.14	11.27	19.21	27.96	32.88
			TC=	.0000	.0002	.0005	.0010	.0020	.0041	.0051
			TS=	.0000	.0001	.0002	.0003	.0006	.0012	.0016
`queens`	1.2550	14.5%	S=	1.73	3.27	6.00	10.26	15.39	20.60	22.28
			TC=	.0003	.0016	.0039	.0081	.0171	.0308	.0396
			TS=	.0000	.0003	.0009	.0019	.0041	.0078	.0098
`rantree`	.4460	12.6%	S=	1.72	3.24	5.53	8.53	11.48	13.56	14.04
			TC=	.0012	.0031	.0087	.0195	.0371	.0635	.0747
			TS=	.0004	.0014	.0039	.0078	.0141	.0236	.0277
`mm`	1.5830	1.5%	S=	1.21	1.85	3.26	5.78	10.23	15.78	18.87
			TC=	.0006	.0094	.0270	.0465	.1017	.1744	.2219
			TS=	.0002	.0016	.0046	.0086	.0190	.0376	.0507
`scan`	1.1900	11.8%	S=	1.35	2.23	3.91	6.54	10.27	14.22	14.78
			TC=	.0001	.0008	.0021	.0043	.0081	.0140	.0189
			TS=	.0000	.0000	.0001	.0003	.0006	.0009	.0012
`sum`	.4460	13.5%	S=	1.34	2.23	3.92	6.46	10.13	12.68	13.28
			TC=	.0001	.0010	.0021	.0043	.0076	.0134	.0178
			TS=	.0000	.0000	.0001	.0003	.0006	.0010	.0012
`tridiag`	3.9880	.6%	S=	1.21	1.79	2.96	5.04	8.49	12.13	15.80
			TC=	.0004	.0023	.0046	.0108	.0221	.0366	.0453
			TS=	.0001	.0001	.0004	.0009	.0018	.0035	.0034
`allpairs`	24.9400	.2%	S=	1.10	1.58	2.62	4.20	6.27	7.88	7.16
			TC=	.0104	.1254	.2553	.4478	.6910	.8898	.9853
			TS=	.0082	.0194	.0642	.1020	.1321	.1532	.2094
`abisort`	5.2800	11.3%	S=	.61	.75	1.23	2.09	3.58	5.57	6.85
			TC=	.0001	.0014	.0030	.0071	.0162	.0347	.0487
			TS=	.0000	.0001	.0002	.0006	.0013	.0024	.0029
`mst`	24.7990	5.7%	S=	.93	1.06	1.38	1.53	1.59	1.37	1.25
			TC=	.0116	.0454	.0889	.1523	.2434	.3294	.3578
			TS=	.0037	.0029	.0055	.0083	.0113	.0147	.0150
`qsort`	.2480	18.7%	S=	1.33	1.63	1.60	1.42	1.22	1.02	1.13
			TC=	.0017	.0101	.0399	.1073	.2394	.4334	.5290
			TS=	.0009	.0061	.0232	.0611	.1313	.2245	.1611
`poly`	2.3580	3.0%	S=	.98	1.08	1.27	1.44	1.41	1.22	.76
			TC=	.1510	.3083	.5142	.7066	.8179	.8458	.6338
			TS=	.0120	.0771	.1520	.1626	.1346	.1763	.0741

Table 5.2: Performance of MP protocol on GP1000.

Program	T_{par}	O_{expose}	Speedup when number of processors is 2	4	8	16	32
fib	.6763	127.8%	.88	1.74	3.23	6.03	11.18
queens	.6338	117.3%	.91	1.77	3.24	5.95	9.49
rantree	.1827	39.0%	1.41	2.70	4.60	7.44	10.79
mm	.6576	60.2%	.93	1.64	3.01	5.64	10.07
scan	.7156	81.3%	.94	1.72	3.01	5.51	8.97
sum	.2471	94.8%	.90	1.67	2.86	4.85	7.82
tridiag	1.6559	56.0%	.92	1.58	2.82	5.02	8.49
allpairs	12.2866	55.0%	.95	1.63	2.85	4.85	7.61
abisort	2.9351	27.1%	.84	1.36	2.41	4.36	7.79
mst	9.5555	43.2%	.85	1.21	1.55	1.81	1.85
qsort	.1740	52.8%	1.17	1.67	1.75	1.80	1.71
poly	.7275	30.2%	.61	.79	1.01	1.19	1.20

Table 5.3: Performance of SM protocol on TC2000.

Program	T_{par}	O_{expose}	Speedup when number of processors is 2	4	8	16	32
fib	.3435	15.7%	1.72	3.37	6.52	11.97	20.46
queens	.3525	20.9%	1.63	3.07	5.58	9.11	13.51
rantree	.1391	5.9%	1.82	3.46	5.76	9.07	11.99
mm	.4198	2.3%	1.29	2.18	3.88	7.09	11.40
scan	.4558	15.5%	1.35	2.40	4.29	7.57	12.20
sum	.1430	12.7%	1.40	2.49	4.43	7.56	11.88
tridiag	1.0907	2.7%	1.21	1.98	3.35	5.83	9.44
allpairs	8.1841	3.2%	1.22	2.03	3.43	5.56	7.80
abisort	2.4107	4.4%	.95	1.48	2.49	4.51	7.93
mst	6.9101	3.6%	1.04	1.34	1.76	1.93	1.92
qsort	.1294	13.6%	1.48	2.07	2.05	1.95	1.62
poly	.5759	3.1%	.77	.92	1.19	1.43	1.29

Table 5.4: Performance of MP protocol on TC2000.

suspensions that occurred. Each entry in the table contains three values computed from these measurements:

- **S** — This is the program's speedup over the **sequential** version of the program (i.e. that has futures and touches removed and that is run with copy-back caching of the stack on the TC2000).

$$\text{S}= \frac{T_{seq}}{run\_time}$$

- **TC** — This is the proportion of lightweight tasks that were transformed into heavyweight tasks.

$$\text{TC}= \frac{heavyweight\_tasks\_created}{N_{future}}$$

- **TS** — This is the number of task suspensions expressed relatively to the number of lightweight tasks.

$$\text{TS}= \frac{number\_of\_task\_suspensions}{N_{future}}$$

Note that a few of the benchmarks (`allpairs`, `mst`, `poly`, and `qsort`) did not run properly with the SM protocol on the GP1000[1]. The tables for the TC2000 only contain the speedup. The speedup data is reproduced as speedup curves in Figures 5.1 through 5.6. The speedup curves for the GP1000 also contain data for runs of the MP protocol with higher and lower intermittency factors. For now, only the curves marked $I = 10$ are considered. TC and TS for the MP protocol on GP1000 are also plotted as a function of the number of processors in Figures 5.7 and 5.8. The benchmark programs can be roughly classified in three groups, according to the shape of their speedup curves.

1. **Parallel and compute bound:** `fib`, `queens`, `rantree`. These programs do not access memory. The speedup curve is initially close to linear speedup, and gradually diverges from it as the number of processors increases (in other words the first derivative of the curve starts at 1 and the second derivative is negative). The flattening out of the curve as the number of processors increases is explained by Amdahl's law (i.e. each program has a maximal speedup).

[1] The bug has stumped me to this day. I suspect that it is a race condition I introduced in the assembly language encoding of the algorithms (Gambit's kernel contains about 5000 lines of hand optimized assembly code). After obtaining a working version of the SM protocol on the TC2000 (written in C), I convinced myself that the problem was not algorithmic. The problem may also be related to a known bug in the parallel garbage collection algorithm.

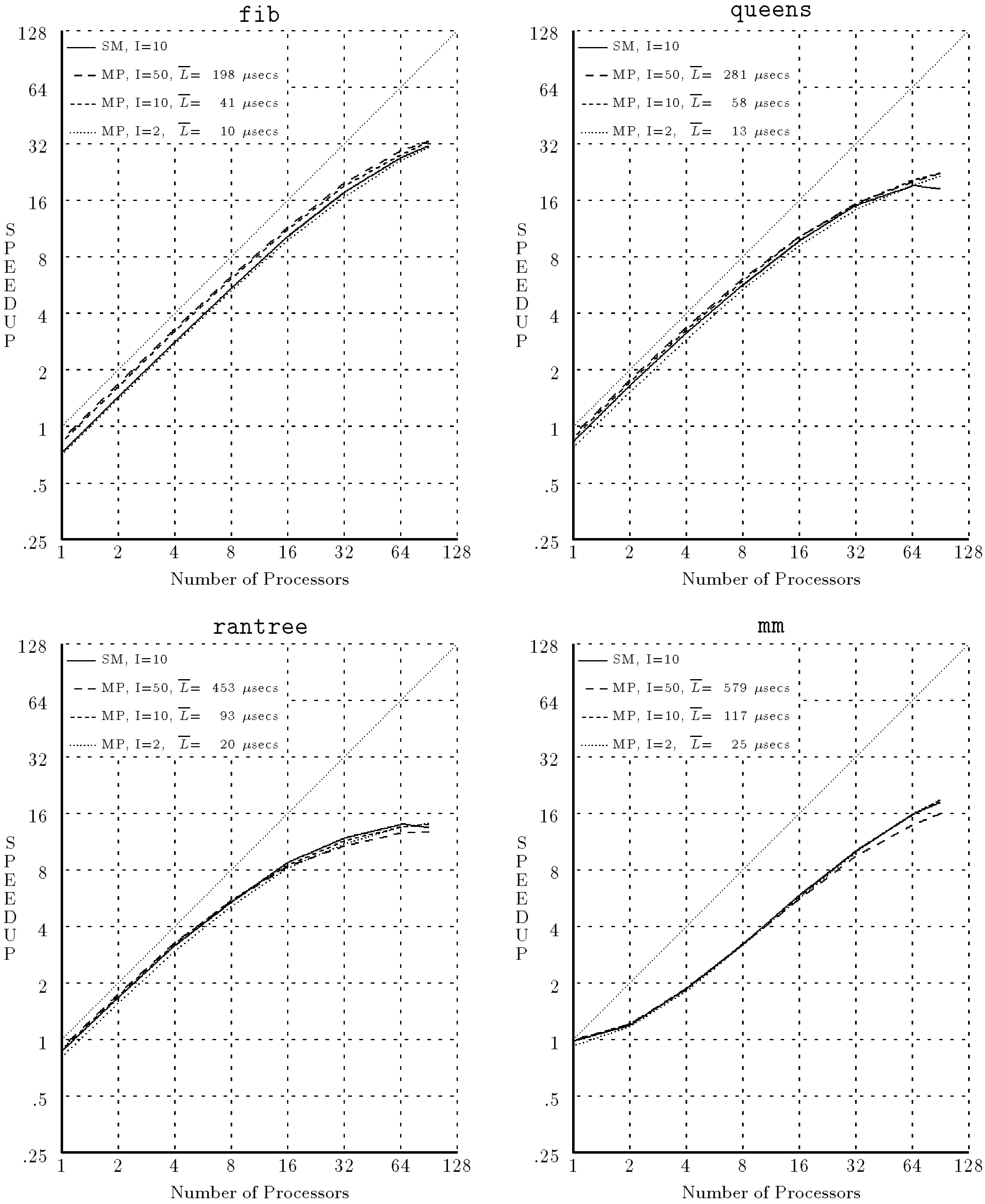


Figure 5.1: Speedup curves for `fib`, `queens`, `rantree` and `mm` on GP1000.

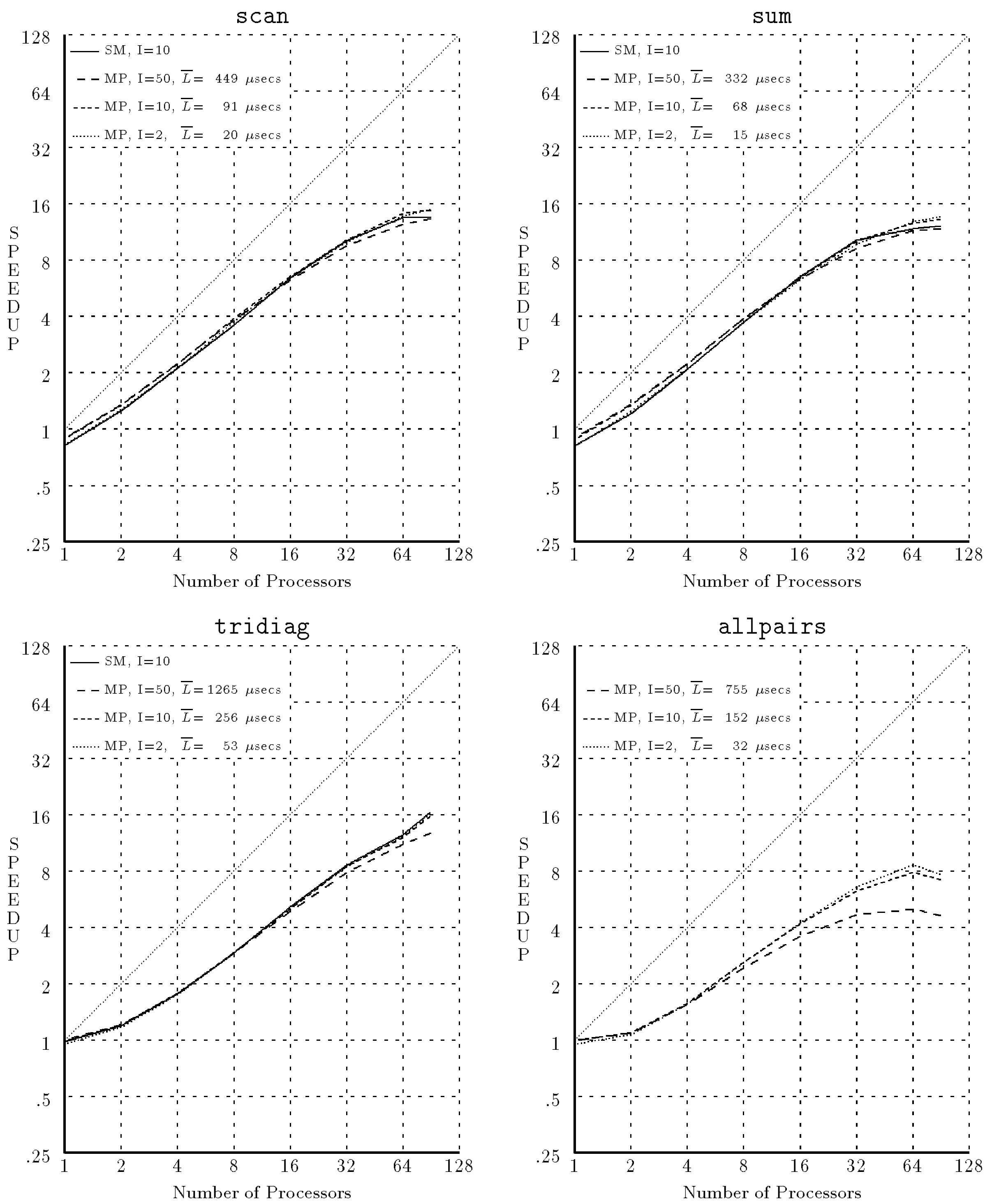


Figure 5.2: Speedup curves for `scan`, `sum`, `tridiag` and `allpairs` on GP1000.

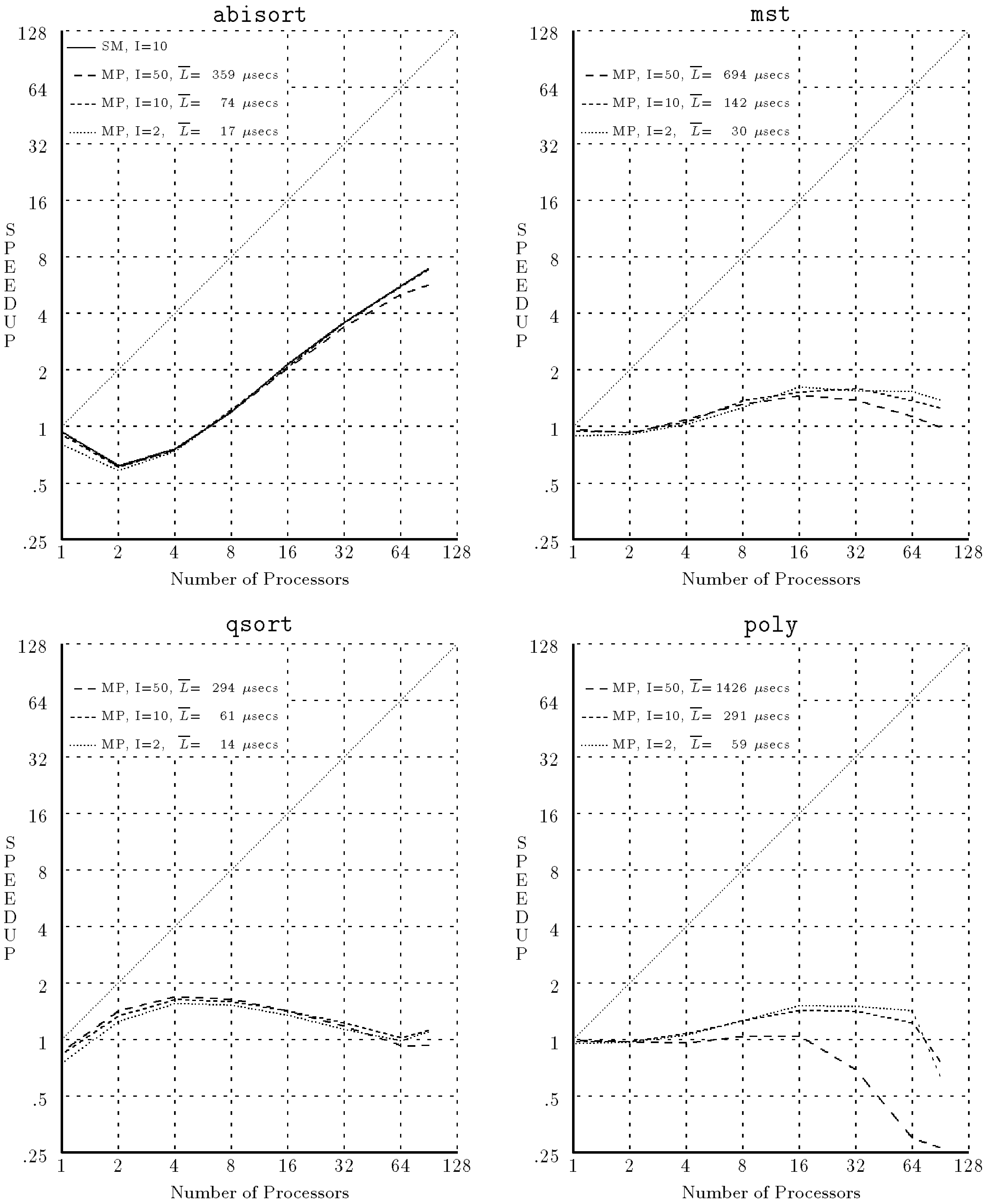


Figure 5.3: Speedup curves for `abisort`, `mst`, `qsort` and `poly` on GP1000.

