Integrating Compile-time and Runtime Parallelism Management through Revocable Thread Serialization

by

Gino K. Maa

B.S., California Institute of Technology (1984)
S.M., Massachusetts Institute of Technology (1988)

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

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author .... [illegible]
Department of Electrical Engineering and Computer Science
January 31, 1995

Certified by .... [illegible]
Anant Agarwal
Associate Professor, EECS
Thesis Supervisor

Accepted by .... [illegible]
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Integrating Compile-time and Runtime Parallelism Management through Revocable Thread Serialization

by

Gino K. Maa

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Efficient decomposition of program and data on a scalable MIMD processor is necessary in order to minimize communication and synchronization costs while preserving sufficient parallelism to balance the workload of the processors. Programmer management of these runtime aspects of computation often proves to be very tedious and error-prone, and produces non-scalable, non-portable code. Previous efforts to apply compiler technology have assumed static timing of runtime operations, but real MIMD processors operate with much asynchrony that is difficult for the compiler to anticipate. Also, real programming environments support features that prevent the compiler from having total knowledge of and control over the runtime conditions.

Most work recognizing the asynchronous behavior of MIMD machines produce many fine-grained tasks and rely on special hardware support for fast and cheap task creation, synchronization, and context switching. The runtime mechanisms prove to be still expensive and cannot adequately improve the performance of massively data-parallel computations, whose dominant communication and synchronization overhead occurs at the data array references. Efficient decomposition of these programs requires information available from compiler analysis of the source program.

We explore a framework for integrating compile-time and runtime parallelism management, whose goal is to have the compiler produce independently-scheduled, asynchronous tasks that will cooperate flexibly and efficiently with a runtime manager to distribute work and allocate system resources. The compiler analyzes the source program to partition parallel loops and data structures, then aligns thread and data partitions from various loops and other parts of the program into groups called *detachments* to maximize data reference locality within each detachment. The threads in a detachment are then provisionally serialized to form a single-threaded *brigade*. In the nominal case each brigade is assigned to a processor and executed by a task to completion in the statically scheduled order; its coarse granularity thus saves much communication, synchronization, and task management overhead. Should runtime load balancing be required, program threads can be split off from a brigade at points prioritized by the compiler and executed by tasks on idle processors. Also, if the pre-scheduled thread ordering becomes incompatible with actual runtime dependencies, a deadlock condition is averted by initiating work on a thread split from the blocked task before returning to the blocked task. We expect an overall performance gain when the compiler chooses the approximately correct granularity and thread ordering, so these exceptional cases are in fact relatively rare. We implement this new framework and examine potential policies controlling its compiler choices, and compare its performance data with alternative parallelism management strategies.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor, EECS

Acknowledgments[†]

I am grateful to a host of people whose generous support and technical contributions made this work possible. I thank my advisor, Anant Agarwal, for running a first-rate research team with just the right balance of focus and latitude. This work would not have been possible without his enthusiasm and guidance. Anant's insistance that I shift my attention from all the fascinating details of an engineering perspective to higher abstractions fit for the audience's consumption has shaped the presentation indelibly. I thank my readers Arvind and Bill Weihl for their valuable comments and feedback on the thesis draft.

I am obliged to David Kranz for fixing and tailoring the Mul-T compiler for all of my specific requirements, to John Kubiatowicz for always ready to help resolve the many runtime and simulation problems expeditiously, and to the whole cast of Alewife Project members past and present for developing the essential systems and tools that this work relies on. I can hardly express my sentiment better than this stolen line:

So long, and thanks for all the fish!

I thank my office mate Jonathan Babb, who provided many interesting topics for conversation, took to my extemporaneous half-baked ideas, and generally kept me company through the long office hours of the last years. And thanks go to Anne McCarthy, who has quietly and proficiently shielded the myriad administrivia from my way.

I am forever indebted to my parents, my brother, and sister for their unconditional sacrifice, love, understanding, and support. I thank my dear friends John, Greg, Joe, Wison, Carmela, Tarn, Rosie, and David for their interest in my endeavors and their moral encouragement. Gregg generously provided the annual Manna from Santa Ana; his tantalizing job offers were, to his chagrin, ultimately ego boosting therby affirming of my graduate career choice.

[†]This research was funded by ARPA contracts #N00014-91-J-1698 and #N00014-94-1-0985, by NSF Experimental Systems grant #MIP-9012773, and by the NSF Presidential Young Investigator Award.

Contents

1 Introduction **15**

1.1 The Problem with Static Global Scheduling 17

1.2 The Problem with Runtime-managed Scheduling 20

1.3 Revocable Thread Serialization 21

1.4 Expected Results and the Performance Comparisons 24

1.5 Summary of Contributions 26

1.6 Related Work 27

1.7 Outline for the Thesis 29

2 Background **31**

2.1 Hardware Environment 32

2.2 Programming Environment 32

2.2.1 The Mul-T Programming Language 34

2.2.2 Mul-T's Runtime Task Manager 37

2.2.3 Toward an Efficient Future Implementation 38

2.2.4 Streamlining Future Implementation — The Lazy Future 40

2.2.5 Summary and a New View of Parallel Loops 46

3 Compile-time Program Transformations: An Example **49**

3.1 Structure of the Compilation Phases 49

3.2 Compiler Walk-through of an Example Program 51

3.2.1 Compiler Loop and Array Partitioning 53

3.2.2 Compiler Thread and Data Alignment 54

3.2.3 Compiler Thread Serialization 54

3.3 Summary 56

4 **Compile-time Program Transformations: Theory and Implementation** **59**

4.1 Partitioning of Parallel Loops and Data Arrays 60

4.1.1 Abstractions for Distributed Loops and Arrays 60

4.1.2 Distributed Iteration and Array Spaces 60

4.1.3 Application of the Inter-space Translations 61

4.1.4 Partitioning of Parallel Loops and Data Arrays 64

4.1.5 Partitioning without Sufficient Information 66

4.2 Aligning Partitions 66

4.2.1 Building the Partition Communications Graph 67

4.2.2 Computing Partition Alignment 69

4.2.3 Aligning Large Numbers of Partitions 70

4.3 Intra-detachment Thread Scheduling 71

4.3.1 Scheduling Independent Loop Iterations 73

4.3.2 Scheduling Partially-dependent Loop Iterations 77

4.3.3 Scheduling Random Parallel Threads 81

4.4 Policies for Loop and Array Partitioning 89

4.5 Summary 90

5 **Performance Characterization and Benchmark Programs** **91**

5.1 Performance of the Revocable Thread Serialization 92

5.2 Impact of Partition Alignment 97

5.3 Cost of Undoing Intractable Serialization 101

5.4 The SIMPLE Benchmark Program 102

5.4.1 Static Alignment of Loop/Array Partitions in SIMPLE 105

5.4.2 Speedup of SIMPLE 105

5.5 Summary 106

6 **Conclusion** **109**

6.1 Future Work 110

A **Program Analyses in Scepter** **113**

A.1 Static Scalar Alias and Dataflow Analyses 114

A.1.1 The Freevars 115

A.1.2 Constructing Alias Sets 116

A.1.3 Deriving Communication Edges 117

A.2 Dynamic Analysis 118
A.2.1 Annotating Dynamic Instances 118
A.2.2 Replicating Subgraphs of Dynamic Instances 121
A.3 Index Analysis 121
A.3.1 Splitting Data Nodes 122
A.4 Escaping References Analysis 122

List of Figures

1-1 *Cost to Stay Synchronized to a Static Global Schedule* 19

1-2 *Processor Profile for Parallel Matrix Multiply* 19

1-3 *Cache Misses of Processors for Parallel Matrix Multiply* 20

1-4 *Expected Tolerance to Dynamic Workload Variations* 24

1-5 *Expected Cost of Responding to Runtime Reordering* 26

2-1 *The Alewife Multiprocessor* 33

2-2 *Parallelism Generation through Futures* 36

2-3 *Dynamic Data Linkage of Future Tasks* 36

2-4 *Dynamic Schedule and Parallelism Profile of Future Tasks* 37

2-5 *Recursively Creating Futures for* `(Fib 5)` 43

2-6 *Runtime Behavior from the Lazy Future Tree of* `(Fib 5)` 44

2-7 *Runtime Unrolling of Parallel Loop Iterations* 46

2-8 *Unrolling of Parallel Loop Iterations with a Future Tree* 48

3-1 *Program Parallelism Restructuring and Compilation Process* 51

3-2 *Compilation Demonstration Program* 52

3-3 *Initial Thread Graph for Example Program* 53

3-4 *Partitioned Thread Graph for Example Program* 54

3-5 *Aligned Thread Graph for Example Program* 55

3-6 *Serialized Thread Graph for Example Program* 57

3-7 *Abstract Output of Transformed Example Program* 58

4-1 *Space Mappings and Translations* 62

4-2 *Partition Boundary-Crossing Array Accesses* 65

4-3 *The Loop and Data Partitions Communication Graph* 68

4-4 *Computing Communications between Partitions* 69

4-5 *Controlling the Loading Ratio of a Lazy Future Tree* 74

4-6 *Binary Lazy Future Fan-out Tree for $n \approx 0.6$* 74
4-7 *Execution Time Schedule for Computing the Loading Ratio* 75
4-8 *Optimum Loading Ratio* vs. *Amount of Work* 76
4-9 *Lazy Future Fan-out Tree Macro for Parallel Loop Iterations* 77
4-10 *Restructuring of Loop Iterations with Intervening Sequential Sections* 78
4-11 *Execution Schedule of a Partitioned Partially-dependent Loop* 79
4-12 *Macro for Unrolling Partitioned Partially-dependent Loop Iterations* 80
4-13 *Scenarios for Thread Scheduling* 84
4-14 *Proper Synchronization in Chaining Data-parallel Loops* 87

5-1 *Load-balancing Test Program with Variable Loop Iteration Length* 93
5-2 *Effects of Varying σ of Thread Size, $m = 200$* 95
5-3 *Effects of Varying σ of Thread Size, $m = 40$* 97
5-4 *Alignment Test Program* 98
5-5 *Effects of Loop/Array Partition Alignment* 100
5-6 *Runtime Serialization Revocation Test Loop* 101
5-7 *Incremental Cost of Revoking Compile-time Serialization* 102
5-8 *Block diagram of SIMPLE* 104
5-9 *Speedup Curve of SIMPLE (64×64)* 106

A-1 *The Alias Set of Variable v within the Alias-relation Lattice* 117
A-2 *Representing Instance–Indices Mapping in WAIF-CG* 123

List of Tables

2.1 *Timing of Selected Alewife Machine Operations* 34

2.2 *Performance of Eager and Lazy Futures* 44

5.1 *Taxonomy of Some Named Parallelism-management Strategies* 92

5.2 *Simulation Results: Varying σ of Thread Size, $m = 200$* 94

5.3 *Simulation Results: Varying σ of Thread Size, $m = 40$* 96

5.4 *Simulation Results: Loop/Array Partition Alignment* 99

5.5 *Loop and Call Structure in SIMPLE* 107

5.6 *Localizing Memory Accesses in the Parallel Loops of SIMPLE (64 $\times$ 64) through Alignment* 108

5.7 *Speedup of SIMPLE (64 $\times$ 64)* 108

Chapter 1

Introduction

Efficient decomposition of program and data on a scalable MIMD processor is necessary in order to minimize the communication and synchronization costs and to preserve sufficient parallelism to balance the workload of the processors. Relying on programmer management of these operational aspects of computation often proves to be very tedious and error-prone, and produces non-scalable, non-portable code. Providing a shared-memory programming abstraction spares programmers the onus of explicitly deciding where to locate data relative to the executing program and coding accesses to all non-local data, but it does not insure an efficient operation. Somehow someone must be mindful of these choices if optimized performance is expected. General-purpose architectural mechanisms and runtime support do not address this problem adequately. Straightforward borrowing of traditional hardware techniques, such as caching to decouple the memory performance from the processor, are not effective unless we can show good probability of reuse. Most methods to parallelize program execution reduce the amount of data reuse. Latency masking by multithreading incurs high expenses in the context switching overhead and increases the processor state manifold. Its use therefore must be carefully justified by some overall performance improvement.

Some previous parallel processor projects relied entirely on runtime load balancing and storage allocation [3, 39]. Others applied aggressive compiler technology [46, 41, 23, 12, 30] assuming some (perhaps statistically) fixed durations for basic machine operations, leading to an essentially static scheduling of runtime operations. The static schedule precludes recursions and dynamic loop bounds and data structures from the input program, and cannot tightly schedule conditionals with branches of unbalanced amounts of computation. The runtime approach is better suited for control-parallel computation, while the static approach is better for data-parallel computation.

In control-parallel computation, communication and synchronization paths usually occur predictably along the fork-join interface between the parent and child tasks, so that a runtime manager has a chance to optimize task scheduling and placement. In data-parallel computation, however, those paths are difficult to identify at runtime, requiring, instead, extensive data dependence and array subscript analyses by the compiler. Experience [34] shows that stringent time allowance and the lack of sufficient program information confine the best runtime policies to those that are simplest. Even then, the large number of fine-grained tasks coming from data-parallel programs can easily overwhelm the runtime manager. Compile-time program reorganization is a way to alleviate the runtime task management burden. On the other hand, real MIMD computing has much inherent asynchrony that is difficult for the compiler to anticipate (*e.g.*, communication latency, cache hits). Also, real programming environments may support features that prevent the compiler from having a total knowledge of and control over runtime conditions (*e.g.*, dynamic storage allocation/reclamation, incremental compilation, exception processing, multiprogramming). Total compiler management forces a MIMD machine to run with global synchronization in either SIMD or SPMD[1] mode, neither of which makes good use of its flexibility that is paid for through its hardware complexity. Hence there is a need to keep both the compiler and the runtime system usefully involved in the process.

We propose a framework to integrate the compiler and the runtime system together in managing program parallelism to allow both control- and data-parallel styles of programs to make efficient use of MIMD machines. But before introducing this strategy, we first explain the greater context in which it was developed: the dichotomy of the purely compiler-based and the purely runtime-based approaches to managing runtime parallelism. To understand the problems in relying on a static global task schedule, we will examine this approach and point out its weaknesses. We will then turn to show the inadequacies in current approaches to compiling asynchronous data-parallel code without assuming a static global schedule. The discussion of the specific problems in a specific domain of programming and machine environment and the developmental steps leading to our solution of an integrated parallelism management approach are found in the next chapter.

[1]Single-Program Multiple-Data, see [15].

1.1 The Problem with Static Global Scheduling

A *global schedule* determines the events of every processor against the time line; therefore, it contains the relative timing among events occurring on different processors. In contrast, a *local schedule* shows the relative timing of events on a single processor, and expresses no information regarding activities on other processors except at points of synchronization. Many attempts [46, 41, 23, 12, 30] to involve a compiler in solving the program decomposition problem either explicitly or implicitly assume a global schedule at compile-time. That is, only with this schedule on hand, which is of course itself dependent on the outcome of the partitioning and scheduling decision, can a program be partitioned and scheduled for minimum parallelization overhead (*i.e.*, from communication and synchronization) that will then compute in the shortest time using p processors. This is because the rate of speedup of a routine with each additional processor allocated to execute the routine,

$$S'(p) = \left. \frac{\Delta(\text{speedup of routine})}{\Delta p} \right|_p$$

is a monotonically diminishing function in general.[2] That is, the speedup curve as a function of the number of processors for which it has been partitioned is convex because the total parallelization overhead increases with decreasing granularity.[3] Thus when two routines to be partitioned are known to be simultaneously ready to execute, the optimal schedule would have both running, each partitioned to use a fraction of the p processors, in a configuration known as *space sharing*. The alternative of partitioning each to use all the available processors and then scheduling them in tandem, known as *time sharing*, is schedule-independent but suboptimal.[4] The computation of the optimum partition sizes and its proof are given in [41]. The essential observation is that this partitioning method thus needs to know which computations are contemporaneous throughout the system — before deciding the partition granularity of a piece of code — and results in its reliance on a static global schedule.