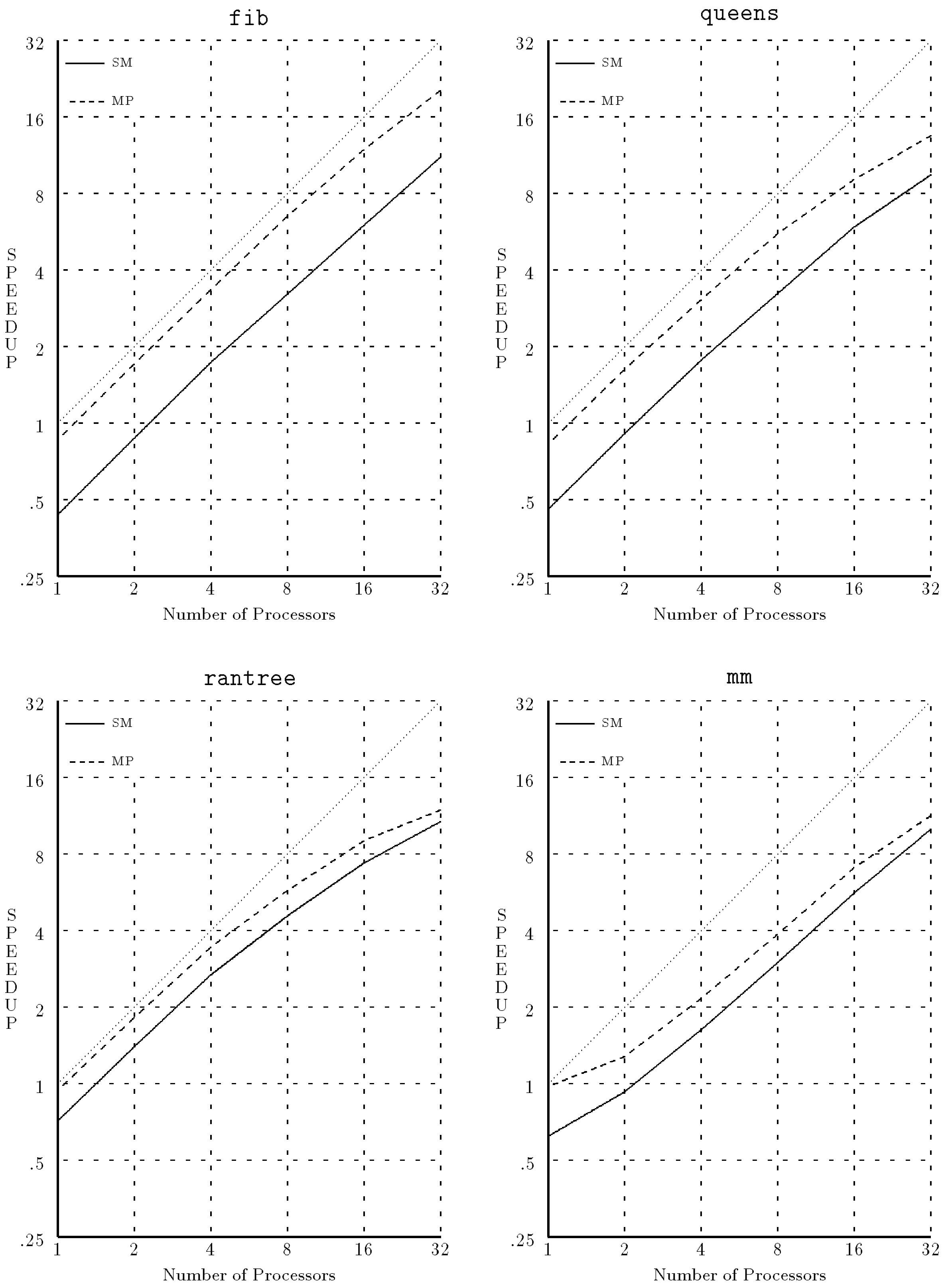


Figure 5.4: Speedup curves for `fib`, `queens`, `rantree` and `mm` on TC2000.

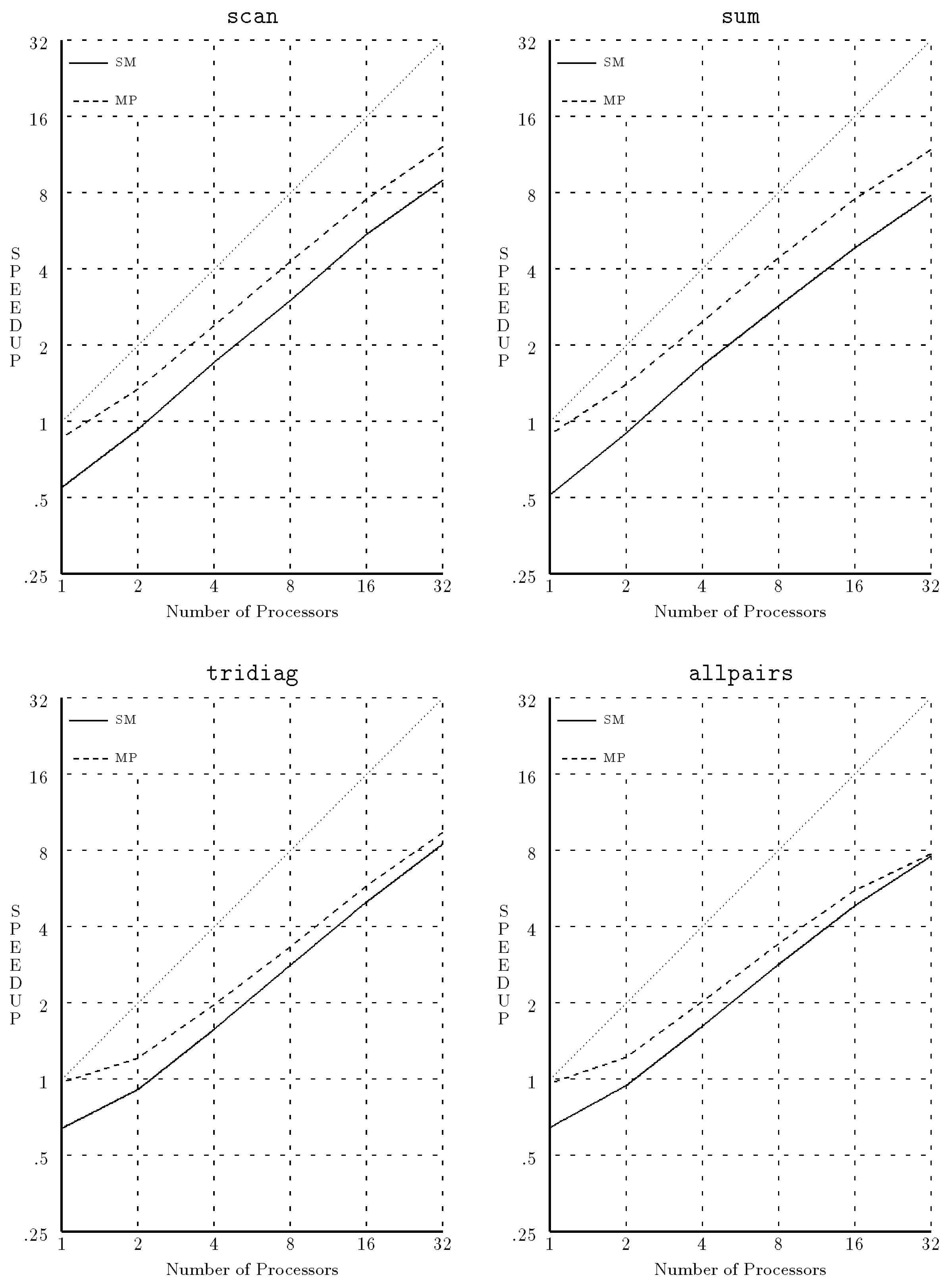


Figure 5.5: Speedup curves for `scan`, `sum`, `tridiag` and `allpairs` on TC2000.

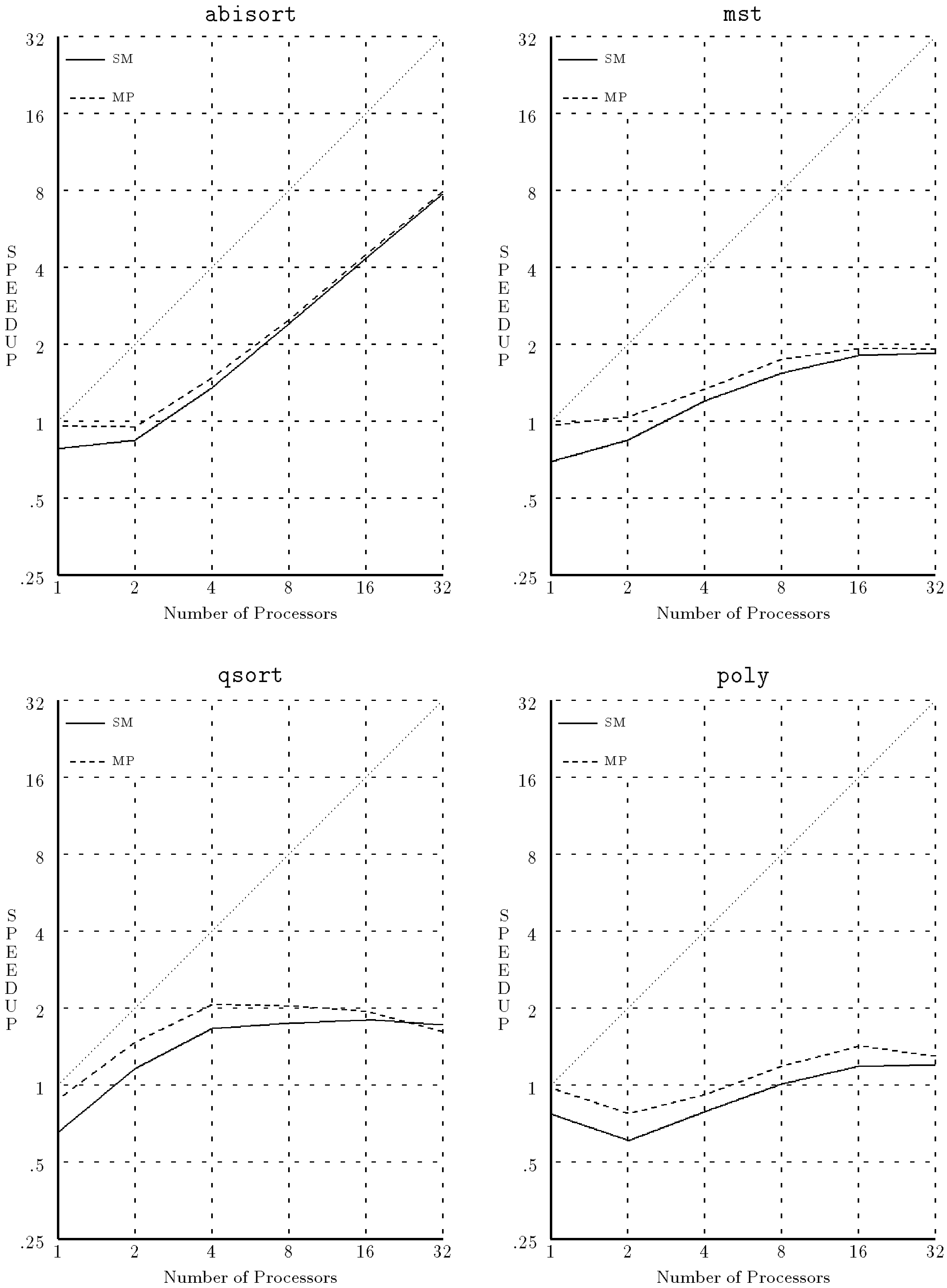


Figure 5.6: Speedup curves for `abisort`, `mst`, `qsort` and `poly` on TC2000.

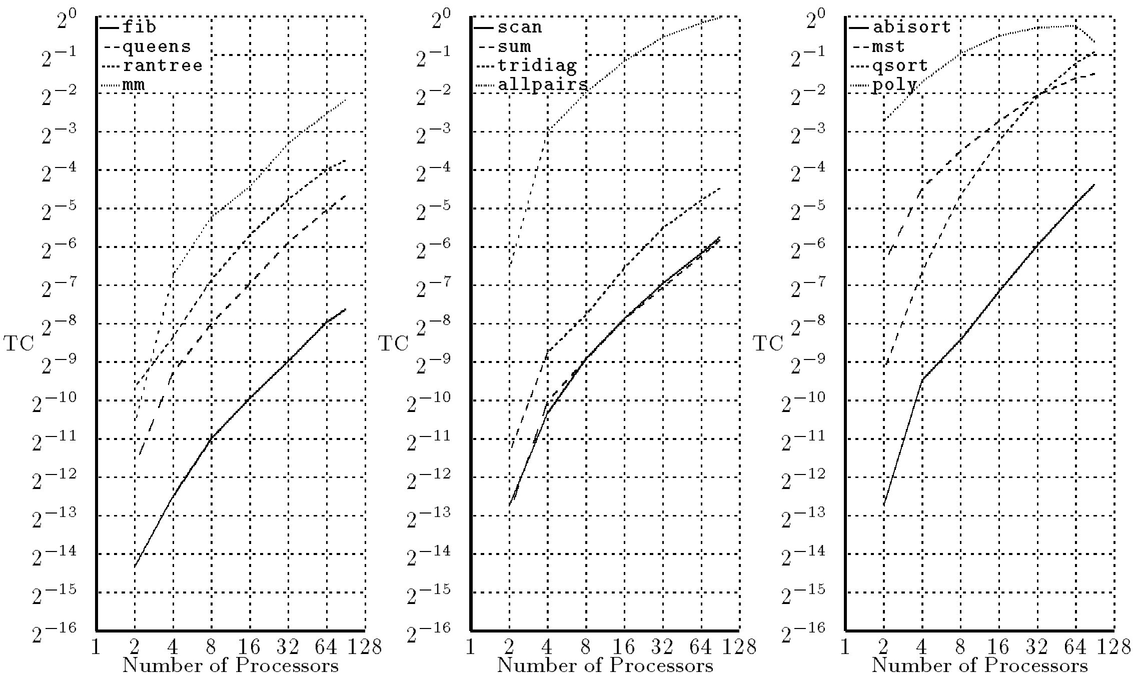


Figure 5.7: Task creation behavior of MP protocol on GP1000.

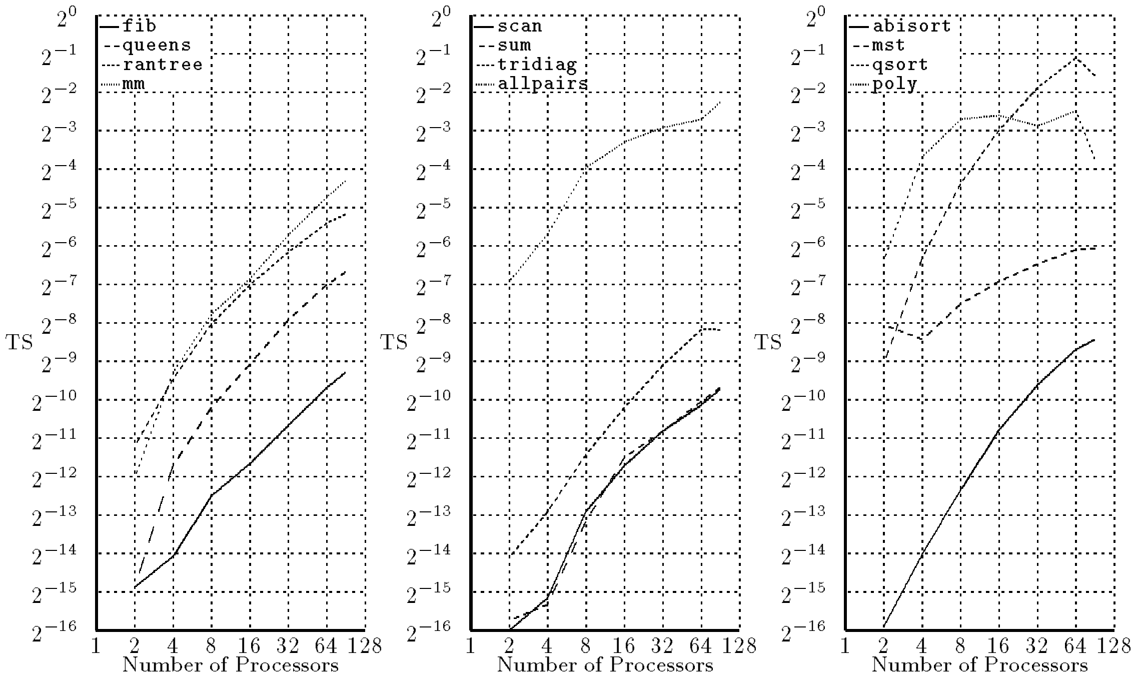


Figure 5.8: Task suspension behavior of MP protocol on GP1000.

2. **Parallel and memory accessing:** `abisort`, `allpairs`, `mm`, `scan`, `sum`, `tridiag`. These programs access memory to various extents. The speedup curves for these programs is "S" like (i.e. the second derivative is initially positive and then negative). A good example is `abisort`. The initial bend in the curve is explained by the increase in cost for accessing shared user data which is distributed evenly across the machine. A memory access has a probability of $\frac{n-1}{n}$ of being to remote memory (where n is the number of processors), so the average cost of an access to shared user data is $\frac{L+R(n-1)}{n}$, where R is the cost of a remote memory access and L is the cost of a local memory access. The bend in the curve is consequently more pronounced for programs which spend a high proportion of their time accessing the heap (e.g. `abisort`, `allpairs`, and `mm`).

3. **Poorly parallel:** `mst`, `poly`, `qsort`. These are programs whose algorithms do not contain much parallelism or that contain a form of parallelism that is not well suited for LTC. The speedup curves for these programs are mostly flat because little of the parallelism is exploited. Generally, the curve starts going down after a certain number of processors because no more parallelism can be exploited but other costs, such as contention and memory interconnect traffic, increase.

5.3.1 Speedup on GP1000

On the GP1000, it is striking how similar the tables and speedup curves are for the SM and MP protocols. The speedup, number of tasks created and the number of task suspensions are normally within a few percent of each other. Nevertheless, the MP protocol typically has a slightly higher speedup, especially for the fine grain programs. This can be explained by the fact that the difference in O_{expose} between protocols is larger for fine grain programs.

Recall that on the GP1000, the SM protocol is using the original continuation semantics and the MP protocol is using the Katz-Weise semantics without legitimacy support. Since the speedup characteristics for both protocols are so similar, it follows that the additional work needed to support the Katz-Weise semantics, mostly that of heapification, is globally negligible. The cost of supporting legitimacy is examined in a later section.

For both protocols, the number of heavyweight tasks created by most programs is a small fraction of what ETC would have created. When `fib` is run on 90 processors, only about .5% of the lightweight tasks are transformed to heavyweight tasks. As suggested by the curves in Figure 5.7, above 4 processors TC increases roughly linearly with the number of processors. The notable exceptions are `allpairs`, `mst` and `poly` whose TC levels off as it nears 1 and `qsort` whose TC first goes up roughly as the square of the number of processors before leveling off as it nears 1. All programs have TC<8% on 90 processors, except `mm` (22-24%), `allpairs` (99%), `mst` (36%), `poly` (63%), and `qsort` (53%). The high TC of these programs can be explained by their coarse granularity and low degree of parallelism (except `qsort` which is explained later). These programs create relatively few lightweight tasks so proportionately more of them need to be stolen to keep the processors working. An extreme example is `allpairs` which on each iteration creates only 116 lightweight tasks (i.e. the maximum parallelism is 117). It isn't surprising that on a 90 processor partition nearly all of the tasks get stolen to balance the load across the machine.

The reason why TC is high for `qsort` (and also `poly`), is that most of the stolen tasks perform very little work (i.e. T_{work} is only a few instructions). Most of `qsort`'s stolen tasks perform a single call to `cons` before they terminate. A handful of similarly simple operations are performed by `poly`'s stolen tasks. Thieves that have just stolen a task will soon be looking for new tasks to steal so the lightweight tasks that are created are likely to get stolen. `Qsort`'s poor speedup is explained by its high TC and low T_{work} combined with its fine granularity (G = 16 μsecs) and heavy remote memory usage ($O_{RemHeap}$ = 3.94).

Similarly Figure 5.8 suggests that, above 4 processors, the number of task suspensions increases fairly linearly with the number of processors for most programs. The notable exceptions are `allpairs`, `mst` and `poly` which have a fairly constant TS above 8 processors.

5.3.2 Speedup on TC2000

On the TC2000, the speedup curves for the MP protocol have a similar shape to those for the MP protocol on the GP1000. The actual speedup is however slightly higher for the TC2000. This is probably due to the TC2000's faster memory system combined with the lower quality of code generated by the compiler (which makes the memory system appear even faster). These factors reduce the relative importance of task management operations and memory accesses. Consequently, a native code implementation on the TC2000 would have a lower speedup (but higher absolute performance!).

The SM protocol however has a consistently lower speedup than that of the MP protocol. For each protocol, the speedup curve starts off at $1/O_{expose}$ on 1 processor (for their respective O_{expose}) and as the number of processors increases the curves tend to get closer. Programs with good speedup characteristics (e.g. `fib` and `sum`) maintain a roughly constant distance between the speedup curves. In other words, the ratio of their run time stays close to the ratio of their O_{expose}. On the other hand, programs with poor speedup characteristics (e.g. `mst` and `qsort`) have speedup curves that become colinear at a high number of processors. This can be explained by the progressive decrease of mandatory work being performed by the program. The main cause of the overhead O_{expose}, that is suboptimally caching the stack and task stack, mostly affects the performance of the mandatory work. The relative importance of suboptimally caching the stack will thus decrease as the programs spend more and more time being idle and/or accessing remote memory.

The only point where the speedup curves cross is for `qsort` at 32 processors. However, the same thing should be expected for other benchmarks on larger partitions because, as the number of processors increases, the benefits of caching decrease whereas the speed of work distribution becomes more critical to performance. Since the SM protocol has a lower steal latency, it will likely outperform the MP protocol on very large partitions. Note however that this might happen at the point where the efficiency (i.e. the ratio of speedup and the number of processors) is so low that it is not cost effective. For instance, the best speedup attained by `qsort` is 2.07 at 4 processors using the MP protocol whereas the best speedup for the SM protocol is 1.8 at 16 processors.

5.4 Effect of Interrupt Latency

In order to study the effect of the interrupt latency on the performance of the MP protocol, the programs were tested on the GP1000 with lower and higher intermittency factors. The previous experiments were performed with $I = 10$ and a new set of measurements were taken with $I = 2$ and $I = 50$. These changes in I cause the interrupt latency to vary roughly by a factor of 5 (decrease and increase respectively). Tables 5.5 and 5.6 contain for each program the value of T_{par}, O_{expose}, and for each partition size: S, TC, and TS. Figures 5.1 through 5.3 contain the speedup curves for each setting of I and also give $\overline{L}$, the average interrupt latency ($\overline{L}$ is T_{par} divided by the number of interrupt checks executed). Note that the average time before an interrupt is detected (T_{detect}) is $\overline{L}/2$.

The settings for I were chosen so that T_{detect} would be roughly comparable to $T_{\texttt{steal\_task}}$, the cost of stealing a task. Experimental measurements put $T_{\texttt{steal\_task}}$ at between 120 and 180 μsecs (depending on the program). When $I = 2$, T_{detect} is normally a fraction of $T_{\texttt{steal\_task}}$ and when $I = 50$, it is normally larger.

Program	T_{par}	O_{expose}		Speedup, TC and TS when number of processors is 2	4	8	16	32	64	90
fib	1.1620	41.4%	S=	1.40	2.76	5.31	9.82	16.69	26.18	30.44
			TC=	.0001	.0002	.0005	.0011	.0022	.0041	.0057
			TS=	.0000	.0001	.0002	.0004	.0007	.0013	.0017
queens	1.4180	29.4%	S=	1.53	2.92	5.37	9.18	14.31	19.09	21.49
			TC=	.0003	.0016	.0043	.0092	.0182	.0325	.0427
			TS=	.0000	.0003	.0009	.0022	.0047	.0084	.0108
rantree	.4880	23.2%	S=	1.57	2.97	5.14	8.20	11.02	13.56	14.14
			TC=	.0012	.0031	.0089	.0192	.0395	.0644	.0843
			TS=	.0004	.0014	.0040	.0078	.0157	.0257	.0327
mm	1.6840	8.0%	S=	1.17	1.82	3.21	5.66	10.02	15.84	18.87
			TC=	.0009	.0122	.0246	.0562	.1081	.1979	.2561
			TS=	.0003	.0026	.0043	.0120	.0237	.0433	.0582
scan	1.2850	20.8%	S=	1.27	2.13	3.76	6.39	10.02	13.75	14.94
			TC=	.0001	.0011	.0023	.0049	.0092	.0159	.0213
			TS=	.0000	.0001	.0002	.0003	.0007	.0012	.0015
sum	.4870	23.9%	S=	1.25	2.11	3.74	6.28	9.73	12.84	13.65
			TC=	.0001	.0010	.0021	.0046	.0089	.0152	.0199
			TS=	.0000	.0001	.0002	.0003	.0008	.0013	.0014
tridiag	4.1510	4.7%	S=	1.17	1.75	2.95	5.07	8.44	12.27	16.18
			TC=	.0005	.0025	.0058	.0126	.0229	.0437	.0587
			TS=	.0000	.0001	.0004	.0012	.0020	.0040	.0051
allpairs	26.1640	5.1%	S=	1.07	1.56	2.60	4.21	6.59	8.62	7.67
			TC=	.0105	.1253	.2752	.4920	.7422	.9464	.9970
			TS=	.0080	.0219	.0658	.1079	.1422	.1741	.2438
abisort	5.9390	25.2%	S=	.58	.74	1.21	2.07	3.59	5.64	6.95
			TC=	.0001	.0014	.0030	.0068	.0168	.0361	.0509
			TS=	.0000	.0001	.0002	.0007	.0016	.0036	.0049
mst	26.3310	12.2%	S=	.91	1.03	1.27	1.62	1.55	1.53	1.38
			TC=	.0120	.0473	.1028	.1828	.2912	.4097	.4511
			TS=	.0038	.0033	.0059	.0094	.0144	.0221	.0249
qsort	.2780	33.0%	S=	1.25	1.56	1.53	1.35	1.13	.99	1.10
			TC=	.0011	.0105	.0431	.1213	.2718	.5112	.6093
			TS=	.0006	.0064	.0250	.0679	.1407	.2402	.2049
poly	2.3990	4.8%	S=	.97	1.05	1.27	1.51	1.51	1.42	.64
			TC=	.1524	.3249	.5235	.7161	.8340	.8846	.5555
			TS=	.0116	.0667	.1504	.1655	.1372	.1715	.0565

Table 5.5: Performance of MP protocol on GP1000 with $I = 2$.

Program	T_{par}	O_{expose}	Speedup, TC and TS when number of processors is							
				2	4	8	16	32	64	90
fib	.9610	16.9%	S=	1.70	3.33	6.34	11.56	19.85	29.36	33.15
			TC=	.0000	.0002	.0004	.0009	.0020	.0033	.0043
			TS=	.0000	.0000	.0001	.0003	.0006	.0009	.0012
queens	1.2140	10.8%	S=	1.78	3.38	6.17	10.29	15.48	20.15	22.28
			TC=	.0003	.0014	.0034	.0079	.0150	.0275	.0340
			TS=	.0000	.0003	.0007	.0018	.0037	.0066	.0083
rantree	.4350	9.8%	S=	1.75	3.29	5.47	8.41	10.77	12.61	12.86
			TC=	.0012	.0028	.0081	.0162	.0293	.0466	.0537
			TS=	.0004	.0013	.0036	.0064	.0109	.0174	.0195
mm	1.5630	.3%	S=	1.21	1.88	3.24	5.62	9.52	13.89	16.04
			TC=	.0010	.0102	.0261	.0538	.0892	.1501	.1862
			TS=	.0000	.0011	.0045	.0086	.0137	.0211	.0260
scan	1.1780	10.7%	S=	1.35	2.24	3.84	6.29	9.60	12.49	13.33
			TC=	.0001	.0009	.0018	.0034	.0063	.0107	.0132
			TS=	.0000	.0000	.0001	.0002	.0004	.0006	.0006
sum	.4350	10.7%	S=	1.37	2.25	3.91	6.38	9.27	11.56	11.77
			TC=	.0001	.0010	.0018	.0034	.0062	.0107	.0139
			TS=	.0000	.0000	.0001	.0002	.0002	.0004	.0005
tridiag	3.9370	-.7%	S=	1.21	1.78	2.93	4.89	7.89	11.18	12.80
			TC=	.0003	.0019	.0044	.0085	.0157	.0241	.0288
			TS=	.0001	.0001	.0003	.0007	.0013	.0018	.0020
allpairs	24.8150	-.3%	S=	1.10	1.56	2.44	3.61	4.68	5.01	4.59
			TC=	.0093	.1238	.2419	.4133	.6207	.8236	.9005
			TS=	.0074	.0155	.0562	.0926	.1298	.1631	.1574
abisort	5.1110	7.7%	S=	.61	.75	1.21	2.04	3.42	5.07	5.69
			TC=	.0001	.0011	.0024	.0057	.0127	.0247	.0310
			TS=	.0000	.0000	.0002	.0004	.0007	.0009	.0009
mst	24.3180	3.6%	S=	.92	1.09	1.32	1.46	1.37	1.12	.99
			TC=	.0090	.0292	.0530	.0791	.0935	.0922	.0911
			TS=	.0034	.0023	.0042	.0053	.0055	.0055	.0054
qsort	.2410	15.3%	S=	1.42	1.69	1.63	1.42	1.17	.93	.93
			TC=	.0009	.0088	.0335	.0853	.1830	.2196	.3396
			TS=	.0003	.0052	.0211	.0534	.1078	.1272	.0997
poly	2.3110	.9%	S=	.97	.96	1.05	1.04	.69	.30	.26
			TC=	.1510	.2819	.4460	.5883	.5713	.3588	.4754
			TS=	.0115	.0892	.1845	.1723	.1206	.0722	.0629

Table 5.6: Performance of MP protocol on GP1000 with $I = 50$.

Overall, the speedup curves indicate that the setting of I does not significantly affect performance. For small partitions, the speedup curves for $I = 50$ are consistently better (but only slightly) than smaller values of I. This is simply due to the slightly lower polling overhead for $I = 50$. As the number of processors increases and the program's work distribution requirements become more critical, the performance for the lower values of I improves and eventually surpasses the performance for $I = 50$. The only exception is `fib` which at 90 processors is still a little faster with $I = 50$. On large partitions, most programs perform better with a setting of $I = 2$ but the performance of $I = 10$ is very close. The difference in performance between $I = 2$ and $I = 10$ at 90 processors is less than 3%, with the exception of `allpairs` (7%) and `mst` (10%). It is interesting to note however that good performance is obtained for all settings of I such that $\overline{L}$ is less than $T_{\texttt{steal\_task}}$ (`allpairs` and `mst` with $I = 10$ are on the border with $\overline{L}$ equal to 152 and 142 μsecs respectively).

5.5 Cost of Supporting Legitimacy

The previous experiments were performed with a version of the MP protocol that did not contain support for legitimacy. To evaluate the cost of supporting legitimacy, the appropriate operations were added to the task management algorithms (i.e. the creation of the legitimacy placeholder, its installation in the stolen task and `end_frame`, and the legitimacy propagation and chain collapsing in `end-body`). The programs were run on the GP1000 with increasingly large partitions (up to 16 processors). Two runs were performed: one with and one without a speculation barrier at the end of the program. The run time was measured and compared to the run time of the version lacking legitimacy support. The overhead (the ratio of run times) is shown in Table 5.7.

The results clearly show that for all programs based on fork-join algorithms, the cost of supporting legitimacy is negligible. In fact, it can hardly be measured at all (the cost is below the noise level of $\pm 2\%$). The collapsing of the legitimacy chain appears to be working out as expected for fork-join algorithms. Only the programs `qsort` and `poly`, which are based on pipeline parallelism, have measurable overheads. The overheads increase with the number of processors, indicating that the legitimacy chain is getting longer and its collapsing is getting more expensive. The highest overhead is 10% for `poly` at 16 processors when a speculation barrier is present. On 16 processors, the overhead is a little lower (by 2 to 3%) when there is no speculation barrier.

Program	1 without	1 with	2 without	2 with	4 without	4 with	8 without	8 with	16 without	16 with
fib	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.99	1.01	1.00
queens	1.00	1.00	1.00	1.00	1.00	1.00	.99	1.01	1.00	.99
mm	1.00	1.00	1.01	1.00	1.00	1.01	1.00	1.00	1.00	1.00
scan	1.00	1.00	.99	1.00	1.00	1.01	.98	1.01	1.01	1.00
rantree	1.00	1.00	1.00	1.00	1.01	1.01	1.00	1.00	.98	.99
sum	1.00	1.00	1.00	1.00	1.00	.99	.99	1.01	1.02	1.02
tridiag	1.00	1.00	1.02	1.00	1.01	1.00	1.02	.99	1.00	.99
allpairs	1.00	1.00	.99	1.00	1.00	1.00	1.01	1.01	1.00	1.00
abisort	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00
mst	1.00	1.00	1.00	.99	1.00	1.00	1.00	1.00	1.00	1.00
poly	1.00	.98	1.00	1.00	1.00	1.00	1.06	1.05	1.03	1.05
qsort	1.00	1.00	1.01	1.01	1.01	1.03	1.05	1.06	1.07	1.10

Table 5.7: Overhead of supporting legitimacy, with and without speculation barrier on GP1000.

5.6 Summary

This chapter has evaluated the performance of the SM and MP protocol implementations of LTC on large shared-memory multiprocessors (up to 90 processors). Experiments were conducted with several benchmark programs on the GP1000 multiprocessor (which lacks a data cache) and the TC2000 (which has a data cache). The results show that

- **The parallelization cost is low** — The overhead of parallelizing a sequential program (by adding futures and touches) is typically less than 20% when using the MP protocol. For the SM protocol, the overhead is twice as large when a cache is not available. However, when a cache is available the overhead is much more important (up to a factor of two on the TC2000) because the SM protocol must cache the stack and LTQ suboptimally.
- **LTC scales well** — Programs with a high degree of parallelism have fairly linear speedup with respect to the sequential version of the program. The SM and MP protocols have almost identical speedup curves when a cache is not available. When a cache is available, the speedup curve for the MP protocol is consistently better due to the difference in caching policy. However, this difference gradually

decreases as the number of processors increases because the caching policy becomes less important (the caching policy has no influence on the idle time and remote memory access time which increase with the number of processors).

- **Interrupt latency can be relatively high** — For the MP protocol, an interrupt latency as high as the time to steal a task provides adequate performance. On a 90 processor GP1000, the run time is usually within 3% of the run time for the best latency.

- **Supporting the Katz-Weise semantics and legitimacy generally has a negligible impact on performance** — There was no noticeable performance difference between a version of the system that supported the Katz-Weise semantics and one that did not. This indicates that the additional cost of heapification is low relatively to the other costs of stealing (in particular, the remote memory references needed to transfer the task between processors). The cost of legitimacy propagation and testing is also very low. The overhead for fork-join programs is too low to measure. However, programs with a less restrictive task termination order exhibit a measurable but small overhead (no more than 10% on 16 processors).

Chapter 6

Conclusion

The initial goal of this work was the implementation of a high-performance Multilisp system. Earlier implementations of Multilisp, such as Concert Multilisp [Halstead, 1984] and MultiScheme [Miller, 1987], gave interesting self relative speedups but because they were based on interpreters it was not clear that the same speedups would apply to a "production quality" system. As a first step of this work, a highly optimizing compiler for Scheme was developed to provide a realistic setting for exploring new implementation strategies for Multilisp and evaluating their performance. This effort resulted in Gambit [Feeley and Miller, 1990], currently the best Scheme compiler in terms of performance of the code generated.

The system was ported to the GP1000 and TC2000 multiprocessors, and support for Multilisp's parallelism constructs added to the compiler. Initially the eager task creation (ETC) method was used to implement futures but it was soon clear that the overhead of task creation would be too high for fine grain programs (as explained in Chapter 2). Work on the lazy task creation (LTC) mechanism was triggered by a comment on "lazy futures" in [Kranz *et al.*, 1989]. LTC postpones the creation of a task until it needs to be transferred to another processor, the "thief". Consequently, the overhead of task creation is mostly dependent on the work distribution needs of the program and not so much the program's granularity. For divide-and-conquer programs, LTC has the nice property of transferring large pieces of work and roughly balancing the work between the thief and victim processors. This helps reduce the number of task transfers needed to keep processors busy. Most tasks end up being executed locally at low cost.

Eric Mohr independently explored the LTC mechanism with the Mul-T system on the Encore Multimax multiprocessor (a UMA computer with up to 20 processors) and ended up using a version of the shared-memory (SM) protocol very similar to the one used here [Mohr, 1991]. In the SM protocol, thief processors directly access the stack of other processors to "steal" tasks. This thesis extends his results in several ways:

- **Experience on large machines** — Experiments on a 90 processor GP1000 with a wide range of benchmarks provide concrete evidence that LTC scales well to large machines and that good speedup is possible for realistic programs.

- **Support of a rich semantics** — The semantics of the Multilisp language does not have to be impoverished to attain good performance. In fact, the laziness of LTC can be exploited to implement several programming features at low cost. These include

  - The Katz-Weise continuation semantics with legitimacy; which provides an elegant semantics for first-class continuations.
  - Dynamic scoping.
  - Fairness.

- **Better implementation of the SM protocol** — A slightly faster implementation of the SM protocol was developed. It requires fewer instructions, fewer memory references, and is simpler to prove correct.

- **The message-passing (MP) protocol** — The main problem with the SM protocol is that all processors must have access to the runtime stack. On machines lacking coherent-caches, such as the TC2000, the stack can only be cached in write-through mode instead of the more efficient copy-back mode. This affects the speed of computation in general (parallel and sequential parts of the programs suffer). A study of several benchmarks in Chapter 3 shows that the stack is one of the most frequently accessed data structures and that the difference in caching policy can account for an important difference in performance (as high as a factor of two on the TC2000).

  In the MP protocol the stack is a private data structure that can be cached optimally. To obtain a task to run, a thief processor sends a work request message to the "victim" processor. When the request is serviced, the victim accesses its own stack to remove a lazy task and packages it in a heavyweight task that is sent back to the thief. This approach would appear to depend on a low latency interrupt mechanism, such as polling, but in fact the experiments indicate that performance is close to optimal when the interrupt latency is comparable to the time required to perform the task steal.

6.1 Future Work

The results of this thesis suggest that task partitioning can be done efficiently on machines that lack an efficient shared memory. Coherent-caches are not really required, as shown by the MP protocol implementation of LTC. There is thus hope that, at least for some problems, Multilisp can run efficiently on distributed-memory machines. A machine like the Thinking Machine's CM-5, which lacks a shared memory but provides a fast message-passing system, would be an ideal candidate.

One of the shortcomings of LTC as implemented here is that it does not address the data partitioning problem. The scheduling algorithm makes no attempt to run a task on (or close to) the processor that contains the data it accesses. As shown in Chapter 3, a substantial performance loss is attributable to the remote memory accesses to user data (up to a factor of 5 on the GP1000 and a factor of 3 on the TC2000). Coherent-caches may help reduce this problem on shared-memory machines but the penalty on distributed-memory machines will be much higher.

Another problem is the overhead of touching. Contrary to Multilisp's original specification, this work has assumed that touches are inserted explicitly by the user. This is hard to do for programs with complex data dependencies. It would be more convenient for the user if touches were inserted automatically by the compiler. Adding a touch on each strict operation is a poor solution because it causes a high overhead. On the GP1000 the overhead is roughly a factor of 2 on typical programs (but a lower overhead may be possible on modern processors which are optimized for register operations). A better solution would be for the compiler to do a data-flow analysis of the program to identify all the strict operations that might be passed a placeholder. Control-flow and data-flow analysis techniques such as [Shivers, 1988, Shivers, 1991] would be a good starting point.

Appendix A

Source Code for Parallel Benchmarks

This appendix contains the source code for the parallel benchmark programs used in chapters 2, chapters 3, chapters 4, and 5. A general description of these programs is given in section 2.9. Half of the programs were originally written in Mul-T by Eric Mohr as part of his PhD thesis work [Mohr, 1991]. These programs were translated to Scheme with superficial changes to suit Gambit's particular features. These changes include

- Macro definitions (Gambit uses the non-standard construct `##define-macro`).
- The definition of record structures (Gambit does not have a predefined construct for defining structures; plain vectors were used instead).
- The performance of `abisort`, `allpairs` and `mst` was improved by partial evaluating the programs by hand. The algorithms are the same but some of the procedure abstractions were removed by replacing procedure definitions by macro definitions.
- The programs `abisort`, `rantree` and `tridiag` originally had a few uses of a non-standard construct to return multiple values. Since Gambit does not have such a feature, the multiple returns were reformulated in standard Scheme. This only affects `rantree`'s performance because the two other programs used multiple value returns exclusively in the initialization phase (which is not measured).
- `Tridiag`, which solves a set of equations, uses only half as many equations (i.e. 32767). This data set just barely fits in the memory available on a single processor node of the GP1000. About 2 Mbytes per processor (out of a total of 4 Mbytes) are

available for the heap after Gambit has started. This makes it possible to evaluate the program in a uniprocessor configuration (which is useful to generate speedup curves). All other programs were run with the same data set size in order to make direct comparisons easier.

The new programs fall into two main classes. The programs `mm` (matrix multiplication), `scan` (parallel prefix operation on a vector), and `sum` (parallel reduction operation on a vector) are based on divide and conquer algorithms. The program `poly` (polynomial multiplication) implements a form of pipeline parallelism and `qsort` (quicksort) is a combination of pipeline and divide and conquer parallelism.

The programs were modified in certain places to address shared-memory problems. To lessen contention to shared data in vectors, the non-standard procedures `make-cvector` and `cvector-ref` were used instead of the corresponding standard vector operations. A `cvector` is a vector with immutable elements (i.e. a "constant vector"). When a `cvector` is created, it is copied to the local memory of each processor. Access to a `cvector` is thus both contention free and fast (as fast as a local memory reference). However, access to the elements of a `cvector` may still exhibit some contention and remote memory reference latency if the elements are memory allocated structures (as is the case in `tridiag`, the only program that uses `cvector`s).