Unfortunately, the timing of computation events in a typical MIMD machine is far less predictable than that in a SIMD or a VLIW machine. Data-dependent conditional branches and loop

[2]Speedup is defined as

$$S(p) = \frac{\text{serial execution time}}{p\text{-processor execution time}}$$

[3]A simplifying assumption is used in this section which ignores such factors as the effects of scaling the total cache size, consideration of which could also yield superlinear speedup results.

[4]Again, discounting the effects of scaling the various memory resources.

terminations alter the instruction count dynamically. Network traffic congestion [38] is rarely precisely accounted at compile time.[5] Similarly, cache activities are difficult to account for, and collisions in direct-mapped caches are proving to have a significant impact on multiprocessor performance.[6] So when the actual relative completion times differ from the optimal static global schedule, our options are either to wait until the slowest processor has caught up with the schedule before advancing everyone, or to proceed with new tasks and to allow the actual schedule to skew and diverge from the static schedule. Either choice voids the optimality conditions on which the initial compile-time partitioning and scheduling decisions were based.

Figure 1-1 shows an instance where runtime adherence to the static schedule could perform worse than simple greedy runtime scheduling.[7] On the left, each of the numbered tasks is executed by a statically assigned block of processors. A certain processor slow in completing Task 3 will cause either a stall in the global schedule (as shown on the right) or further timing skew later on, if the scheduler tries to schedule around the delayed task. This is due to its inability to balance workload dynamically across processor blocks, and often even among processors within a block. Furthermore, under a common strategy known as *gang scheduling*, a contiguous block of processors, if not a particular block (in this case the entire block of processors assigned to Task 3), has to be free to allow a succeeding task to be scheduled. Thus the compiler's dependence on a static global schedule to partition a program could be far from optimal in practice, unless that schedule really anticipates the detailed event timing. Such details include the effect of control decisions on the instruction stream (*à la* strict trace scheduling of VLIW, treating the processors as functional units and the network as gated data paths), and runtime variables of cache misses and network routing traffic.

An actual instance of cache-miss-induced schedule skew on a MIMD machine is cited in [2]. The active-processors profile of Figure 1-2 shows that, for a very regular and easily-partitioned matrix multiply code, static partitioning led to a scenario where some processors were taking much longer than the rest in completing its statically assigned block of loop iterations. The long tail falls first at 1,130,000 to 3 active processors, then at 1,220,000 to a lone active processor. It finally finished at 1,440,000, when all processors entered termination signaling phase. Figure 1-3 shows

[5]Although when the processors are completely statically scheduled, the network behavior really can be fully anticipated and its routes pre-computed and statically programmed [30, 51, 5].

[6]Recent work [2] gives higher priority to minimizing the skew in the global schedule than simply to minimizing the total work — skew due to nonuniform cache behavior.

[7]A purely greedy schedule provably performs, at worst, half as well as the optimal schedule [20].

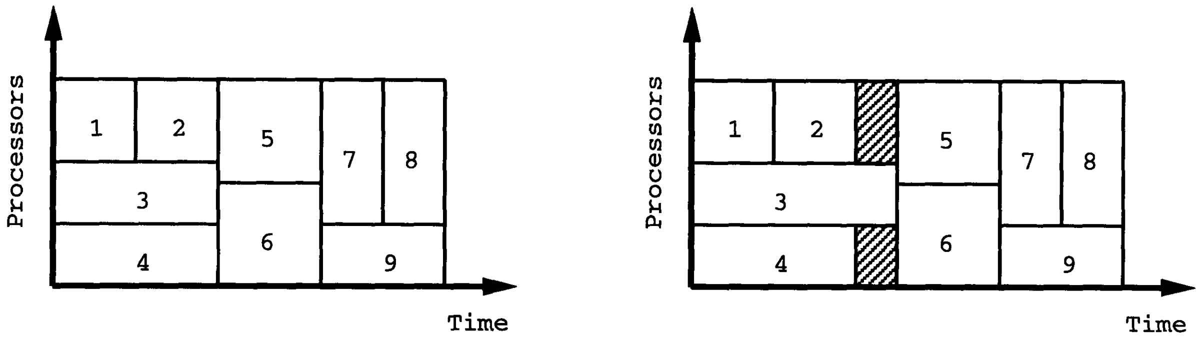


Figure 1-1: *Cost to Stay Synchronized to a Static Global Schedule*

the uneven caches misses incurred by the processors, each executing an equal number of iterations. In this particular case, the problem is due primarily to thrashings from the direct-mapped cache organization, which is a common choice among current processor designs. A particular solution for this problem has been to skew the heap starting address for each processor so that accesses to the same data in different partitions do not contend for the same cache line.

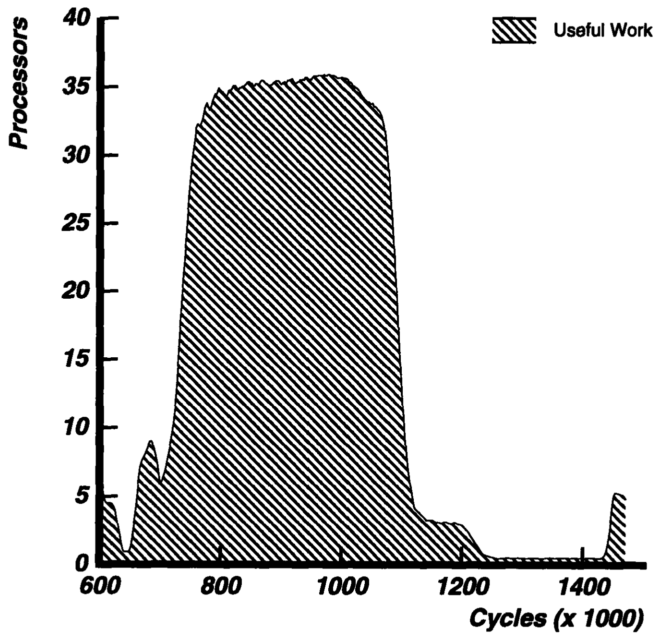


Figure 1-2: *Processor Profile for Parallel Matrix Multiply*

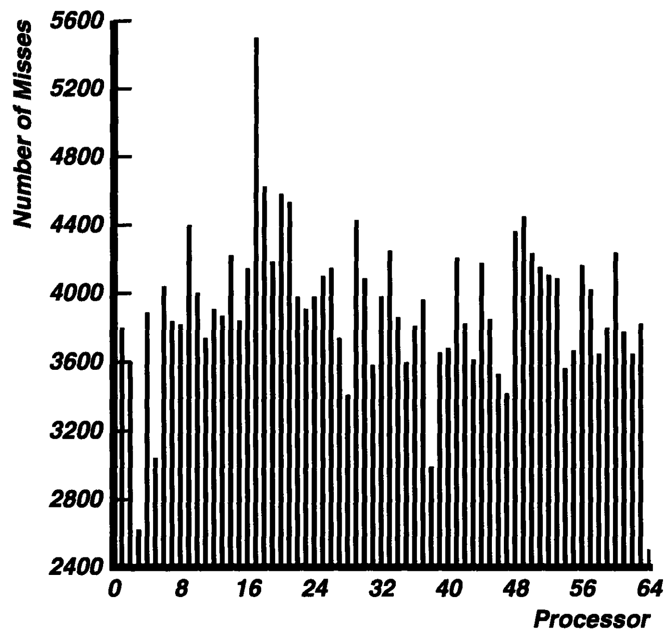


Figure 1-3: *Cache Misses of Processors for Parallel Matrix Multiply*

1.2 The Problem with Runtime-managed Scheduling

Without the static global schedule, the compiler is left without a means to determine the partition granularity for optimal trade-off between maximizing useful parallelism and minimizing its overhead cost. Developments along this path often have the compiler producing many very fine-grained tasks, and rely on additional special hardware support [22, 3, 21, 15, 1, 14] for fast task creation, synchronization, suspension/switching, and termination. Refinements have been introduced to delay runtime task creation until needed [16], or to offer programmers new language features to control runtime in-lining of tasks [19]. These are general mechanisms that, nevertheless, do not make use of information that may be available to a compiler and may be particularly useful in efficiently addressing the following needs of the massively data-parallel procedures:

- The dominant inter-task communication usually occurs at the interface between the parallel producer and consumer tasks of the large data structures, rather than those at the interface between the parent and child tasks. A consumer task often cannot start until the corresponding producer has computed its value, thus creating it as an independent task needlessly burdens the runtime manager and congests its queues.

- Preservation of communication locality across such parallel producer- and consumer-task interface is essential to minimize network traffic and access latency when scaling the system and problem sizes.
- In these instances the existing hardware and runtime parallelism-controlling mechanisms mentioned above do not provide for alignment of tasks along the appropriate direction — according to accessors of parallel data structures.
- By itself the runtime manager is unable to identify which tasks are on the critical computation path and to schedule it preferentially to allow other, dependent tasks to proceed or to be created, thus maximizing the available parallelism.

These are some key motivating concerns for the present work, which searches for some viable solutions. Even after addressing these concerns, there may still be large excess of parallel tasks: Allowing 100,000 tasks to be concurrently created when there are only 100 processors available is probably suboptimal, given both the task creation costs and the convexity property of the speedup curve. Extra storage is needed, as well, to hold the context states of pending tasks. Providing the compiler an estimated runtime machine size and allowing it to fold this excess parallelism could reduce these runtime overhead significantly.

Yet another approach uses compiler analyses to form loop iteration-based tasks and runtime scheduling of these tasks to achieve a degree of load balancing and machine-size independence [39, 15]. They exploit parallelism from iterative loops only and have a fixed partition granularity, namely one loop iteration.

1.3 Revocable Thread Serialization

We introduce an integrated compile-time and runtime program decomposition strategy called *Revocable Thread Serialization* (RTS) that provides separately a framework and a controlling policy to partition, schedule, and place parallel programs and their data structures automatically. The intent is to incorporate information of statically-characterizable iteration constructs and array index expressions from the compiler to structure the initial allocation and sequencing of computation among the processors, while leaving the runtime system the flexibility to revoke any of the static arrangements in order to allow load balancing and to avert deadlocking. The major phases of this framework performed by the compiler are as follows:

- *Partitioning:* the compiler analyzes the source program to partition parallel loop iterations and data arrays by splitting them into *partitions* to maximize reuse-type locality within the partition.

- *Alignment:* the compiler aligns loop and array partitions and individual threads and data objects that communicate the most by merging them into *detachments*.

- *Provisional Serialization:* the compiler then provisionally serializes for each detachment the threads within to form a coarse-grained, single-threaded *brigade* — by choosing a default order of execution and then juxtaposing them with a runtime annotation,[8] for undoing the serialization.

The runtime part of this framework requires a compact runtime manager, also known as the scheduler, to run on each processor when no user task is active. The runtime manager supports these operations on tasks:

- Creating an initial task on some processor to execute each dynamic instance of a brigade.

- When a processor becomes idle, attempting to find a ready-to-execute task or one of the provisionally serialized threads from another processor.

- Once a ready task is found, migrating it to the idle processor and starting it immediately. On the other hand, if a provisionally serialized thread, splitting the piece of work off from its brigade cleanly and migrating it to the idle processor for execution.

The compiler's ability to analyze statically the loop structures and communication paths between parts of the program and the data structures is limited by the complexity of the program constructs. We contend that a large number of data-parallel numerical programs do have very simple control and data structures at their compute-intensive core. This class includes programs that can be coded in a variety of Fortran and analyzed by some parallelizing compiler. But, because of the runtime support, our compiler can take a much less strictly-top-down procedure characteristic of static global schedule-based schemes. It is able to resort as well to a bottom-up approach to construct aligned thread and data partitions that does not require absolute knowledge of loop bounds or the number of processors, or require resolution of all of the communication paths in a program. With RTS,

[8]This annotation will be introduced later in sections on implementation strategies as the *lazy future* mechanism.

the compiler participates in the management of runtime parallelism without losing the flexibility of a runtime manager to respond to runtime conditions. The RTS framework, in essence, allows the compiler to play a classic game in computer systems engineering: optimizing for the common case while maintaining a reasonable cost to handle the exceptional cases. If the common case can then be arranged to apply most of the time, the system may still be more efficient overall even if penalties must be paid occasionally for the exceptions. Those exceptions are the instances when the compiler could not foresee the runtime behavior or system configuration and made a wrong static partitioning or serialization choice that would need to be revoked later.

The controlling policy is a set of heuristics specifying some variables in tuning the effectiveness of the compilation framework: the granularity of static partitions the compiler assumes, the amount of work from a brigade to migrate among processors, and which parcel of work is target for migration.

There are two major advantages to the RTS approach: adherence to true MIMD operation and competitive performance on a broader range of programs. First, it is more compatible with the inherent asynchrony in much MIMD hardware operation, without requiring very frequent global synchronization to force close lockstep operation or otherwise to bring the runtime state to some pre-determined condition. Second, it functions in the context of a more general, "symbolic" programming paradigm within a mostly runtime-managed environment, while still allowing its data-parallel computations to benefit from compile-time analyses and static program and data decomposition. In such problem domains we therefore hope to see performance efficiency close to par with parallel Fortran on SPMD-type machines, where compiler optimizations have been applied with much more success.

In comparison, the CM-5 [8], currently a leading-edge commercial multiprocessor from Thinking Machines Corporation, for example, offers a message-passing facility, CMMD, for explicit user control of tasks and their communications, and explicit user partitioning of distributed data structures. The CM-5 also offers a CM Fortran programming system, under which array data and program loops are automatically partitioned and distributed, but users are not to declare any parallelism beyond those the compiler infers from those program loops involving array data (such as having two procedures execute in parallel on different processors). These two programming models are inherently incompatible. So when CMMD is used in a CM Fortran program, the Fortran code usually serves only to describe the intra-node-level computations, thus defeating the automatic global partitioning, scheduling, and communication management done under the data-parallel CM Fortran.

1.4 Expected Results and the Performance Comparisons

We expect to see performance for the three classes of approaches — the purely static, the purely runtime, and the RTS parallelism management — to differ based on program characteristics and runtime conditions. Our analyses and presentation will focus on the most salient metrics that will distinguish the behavior of our proposed scheme from that of the others. We claim that RTS is able to tolerate non-uniformity in actual execution time of threads from the statically calculated execution time. Such variations arise from data-dependent control structures of the program, as well as from asynchronous hardware operation or network routing described in Section 1.1. Figure 1-4 shows the expected performance degradation from variations in actual execution time of a set of statically identical threads.

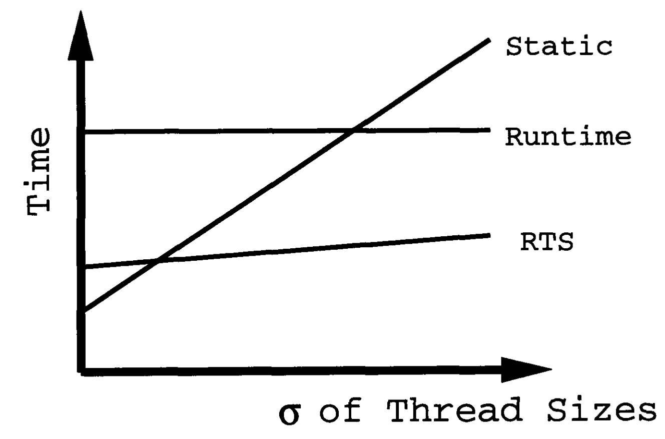


Figure 1-4: *Expected Tolerance to Dynamic Workload Variations*

Obviously, static partitioning does best with no variation in execution time among the threads, since there is almost no additional runtime overhead. It groups the threads into exactly as many partitions as there are processors and assigns a partition to each processor. But with some introduced variations, the fixed partitions cannot shift work around. Thus, the completion time would be determined by the slowest partition, which is linearly proportional to the standard deviation (σ) of a normally-distributed sample. The purely runtime-scheduled system, on the other hand, has enough short tasks in the scheduler queue that load balancing is not a severe problem, provided that the scheduler does not itself become a point of severe contention. Its performance should be largely independent of variations in task sizes, although the runtime management overhead would be consistently high for fine-grained sizes. Under RTS, the compiler attempts to partition a program in the same way that static partitioning does. Given similarly sophisticated program analysis and transformation techniques, and the same amount of information on the input program, compiled RTS partitions would incur the extra cost of the thread joint annotations in comparison to the static partitions. With no size variations, no joint will be revoked since all partitions should finish simultaneously. But, as thread sizes vary, those processors that finish early would take work from those still working. This action will add to the total overhead, although a significant part of this is incurred by processors that would otherwise be idle. The outcome is that the RTS curve should rise, but much more slowly than the static one.

Figure 1-5 shows the expected behavior when the compiler has scheduled threads in an order incompatible with the actual runtime dependencies. We use the more tangible label of the "amount of forward references" to quantify this notion. A forward reference arises from a computation requesting results from some other computation that it or another processor has yet to perform. Obviously, statically sequenced partitioning cannot tolerate any out-of-order scheduling from the compiler, while the runtime scheduled system would be, in principle, impervious to the problem.[9] Lastly, RTS is expected to outperform a purely runtime-scheduled system because it would have to create many fewer tasks if all of its provisionally scheduled thread orderings are confirmed valid. As more of its scheduled orderings are revoked at runtime, the higher cost of revocation would drive up the overall cost, perhaps at some point exceeding that of the runtime scheduling.

These are assessments of what we expect to see from the work to be presented here. They serve to motivate and to cast in more specific terms the types of target behaviors we seek. We will return to these metrics later and compare them using figures from detailed simulations to the alternative parallelism management schemes. The simulation results to be presented in Chapter 5 confirm all the qualitative behavior comparisons described in this section.

[9] Although it should be noted that if there were a large number of interdependent tasks, and a runtime scheduler is used to pick randomly the order of execution, there would likely be many task suspensions and resumptions, at a great cost, before the right order is empirically reached.

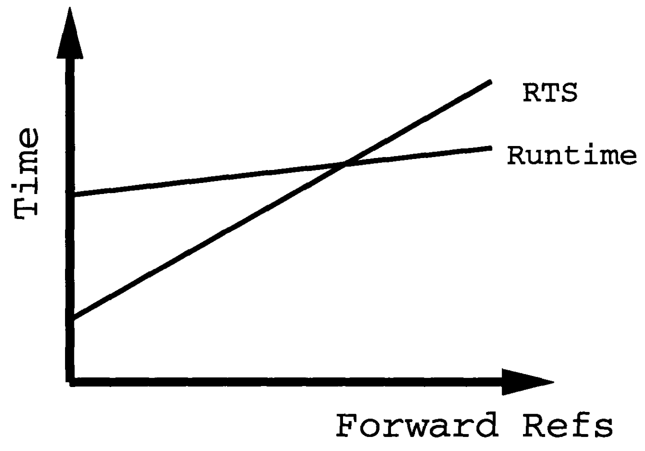


Figure 1-5: *Expected Cost of Responding to Runtime Reordering*

1.5 Summary of Contributions

We have introduced an integrated framework to manage asynchronous tasks for parallel processors and proposed some of its tuning policies. The top-level contribution of this work is the conception of the framework itself: a compiler to partition program and data to preserve locality and to serialize fine-grained threads provisionally to reduce runtime overhead, combined with a runtime system that decomposes partitions by revoking some serialization and migrates tasks among processors to maintain load balance and to avert deadlocks. The compiler implementation is realized as a source-to-source program restructuring package, called Scepter, placed before a serial machine-code compiler. The major transformation phases — partitioning of loops and array data, alignment of partitions into detachments, and scheduling of threads in each detachment into brigades — start after the program analyses. An internal program and thread/data communication representation form, called WAIF, to capture the results of the program analyses, has been defined to facilitate the required transformations. The entire framework has been implemented to the extent where a substantial, 1200-line standard benchmark program has been compiled successfully (without manual intervention before or after the compilation) and run on up to a 64-node simulated machine. Alternative serialization strategies have also been implemented under the same framework, using the same machine and runtime environment, so that direct comparisons with them are possible. Our specific contributions in the implementation include the alignment of unequal-sized and -shaped partitions and the revocable serialization of independent threads that allows runtime adjustment of task granularity and thread scheduling order. The concepts of lexical-environment and control-order preservation for parallel threads are introduced and enforced throughout the serialization process.

Lastly, we report and analyze simulation results of system performance with some of the tuning policies, using both synthetic and standard benchmark programs.

1.6 Related Work

Sarkar [46] and Prasanna [41] both do strict top-down partitioning based on an assumed static global schedule. They both produce atomic task partitions: all inputs are ready before a task starts; thereafter, the task executes uninterruptedly.[10] This constraint would often severely limit the maximum size of tasks or reduce the amount of actual runtime parallelism. Static task placement could co-locate related tasks, so maintaining communication locality is not a problem. But if tasks are allowed to migrate, then maintaining larger tasks becomes desirable. Sarkar uses estimates of execution times of macro operator nodes such as for iteration counts, probability distribution of conditional branches, and recursion depths that are in fact very input dependent.[11] Sarkar also ignores data structure decomposition, assuming the data structures to be functional and treating each as an indivisible unit, *e.g.*, when passed along the graph edges.

Prasanna's work has only been applied to very specific and regular programs, namely nested matrix additions and multiplications, because it requires information such as the speedup function of a parallel program and its derivative function before the method can be applied. Aside from the difficulty of determining such functions, many program routines do not have smooth speedup function, nor for that matter the same speedup function through its execution time.[12]

Lo, *et al.* [30] describe the OREGAMI project. Its static task-to-processor mapping takes, separately from the program itself, a synchronous task communication specification, and, after trying a few regular, pre-computed solutions, resorts to a general node contraction algorithm. The contraction is a two-step process using a greedy heuristic that merges adjacent nodes to form a graph of no more than twice the final node partitions, followed by an optimal maximal-weighted matching function that reduces to the desired number of equal-sized minimized-communication partitions. This system's synchronous nature requires specifying the computation tasks and their communication for each computation phase. Barrier synchronization between phases is assumed, as

[10]Sarkar refers to this task partitioning discipline as a "convexity constraint".

[11]The static part of our work also relies on approximated or assumed dynamic information; however, our integration of runtime inputs and controls means that the use of inexact compile-time information would be much less detrimental.

[12][31] introduces the program *parallelism profile*, which shows the maximum speedup at each instant during program execution. Its sharp peaks and troughs shows the time varying nature of speedup functions.

well as migration of each task to the processor it is assigned for the next phase. Dynamically-spawned tasks and intra-phase computing time and communication variations are difficult to accommodate efficiently.

A line of investigation not based on the top-down, structured approach focuses on parallelizing independent loops [33, 15], and partially-dependent loops [12]. These are the synchronous analogues of the schemas described in Sections 4.3.1 and 4.3.2, respectively. The fundamental differences in the analogy are noted in those sections. The independent loop iterations are then dispatched from a central site one at a time to requesting processors under self-scheduling [49], or a block of iterations at a time under guided self-scheduling [40], whose iteration block size is a constant fraction of the number of iterations remaining to be dispatched.

Work on SIMD machines showed potential for great communication cost-saving in aligning data arrays between consecutive phases of parallel operations [26]. The partition alignment procedure in conjunction with the schema in Section 4.3.3 represents our attempt to address this problem.

The dataflow program graph works of Traub [50] and Culler [11] address the compilation of non-strict functional programs into sequential threads. But they provide no means to capture the notion of data locality, nor attempt to analyze the data references needed to decide the partition and placement of task and data. These efforts are concerned with exposing maximum parallelism, assuming sufficient storage and communication bandwidth and relying on masking latencies with excess parallelism. In [11] the program "partitions" are those along procedure activation boundaries. In general, the low-level nature of the dataflow graph intermediate form tends to obscure the control structure of the program and make analyses tedious.[13] We have not found published descriptions of work that extract from dataflow program graphs information typically required of structural program transformations; currently they have only been used as target object code for physical or abstract graph interpreters.

Rogers and Pingali [44] proposed a way to translate conventional shared-memory programs to dataflow program graphs. They assume programmer-specified data partition and placement, as do some imperative-language-based work [23].

Runtime managers in dynamic dataflow [10, 45] have focused on controlling the number of live tasks for the sake of limiting resource usage to prevent deadlocks. The mechanisms control

[13]In the same way that inferring program flow or the extent of variables' values is more tedious, though sometimes not impossible, in a non-block-structured program using arbitrary `go tos`.

how many loop iterations are unrolled, but none takes deliberate position on which iterations of which loops to unroll. The implicit assumption in both — to unroll in program order — is that such controlled runtime unrolling will not cause program to deadlock logically, although the functional language semantics of Id and others does not impose such restriction. A further point of note is that these proposals present controlling mechanisms, leaving unsolved a significant policy issue of what should be the appropriate unrolling bounds. Determining this likely requires either programmer annotations, or a static global analysis of the program's control structure and the actual loops' iteration bounds.

The lazy future mechanism [16] is introduced to reduce the cost of creating many unneeded futures, often a result of over-specifying parallelism necessary for scalable programming. This is a low-level runtime solution which by itself cannot address data-parallel computation needs adequately, as we have argued earlier. Our proposal builds on this basic mechanism, providing compile-time restructuring of program threads and data so that applications of the lazy future can be directed at the appropriate place and time.

Earlier work on integrating control- and data-parallel computation included Chen's [7] efforts to create functional abstractions for standard data-parallel array manipulation operations so they could be incorporated into the context of the Crystal functional language. The array operations are then meant to execute on specialized data-parallel hardware.

1.7 Outline for the Thesis

In the next chapter we describe a specific machine and programming environment that will be the base for introducing RTS. We explore the implications of this system under various program scenarios to show its strengths and weaknesses. A strategy to improve this system motivates a solution implementation in the compiler. Chapter 3 lays out the design structure of the compiler, and shows how an input program is transformed at each compiler stage to arrive at an object program whose structure meets the requirements for efficient runtime parallelism management and maximizes cache and local memory utilization. Chapter 4 describes the detailed implementation considerations for the static parallelism management stages of the compiler and policy options for tuning the performance of the framework. Chapter 5 presents the benchmarks, using both synthetic kernels for testing particular aspects of the system and an entire application for showing its general performance in adapting to solving real-world computing problems. Chapter 6 gives the conclusions, and hints at future directions of exploration.

Chapter 2

Background

This chapter describes a specific parallel machine and programming environment that will be the base for introducing an implementation of the Revocable Thread Serialization framework. The existing language and runtime system (Sections 2.2.1 and 2.2.2) support a mostly runtime-managed scheduling scheme operating on fine-grained tasks. Section 2.2.3 notes how the runtime scheduling interacts with a running program and the expensive runtime costs that result from the scheduler intervention. Section 2.2.4 examines a runtime mechanism introduced by Mohr, *et al.*, [16] that has an effect of automatically adjusting task granularity by provisionally joining threads together according to their serial placement, thus reducing the amount of scheduler intervention required. The runtime part of the RTS framework adopts this mechanism from Mohr. We will then provide evidence that this mechanism indeed works quite well for one broad class of programs, although we will also show how it would become a pure burden for another broad class of programs. Finally, Section 2.2.5 analyzes the underlying reasons for the mechanism's success and failure and speculates that a compiler might be used to restructure a program so that the latter class will benefit equally from the runtime mechanism as the former class. This exposition will motivate the need for, as well as setting up a complete problem context leading to, the presentation of the compiler phases of RTS in the next two chapters. Readers who are familiar with the concepts, runtime behaviors, and insights of a lazy future-based multiprocessor programming model should skip this chapter.

2.1 Hardware Environment

The Alewife multiprocessor [1] is the centerpiece of the Alewife research project at MIT. Its goal is to explore the utility and cost-effectiveness of providing a shared-memory programming model based on physically-scalable hardware: homogeneous processor-memory nodes with hardware-assisted directory-based coherent caching, embedded in a low-dimension network that provides asynchronous message delivery, shown in Figure 2-1.

The cost structure dictated by the hardware greatly influences compiler optimizations; an intra-node access (*i.e.*, to the local memory) is much less costly than a network-bound access, while the incremental cost of reaching a more distant node is only second order with the use of cut-through routing. Table 2.1 summarizes some raw figures of the current implementation of an Alewife machine processor node that are of interest to us. A uniform global memory address space is supported in hardware through the custom-designed cache controller (CMMU) [27] that interfaces to the processor, local memory, local cache, and the network [18] to provide the function of a coherent processor-side cache.

The global-memory address mapping of the distributed memory is straightforward: the node number is the most-significant-digits prefix of each global address, followed by an offset indicating which memory location within the chosen node. Any distributed data structure must be managed by the user or compiler explicitly, as the hardware does not support finely interleaved global-memory addressing. The assumption of a locally-cached shared-memory allows us to abstract the details of generating remote fetch operations, and to focus our concerns on how best to use the provisions of these mechanisms: the automatic local replication of global or loop constants, and the easy runtime relocation of tasks and data. The global address space and the caching can certainly both be emulated in software at a higher cost. The compiler can perhaps then be called on to optimize these operations. We choose not to pursue these detail issues here; the available hardware mechanisms should adequately provide the functions that are sought.

2.2 Programming Environment

Although there certainly are blended varieties, current parallel programming can be broadly classified into the following three camps:

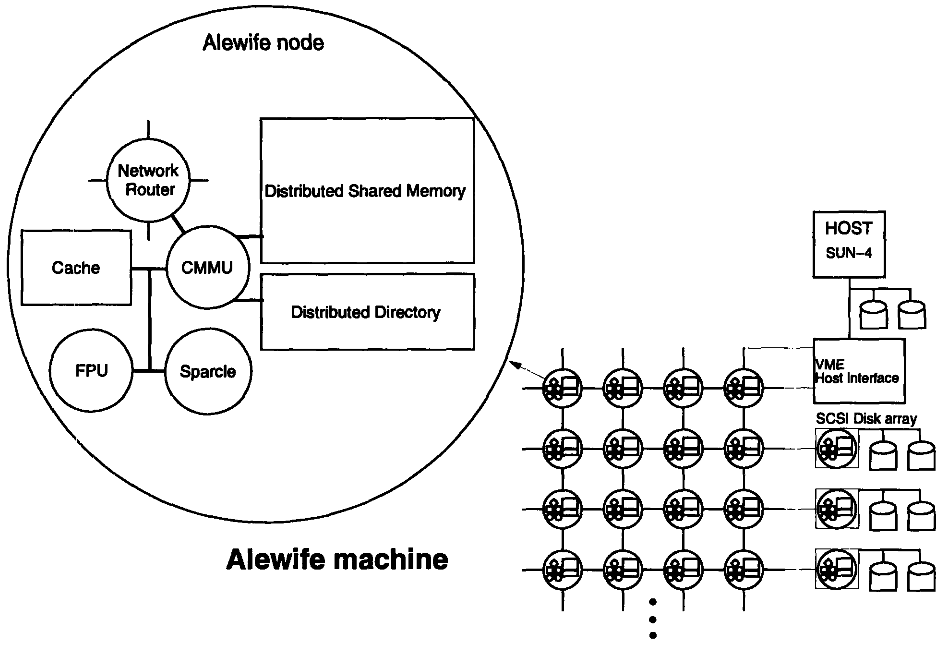


Figure 2-1: *The Alewife Multiprocessor*

- Serial programming: Program is converted to an appropriate parallel version suitable for the execution system by automatically discovering the necessary parallelism.