When the shared data was in mutable vectors (i.e. the programs `allpairs`, `mm`, `mst`, `scan` and `sum`), the non-standard procedures `make-dvector`, `dvector-ref` and `dvector-set!` were used instead of the corresponding standard vector operations. A `dvector` is a vector whose entries are evenly allocated across the machine (i.e. a "distributed vector"). If entry i is in the local memory of processor j, then entry $i+1$ is on processor $j+1$ (modulo the number of processors). On an n processor machine, a reference to the vector will correspond to a local memory reference with probability $\frac{1}{n}$ and to a remote reference with probability $\frac{n-1}{n}$. This means that the average cost of an access to a `dvector` increases with the number of processors, quickly approaching the cost of a remote reference. `Dvector`s have good contention characteristics because during a given cycle there can be as many accesses to `dvector`s as there are processors. The average number of contention free accesses will be lower, but this is more of an academic question since in general, processors do not all access memory at the same moment.

Record structures were similarly distributed where possible (i.e. the programs `abisort`, `mst` and `tridiag`). This was done with a call to the procedure `make-vector-chain` which builds a chain of fixed size vectors that are evenly distributed across the machine.

The creation of all these special data structures happens once and for all in the initialization phase of the programs. Thus, it doesn't contribute to the measurements. Memory allocation in the main part of the program only occurs for `qsort` and `poly` and is done with the standard `cons` procedure. This means that space is allocated in the local memory of the processor doing the allocation.

The programs were all compiled with special declarations meant to improve performance. All references to predefined variables, such as `cons` and `car`, were assumed to be to the corresponding primitive procedure. This essentially means that inline code was generated for calls to simple predefined procedures. All arithmetic operations were assumed to be on small integers (fixnums), except for the program `poly` which uses generic arithmetic.

In the code that follows, `FUTURE` and `TOUCH` have been underlined to make them stand out. The last line of each program is a call to the macro `benchmark`, which starts the run. The subforms passed to `benchmark` are in order: the name of the program, the expression used to initialize the input data and the expression that starts the part of the program being measured. A brief description is included with each program.

A.1 abisort

This program sorts 16384 integers using the adaptive bitonic sort algorithm described in [Bilardi and Nicolau, 1989].

```
(##define-macro (make-node) `(make-vector 3 #f))
(##define-macro (node-left  x) `(vector-ref ,x 0))
(##define-macro (node-value x) `(vector-ref ,x 1))
(##define-macro (node-right x) `(vector-ref ,x 2))
(##define-macro (node-left-set!  x v) `(vector-set! ,x 0 ,v))
(##define-macro (node-value-set! x v) `(vector-set! ,x 1 ,v))
(##define-macro (node-right-set! x v) `(vector-set! ,x 2 ,v))

(##define-macro (swap-left l r)
  `(let ((temp (node-left ,l)))
     (node-left-set! ,l (node-left ,r))
     (node-left-set! ,r temp)))

(##define-macro (swap-right l r)
  `(let ((temp (node-right ,l)))
     (node-right-set! ,l (node-right ,r))
     (node-right-set! ,r temp)))

(##define-macro (fixup-tree-1 root up?)
  `(let loop ((pl (node-left ,root))
              (pr (node-right ,root)))
     (if pl
         (compare-and-swap pl pr ,up?
           ;swap right subtrees, search path goes left
           (begin (swap-right pl pr)
                  (loop (node-left pl) (node-left pr)))
           ;search path goes right
           (loop (node-right pl) (node-right pr))))))

(##define-macro (fixup-tree-2 root up?)
  `(let loop ((pl (node-left ,root))
              (pr (node-right ,root)))
     (if pl
         (compare-and-swap pl pr ,up?
           ;swap left subtrees, search path goes right
           (begin (swap-left pl pr)
                  (loop (node-right pl) (node-right pr)))
           ;search path goes left
           (loop (node-left pl) (node-left pr))))))
```

```
(##define-macro (compare-and-swap node1 node2 up? true false)
  `(let ((v1 (node-value ,node1))
         (v2 (node-value ,node2)))
     (cond ((,(if up? `>= `<) v1 v2)
            (node-value-set! ,node1 v2)
            (node-value-set! ,node2 v1)
            ,true)
           (else ,false))))

(##define-macro (pbimerge root spare up?)
  `(let loop ((root ,root) (spare ,spare))
     (compare-and-swap root spare ,up?
       (fixup-tree-1 root ,up?)
       (fixup-tree-2 root ,up?))
     (cond ((node-left root)
            (let ((left-half (FUTURE (loop (node-left root) root))))
              (loop (node-right root) spare)
              (TOUCH left-half))))))

(define (pbisort-up root spare)
  (let ((left (node-left root)))
    (if left
      (let ((left-half (FUTURE (pbisort-up left root))))
        (pbisort-down (node-right root) spare)
        (TOUCH left-half)
        (pbimerge root spare #t)))
      (compare-and-swap root spare #t #t #f)))

(define (pbisort-down root spare)
  (let ((left (node-left root)))
    (if left
      (let ((left-half (FUTURE (pbisort-down left root))))
        (pbisort-up (node-right root) spare)
        (TOUCH left-half)
        (pbimerge root spare #f)))
      (compare-and-swap root spare #f #t #f)))

(define (new-node l r v)
  (let ((node (make-node*)))
    (node-left-set!  node l)
    (node-right-set! node r)
    (node-value-set! node v)
    node))
```

```
(define node-chain #f)

(define (init-node-chain n) ; make a chain of 3 element vects
  (set! node-chain (make-vector-chain n 3)))

(define (make-node*)
  (let ((node node-chain))
    (set! node-chain (vector-ref node 0))
    node))

(define (make-inorder-tree depth)
  (let loop ((i 0)
             (depth depth))
    (if (= depth 1)
        (cons (new-node #f #f i) i)
        (let* ((x (loop i (- depth 1)))
               (l-tree (car x))
               (l-imax (cdr x)))
          (let* ((y (loop (+ l-imax 2) (- depth 1)))
                 (r-tree (car y))
                 (r-imax (cdr y)))
            (cons (new-node l-tree r-tree (+ l-imax 1)) r-imax))))))

(define r #f)
(define s #f)

(define k 14)

(define (init)
  (init-node-chain (expt 2 k))
  (let* ((x (make-inorder-tree k))
         (root (car x))
         (imax (cdr x)))
    (let ((spare (new-node #f #f (+ imax 1))))
      (set! r root)
      (set! s spare))))

(benchmark ABISORT (init) (pbisort-up r s))
```

A.2 allpairs

This program computes the shortest paths between all pairs of 117 nodes using a parallel version of Floyd's algorithm.

```
(##define-macro (do-all var lo hi . body)
  `(let loop ((,var ,lo) (hi ,hi))
     (if (= ,var hi)
         (let () ,@body)
         (let* ((mid (quotient (+ ,var hi) 2))
                (lo-half (FUTURE (loop ,var mid))))
           (loop (+ mid 1) hi)
           (TOUCH lo-half)))))

(define (apsp/par a n)
  (let ((n-1 (- n 1)))
    (do ((k 0 (+ k 1)))
        ((= k n))
      (let ((k*n (* k n)))
        (do-all i 0 n-1
          (let* ((i*n (* i n))
                 (i*n+k (+ i*n k)))
            (do ((j 0 (+ j 1)))
                ((= j n))
              (let* ((kpath (+ (dvector-ref a i*n+k)
                               (dvector-ref a (+ k*n j))))
                     (i*n+j (+ i*n j)))
                (if (< kpath (dvector-ref a i*n+j))
                    (dvector-set! a i*n+j kpath))))))))))

(define (make-linear-adjacency-matrix n)
  (let ((a (make-dvector (* n n) (quotient most-positive-fixnum 2))))
    (dvector-set! a 0 0)
    (do ((i 1 (+ i 1)))
        ((= i n))
      (dvector-set! a (+ (* i n) i) 0)
      (dvector-set! a (+ (* (- i 1) n) i) 1)
      (dvector-set! a (+ (* i n) (- i 1)) 1))
    a))

(define a #f)

(define n 117)

(define (init)
  (set! a (make-linear-adjacency-matrix n)))

(benchmark ALLPAIRS (init) (apsp/par a n))
```

A.3 fib

This program computes F_{25}, the 25^{th} fibonacci number, using the "standard" doubly recursive algorithm.

```
(define (pfib n)
  (let fib ((n n))
    (if (< n 2)
        n
        (let* ((f1 (FUTURE (fib (- n 1))))
               (f2 (fib (- n 2))))
          (+ (TOUCH f1) f2)))))

(benchmark FIB #f (pfib 25))
```

A.4 mm

This program multiplies two matrices of integers (50 by 50).

```
(define (mm m1 m2 m3) ; m1 * m2 -> m3

  (define (compute-entry row col) ; loop to compute inner product
    (let loop ((i (+ row (- n 1)))
               (j (+ (* n (- n 1)) col))
               (sum 0))
      (if (>= j 0)
        (loop (- i 1)
              (- j n)
              (+ sum (* (dvector-ref m1 i) (dvector-ref m2 j))))
        (dvector-set! m3 (+ (+ i 1) col) sum))))

  (define (compute-cols-between row i j) ; DAC over columns
    (if (= i j)
      (compute-entry row i)
      (let ((mid (quotient (+ i j) 2)))
        (let* ((half1 (FUTURE (compute-cols-between row i mid)))
               (half2 (compute-cols-between row (+ mid 1) j)))
          (TOUCH half1)))))

  (define (compute-rows-between i j) ; DAC over rows
    (if (= i j)
      (compute-cols-between (* i n) 0 (- n 1))
      (let ((mid (quotient (+ i j) 2)))
        (let* ((half1 (FUTURE (compute-rows-between i mid)))
               (half2 (compute-rows-between (+ mid 1) j)))
          (TOUCH half1)))))

  (compute-rows-between 0 (- n 1)))

(define m1 #f)
(define m2 #f)
(define m3 #f)

(define n 50)

(define (init)
  (set! m1 (make-dvector (* n n) 2))
  (set! m2 (make-dvector (* n n) 2))
  (set! m3 (make-dvector (* n n) #f)))

(benchmark MM (init) (mm m1 m2 m3))
```

A.5 mst

This program computes the minimum spanning tree of a 1000 node graph. A parallel version of Prim's algorithm is used.

```
(##define-macro (make-city) `(make-vector 4 #f))
(##define-macro (city-x        x) `(vector-ref ,x 0))
(##define-macro (city-y        x) `(vector-ref ,x 1))
(##define-macro (city-closest  x) `(vector-ref ,x 2))
(##define-macro (city-distance x) `(vector-ref ,x 3))
(##define-macro (city-x-set!        x v) `(vector-set! ,x 0 ,v))
(##define-macro (city-y-set!        x v) `(vector-set! ,x 1 ,v))
(##define-macro (city-closest-set!  x v) `(vector-set! ,x 2 ,v))
(##define-macro (city-distance-set! x v) `(vector-set! ,x 3 ,v))

(define (new-city x y closest distance)
  (let ((city (make-city*)))
    (city-x-set! city x)
    (city-y-set! city y)
    (city-closest-set! city closest)
    (city-distance-set! city distance)
    city))

(define (prim cities ncities find-closest-city)
  (let* ((max-i (- ncities 1))
         (target0 (dvector-ref cities max-i)))
    (city-closest-set! target0 target0)  ;; makes drawing easier
    (let loop ((max-i (- max-i 1))
               (target target0))
      (if (= max-i 0)
          (add-last-city (dvector-ref cities 0) target)
          (let* ((closest-i (find-closest-city cities max-i target))
                 (newcity (dvector-ref cities closest-i)))
            (dvector-set! cities closest-i (dvector-ref cities max-i))
            (dvector-set! cities max-i newcity)
            (loop (- max-i 1) newcity))))))

(define (add-last-city city newcity)
  (let* ((newdist (distance city newcity))
         (olddist (city-distance city)))
    (cond ((< newdist olddist)
           (city-distance-set! city newdist)
           (city-closest-set! city newcity)))))
```

```
(define (distance c1 c2)
  (let ((dx (- (city-x c1) (city-x c2)))
        (dy (- (city-y c1) (city-y c2))))
    (+ (* dx dx) (* dy dy))))

(##define-macro (combine-interval/ptree lo hi f combine)
  `(let ((lo ,lo) (hi ,hi))
     (let* ((n (+ (- hi lo) 1))
            (adjust (- lo 1))
            (first-leaf (quotient (+ n 1) 2))
            (treeval
             (let loop ((i 1))
               (cond ((< i first-leaf)
                      (let* ((left (FUTURE (loop (* i 2))))
                             (right (,combine (loop (+ (* i 2) 1))
                                              (,f (+ i adjust)))))
                        (,combine right (TOUCH left))))
                     (else
                      (,f (+ i adjust)))))))
       (if (even? n)
           (,combine treeval (,f hi))
           treeval))))

(define (find-closest-city/ptree cities max-i newcity)
  (combine-interval/ptree 0 max-i
    (lambda (i) (update-city i cities newcity))
    (lambda (i1 i2)
      (if (< (city-distance (dvector-ref cities i1))
             (city-distance (dvector-ref cities i2)))
          i1
          i2))))

(define (update-city i cities newcity)
  (let* ((city (dvector-ref cities i))
         (newdist (distance city newcity))
         (olddist (city-distance city)))
    (cond ((< newdist olddist)
           (city-distance-set! city newdist)
           (city-closest-set! city newcity)))
    i))
```

```
(define city-chain #f)

(define (init-city-chain n) ; make a chain of 4 element vects
  (set! city-chain (make-vector-chain n 4)))

(define (make-city*)
  (let ((city city-chain))
    (set! city-chain (vector-ref city 0))
    city))

(define random (make-random 3434534))
(define random-range 1000)

(define (make-random-vector-of-cities n)
  (let ((cities (make-dvector n)))
    (do ((i 0 (+ i 1)))
        ((>= i n) cities)
      (dvector-set! cities i
        (new-city (modulo (random) random-range)
                  (modulo (random) random-range)
                  '()
                  most-positive-fixnum
                  )))
    cities))

(define c #f)

(define n 1000)

(define (init)
  (init-city-chain n)
  (set! c (make-random-vector-of-cities n)))

(benchmark MST (init) (prim c n find-closest-city/ptree))
```

A.6 poly

This program computes the square of a 200 term polynomial of x (with integer coefficients) and evaluates the resulting polynomial for a certain value of x.

```
(##declare (generic)) ; use generic arithmetic

(define (poly* p1 p2) ; compute p1*p2
  (if (or (null? p1) (null? p2))
    '()
    (poly+*k (cons 0 (poly* p1 (cdr p2)))
             p1
             (car p2))))

(define (poly+*k p1 p2 k) ; compute p1+p2*k
  (if (null? p2)
    p1
    (if (null? p1)
      (let ((rest (FUTURE (poly+*k '() (cdr p2) k))))
        (cons (* (car p2) k) rest))
      (let ((rest (FUTURE (poly+*k (TOUCH (cdr p1)) (cdr p2) k))))
        (cons (+ (car p1) (* (car p2) k)) rest)))))

(define (poly-eval p x) ; compute value of p at x
  (let loop ((p p) (y 1) (sum 0))
    (if (pair? p)
      (loop (TOUCH (cdr p)) (* x y) (+ sum (* (car p) y)))
      sum)))

(define p ; 200 terms
  '(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))

(benchmark POLY #f (poly-eval (poly* p p) 1))
```

A.7 qsort

This program sorts a list of 1000 integers using a parallel version of the Quicksort algorithm.

```
(define (qsort lst)

  (##define-macro (filter keep? lst)
    `(let loop ((lst ,lst))
       (let ((lst (TOUCH lst)))
         (if (pair? lst)
           (let ((head (car lst)))
             (if (,keep? head)
               (cons head (FUTURE (loop (cdr lst))))
               (loop (cdr lst))))
           '()))))

  (define (qs lst tail)
    (if (pair? lst)
      (let ((pivot (car lst))
            (other (cdr lst)))
        (let ((sorted-larger
                (FUTURE (qs (filter (lambda (x) (not (< x pivot))) other)
                            tail))))
          (qs (filter (lambda (x) (< x pivot)) other)
              (cons pivot sorted-larger))))
      tail))

  (qs lst '()))

(define (walk lst)
  (let loop ((lst lst))
    (let ((lst (TOUCH lst)))
      (if (pair? lst) (loop (cdr lst)))))
  lst)

(define l ; randomized list of numbers 0 to 999
  '(34 313 852 803 941 931 63 581 309 569 62 561 602 572 353 253 815 869
    928 472 247 808 88 698 315 152 58 465 881 888 652 312 47 69 279 418
...
    361 762 53 664 892 768 778 685 190 52 665 289 558 188 455 408 381 805
    791 68 293 827 529 301 825 357 202 365 955 746 449 927 823))

(benchmark QSORT #f (walk (qsort l)))
```

A.8 queens

This program computes the number of solutions to the n-queens problem, with $n = 10$.

```
(define (queens n)
  (let try ((rows-left n)
            (free-diag1 -1)              ;all bits set
            (free-diag2 -1)
            (free-cols (- (ash 1 n) 1)))  ;bits 0 to n-1 set
    (let ((free (logand free-cols (logand free-diag1 free-diag2))))
      (let loop ((col 1))
        (cond ((> col free)
               0)
              ((= (logand col free) 0)
               (loop (* col 2)))
              ((= rows-left 1)
               (+ 1 (loop (* col 2))))
              (else
               (let* ((sub-solns
                       (FUTURE
                        (try (- rows-left 1)
                             (+ (ash (- free-diag1 col) 1) 1)
                             (ash (- free-diag2 col) -1)
                             (- free-cols col))))
                      (other-solns (loop (* col 2))))
                 (+ (TOUCH sub-solns) other-solns))))))))

(benchmark QUEENS #f (queens 10))
```

A.9 rantree

This program models the traversal of a random binary tree with on the order of 32768 nodes. The branching factor is 50%.

```
(define (lehmer-left  seed) (+ 1 (* seed #xface475)))
(define (lehmer-right seed) (+ 1 (* seed #x283feed)))

(define (pseudo-random-tree n)
  (let loop ((n n) (seed 1))
    (cond ((<= n 2)
           n)
          ((> seed 0)
           (let* ((ln (+ 1 (modulo seed (- n 2))))
                  (rn (- (- n 1) ln))
                  (left (FUTURE (loop ln (lehmer-left seed))))
                  (right (loop rn (lehmer-right seed))))
             (+ (TOUCH left) (+ right 1))))
          (else
           (+ 1 (loop (- n 1) (lehmer-left seed)))))))

(benchmark RANTREE #f (pseudo-random-tree 32768))
```

A.10 scan

This program computes the parallel prefix sum of a vector of 32768 integers. The vector is modified in place. A given element is replaced by the sum of itself and all preceding elements.

```
(##define-macro (scan f c v)
  `(let ((c ,c) (v ,v))
     (let ((n (dvector-length v)))

       (define (pass1 i j)
         (if (< i j)
           (let* ((m      (quotient (+ i j) 2))
                  (left   (FUTURE (pass1 i m)))
                  (right  (pass1 (+ m 1) j))
                  (result (,f (TOUCH left) right)))
             (dvector-set! v j result)
             result)
           (dvector-ref v j)))

       (define (pass2 i j c)
         (if (< i j)
           (let* ((m     (quotient (+ i j) 2))
                  (left  (FUTURE (pass2 i m c)))
                  (cc    (,f c (dvector-ref v m)))
                  (right (pass2 (+ m 1) j cc)))
             (dvector-set! v m cc)
             (TOUCH left))))

       (if (> n 0)
         (let ((j (- n 1)))
           (pass1 0 j)
           (pass2 0 j c)
           (dvector-set! v j (,f c (dvector-ref v j))))))))

(define (scan+ c v) (scan + c v))

(define v #f)

(define n 32768)

(define (init)
  (set! v (make-dvector n 0)))

(benchmark SCAN (init) (scan+ 0 v))
```

A.11 sum

This program computes the sum of a vector of 32768 integers.

```
(define (sum vect l h) ; sum vector from 'l' to 'h'
  (if (= l h)
    (dvector-ref vect l)
    (let* ((mid (quotient (+ l h) 2))
           (lo  (FUTURE (sum vect l mid)))
           (hi  (sum vect (+ mid 1) h)))
      (+ (TOUCH lo) hi))))

(define v #f)

(define n 32768)

(define (init)
  (set! v (make-dvector n 1)))

(benchmark SUM (init) (sum v 0 (- n 1)))
```