- Explicit parallel programming: Parallelism is imperatively specified for (and explicitly placed on) a particularly configured machine.

- Declarative parallel programming: Program specifies all the parallelism and relies on automatic reorganization to run efficiently on a given system configuration.

The first and last of these provide configuration-independent machine models to the programmer, thus achieving practical system scalability. Both therefore must address the problems of how to automate the management of parallelism. We have chosen the last to bypass the much explored issues of parallelizing serial code. The generality of this approach can be argued thus: in principle, a serial program can first be parallelized using any of the existing program parallelization efforts. The resultant output will then be ready for our processing, which, in fact, focuses on raising the actual task granularity through a cheap and flexible serialization to achieve efficient operation on a finite-sized system.

Operation	Processor Cycles
Load Instruction	2
Store Instruction	3
Context-switch (Data Request)	14
Private Cache-miss Penalty	9
Shared Local Cache-miss Penalty	11
Remote-initiated Cache Flush	6
Remote Read-miss Penalty (Clean)	28 + 2*distance
(Dirty in Home Node)	35 + 2*distance
(Dirty in Third Node)	58 + 2*distance
Directory-read (cache)	5
Directory-write (cache)	6
Fast Task Dispatch	168 + distance

Table 2.1: *Timing of Selected Alewife Machine Operations*

2.2.1 The Mul-T Programming Language

We have chosen to build this work on the Mul-T language [13], which has an explicit declaration construct for parallel computations. Mul-T offers a convenient way to specify fine-grained parallel computations, while it still preserves the ability to express side-effect-based algorithms and serial code sections as naturally and efficiently as a sequential language does. Mul-T is an extension of T [47], a Scheme-like language, for parallel computation.

The principal parallelism generator introduced is the *future* construct [22]. An invocation of `(future` *<expression>*`)` evaluates the enclosed *<expression>* concurrently with the rest of the program following the future, known as the *continuation* of the future. We will sometimes refer to the expression inside the future as the *child thread*, and the future caller and the continuation together as the *parent thread*. Future conforms to the sequential ordering semantics of the rest of the language by creating a future task to evaluate the expression and immediately returning a special value, which is a reference to an empty *placeholder* object, so that the continuation may proceed as if the expression has already been computed. The placeholder is a self-synchronizing data storage that enforces the read-after-write access order. While the placeholder is empty, an attempt to read the value of a placeholder, known as *touching* the future, causes the reading task to be suspended. When the future expression is eventually evaluated, the placeholder is filled with the result value. The placeholder then becomes *determined*, and all tasks blocked on its reading are again enabled. Reading a determined placeholder yields its value just as does with a regular variable. A future expression that returns no value (*i.e.*, executed for side effects only) does not require creation of the placeholder object.

The following short example function is used to illustrate parallelism generation in the runtime system.

```
(define (main x)
  (let ((a (future (f x)))
        (b (future (g x))))
    (+ a b)))
```

For the purpose of this discussion, we assume `f` and `g` are both entirely serial. Figure 2-2 depicts an execution of `(main 15)`, spawning two futures for computing `(f x)` and `(g x)`. These futures are entered in a queue of available futures. Any processor, including the initial creator of the future, that has become idle can *steal* a future by taking one off a queue and initiate the task to compute its value. Thus, a total of three tasks are called for to complete the specified work. Figure 2-3 shows the argument/results data communication links for the `(f x)` future. At future creation, the bindings of the free variables in the future expression are captured, since they must refer to the lexical environment valid at the place the future is declared.[1] During evaluation of the future expression, the values of the free variables may be read and, in general although not here, their contents may be written.[2] Also, the result of the future expression is finally written into the value slot of the placeholder object that is in $t1$.

Figure 2-4 shows the program's *parallelism profile*, the number of active tasks as a function of time, assuming there is an unbounded number of available processors so that newly created futures become running tasks immediately. The graph is shaded only to relate an accounting of the active tasks to parts of the program text. At the beginning, `main` alone runs. At times T_{fb} and T_{gb}, tasks `f` and `g` are started. Almost immediately afterwards, at time T_{ms1}, `main` is suspended, waiting on the touch of the future value of `a`. As task `f` ends at time T_{fe}, `main` resumes at T_{mr1} and quickly suspends again at T_{ms2}, now waiting for the value of `b`. When task `g` ends at time T_{ge}, `main` resumes and eventually ends at T_{me}.

Mul-T presents a shared-memory model; that is, all data is accessible, without explicit copying, to any part of a program.[3] It has a store-based semantics, meaning that variables refer to storage

[1]This may be accomplished by creating a closure [48] for the future expression at that time.

[2]Of course, function `f` itself may contain free variables that refer to the lexical environment at the place of its definition.

[3]Stack-resident and other dynamically-allocated data, of course, have limited lifetimes.

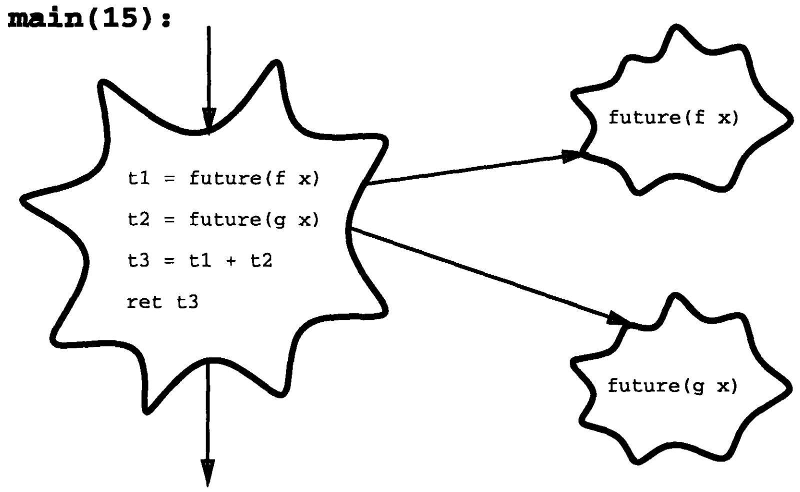


Figure 2-2: *Parallelism Generation through Futures*

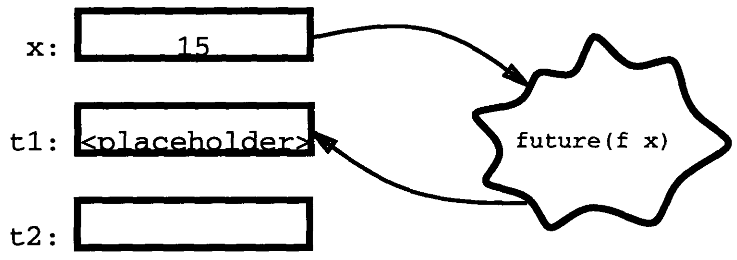


Figure 2-3: *Dynamic Data Linkage of Future Tasks*

locations that contain values. Its standard data structure types include lists, structures (or records), and arrays. In addition, there are the self-synchronizing data types to provide fine-grained synchronization for enforcing the producer-consumer relationships in data-parallel programs: J-structure and L-structure. The J-structure, derived from the I-structure [4], is an array each of whose elements has an associated full/empty (F/E) flag. The F/E flag is initialized to empty. A J-structure read operation reads the content of the location if the flag was full, otherwise it suspends the reading task and enters it in a queue of waiting tasks for the given location. A J-structure write operation stores a value at the location and sets the flag to full if the flag was empty, otherwise it signals a double-write error. A valid write also wakes up all the waiting reading tasks by moving them onto a queue of runnable tasks. There is much similarity between the future's placeholder object and a J-structure element. The L-structure is a J-structure with a resettable F/E flag and a more symmetrical read/write semantics. We will not be referring to the L-structure in this work.

In this work an instantiation of the body of a future is also referred to more generally as a *thread*, although we shall see other ways to produce threads. A thread is not concerned, by itself,

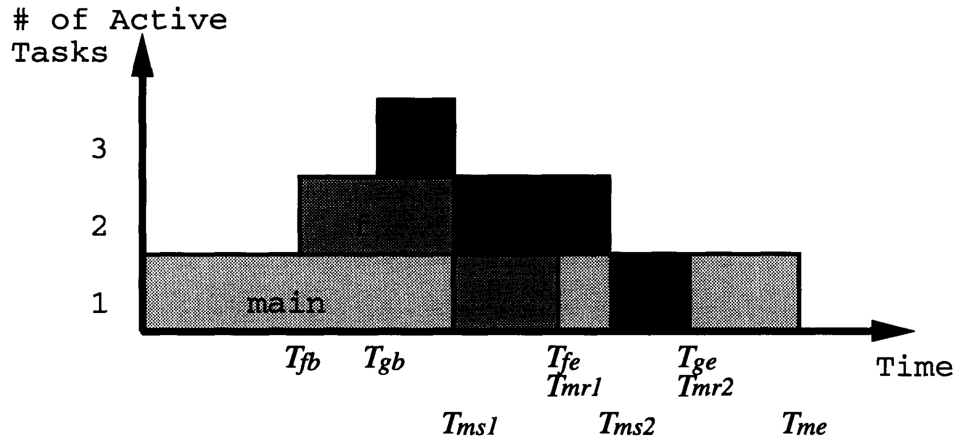


Figure 2-4: *Dynamic Schedule and Parallelism Profile of Future Tasks*

with the future's placeholder linkage setup. The term *task* refers to the basic runtime-scheduled entity; that is, each task has an entry in a runtime manager task queue of some processor. The task queue is distinct from the future queue mentioned earlier. Although for now task and future can be treated synonymously, it will become obvious that they need to be different later, when a "lazy" implementation of futures is introduced. A task is needed to execute a thread or a brigade (a thread that serializes a collection of threads). Since our tasks all share the same address space, they are more closely related to the notion of lightweight threads in the operating systems literature, where tasks are usually synonymous with processes. We use the term *job* to refer to the collection of tasks that, together, computes a single user program. There may be other jobs running concurrently in the system, whether under a space-partitioned scheme[4] or not. We do not address the issue of protection amongst the tasks or the jobs.

2.2.2 Mul-T's Runtime Task Manager

Our runtime task manager is a system program that places runtime tasks onto processors, schedules a task to each idle processor from queues of runnable tasks, and on demand migrates tasks among processors to fill their idle time. But more specifically, it has no notion of the actual requirements of communication locality among tasks and data, nor the ability to prioritize effectively the scheduling of tasks; these requirements must be imposed within a task before the runtime. The runtime manager treats all tasks in its FIFO task queue with equal preference in regards to scheduling and

[4]A non-overlapping subset of processors is explicitly allocated for each job.

task migration choices because it has to be kept very simple and minimal to limit its own impact on the system performance. Examples of the prescribed runtime manager are described in [36]. When a processor becomes idle, the manager repeatedly tries all tasks in its task queue until one is found that is able to continue.[5] Thus, it is desirable for tasks that are obviously dependent on others' completion not be created earlier.

In our conception, the division of labor between the static and runtime task management will be as follows. For sections of code that the compiler is able to analyze, it decides how many initial runtime tasks to create[6], which threads are computed in what order by which runtime tasks, and which new tasks to create should the runtime manager demand more. The runtime system distributes and redistributes the tasks to keep processors from idling, and occasionally asks for more tasks when there are not enough to do its work efficiently.

2.2.3 Toward an Efficient Future Implementation

This section explores the interaction of runtime task scheduling with a running program to reveal the runtime costs that result from the scheduler intervention. Shown is the variability in the amount of required scheduler intervention and the importance in controlling the overall costs of creating and scheduling tasks. Finally, ideas are suggested in this section to direct the runtime scheduler from the compiler's output code.

Section 2.2.1 noted that, for the example program presented, two futures were invoked, and thus three tasks were needed, to complete the computation. They would always be created and then scheduled for some processor by the runtime system described whether there are actually enough processors to speed up the execution through parallel processing. In realistic scenarios, often this extra work will never reap any benefit, because far too many futures are created by recursive or looping programs when compared to the number of processors used to execute such a job. Instead of generating useful parallelism, the excess tasks will pile on the task queues, for each task a substantial creation overhead will have already been paid. Before showing the existing solution to this problem, we note another penalty of having many more tasks around than necessary: it is expensive to juggle tasks.

[5]Tasks that are blocked on a future placeholder or a self-synchronizing data store are entirely taken out of the task queue and therefore do not affect scheduling of executable tasks.

[6]The number of initial runtime tasks is equal to the dynamic count of the detachments generated by the compiler.

In the runtime scenario depicted in Figure 2-4, if there are no other processors available to compute the two spawned futures, the lone processor will switch to and start task f after suspending main at T_{ms1} on touching a. When f finishes and writes a, main will be awakened by the placeholder determination, resumed, and quickly suspended again on touching b. Then task g will start and when it finishes, main will resume and eventually terminate. Compare this execution description to the case where a serial version (*i.e.*, the same program without the future forms) is run on the lone processor — enter main; compute f; store a; compute g; store b; finish main — it becomes apparent that in addition to the cost of creating the tasks, much scheduling operations overhead, those of suspending, queuing, and resuming tasks, are often involved.

Changing the default order of which task to compute first at a task branch point (*e.g.*, when a future is "called") sometimes can substantially reduce the task management cost. Consider an alternative where the processor, after creating a new future, suspends the parent task and enters it on a task queue. It then starts a task for the child thread. With this scheduling order, a lone processor executing the example parallel program will behave exactly as it does the serial version — no more repetitive suspension and task switching — except that the two futures are still unconditionally created. An indirect cost born by this choice is that, should the parent task be stolen by another processor, its context and state, most of which are likely to reside already on the local node, will have to be migrated to or remotely fetched by that processor. In contrast, the original scheduler would have left the child task as target for a load-stealing processor. The child task, although still needing access to the free variables captured by its future closure, can migrate to a new processor before allocating its own local stack frame of local variables.

It is worthwhile examining the generality of this scheduling order: is the *child-first* order always optimal? Since it mimics the depth-first traversal of traditional stack-based serial programming semantics, any parallel program derived from adding future constructs to a serial program that already operates correctly and without deadlocking will similarly execute on a single processor, suspension-free, using child-first scheduling. Programs not parallelized from serial programs in this way do not necessarily have a suspension-free left-to-right, top-to-bottom serialization. Thus, our simple alternative strategy cannot guarantee suspension-free task scheduling, only that it is probably very likely with most programs. More sophisticated compiler-directed strategies will be explored later in Chapter 4, although in practice most Mul-T programs are serializable in a left-to-right, top-to-bottom textual order. Altering the scheduling order can also dramatically change the amount of runtime parallelism generated. We will consider the detailed aspects of manipulating parallelism through compiler control of such scheduling order later.

2.2.4 Streamlining Future Implementation — The Lazy Future

We have identified and highlighted two problems in implementing a future-based parallel computing environment: reducing the amortized costs of creating futures and of scheduling tasks. Achieving a degree of success here will encourage futures to be used more liberally, thus increasing the maximum parallelism expressed in source programs, yet without incurring the parallelization costs when the excess parallelism is not needed or realized at runtime. This section examines a runtime mechanism introduced by Mohr, *et al.*, that has an effect of automatically adjusting task granularity by provisionally joining threads together and dictating scheduling order among them, thus reducing the amount of scheduler intervention required. Evidence they provided shows that the new runtime mechanism indeed works quite well for a broad class of programs, although we will also cite how it would become a pure burden for another broad class of programs.

A mechanism known as the *lazy future* [16] was introduced and implemented to gain runtime control of task granularity. It functions by effectively ignoring the future declaration and executing the child thread in-line when enough tasks has already been created to keep the available processors busy. That is, for the program segment

```
    (f (future (g x)))
```

if no processors are idle, instead of creating a future to evaluate `(g x)`, the parent task directly computes it itself first, and then proceeds with the function `f`. Rather than commit firmly to a decision upon calling the future (*e.g.*, [19]), a lazy future provisionally in-lines the future by keeping enough runtime information so that the decision can be revoked later. This flexibility is maintained through permitting any processor (including, as we shall see why, the originating processor that is now computing the child thread) to steal `f`, the continuation, any time while `g`, the child, is being computed.