A.12 tridiag

This program solves a tridiagonal system of 32767 equations.

```
(##define-macro (a obj) `(vector-ref ,obj 0))
(##define-macro (b obj) `(vector-ref ,obj 1))
(##define-macro (c obj) `(vector-ref ,obj 2))
(##define-macro (y obj) `(vector-ref ,obj 3))
(##define-macro (x obj) `(vector-ref ,obj 4))
(##define-macro (a-set! obj v) `(vector-set! ,obj 0 ,v))
(##define-macro (b-set! obj v) `(vector-set! ,obj 1 ,v))
(##define-macro (c-set! obj v) `(vector-set! ,obj 2 ,v))
(##define-macro (y-set! obj v) `(vector-set! ,obj 3 ,v))
(##define-macro (x-set! obj v) `(vector-set! ,obj 4 ,v))

(define (reduce/par equ imid)

  (define (reduce-equation i delta)
    (let* ((equ-ileft  (cvector-ref equ (- i delta)))
           (equ-iright (cvector-ref equ (+ i delta)))
           (equ-i (cvector-ref equ i))
           (e (- (quotient (a equ-i) (b equ-ileft))))
           (f (- (quotient (c equ-i) (b equ-iright)))))
      (a-set! equ-i (* e (a equ-ileft)))
      (c-set! equ-i (* f (c equ-iright)))
      (b-set! equ-i (+ (b equ-i)
                       (+ (* e (c equ-ileft))
                          (* f (a equ-iright)))))
      (y-set! equ-i (+ (y equ-i)
                       (+ (* e (y equ-ileft))
                          (* f (y equ-iright)))))))

  (let do-branch ((i imid)
                  (delta (quotient imid 2)))
    (if (= delta 1)
        (reduce-equation i delta)
        (let* ((ileft  (- i delta))
               (iright (+ i delta))
               (l (FUTURE (do-branch ileft (quotient delta 2)))))
          (do-branch iright (quotient delta 2))
          (TOUCH l)
          (do ((d 1 (* d 2)))
              ((> d delta))
            (reduce-equation i d))))))
```

```
(define (backsolve/par equ imid)
  (let loop ((i imid) (delta imid))
    (let ((equ-i (cvector-ref equ i)))
      (x-set! equ-i (quotient (- (y equ-i)
                                 (+ (* (a equ-i)
                                       (x (cvector-ref equ (- i delta))))
                                    (* (c equ-i)
                                       (x (cvector-ref equ (+ i delta))))))
                              (b equ-i)))
      (if (> delta 1)
          (let* ((new-delta (quotient delta 2))
                 (l (FUTURE (loop (- i new-delta) new-delta))))
            (loop (+ i new-delta) new-delta)
            (TOUCH l))))))

(define abcyx-chain #f)

(define (init-abcyx-chain n) ; make a chain of 5 element vects
  (set! abcyx-chain (make-vector-chain n 5)))

(define (make-abcyx*)
  (let ((node abcyx-chain))
    (set! abcyx-chain (vector-ref node 0))
    node))
```

```
(define n #f)
(define imid #f)
(define equ #f)

(define k 15)

(define (init1)
  (let ((n+1 (expt 2 k)))
    (set! n (- n+1 1))
    (set! imid (quotient n+1 2))
    (init-abcyx-chain (+ n 2))
    (set! equ (make-cvector (+ n 2) make-abcyx*))))

(define (init2)
  (do ((i (+ n 1) (- i 1)))
      ((< i 0))
    (let ((equ-i (cvector-ref equ i)))
      (a-set! equ-i 1)
      (b-set! equ-i 1)
      (c-set! equ-i 1)
      (y-set! equ-i 3)
      (x-set! equ-i 0)))
  (let ((equ-1 (cvector-ref equ 1)))
    (a-set! equ-1 0)
    (b-set! equ-1 1)
    (c-set! equ-1 1)
    (y-set! equ-1 2))
  (let ((equ-n (cvector-ref equ n)))
    (a-set! equ-n 1)
    (b-set! equ-n 1)
    (c-set! equ-n 0)
    (y-set! equ-n 2)))

(define (run)
  (reduce/par equ imid)
  (backsolve/par equ imid))

(benchmark TRIDIAG (begin (init1) (init2)) (run))
```

Appendix B

Execution Profiles for Parallel Benchmarks

This Appendix contains "execution profiles" for each of the parallel benchmarks of Appendix A. An execution profile is a plot representing the activity of the processors as a function of time. Profiles are useful to visualize the behavior of parallel programs. They are also an invaluable tool to detect performance related problems with algorithms and the language implementation.

To generate the profiles, the programs were compiled with the default polling settings with an intermittency factor of 10. The message-passing protocol supporting the Katz-Weise continuation semantics and legitimacy was used but fairness was disabled. The programs were run on the GP1000 with 64 processors. Processors can be in one of six distinctive states in the message-passing protocol

1. **Interrupt** — The processor is servicing a steal request. This state accounts for heapifying the parent continuation, creating the task, the result and legitimacy placeholders, and responding to the thief.
2. **Working** — The processor is running the main body of the program (i.e. "user code"). This accounts not only for all the work that is strictly required by a sequential version of a program, but also includes the following extra work needed to support parallelism: pushing and popping lazy tasks, checking for placeholders (as part of `TOUCH`), waiting for references to remote memory and restoring continuations[1].

[1] Measuring all these cases independently would be useful; unfortunately, it is impossible to do in an

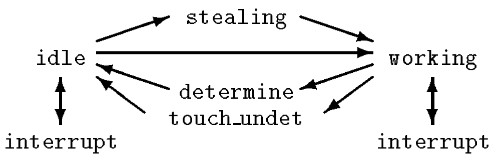
3. **Idle** — The processor is looking for work but hasn't yet found an available task in a work queue or a victim processor to interrupt.

4. **Touching an undetermined placeholder** — An undetermined placeholder was touched. This state indicates the suspension of a task.

5. **Determine** — A placeholder is being determined prior to the termination of a task.

6. **Stealing** — The processor has found a victim processor, sent a steal request and is waiting for a response. The cost of restarting the task is also included except for restoring the task's continuation.

Only certain transitions between these states are possible, as defined by the following diagram



Note that it is possible to go directly from the idle state to the working state. This happens when a task is taken from a processor's HTQ. Also, note that interrupts can only be serviced in the idle state and in the working state.

For the profiles to be significant, it is important to minimize the impact of monitoring on the behavior of the system. The profiles were obtained by having each processor log an event in a table in local memory whenever there was a state transition. The extra code needed to do this is confined to the runtime system, user code is not changed in any way. Each event indicates the state being entered and the current time taken from a real time clock with a 62.5 μsecs resolution. These tables were then dumped to disk for later processing by the analysis program generating the profiles. The cost of logging an event in this way is about 6 μsecs. This is relatively small compared to the typical duration of states (usually much more that 100 μsecs).

A profile is divided into three sections. The top part displays the instantaneous activity of the machine. That is, what proportion of all the processors are in each state as a function of time (time is always expressed in milliseconds). Below this is the

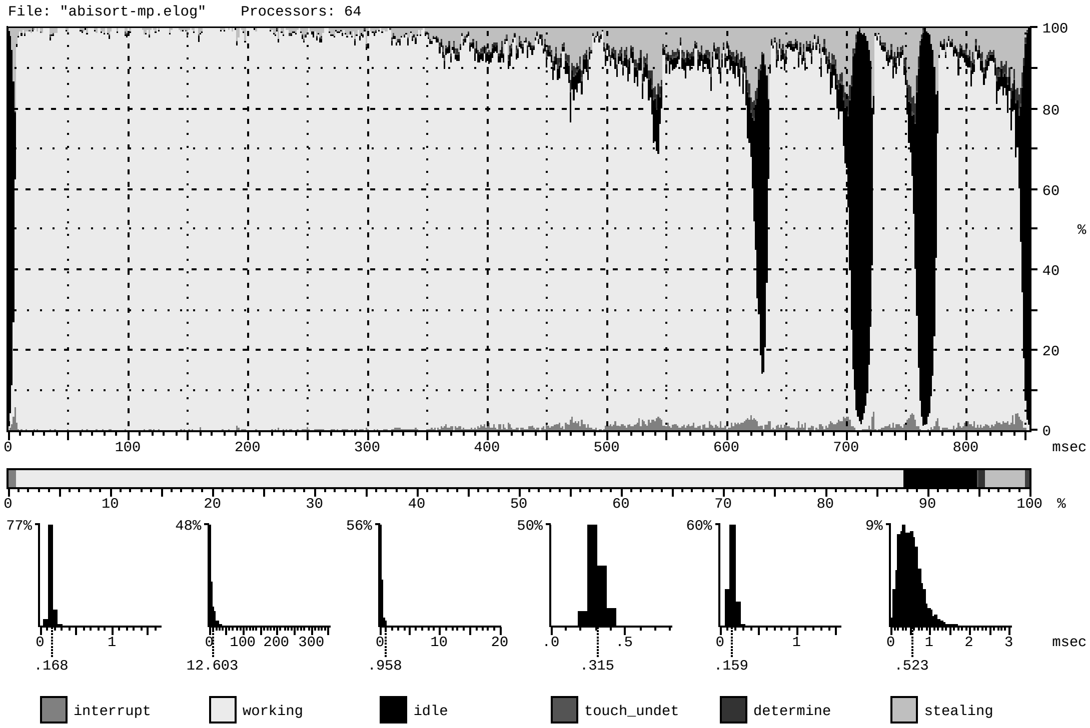
unintrusive way. This is why all these different cases were grouped together in one state. Time spent in the "working" state can only serve as an approximation of the work required by a sequential version of the program.

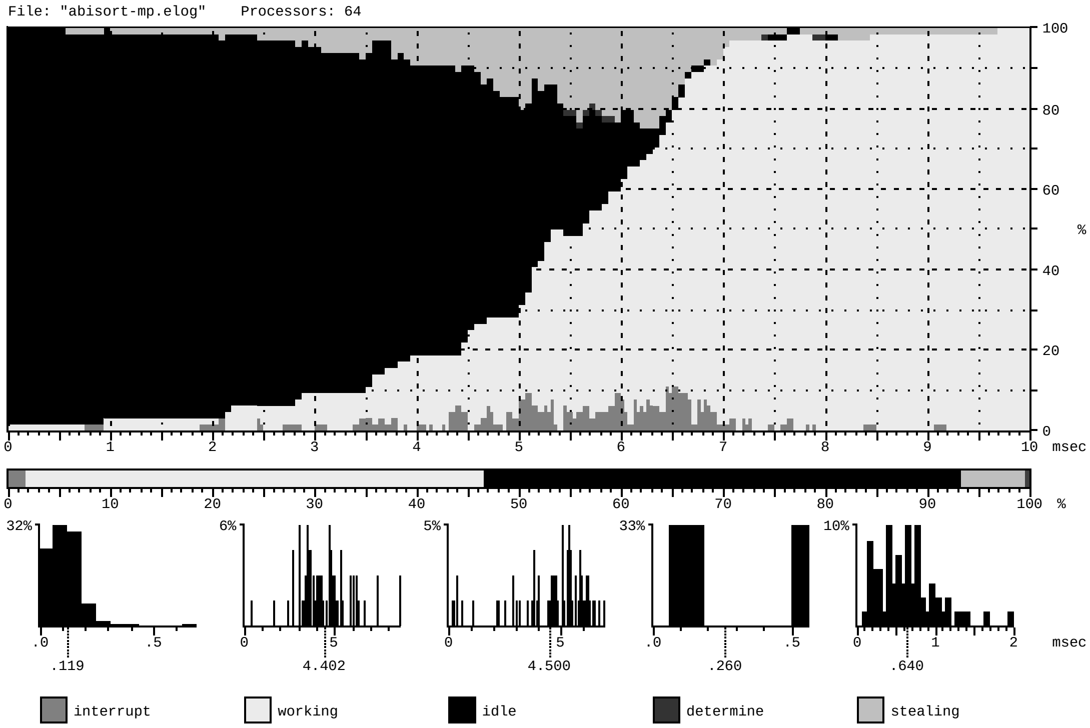
global activity chart. It indicates what percentage of the run time is spent in each of the states (in other words it gives the area covered by each state in the instantaneous activity chart). The bottom section consists of state duration histograms for every state. Each histogram indicates the distribution of state durations and also the average duration[2]. Note that each state is represented by a different shade of gray. To help distinguish the shades, the states are always in the same order; from bottom to top in the instantaneous activity chart and from left to right in the global activity chart.

For each benchmark two profiles are given. The first is for the complete run and the second is a close-up of the beginning of the run.

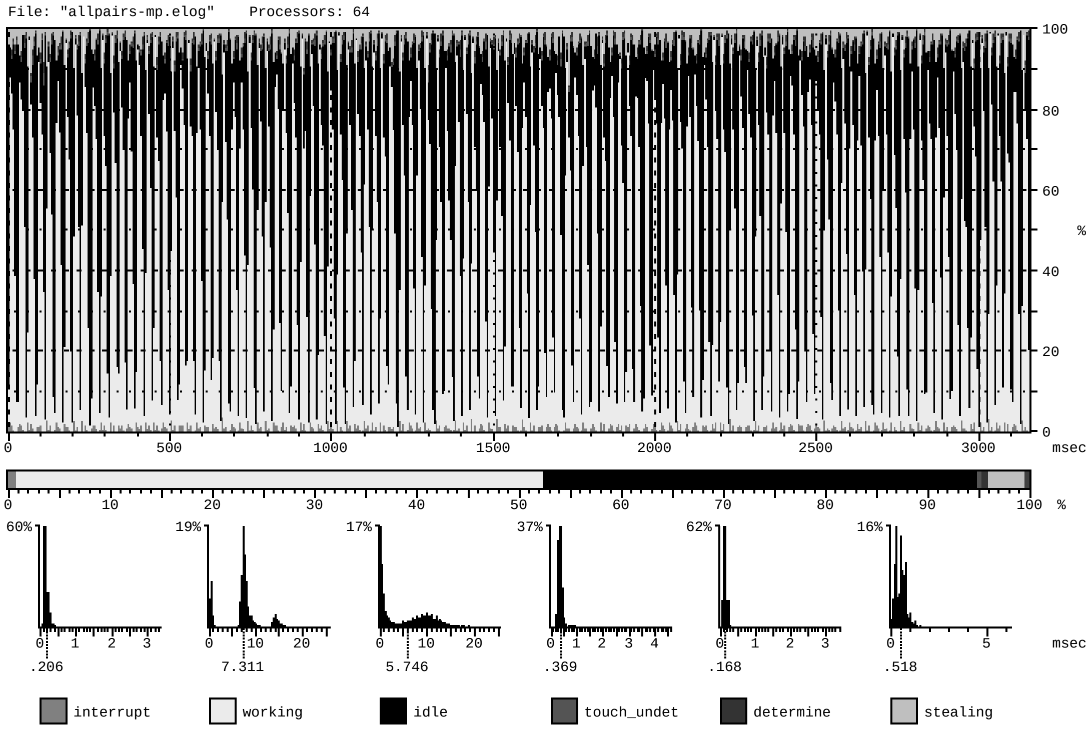
[2]The time spent servicing interrupts is ignored to compute the duration of the working and idle states.

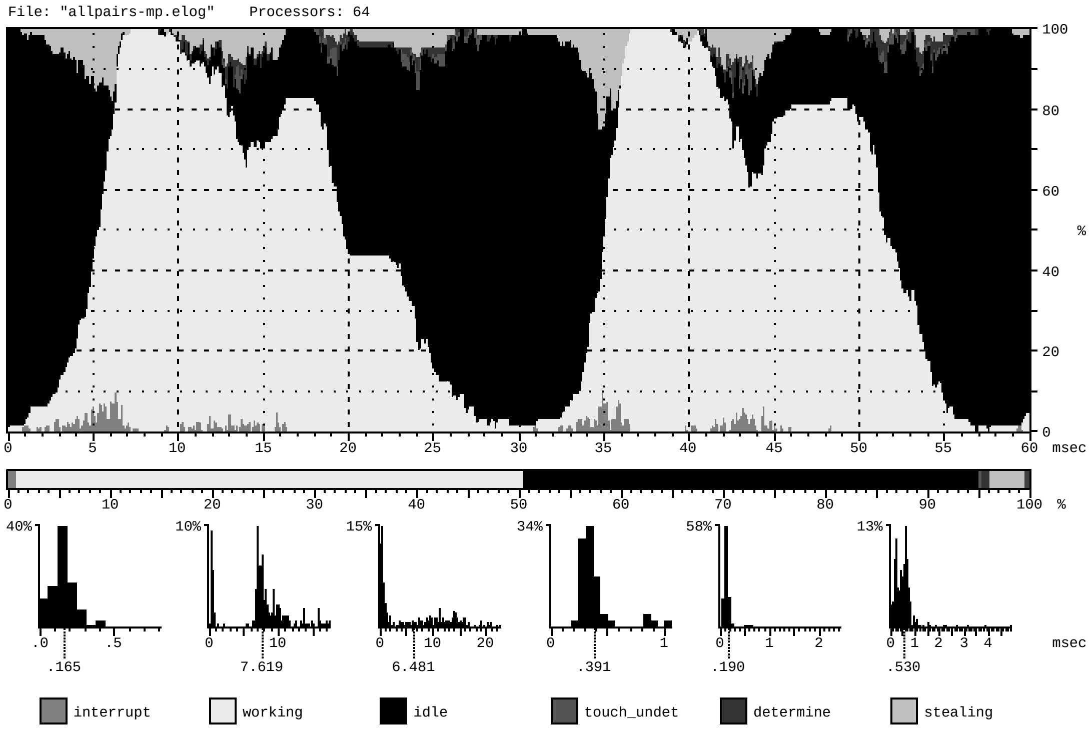
B.1 abisort


File: "abisort-mp.elog" Processors: 64
100
80
60
%
40
20
0
0
100
200
300
400
500
600
700
800
msec
0
10
20
30
40
50
60
70
80
90
100 %
77%
48%
56%
50%
60%
9%
.168
12.603
.958
.315
.159
.523
msec
interrupt
working
idle
touch_undet
determine
stealing

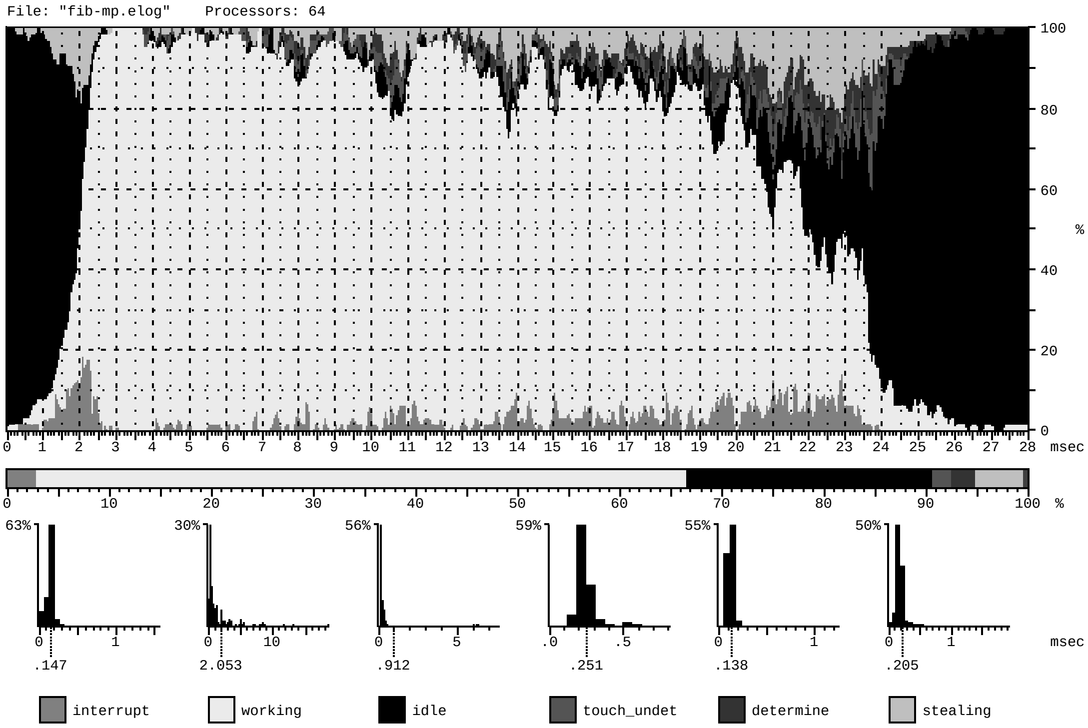

File: "abisort-mp.elog" Processors: 64
100
80
60
%
40
20
0
0
1
2
3
4
5
6
7
8
9
10 msec
0
10
20
30
40
50
60
70
80
90
100 %
32%
6%
5%
33%
10%
.119
4.402
4.500
.260
.640
msec
interrupt
working
idle
determine
stealing