The implementation of the lazy future mechanism consists of three key sections, at the points of the future call, steal, and return. When a lazy future is called, a pointer to the continuation stack frame is entered on the future queue to denote a stealable point. Upon returning from the future, if the continuation has remained intact, the pointer is removed from the future queue and the continuation is resumed as for a normal procedure call. But while the future is in progress, another processor can steal the continuation by getting a pointer from the future queue, creating a placeholder for the future's result, and substituting a determination procedure in place of the continuation frame in the stack. The stealing processor then resumes the continuation by "returning" the placeholder to it. When the future then returns to the substituted determination procedure, it writes the placeholder with the returned value and enters the runtime scheduler, which will seek for another runnable task. The detailed specific implementations of lazy future, including considerations for shared- and distributed-memory machines, are described in [16].

In analyzing the cost of the lazy future, we see that the case of a future returning to continue its original parent is now quite efficient even in comparison to a regular function call. The only extra work required is for the maintenance of the future queue. Neither a future nor a placeholder object has to be created, and no scheduler intervention is involved. The work required for the case when a future is actually stolen is equivalent to that of a regular (or "eager") future: both a placeholder and a task, with states copied from the parent environment, will need to be created. The lazy future mechanism does nominally fix the scheduling order for the child and the continuation thread. Unless the programmer or the compiler swaps the two to enforce a particular order, the child thread will compute before the continuation by default.

Performance of the Lazy Future

We now examine how the lazy future mechanism works in the context of a program to reduce runtime overhead and present its performance in actual implementations. We will also show an example program for which the lazy future is very ill-suited. We will discuss what is fundamentally different in the new program and adapt a new strategy based on the original intent of the lazy future that will address the new scenario properly.

Consider computing the n-th fibonacci number in a two-branched recursion.

```
(define (fib n)
  (if (< n 3) 1
      (+ (future (fib (- n 1)))
         (future (fib (- n 2))))))
```

The structure of this program is representative of the general class of recursive-descent or divide-and-conquer algorithms. At each invocation for which $n > 2$, two future threads are produced.[7]

[7]Only one future is really needed, provided we know the evaluation order of the arguments of +. If the evaluation

With the eager future, $O(2^n)$ tasks will be created all together, as shown by the future tree in Figure 2-5, independent of the runtime resources (*i.e.*, whether there are 4 or 4000 processors). With the lazy future, there would be exactly as many lazy future calls and hence as many threads, but we expect many less actual tasks. The new threads diagram of threads in Figure 2-6 depicts future production for the same program. The executing processor descends down the path on the right, shown by the dark arrow, stepping into child threads produced by the first future call of the recursion. The first processor seeking work will, with the *oldest-first* load-stealing policy, sever the execution stack at the cut indicated and steal the oldest lazy future on the call stack, thus taking the left-most part of the tree for itself. The next steal will take the next oldest future on the stack, and get the middle piece. It is also possible that the second (or a later) steal takes a part of the left-most piece that was itself stolen. With three processors allocated for the job, initially only three tasks are created, as described. But as a processor finishes its share of work, it will attempt to steal from others' queues. The actual scheduling behavior is thus quite dynamic and hard to predict statically, but the net effect is to balance the workload. Tasks are created from available lazy future threads only when a processor is in need of work.

Since the lazy future call is incurred in in-line with every future in the program, its cost must be minimal. In the Alewife implementation, a lazy future call is 9 Sun SPARC instructions (or 14 cycles). Table 2.2, excerpted from [16], shows the actual execution time and the number of tasks created with eager and lazy futures. Execution time of a purely sequential version is also shown for comparison.

---

occurs left to right, we could write

```
(+ (future (fib (- n 1)))
   (fib (- n 2)))
```

and achieve the same parallelism with less overhead. Alas, the definition for T [47], as that for Scheme, specifies a random ordering of argument evaluation. In the discussions, we assume evaluation is left-to-right for clarity. The following is one way to ensure an exact evaluation order.

```
(let* ((a (future (fib (- n 1))))
       (b (fib (- n 2))))
  (+ a b))
```

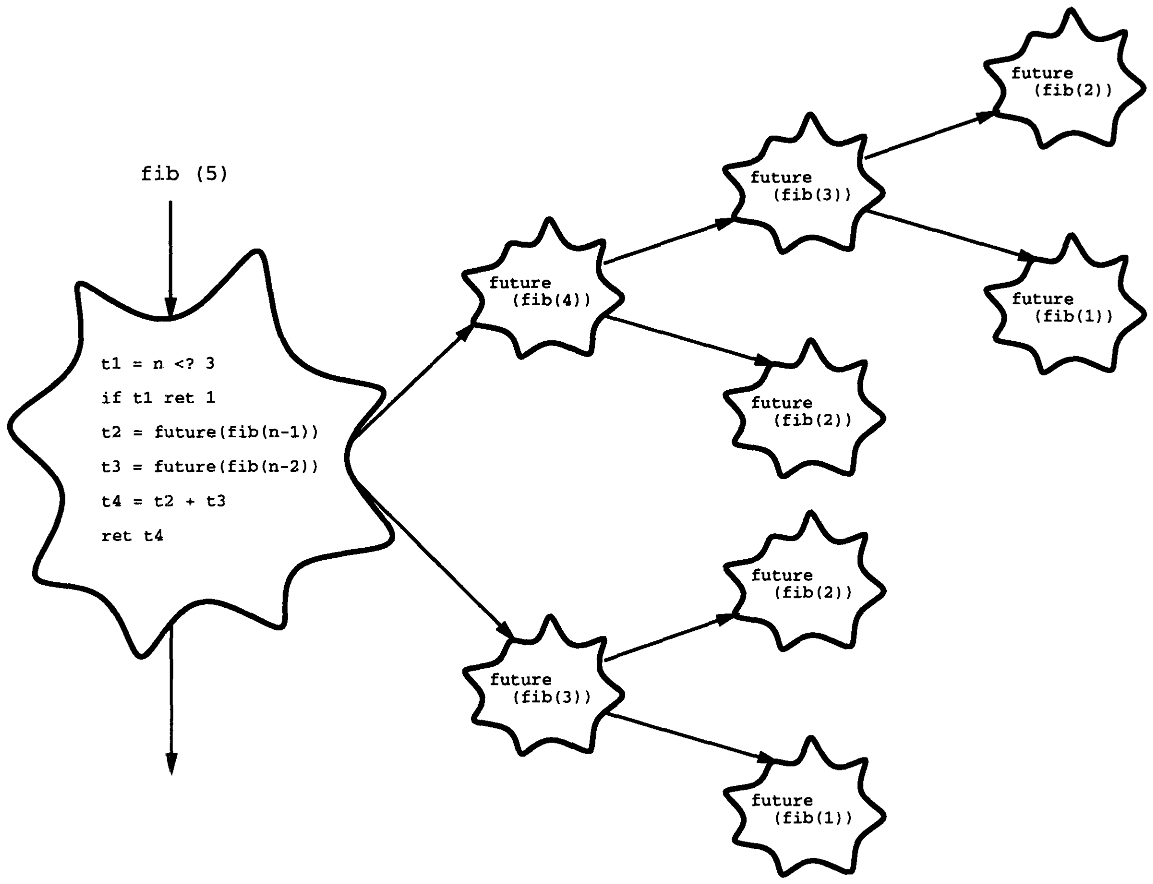


Figure 2-5: *Recursively Creating Futures for* `(Fib 5)`

Evaluation of Lazy Future Effectiveness

We see that lazy future effectively eliminates most of the unnecessary task creations. But why use it at all? Besides lazy future, there are the options to remove excess futures at either compile-time or runtime. The lazy future achieves two unique goals compared to removing futures via in-lining: the allowance for runtime load balancing and deadlock avoidance. Compiler in-lining, of course, does not provide load balancing below the granularity of the blocks of in-lined code. Although runtime in-lining allows a degree of load balancing, lazy future defers assignment of work to processors to a still later time. Unlike in-lining, the lazy future accommodates out-of-order dependencies by allowing suspension of a task blocked on unavailable data or other synchronization failures, thus preserving the independently-scheduled semantics of futures. Even compiler analyses cannot resolve all of these dependencies without unduly restricting the expressivity of the language. Where a future and its continuation thread contend for monitor locks, in-lining could certainly result in

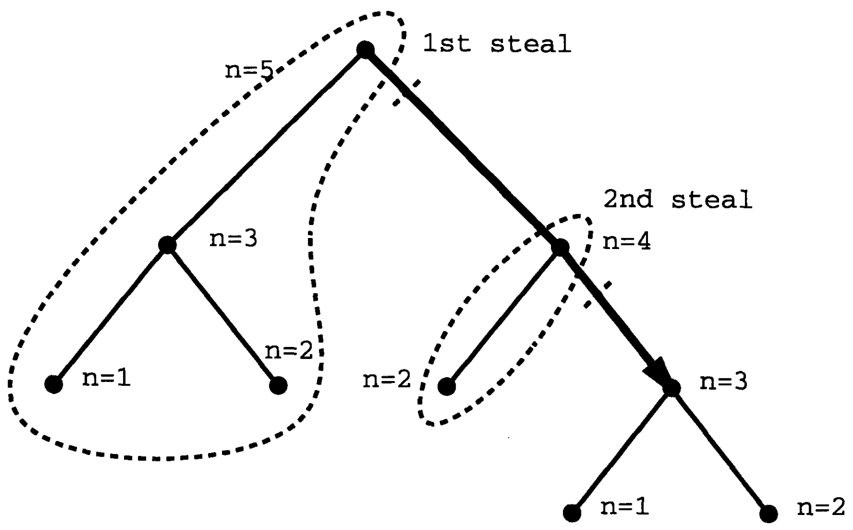


Figure 2-6: *Runtime Behavior from the Lazy Future Tree of* `(Fib 5)`

fib(16) Execution Time and Created Tasks				
Processors	Eager Future		Lazy Future	
	Time	# Tasks	Time	# Tasks
seq	62	—	59	—
1	881	3192	88	0
2	442	3192	46	6
4	223	3192	26	35
8	114	3192	17	104
16	60	3192	11	190

Table 2.2: *Performance of Eager and Lazy Futures*

deadlocking the program. Hence, it is legitimate for a processor's needing to steal work from its own lazy-future queue sometimes.

We have been fortunate in our choice of the example program used to validate the efficiency of the lazy future. Suppose we are to examine the behavior of another common routine: element-by-element summing of two arrays.

```
(doall (i 0 n 1)
  (set (aref c i)
       (+ (aref a i) (aref b i))))
```

where `doall` is syntactic sugar for a loop with each iteration of its body an independent parallel thread.[8] The structure of the program is representative of array computation, a popular use for

[8]

massively-parallel systems. As many threads will be created at runtime as iterations of the loop. The threads diagram shown in Figure 2-7 of the looping construct has a long backbone with shallow (single-deep) futures forked from it. The backbone is the unrolling path of the iterated loop.

With either type of future, we will have great difficulty choosing a good scheduling order for the future calls in this program. If a single processor executes the backbone thread of loop unrolling (as the dark arrow in Figure 2-7-A shows), it alone is responsible for creating all the loop threads. Each stealing processor will take only a single thread each time, regardless of the stealing policy; thus creating a tremendous bottleneck for timely distributing tasks around the system. Further, note that since every thread is converted to a task during the stealing process, the lazy implementation of futures achieves no gain.

Now consider the alternative scheduling order: proceed into the child thread first and leave the continuation (*i.e.*, the loop unrolling path) for another processor to steal (as the dark arrow in Figure 2-7-B shows). It results in each processor "peeling off" a loop iteration and passing the continuation around the system. The overhead of migrating a task is now included in the critical path of loop unrolling. Again, a slow serialization of the parallelism generation phase occurs and, even using lazy futures, every thread is turned into a task.

Slow parallelism generation can severely restrict the amount of parallelism actually realized, when it is unable to unroll a loop fast enough to keep all available processors busy, even when there is otherwise sufficient concurrent loop iterations to saturate a system. The problem is exacerbated by a relatively small loop body. A simple model, ignoring the accounting of which part of a task creation should be charged to the parent or the child task, predicts there will be unused processors when

$$\text{time to unroll an iteration} > \frac{\text{time to compute loop body}}{\text{number of processors}}$$

And the upper bound of actual parallelism is the lower of the number of independent loop iterations or

$$\frac{\text{time to compute loop body}}{\text{time to unroll an iteration}}$$

```
(define-syntax (doall (var init limit step) . body)
  `(do ((,var ,init (+ ,var ,step)))
       ((> ,var ,limit))
     (future (block ,@body))))
```

So, a straightforward application of future to denote concurrent iterations for looping structures cannot be supported efficiently by the same runtime mechanisms.

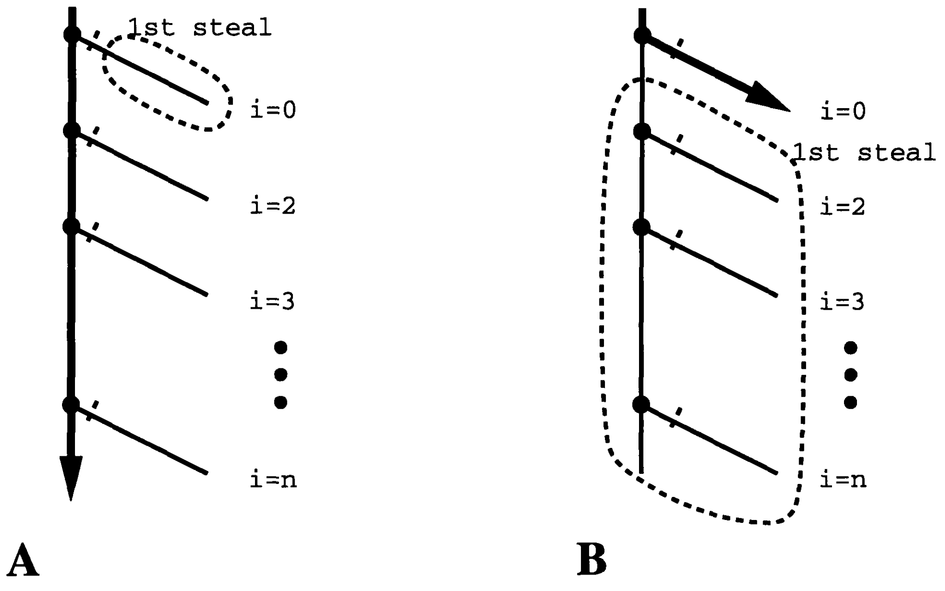


Figure 2-7: *Runtime Unrolling of Parallel Loop Iterations*

2.2.5 Summary and a New View of Parallel Loops

In retrospect of the present state of the programming and runtime systems, we note that they are well suited for control-parallel programs — ones in which the task boundaries are split along function calls such that the caller and the called functions are executed by different tasks. The inter-task communication consists mostly of the direct passing of argument and result values of the function call. We have described a simple case of `fib` whose structure represents a general class of recursive algorithms, and briefly presented its runtime performance. More cases of this variety have been tested and their similar outcomes documented in [16]. But because of the difficulties in predicting at compile time the behavior of control-parallel programs, such as the number of threads actually produced at runtime, most multiprocessor compilers require programmers to manage parallel tasks manually or to confine them to the iterations of the regularly-structured array-accessing loops. Those that devised compiler-based methods as solutions are difficult to realize in practice. They have assumed a customized (with respect to the user programs' specific algorithms) compiler [41], assumed statistical runtime data [46], or asked user to describe task structure and interaction using special language [30]. Except for committing the substantial respective efforts that would be required of each user program under one of these compiler optimization strategies, the environment presented in this chapter provides a painless alternative. By choosing to preserve it, we keep full and efficient support of the simple and general future-based parallel programming methodology. We now seek ideas to extended this environment to handle parallel loops efficiently, so that data-parallel computations can be accommodated as well.