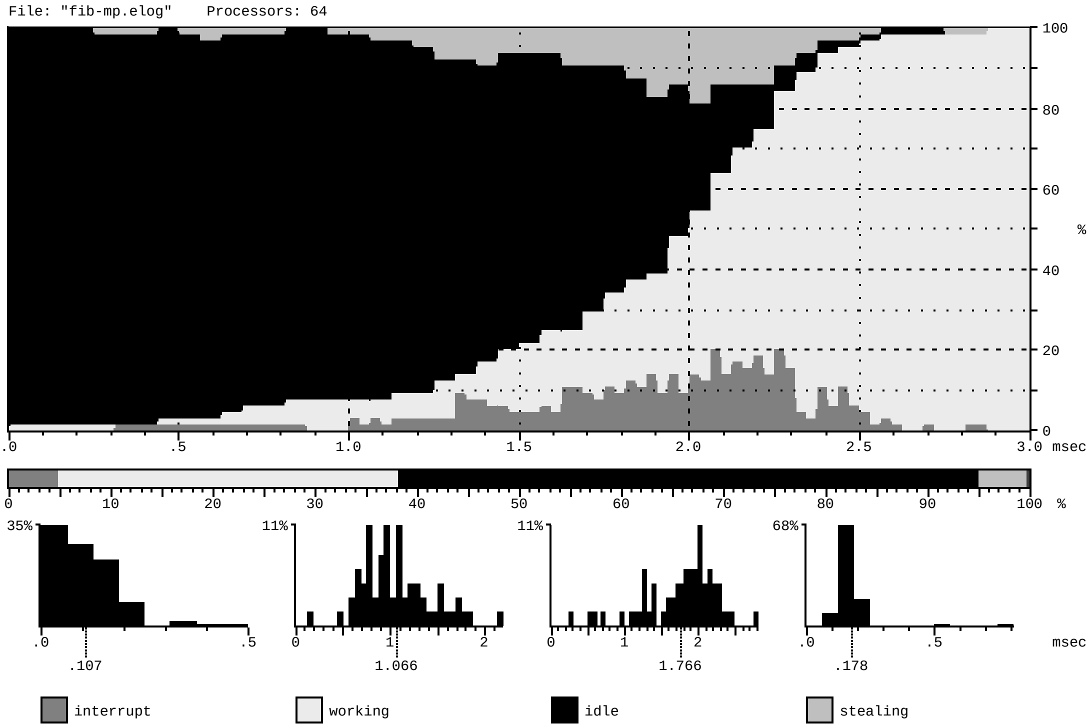
B.2 allpairs


File: "allpairs-mp.elog" Processors: 64
100
80
60
%
40
20
0
0
500
1000
1500
2000
2500
3000
msec
0 10 20 30 40 50 60 70 80 90 100 %
60%
.206
19%
7.311
17%
5.746
37%
.369
62%
.168
16%
.518
msec
interrupt
working
idle
touch_undet
determine
stealing

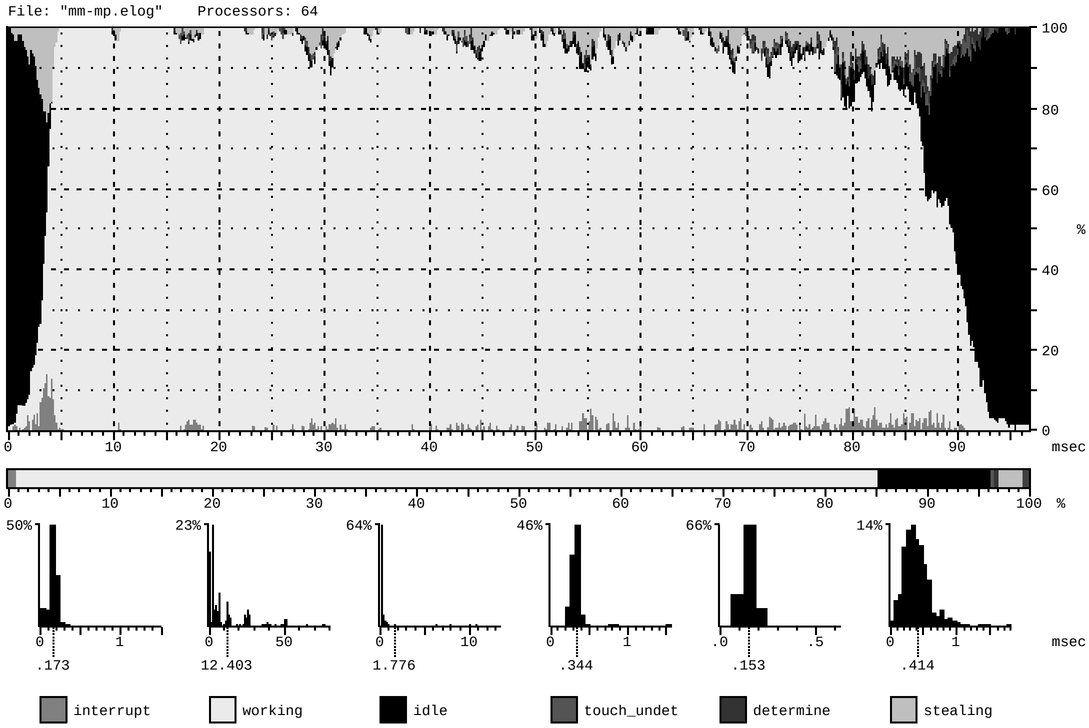

File: "allpairs-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 5 10 15 20 25 30 35 40 45 50 55 60 msec
0 10 20 30 40 50 60 70 80 90 100 %
40%
.165
10%
7.619
15%
6.481
34%
.391
58%
.190
13%
.530
msec
interrupt
working
idle
touch_undet
determine
stealing

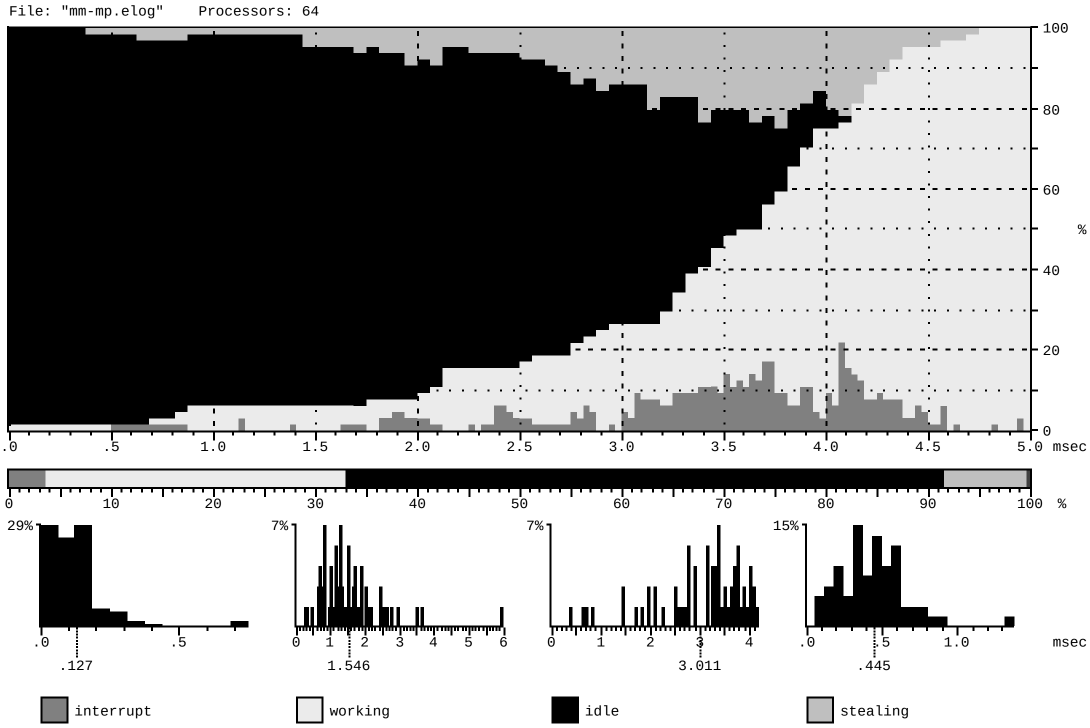
B.3 fib


File: "fib-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 msec
0 10 20 30 40 50 60 70 80 90 100 %
63% 30% 56% 59% 55% 50%
0 1 0 10 0 5 .0 .5 0 1 0 1 msec
.147 2.053 .912 .251 .138 .205
interrupt
working
idle
touch_undet
determine
stealing

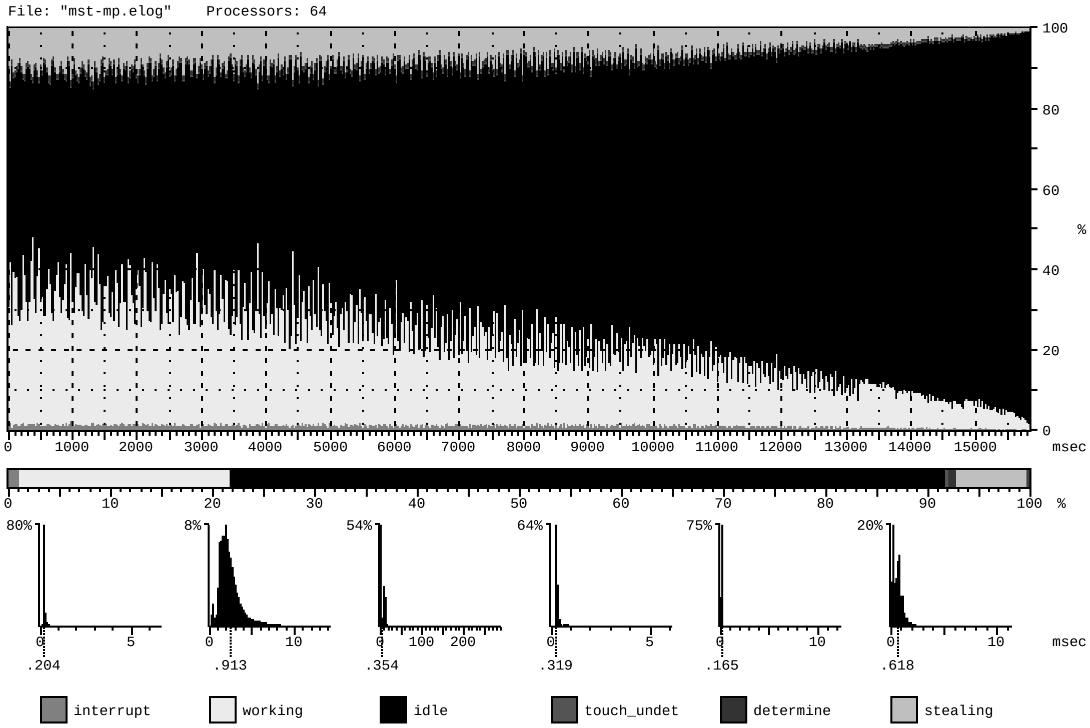

File: "fib-mp.elog" Processors: 64
100
80
60
%
40
20
0
.0 .5 1.0 1.5 2.0 2.5 3.0 msec
0 10 20 30 40 50 60 70 80 90 100 %
35% 11% 11% 68%
.0 .5 0 1 2 0 1 2 .0 .5 msec
.107 1.066 1.766 .178
interrupt
working
idle
stealing

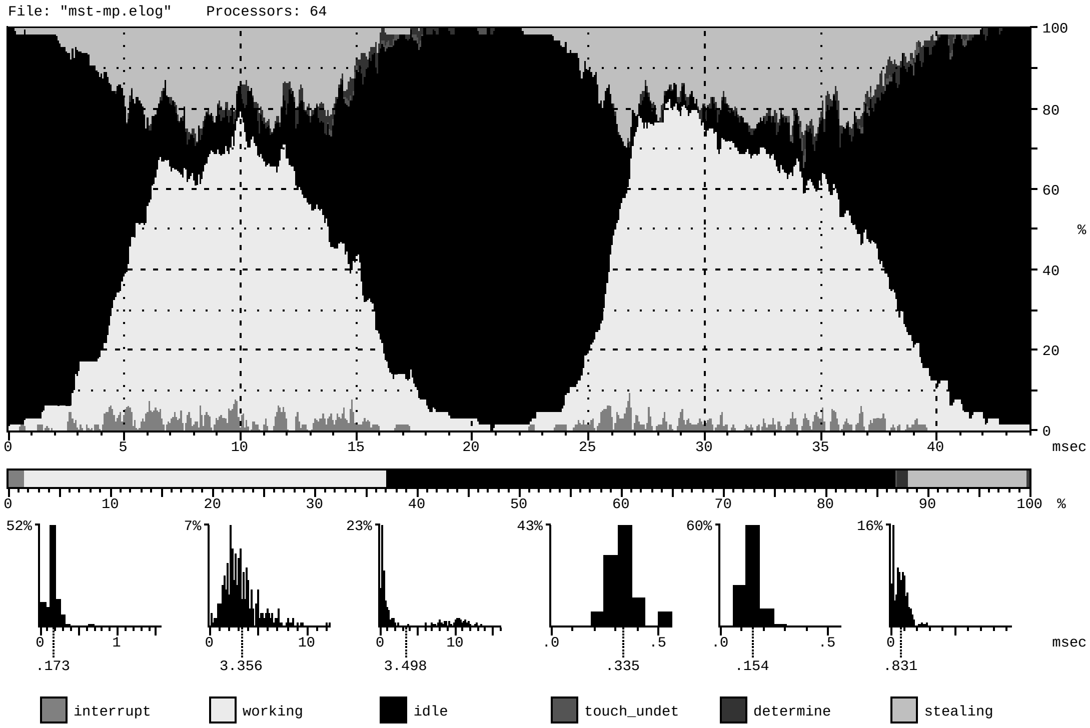
B.4 mm


File: "mm-mp.elog" Processors: 64
msec
%
.173
12.403
1.776
.344
.153
.414
interrupt
working
idle
touch_undet
determine
stealing

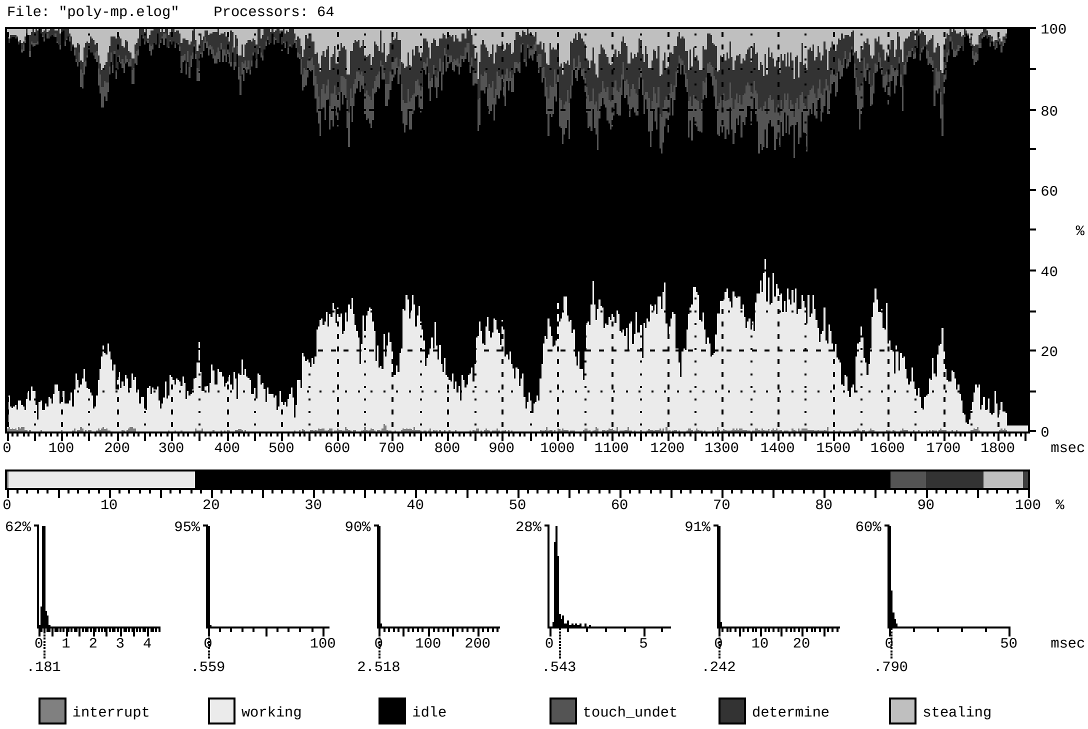

File: "mm-mp.elog" Processors: 64
msec
%
.127
1.546
3.011
.445
interrupt
working
idle
stealing

B.5 mst

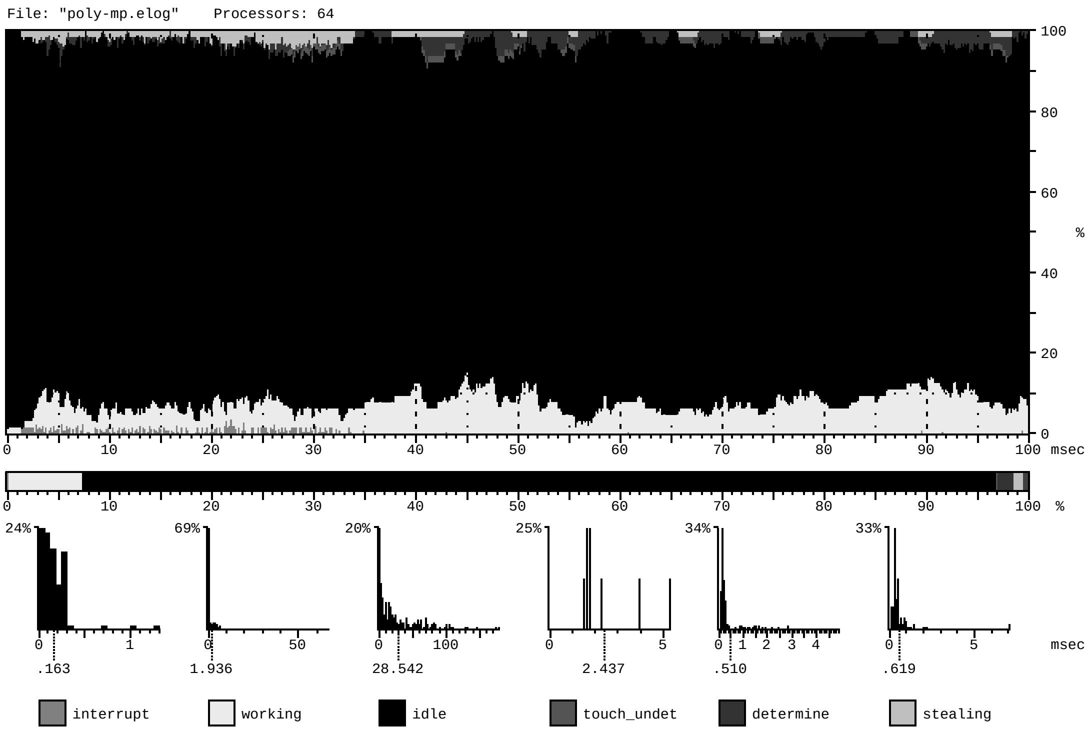

File: "mst-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000
msec
0 10 20 30 40 50 60 70 80 90 100 %
80%
8%
54%
64%
75%
20%
0 5
0 10
0 100 200
0 5
0 10
0 10
msec
.204
.913
.354
.319
.165
.618
interrupt
working
idle
touch_undet
determine
stealing


File: "mst-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 5 10 15 20 25 30 35 40
msec
0 10 20 30 40 50 60 70 80 90 100 %
52%
7%
23%
43%
60%
16%
0 1
0 10
0 10
.0 .5
.0 .5
0
msec
.173
3.356
3.498
.335
.154
.831
interrupt
working
idle
touch_undet
determine
stealing