It appears that the thread structure of an unrolling loop, when interpreted directly as defined, yields a long serial path with shallow threads forked out therefrom that is unsuited to parallel execution. An alternative view of the above `doall` loop is as a macro operator that simultaneously forks out all the iteration threads, each with its unique induction variable value. To accomplish further along this line that would still incorporate the same runtime system developed thus far, the macro operator has to be implemented in a way such that these goals are realized:

- The parallelism generation phase is itself parallelizable.
- The macro operator can be designed so as to take advantage of the lazy future mechanism.
- The amount of work taken from a processor by a load steal is roughly proportional to the amount of work found at that processing node, as was done for the recursive fibonacci function.

If special forms, like `doall` with constant bounds, are required explicitly from the programmer, the compiler can directly build a binary lazy future fan-out tree [9] with leaf nodes of iteration threads, as shown in Figure 2-8. This is in fact the model that will be expanded greatly in the next two chapters on the work the compiler accomplishes. Generalizations beyond strictly-independent loop iterations and partitioning and alignment of iterations into detachments of work that can be directly dispatched to processors will be discussed presently.

[9]It is binary because a future forms a two-branched task fork.

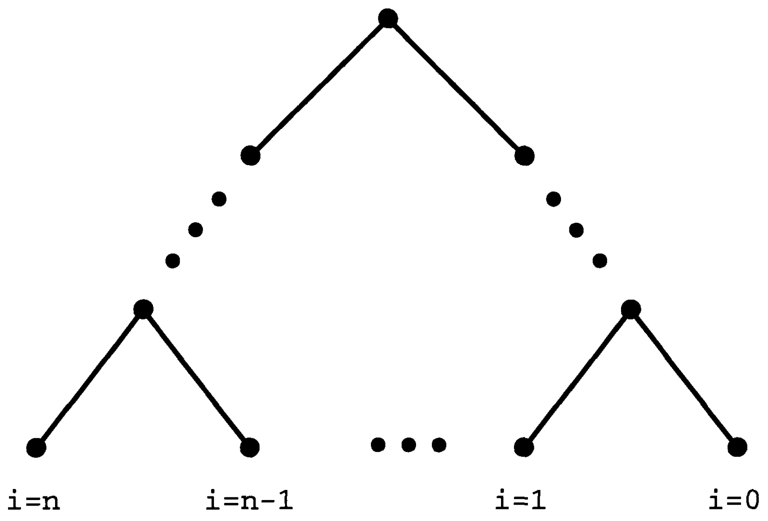


Figure 2-8: *Unrolling of Parallel Loop Iterations with a Future Tree*

Chapter 3

Compile-time Program Transformations: An Example

In this chapter and the next we focus on the compiler parallelism management phases of the Revocable Thread Serialization framework. The primary intention of the compile-time processing is to use information from static program analyses to improve the performance of data-parallel computation. In this chapter we first describe the top-level structure of the compiler itself. We then follow the progress of a short sample program, examining the transformations that it undergoes at each compiler phase and motivating why they are needed. With an understanding of what overall effects are needed and how they are coordinated among the compiler phases, the implementation considerations of each of the phases to be covered in the next chapter can then be better grasped in their proper context.

3.1 Structure of the Compilation Phases

As we concluded at the end of last chapter, the purpose of the compiler parallelism management phases of this framework is to restructure the input program to assure efficient execution of various looping constructs associated with data-parallel computations only. To this end, the following specific goals have to be met:

- Parallel loops and data arrays are partitioned for an expected number of processors. The parceled work is directly distributed among the processors so that maximal system resources can be deployed quickly.

- Since these loops often take data from arrays generated by other loops and produce arrays to be used by succeeding loops, all communication-related loops and arrays are aligned along their partitions to minimize inter-processor data transfer.

- Generate future fan-out tree control structures of the type that had worked well in the last chapter under the runtime management scheme.

The program restructuring strategy to implement the RTS framework is incorporated into the entire compilation process as a source-to-source transformation package called Scepter. It functions as the front-end to Orbit, a Mul-T compiler that knows about sequential language constructs and the parallelism generation primitives to create tasks on some specified processor or to demarcate the stack frame with lazy future steal points. Orbit solves the problems related to machine code generation for a target processor, such as call linkage, stack frame maintenance, register allocation, *etc.*Scepter performs scalar, alias, escape, dynamic, and index analyses on the source program, identifying data references and collecting looping and other control structure information. These are used to build a dynamic thread/data communication graph (WAIF-CG) consisting of threads and data nodes and the communication edges among them.

The compilation process is represented in Figure 3-1. The derivation of WAIF-CG from the source, as well as the internals of WAIF-CG nodes and edges, are described in the Appendix and defined in [32]. The graph records dynamic instance information that Scepter propagates from known iteration constructs to annotate the statements in the loop bodies. Using this information Scepter assesses how many times each textual statement is executed and therefore the number of runtime threads each `future` creates. An important distinction to note is that, although the program analyses and transformations in Scepter seem at times similar to the standard static global analyses and transformations, the process can tolerate any lack of or undecidability in the static information. We will later see how none of the static information is essential to the process, because of the bottom-up compiler program-transformation strategies and the flexible runtime support. To simplify the problem somewhat, we are treating partitioning, alignment, and scheduling as mostly independent phases and present them in that linear order. As most compiler designers know all too well, the mutual dependence and thus the ordering of the transformation and optimization phases of a compiler are rarely non-circular.

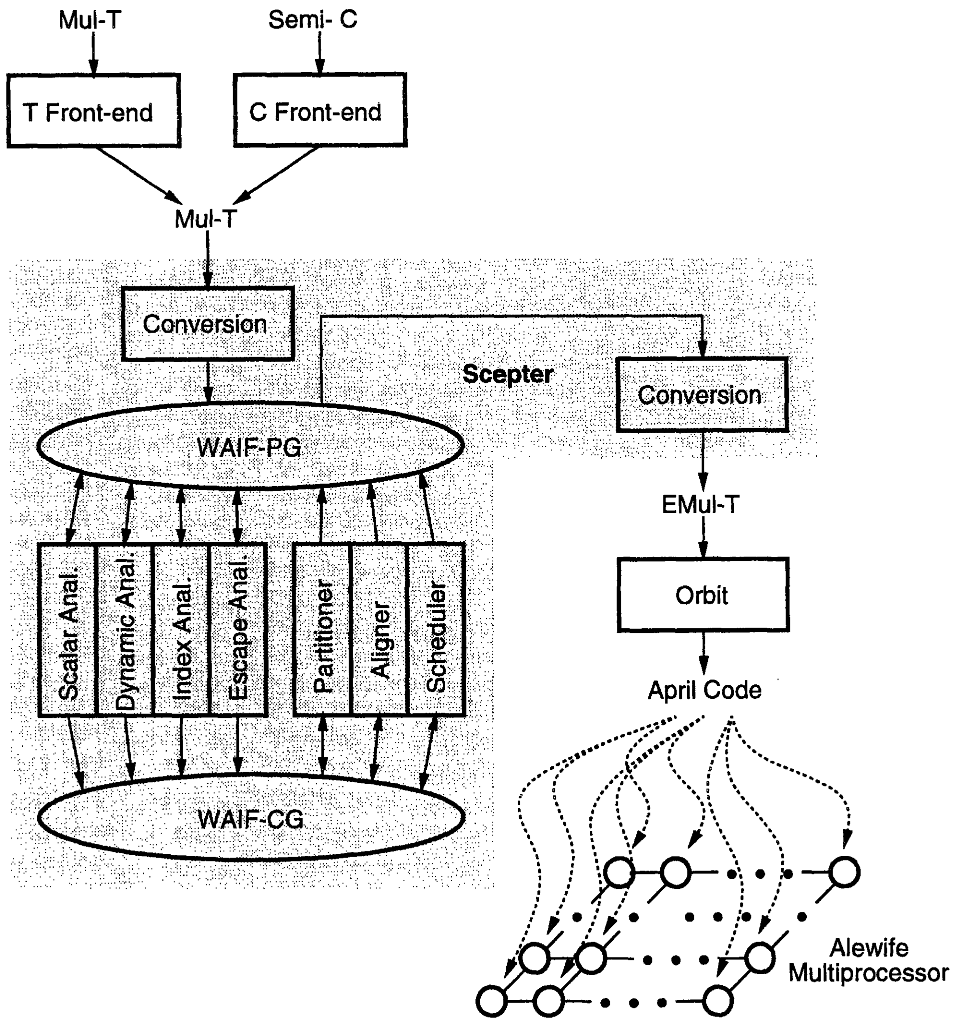


Figure 3-1: *Program Parallelism Restructuring and Compilation Process*

3.2 Compiler Walk-through of an Example Program

The program in Figure 3-2 serves as the example for demonstrating what transformations occur at successive Scepter stages. In this program, x is a J-structure (array) of size 7 created by `make-jstruct`. Its cells x_0 through x_5 are set by `SetCells`, while `SetBounds` defines some boundary values that are stored in y. `RefCells` uses successively the boundary value in y_0, and then x_1 through x_5, followed by the other boundary value $y_1 + x_0$, in computing further values. J-structures are used here instead of plain arrays because threads in `SetCells`, `SetBounds`, and `RefCells` are all concurrently executing, according to the semantics of the future. Without the data-synchronization J-structure, it would not be semantically incorrect if `RefCells` read a value from an array cell before it was set by `SetCells` or `SetBounds`. We note that the numbers of loop iterations and array cells are made deliberately small to keep the illustrations clear. In real programs, and thus in the compilation steps and in the code generation, all considerations assume that these numbers will be very large (*e.g.*, we will in fact build a generalized loop to execute what is ostensibly one or two iterations rather than enumerate them in-line.)

```
(define (Example)
  (let ((x (make-jstruct 7)))
    (SetCells x)
    (let ((y (SetBounds)))
      (RefCells x y))))

(define (SetCells x)
  (doall (i 0 6 1)
    (set (jref x i) (f i))))

(define (SetBounds)
  (let ((z (f 0))
        (y (make-jstruct 2)))
    (future (set (jref y 1) z))
    (future (set (jref y 0) (/ z 2)))
    y))

(define (RefCells x y)
  (future ( ... (jref y 0)))
  (doall (i 0 4 1)
    (... (jref x (+ i 1))))
  (future ( ... (+ (jref y 1) (jref x 0)))))
```

Figure 3-2: *Compilation Demonstration Program*

After the static program analyses described in the Appendix, a thread structure is extracted from the program text. A simplified representation of the thread graph is shown in Figure 3-3. The internal nodes are future forks; the leaf nodes are parallel threads; the solid lines are the program control flow; the dashed lines the data communications, which come from the possibility of accesses to the same data locations — rather than data dependencies. Since the "def" and "use" points in the program are concurrent, only runtime synchronizations (none for plain arrays) enforce the actual directions of data propagation.

The program starts at top center of the graph and exits at bottom center. Three subroutines are called, each of which creates parallel threads. The definition of `doall` is as given in the last chapter, although each parallel loop is now represented by a single node with many out-edges in the thread graph instead of a long unrolling thread. Note that data array creators are recognized and given its own special thread node. As we will see, in these transformations they represent the shape, size, and location of the data memory they create.

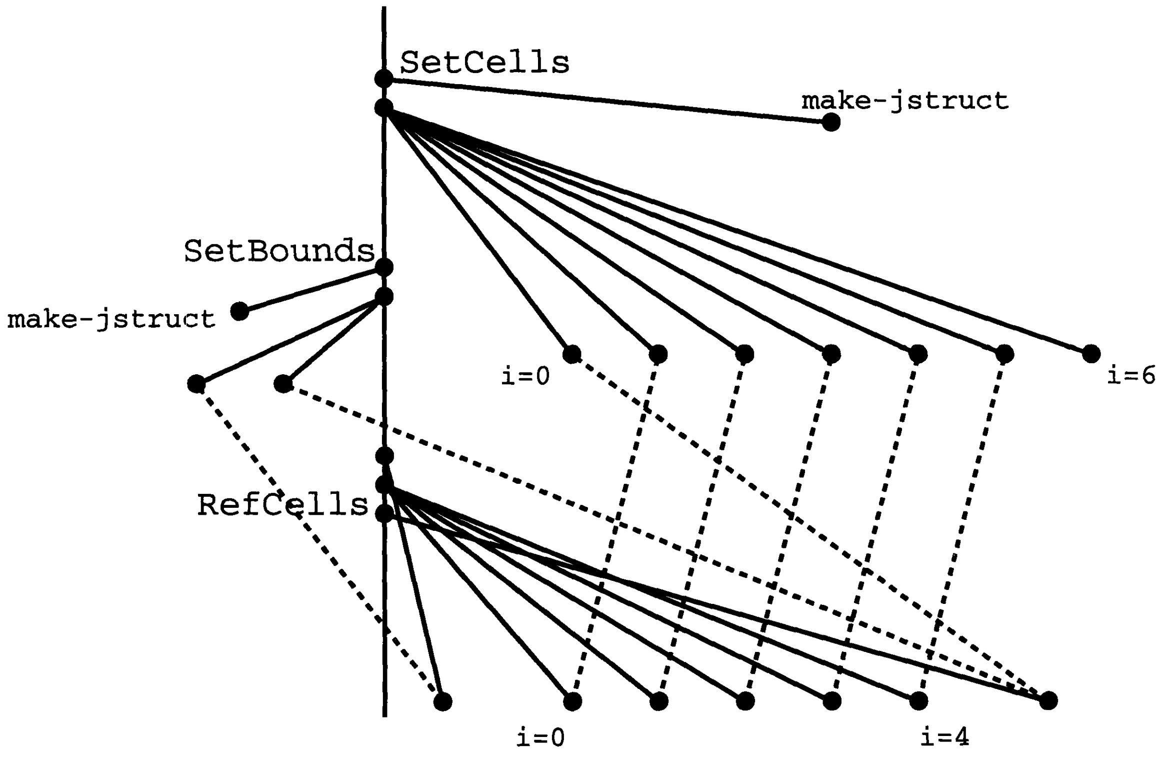


Figure 3-3: *Initial Thread Graph for Example Program*

3.2.1 Compiler Loop and Array Partitioning

The partitioning process forms groups of concurrent loop iterations and segments of data arrays. The purpose of this and the subsequent alignment transformations is to restrict all threads and data within a partition, and eventually within a detachment, to be dispatched to run on the same processor node; therefore, within a partition concurrency is limited, but locality of access is improved. Also, it reduces the number of initial tasks that will be created for the loops, and the compiler will further be able to pre-schedule intra-partition (and intra-detachment) threads. The chunk size of the thread groups and array segments is controlled by the policy.

The partitioning phase, given the thread graph and a parameter for the number of processors to partition for, returns a graph shown in Figure 3-4 as partitioned for four processors. Data reference patterns are examined to decide the partition shape. The array creators have been partitioned into segments, and the threads in each loop of `SetCells` and `RefCells` have been grouped into partitions, as indicated by dashed ovals. Note that the grouping of the iterations of these loops are different — the iterations pairings do not have the same starting offsets because this produces the more optimal access locality of the array.

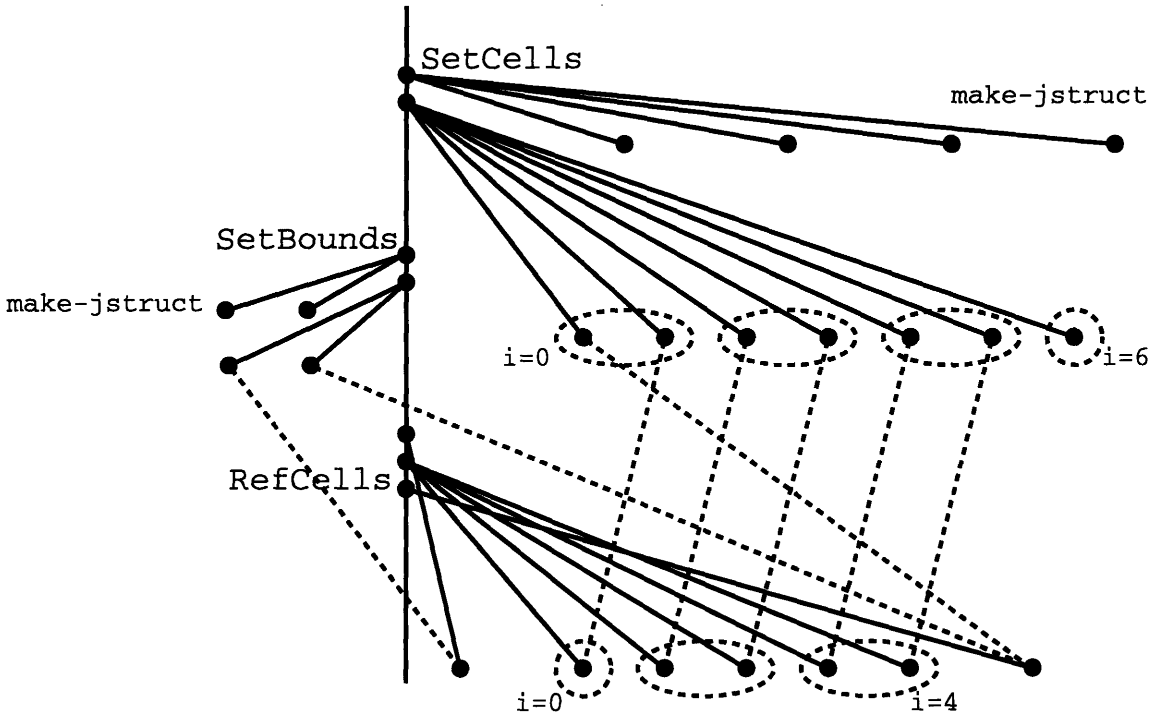


Figure 3-4: *Partitioned Thread Graph for Example Program*

3.2.2 Compiler Thread and Data Alignment

The alignment phase matches the communication-linked partitions from different loops and array creators, as well as independent threads, by placing them into the same detachment as shown in Figure 3-5 with the dashed bubbles. We have labeled the detachments with distinct numbers. Note that no more than one partition from each loop or array creator is placed into each detachment. This is intended to prevent the partitions from collapsing, to ensure enough detachments remain for all expected processors to get an initial work allocation.

3.2.3 Compiler Thread Serialization

After partitioning and alignment, threads and arrays are put into detachments, leaving the runtime system to sort out which threads are actually executable. Without further static serialization, each

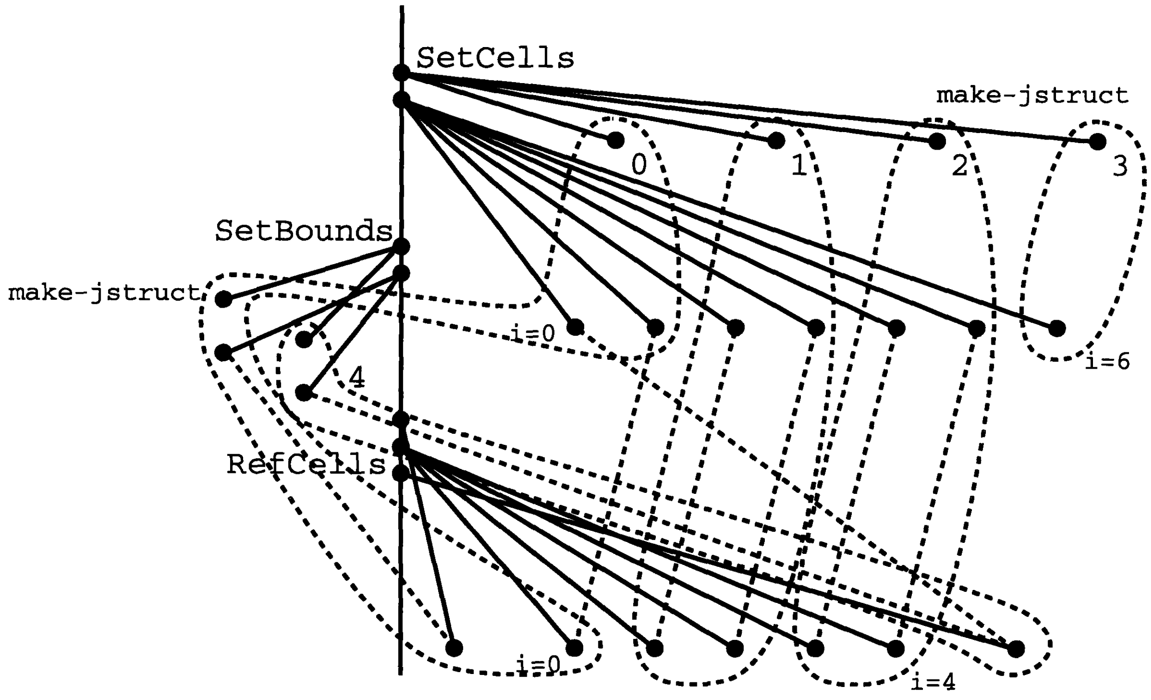


Figure 3-5: *Aligned Thread Graph for Example Program*

thread will be placed on its compiler-selected processor and executed there as an independently scheduled runtime task. The resulting fine granularity will incur high runtime overhead from creation of many small tasks that can also cause the runtime scheduler to thrash about much, possibly holding up resources long before they are required, and executing unnecessary synchronization code. These problems were discussed in Section 2.2.3. The tasks should be lengthened as much as possible via static serialization of threads within a detachment.

The compile-time thread scheduler is given a list of detachments. The default ordering for the threads of each detachment is to follow their original serial program execution order, as we have argued in Chapter 2, constructing a top-level control thread that invokes each of them in the desired order. Loop iteration threads, as we have pointed out, need to be serialized in a more particular fashion. Another set of rules constraining the serializability of detachment threads must be observed as well. Applied to our example program, it is the evaluation of `(f 0)` in `SetBounds` that prevents a complete serialization of the detachments. Function `f` has be called and variable *z* set after the code for `SetCells` is forked out but before the rest of `SetBounds` and `RefCells`, both of which need access to *z*, can be started. The general rules and transformations regarding this scenario will be described more fully in the next chapter, under the section on scheduling of random parallel threads. The final serialization thus has a "break" in each of the parallel detachments, creating two brigades for each detachment as shown in the diagram in Figure 3-6, and in abstract code in Figure 3-7. That is, a set of parallel tasks are spawned to create the array partitions and set their initial values; without first requiring their completion, `(f 0)` is evaluated for z and another set of parallel tasks are spawned to set boundary values and to read out the array cells for further work.[1]

Note that the placement of arrays is realized by having a chosen processor execute a local make-array type of system call of the proper segment size. The `do-parallel-tasks` form creates one task on each processor executing its code body, with variable mt bound for each task to the task sequence number (*i.e.*, from 0 to 3). `align` does the alignment mapping to allow a task to execute the partition that is assigned to its brigade. The value of m returned tells the task which partition number it is to execute. The compiler inserts in each `align` expansion its static alignment map. This mechanism will be described in Section 4.2 on alignment. `doall-segment` is a binary future tree macro that will execute at its leaf nodes the iteration range assigned to the loop partition. The detailed construction of this future tree will be discussed later in Section 4.3.1. The variable l is bound to values that step through the assigned range (in the loops found here, two iterations with partition offsets of 0 and 1. The different offsets for the loop partitions was pointed out in Section 3.2.1). Array index is generated by `j-addr`. These are necessitated by the loop iteration- and array index-space conversions to and from the partition-spaces introduced through the m and l variables. Also serving this purpose is `index-i`, which converts m, l to the original i value. These macros implement the space-mapping translations to be described in Section 4.1.1.

3.3 Summary

This chapter has described in functional terms the compiler end of the Revocable Thread Serialization framework, and demonstrated the intended transformations performed at each compile-time phase. We showed what the structure of the output code would look like. When executing, this code will spawn two sets of four tasks, concurrently with the main program thread, running from its own task that will be sharing Processor 0 with two of the other eight tasks. In addition to this runtime crowding, note also that the amount of work in the parallel brigades is slightly uneven. In a real

[1]Were we to analyze and re-serialize the intra-thread instructions as well, and if `f` is determined to be side-effect and internal-state free, `(f 0)` could be moved to before the code for `SetCells`, thus allowing an uninterrupted serial sequence in each detachment.

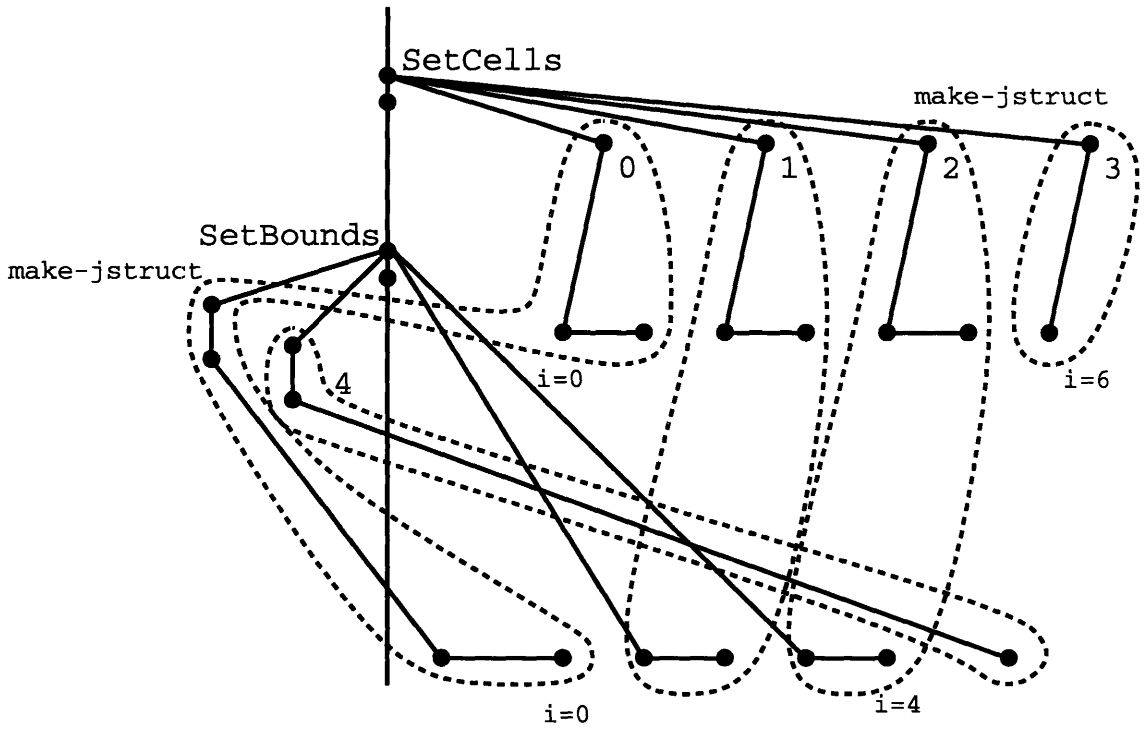


Figure 3-6: *Serialized Thread Graph for Example Program*

situation, these differences would be dwarfed by the amount of work from the partitioned loops, and by the uneven hardware operation and runtime scheduling. And, of course, in any case the static arrangements will still be subjected to runtime load balancing.

```
(define (Example)
  (let ((x (make-jstruct 4)))       ;; make dope vector

    (do-parallel-tasks mt           ;; put on each processor
      (let ((m (align mt)))         ;; partition alignment mapping macro
        (future (set (jref x m) (make-jstruct 2))) ;; create array segmer
        (doall-segment 1 2 0        ;; loop through iterations in partitic
          (set (j-addr x m 1) (f (index-i m 1))))))

    (let ((z (f 0))
          (y (make-jstruct 2)))

      (do-parallel-tasks mt
        (cond ((= mt 0)             ;; do aligned individual threads
               (future (set (jref y 0) (make-jstruct 1)))
               (set (j-addr y 0 0) z)
               (future ( ... (jref y 0 0))))
              ((= mt 1)
               (future (set (jref y 1) (make-jstruct 1)))
               (set (jref y 1 0) (/ z 2))))
        (let ((m (align mt)))       ;; do aligned loop partitions
          (doall-segment 1 2 1
            (... (j-addr x m 1) (+ (index-i m 1) 1))))
        (if (= m 1)
            (future ( ... (+ (jref y 1 0) (jref x 0 0)))))))))
```

Figure 3-7: *Abstract Output of Transformed Example Program*

Chapter 4

Compile-time Program Transformations: Theory and Implementation

We saw from the last chapter how a program undergoes a set of compiler transformations and what the final output code looks like in the abstract. In this chapter we explore more details of what are under the abstractions in that code listing, how they are actually constructed, and some performance tuning issues. But first Section 4.1.1 explains the implementation of loops and arrays in the distributed processor and memory space, because that has a pervasive impact on the resulting code and on its performance, even though the scheme itself is not particularly novel and merely achieves the desired low-level function. Understanding the scheme is also essential to seeing how the results of the compiler parallelism restructuring is realized in the output code. Thereafter, the three main compiler back-end parallelism transformation processes are described (in Sections 4.1.4, 4.2, and 4.3), as well as some policy alternatives used for tailoring the framework for specific input conditions and system parameters (in Section 4.4). The effectiveness of these strategies and processes will be covered in the next chapter. As noted in the last chapter, the front-end program analyses and the WAIF intermediate form building phases of Scepter will be skipped here. A description of them is found in the Appendix.

4.1 Partitioning of Parallel Loops and Data Arrays

4.1.1 Abstractions for Distributed Loops and Arrays

This section describes a scheme to support the distribution of numerically-indexed instances of array elements and loop iterations automatically across a range of processing and memory units. Although hardware interpretation of global memory addresses has been assumed, we only rely on it as a low-level address-space resolution and memory access mechanism, not as an implementation of storage allocation policies. Thus there is no hardware reconfigurable memory blocking or low-order-bit address interleaving, as on some shared-memory machines [37]. Instead, a global memory address is simply a concatenation of a processor node number prefix and the linear offset within the global memory block at that node. The compiler and the runtime system together implement whatever storage mapping is desirable for the memory access demands of a particular program. This design choice simplifies the hardware configuration and ultimately provides more flexibility to accommodate a diversity of storage mapping policies. Thus, references to distributed or multi-dimensional arrays must be re-expressed as native linear vector references with the necessary processor node crossings made explicit through coordinate translations. A similar problem occurs in partitioning and distributing a loop for multiple processors, fragmenting the contiguous iteration space generated by the sequence of values of its induction variables. These considerations apply to the following program constructs:

- Array creations (those that are accessed from parallel loops or otherwise too large to fit on a single node) modified to distribute to multiple nodes.
- Parallel loop control and the induction variables modified for distribution to multiple nodes. These variables often also form the index expressions of distributed arrays.
- Array elements references modified to access those distributed arrays.

4.1.2 Distributed Iteration and Array Spaces

At the core of these translations is the maintenance of three distinct spaces by the compiler, while giving the illusion of only the one seen in the source program. For this discussion we confine loops and arrays to be two-dimensional, although all equations derived are for any number of dimensions.