B.6 poly

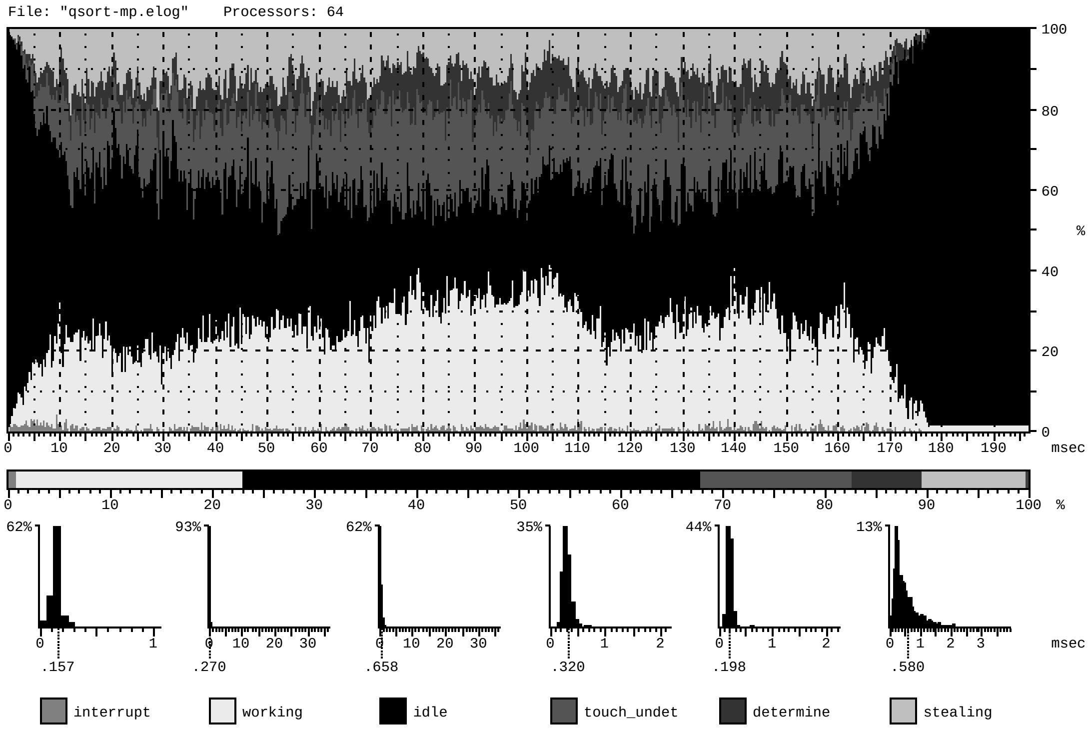


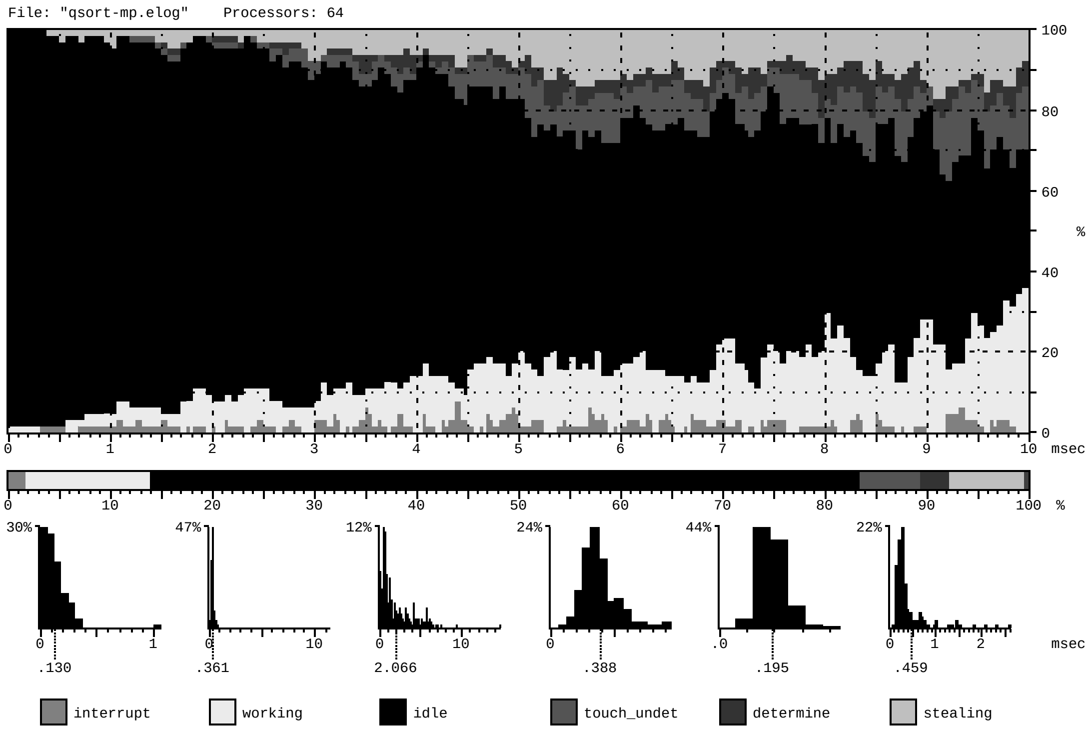
File: "poly-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 1600 1700 1800 msec
0 10 20 30 40 50 60 70 80 90 100 %
62%
95%
90%
28%
91%
60%
msec
.181
.559
2.518
.543
.242
.790
interrupt
working
idle
touch_undet
determine
stealing



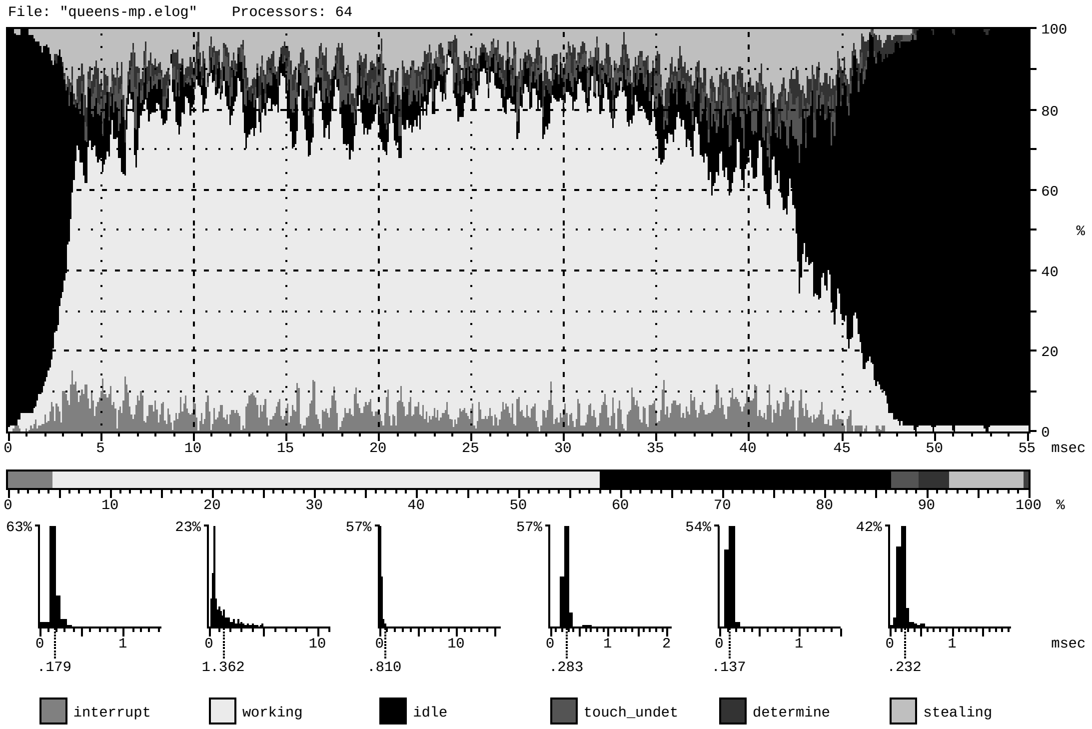
File: "poly-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 10 20 30 40 50 60 70 80 90 100 msec
0 10 20 30 40 50 60 70 80 90 100 %
24%
69%
20%
25%
34%
33%
msec
.163
1.936
28.542
2.437
.510
.619
interrupt
working
idle
touch_undet
determine
stealing

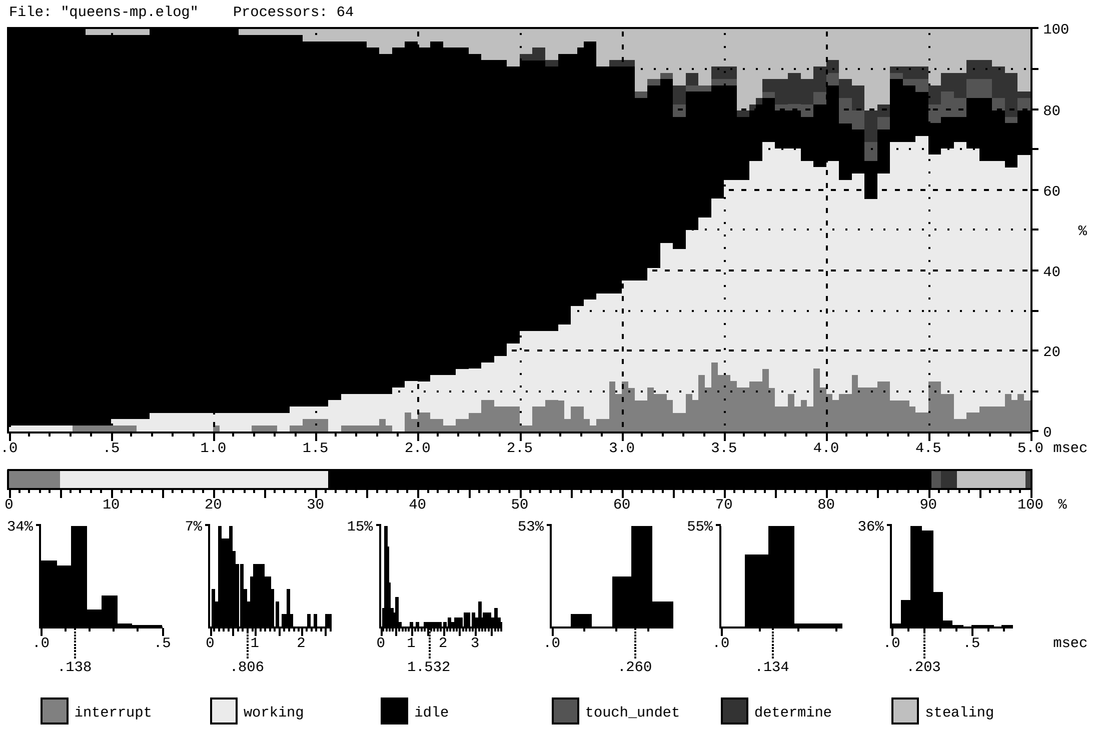
B.7 qsort


File: "qsort-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 msec
0 10 20 30 40 50 60 70 80 90 100 %
62%
93%
62%
35%
44%
13%
msec
.157
.270
.658
.320
.198
.580
interrupt
working
idle
touch_undet
determine
stealing

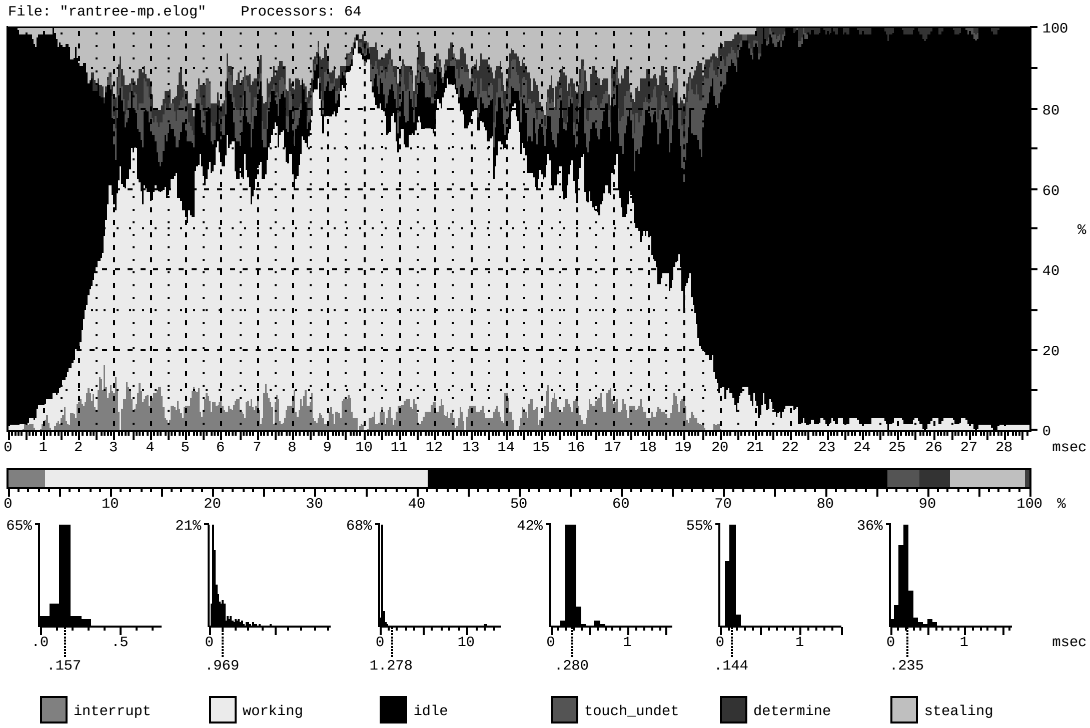

File: "qsort-mp.elog" Processors: 64
100
80
60
%
40
20
0
0 1 2 3 4 5 6 7 8 9 10 msec
0 10 20 30 40 50 60 70 80 90 100 %
30%
47%
12%
24%
44%
22%
msec
.130
.361
2.066
.388
.195
.459
interrupt
working
idle
touch_undet
determine
stealing

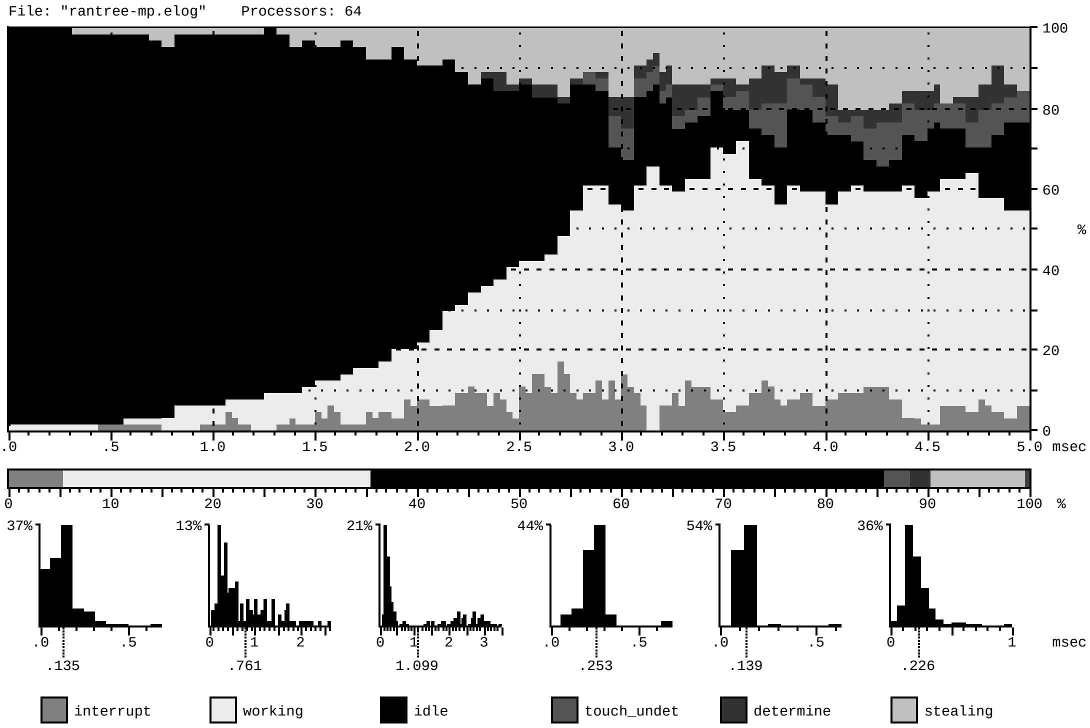
B.8 queens


File: "queens-mp.elog" Processors: 64
msec
%
interrupt
working
idle
touch_undet
determine
stealing

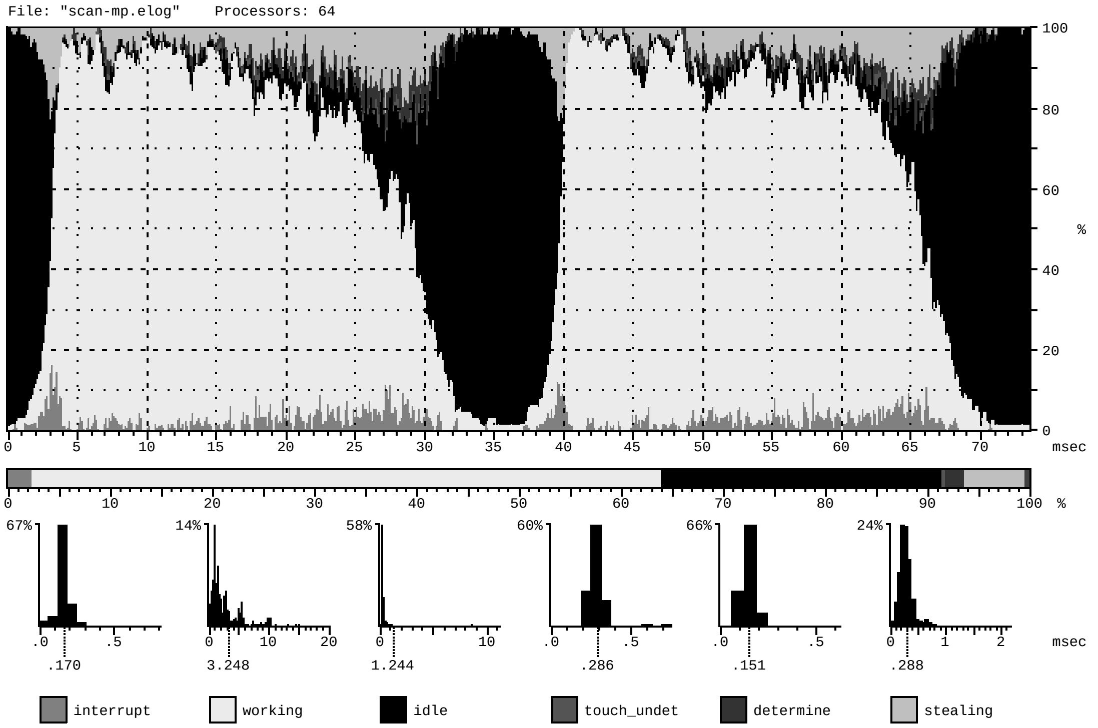

File: "queens-mp.elog" Processors: 64
msec
%
interrupt
working
idle
touch_undet
determine
stealing

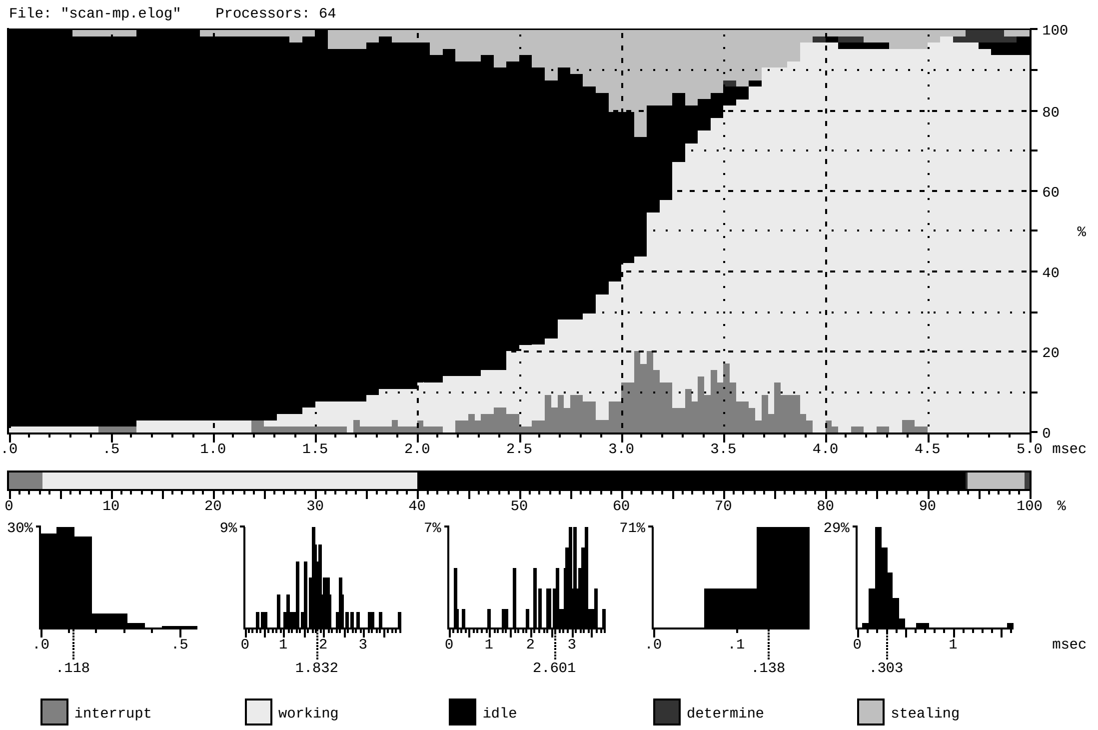
B.9 rantree


File: "rantree-mp.elog" Processors: 64
msec
%
65%
21%
68%
42%
55%
36%
.157
.969
1.278
.280
.144
.235
interrupt
working
idle
touch_undet
determine
stealing

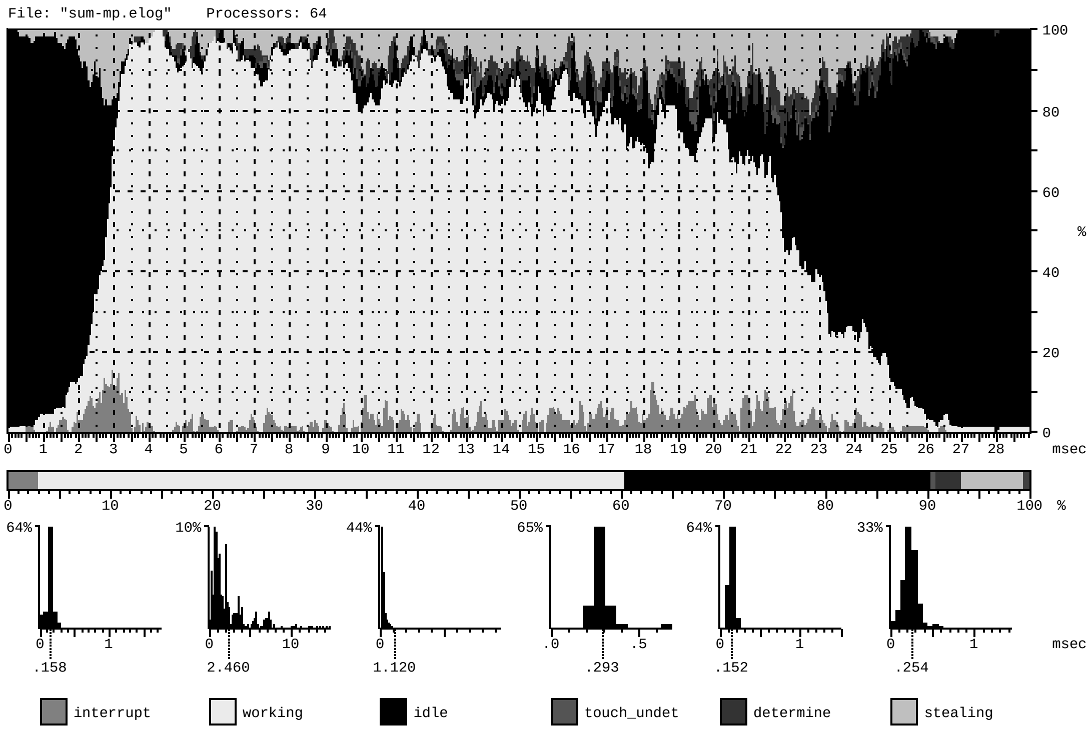

File: "rantree-mp.elog" Processors: 64
msec
%
37%
13%
21%
44%
54%
36%
.135
.761
1.099
.253
.139
.226
interrupt
working
idle
touch_undet
determine
stealing

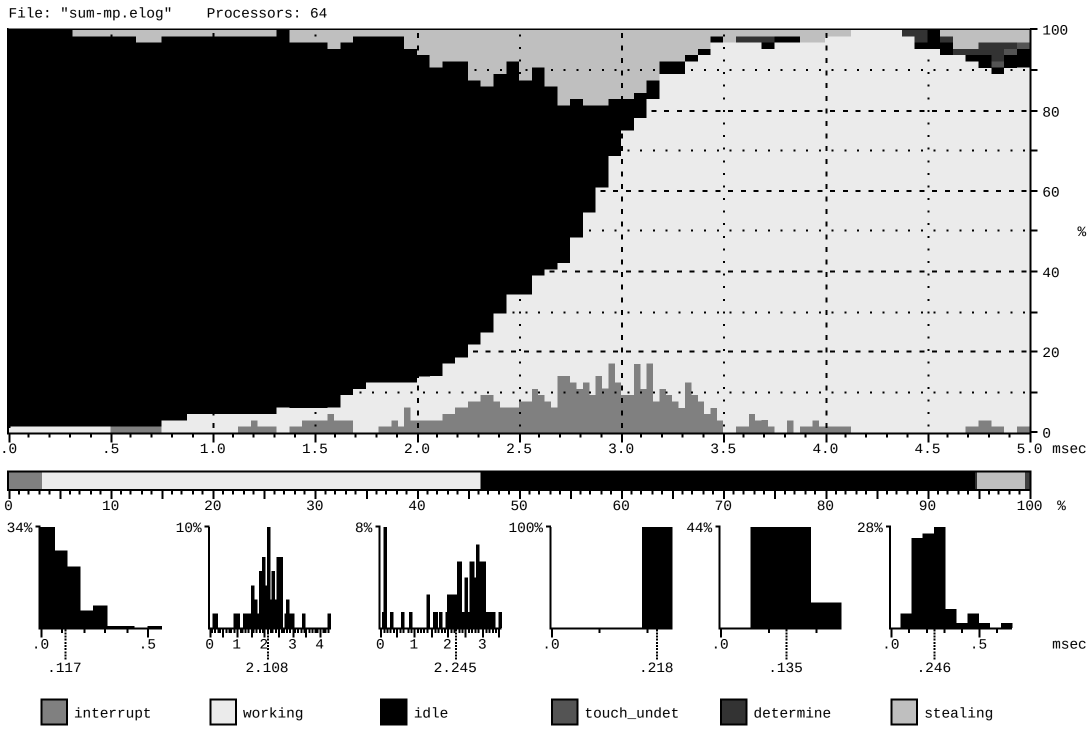
B.10 scan


File: "scan-mp.elog" Processors: 64
100
80
60
%
40
20
0
msec
67%
14%
58%
60%
66%
24%
.170
3.248
1.244
.286
.151
.288
interrupt
working
idle
touch_undet
determine
stealing

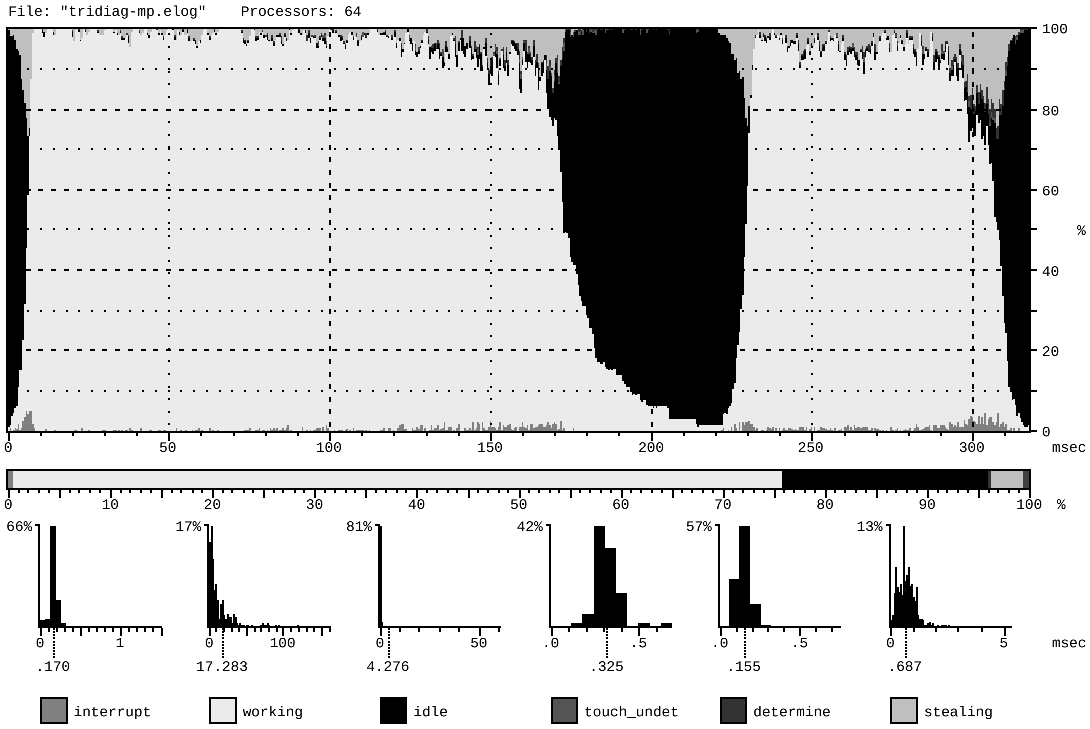

File: "scan-mp.elog" Processors: 64
100
80
60
%
40
20
0
msec
30%
9%
7%
71%
29%
.118
1.832
2.601
.138
.303
interrupt
working
idle
determine
stealing

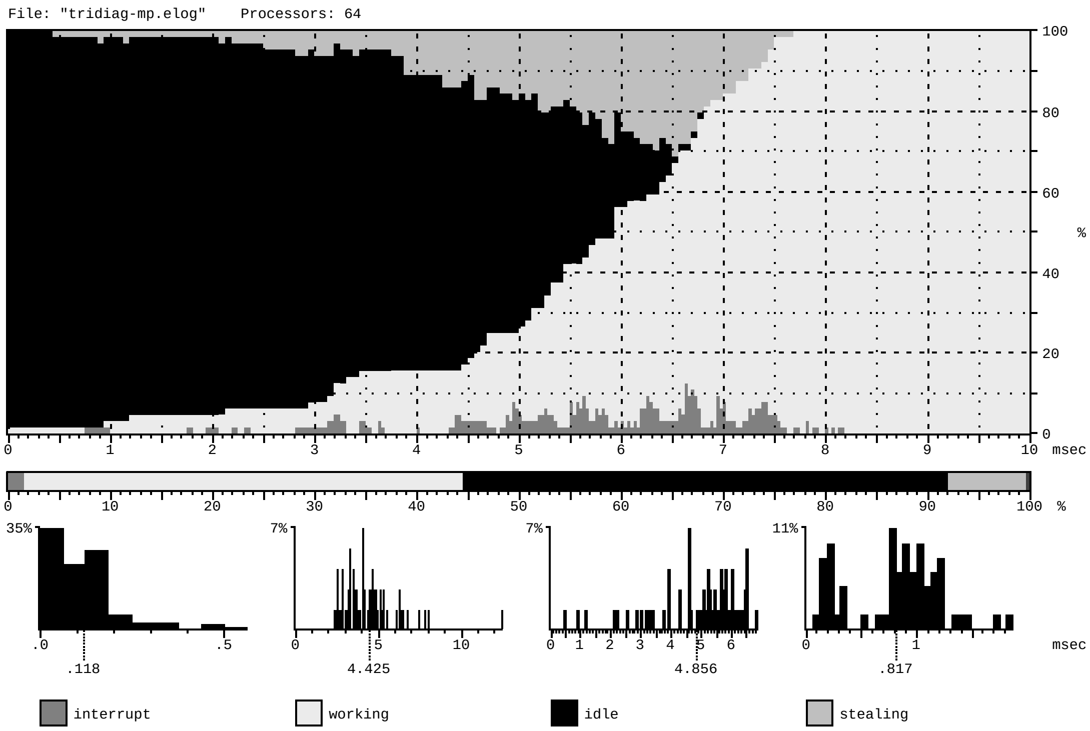
B.11 sum


File: "sum-mp.elog" Processors: 64
msec
%
.158
2.460
1.120
.293
.152
.254
interrupt
working
idle
touch_undet
determine
stealing


File: "sum-mp.elog" Processors: 64
msec
%
.117
2.108
2.245
.218
.135
.246
interrupt
working
idle
touch_undet
determine
stealing

B.12 tridiag


File: "tridiag-mp.elog" Processors: 64
msec
%
.170
17.283
4.276
.325
.155
.687
interrupt
working
idle
touch_undet
determine
stealing


File: "tridiag-mp.elog" Processors: 64
msec
%
.118
4.425
4.856
.817
interrupt
working
idle
stealing

Bibliography

[Adams and Rees, 1988] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, August 1988.

[Agarwal, 1991] A. Agarwal. Performance tradeoffs in multithreaded processors. Technical Report MIT/LCS/TR-501, Massachusetts Institute of Technology, Cambridge, MA, April 1991.

[Appel, 1989] A. W. Appel. Allocation without locking. *Software Practice and Experience*, 19(7):703–705, July 1989.

[Arvind and Nikhil, 1990] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[Baker and Hewitt, 1978] H. Baker and C. Hewitt. The incremental garbage collection of processes. Technical Report AI Memo 454, Mass. Inst. of Technology, Artificial Intelligence Laboratory, March 1978.

[BBN, 1989] BBN Advanced Computers Inc., Cambridge, MA. *Inside the GP1000*, 1989.

[BBN, 1990] BBN Advanced Computers Inc., Cambridge, MA. *Inside the TC2000 Computer*, 1990.

[Bilardi and Nicolau, 1989] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing*, 12(2):216–228, April 1989.

[Callahan and Smith, 1989] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Papers from the Second Workshop* *on Languages and Compilers for Parallel Computing*, pages 95–113. University of Illinois at Urbana-Champaign, 1989.

[Censier and Feautrier, 1978] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, pages 1112–1118, December 1978.

[Chaiken *et al.*, 1991] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *ASPLOS IV: Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 1991.

[Clinger *et al.*, 1988] W. Clinger, A. Hartheimer, and E. Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, Snowbird, UT., July 1988.

[Clinger, 1984] W. Clinger. The Scheme 311 compiler: an exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, 1984.

[Dijkstra, 1968] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[Dubois and Scheurich, 1990] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[Feeley and Miller, 1990] M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[Feeley, 1993] M. Feeley. Polling efficiently on stock hardware. In *Proceedings of the 1993 ACM Conference on Functional Programming Languages and Computer Architecture*, 1993.

[Fra, 1990] Franz Inc., Berkeley, CA. *Allegro CL User Manual*, 1990.

[Friedman and Haynes, 1985] D. P. Friedman and C. T. Haynes. Constraining control. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 245–254, New Orleans, LA., January 1985. ACM.

[Friedman *et al.*, 1992] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

[Gabriel and McCarthy, 1984] R. P. Gabriel and J. McCarthy. Queue-based multiprocessing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 25–44, Austin, TX., August 1984.

[Gabriel, 1985] R. P. Gabriel. *Performance and Evaluation of Lisp Systems*. Research Reports and Notes, Computer Systems Series. MIT Press, Cambridge, MA, 1985.

[Gharachorloo *et al.*, 1991] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257. ACM, April 1991.

[Goldman and Gabriel, 1988] R. Goldman and R. P. Gabriel. Preliminary results with the initial implementation of Qlisp. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 143–152, Snowbird, UT., July 1988.

[Goodman, 1983] J. R. Goodman. Using cache memory to reduce processor-memory traffic. *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, June 1983.

[Gray, 1986] S. L. Gray. Using futures to exploit parallelism in Lisp. Master's thesis, Mass. Inst. of Technology, 1986.

[Halstead and Fujita, 1988] R. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, 1988.

[Halstead *et al.*, 1986] R. Halstead, T. Anderson, R. Osborne, and T. Sterling. Concert: Design of a multiprocessor development system. In *Int'l. Symp. on Computer Architecture*, volume 13, pages 40–48, June 1986.

[Halstead, 1984] R. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, TX., August 1984.

[Halstead, 1985] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[Halstead, 1987] R. Halstead. Overview of concert Multilisp: A multiprocessor symbolic computing system. *ACM Computer Architecture News*, 15(1):5–14, March 1987.

[Haynes *et al.*, 1984] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, TX., 1984.

[Haynes, 1986] Christopher T. Haynes. Logic continuations. In *Proceedings of the Third International Conference on Logic Programming*, pages 671–685. Springer-Verlag, July 1986.

[Hieb *et al.*, 1990] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[Hockney and Jesshope, 1988] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, Bristol and Philadelphia, 1988.

[IEEE Std 1178-1990, 1991] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[Ito and Matsui, 1990] T. Ito and M. Matsui. A parallel Lisp language PaiLisp and its kernel specification. In *Parallel Lisp: Languages and Systems*, pages 58–100. Springer-Verlag, 1990.

[Katz and Weise, 1990] M. Katz and D. Weise. Continuing into the future: on the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[Kessler and Swanson, 1990] R. Kessler and M. Swanson. Concurrent Scheme. In *Parallel Lisp: Languages and Systems*, pages 200–234. Springer-Verlag, 1990.

[Kessler *et al.*, 1992] R. Kessler, H. Carr, L. Stroller, and M. Swanson. Implementing concurrent Scheme for the Mayfly distributed parallel processing system. *Lisp and Symbolic Computation: An International Journal*, 5(1/2):73–93, 1992.

[Kranz *et al.*, 1989] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 81–90, June 1989.

[LeBlanc and Markatos, 1992] T. J. LeBlanc and E. P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. Technical report, University of Rochester, April 1992.

[Lenoski *et al.*, 1992] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[Miller, 1987] J. S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Mass. Inst. of Technology, August 1987. Available as MIT LCS/TR/402.

[Miller, 1988] J. S. Miller. Implementing a Scheme-based parallel processing system. *International Journal of Parallel Processing*, 17(5), October 1988.

[Mohr, 1991] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Yale University Department of Computer Science, October 1991.

[Mou, 1990] Z. G. Mou. A formal model of divide-and-conquer and its parallel realization. Computer science research report #795 (PhD dissertation), Yale University, 1990.

[Murray, 1990] K. Murray. The future of Common Lisp: Higher performance through parallelism. In *The first European Conference on the Practical Application of Lisp*, Cambridge, UK, March 1990.

[Nikhil *et al.*, 1991] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. Technical Report Computations Structures Group Memo 325–1, Mass. Inst. of Technology, Laboratory for Computer Science, Cambridge, MA, November 1991.

[O'Krafka and Newton, 1990] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 138–147. ACM, May 1990.

[Osborne, 1989] R. Osborne. *Speculative Computation in Multilisp.* PhD thesis, Mass. Inst. of Technology, 1989. Available as MIT LCS/TR/464.

[Peterson, 1981] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[Pfister *et al.*, 1985] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. *International Conference on Parallel Processing*, pages 764–771, 1985.

[R3RS, 1986] Revised^3 report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21(12), December 1986.

[R4RS, 1991] Revised[4] report on the algorithmic language Scheme. Technical Report MIT AI Memo 848b, Mass. Inst. of Technology, Cambridge, Mass., November 1991.

[Rettberg *et al.*, 1990] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The Monarch parallel processor hardware design. *IEEE Computer*, 23(4):18–30, April 1990.

[Rozas and Miller, 1991] G. Rozas and J. S. Miller. Free variables and first-class environments. *Lisp and Symbolic Computation: An International Journal*, 3(4):107–141, 1991.

[Rozas, 1987] G. Rozas. A computational model for observation in quantum mechanics. Master's thesis, Mass. Inst. of Technology, 1987. Available as MIT AI/TR/925.

[Shivers, 1988] O. Shivers. Control flow analysis in Scheme. In *ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, June 1988.

[Shivers, 1991] O. Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation.* The MIT Press, Cambridge, Mass., 1991.

[Srini, 1986] V. P. Srini. An architectural comparison of dataflow systems. *IEEE Computer*, 19(3):68–88, March 1986.

[Steele, 1978] G. L. Steele. Rabbit: a compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[Steinberg *et al.*, 1986] S. Steinberg, D. Allen, L. Bagnall, and C. Scott. The Butterfly Lisp system. In *Proc. 1986 AAAI*, volume 2, Philadelphia, PA, August 1986.

[Swanson *et al.*, 1988] M. Swanson, R. Kessler, and G. Lindstrom. An implementation of portable standard Lisp on the BBN Butterfly. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 132–141, Snowbird, UT., July 1988.

[Wand, 1980] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[Weening, 1989] J. S. Weening. *Parallel Execution of Lisp Programs.* PhD thesis, Stanford University, Department of Computer Science, 1989. Available as STAN-CS-89-1265.

[Zorn *et al.*, 1988] B. Zorn, P. Hilfinger, K. Ho, J. Larus, and L. Semenzato. Features for multiprocessing in SPUR Lisp. Technical Report Report UCB/CSD 88/406, University of California, Computer Science Division (EECS), March 1988.