We call the original loop iteration space from the program the (i, j)-space, where the programmer presumably used i and j as the loop induction variables and in the array index expressions. The implementation of partitioned loops and segmented arrays described in this report creates two more spaces: the partitioned loop iteration space we call the (m, l_i, l_j)-space and the segmented array index space the (g, o)-space. Figure 4-1 shows the relationship of these instance spaces and how a specific program coordinate (x, y) (*i.e.*, loop iteration (x, y) which accesses an array's (x, y)-th element) is mapped within these distributed spaces. m and g range through valid processor node numbers; $(l_i \times l_j)$ spans the subrange of loop iterations assigned to each partition, and o spans the segment of an array allocated to each partition. The compiler's space translations to be described in the next section will thus ensure that processor 4 will execute the iteration (x, y) and that its reference to the array's element (x, y) will be directed to processor g's local memory at address o corresponding to the mapping of (x, y) in the (g, o)-space. The space mapping used here and the partition strategy used in a following section only cover rectangular blocks, which as fairly general and efficient — extensions to others such as linear or blocked stripes are trivial, whereas non-rectangular shapes, such as hexagons, are significantly less efficient to compute.

4.1.3 Application of the Inter-space Translations

Now we show how the space translations are realized in the compiler program transformations. Inspecting the output code in Figure 3-7, we see that the space translation-related program transformations cited in the list above are buried under the abstraction of procedure calls. A distributed array is shown formed by creating a dope vector of one cell for each data segment. A set of parallel tasks is then spawned on the participating processors to create a local segment on each node. In this way, each node maintains its own storage allocation, and the addresses of the new segments are entered into the centrally-located dope vector, whose address is given as the reference for the entire array. During execution, the dope vector values, which are written once and thereafter strictly read-only, required by each node is read once and locally cached, so there is not the hotspot problem that might otherwise be anticipated.

In `doall-segment`, a loop starts at the conformal offset for the (s_i, s_j) position as determined by m, and generates locally its own series of (l_i, l_j). When required, the original (i, j) coordinate can be reconstructed with $(s_i + l_i, s_j + l_j)$. This is what `index-i` does. As demonstrated by the (m, l_i, l_j)-space example in Figure 4-1, a partition tile may be truncated short by the actual

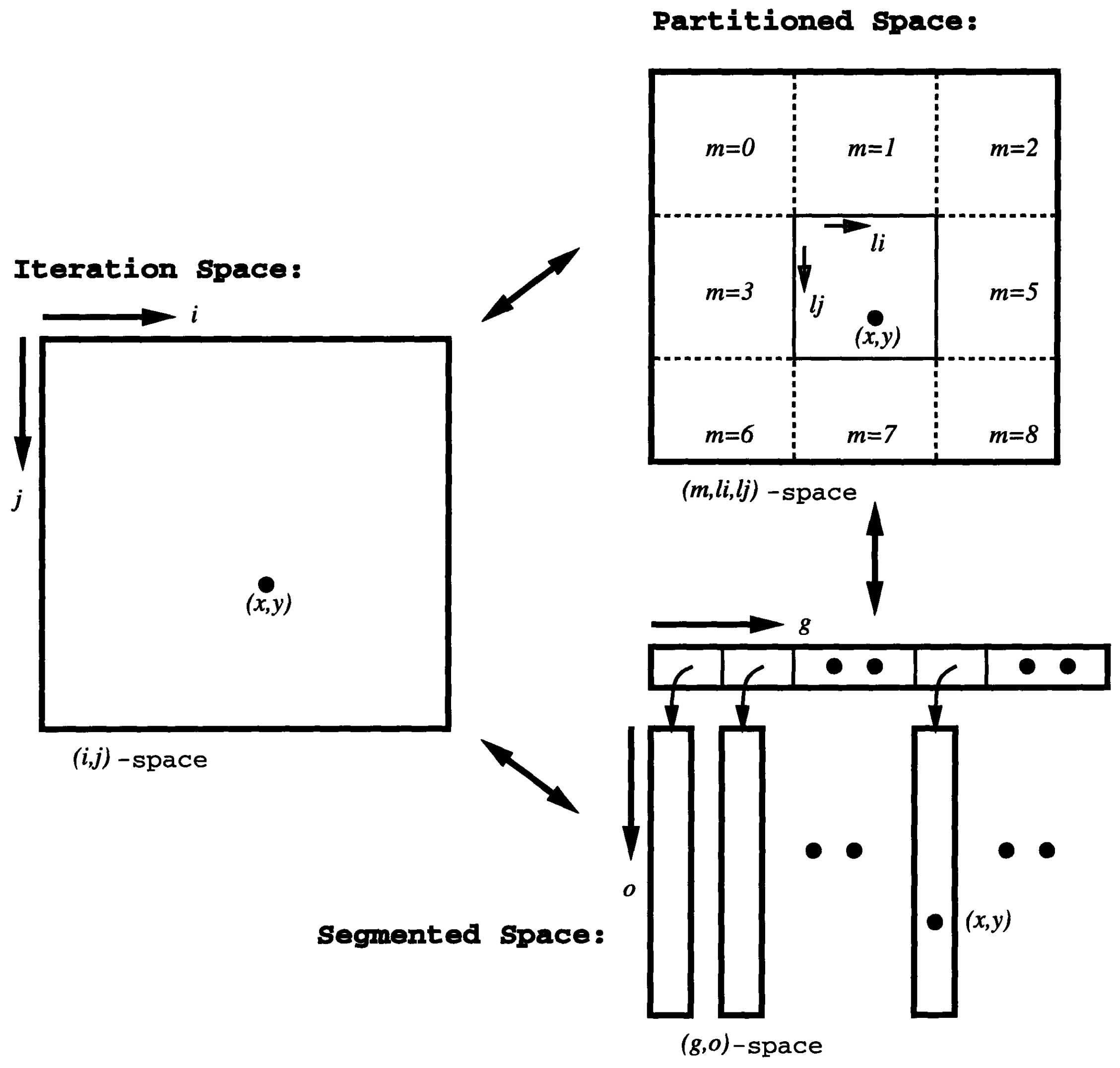


Figure 4-1: *Space Mappings and Translations*

loop bounds. The v-dimensional starting iteration coordinate $(s_v, \ldots, s_k, \ldots, s_0)$ of partition m is determined by

$$s_k = \max(init_k, \left\{ \begin{array}{ll} 0 & \text{if } n_k = 1, \\ w_k \text{mod}(m, n_k) & \text{if } k = 0, \\ w_k \text{mod}(m/(\prod_{d=0}^{k-1} n_d), n_k) & \text{otherwise.} \end{array} \right\})$$

where $init_k$ is the programmed starting value of the k-th induction variable[1], w_k is the width of the partition in the k-th dimension, n_k is the number of partitions to tiles the k-th dimension of the iteration space, and $\text{mod}(a, b)$ is modulo b of a. The v-dimensional ending iteration coordinate

[1]See `doall` definition on Page 44.

$(e_v, \ldots, e_k, \ldots, e_0)$ is determined by

$$e_k = \min(limit_k, s_k + w_k)$$

where $limit_k$ is the terminal loop bound.

Distributed-array index expressions must be translated from either the (m, l_i, l_j)-space, if found with induction variables of a distributed loop, or the (i, j)-space otherwise, to its (g, o)-space coordinate. `j-addr` is seen in Figure 3-7 in place of such an index expression translation for a J-structure reference. The (g, o) coordinate for a written index expression $(i_v, \ldots, i_k, \ldots, i_0)$ is

$$g = i_0/w_0 + \sum_{d=1}^{v}(\prod_{e=0}^{d-1} n_e)(i_d/w_d)$$

$$o = \text{mod}(i_0, w_0) + \sum_{d=1}^{v}(\prod_{e=0}^{d-1} w_e)\text{mod}(i_d, w_d)$$

Because on many processors, the Sun SPARC being among the worst, integer multiply and divide operations are very costly, often being much slower even than their floating-point equivalents, the coordinate translations must be streamlined not to impose a large bias on measured performance of the parallelized code. In all our benchmarks, the compiler rounds off partition and segment sizes to powers of two, and the code generator then substitutes cheap binary shifts and bit masking operations for the multiply, divide, and modulo functions. Also, each array access currently undergoes a full coordinate translation, most of which could be manually or automatically simplified with standard compiler optimizations for constant expressions or common sub-expression.

Because the compiler can be invoked incrementally in a T system and because in reality there are many situations where we simply cannot analyze and trace the identity of all the array references in an entire program, a means must be provided for addressing the distributed arrays when the compiler is unable to substitute the proper address translation expression. Through a compiler analysis on escaping variable values[2], (*e.g.*, among the arguments passed in calling a foreign procedure,) partitioned arrays that exit a compiler-controlled program context are noted, and their descriptor headers, where type and size information normally reside, annotated with an index translation function. On the accessor end, an array reference whose origin cannot be identified through the alias and dataflow analyses will be compiled to examine the descriptor for an index translation function. When the look up fails, the array reference would assumed be for a standard contiguous array.

[2]See the Appendix.

4.1.4 Partitioning of Parallel Loops and Data Arrays

The function of the compiler partitioning phase is to group the parallel loop iterations and to segment the data arrays that these loops access according to the expected number of processors allocated at runtime and to the array reference traffic obtained through static program analyses. Automatic partitioning of loops is a much-studied problem for parallelizing compilers with many ready solutions in the field [25, 42, 24, 52]. We have adopted and implemented a simple strategy suggested in [24]. Our relatively modest goals are to produce no more initial tasks to compute the loop than the expected number of processors and to improve processor utilization through maximizing reuse of entries in the local data cache. To simplify loop induction variable generation and the in-line translation computations for the array accesses, all partitions for each loop or array are constrained to be rectangular and of identical size (except at the boundaries). They must of course also "tile" (*i.e.*, completely cover but without any overlap) the entire iteration or index space.

Since the p partitions must tile the entire iteration or array space of area A, each one will have an area of A/p. The decision of what length and width a partition should be becomes a matter of minimizing the communication cost, considering the effects of local data caching. For example, computing $a_{i,j} = a_{i,j} + a_{i,j+5}$, as in the loop

```
(doall (i ... )
  (doall (j ... )
    (set (aref a i j) (+ (aref a i j) (aref a i (+ j 5))))))
```

would gain from a partition of maximum length along the j dimension. This is because, to compute a partition, in addition to fetching the A/p array values under the partition tile, it also has to fetch past the edge extra values along its j dimension. Therefore, lengthening the j dimension will reduce the number of rows (*i.e.*, in the i dimension) that will be fetching past its end, thus minimizing communication with a neighboring partition. There is no such boundary crossings along the i dimension, thus no point in attempting to reduce its boundary length.

To generalize this problem, we consider in detail the case of

$$a_{i,j} = a_{i,j} + a_{i+r_i,j+r_j}$$

Figure 4-2 shows in a combined iteration and array space a partition and its boundary-crossing accesses for $r_i = 1$ and $r_j = 2$. The access area outside of the partition is

$$A_e = w_i r_j + w_j r_i - r_i r_j$$

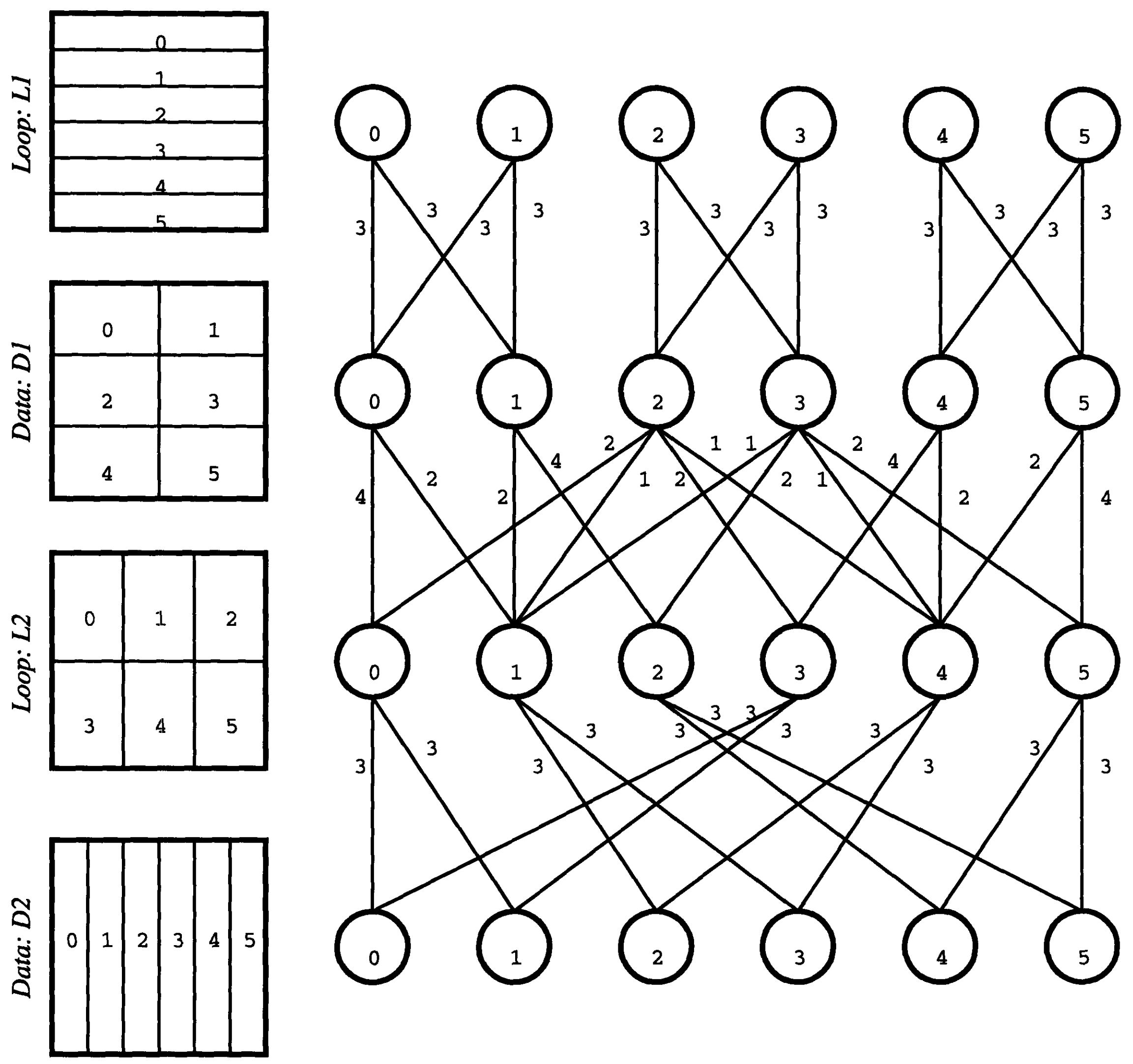


Figure 4-3: *The Loop and Data Partitions Communication Graph*

E of each loop iteration partition, which are $[\ldots, f_c(S), \ldots, f_0(S)]$ and $[\ldots, f_c(E), \ldots, f_0(E)]$, among the array partitions, as shown in Figure 4-4. The next step is to calculate, for those array partitions covered, the region of overlap in each. This corresponds in the figure to the gray region cast by loop partition 1 inside the array partitions 0, 1, 3, 4, 6, and 7, representing the proportion of communication of the loop partition with each of these array partitions induced by the index expression. Along each dimension c, the overlap is computed as from $\max(s_c, \min(f_c(S), f_c(E)))$ to $\min(e_c, \max(f_c(S), f_c(E)))$, and the total overlap volume is the product of these along all dimensions. This volume, multiplied by the weight of the WAIF data edge associated with the program array reference, is added to the weight of the communication edge between the loop and array partitions.

shapes of loop partitions and array segments with non-uniform, possibly overlapping, and inexact communication patterns can all be accommodated. This degree of flexibility is necessary especially since a consequence of our choice to perform partitioning and alignment as two independent phases is that the partitions to be aligned often will be of different sizes or even shapes.

The input to be aligned is a set of partitioned loop and array nodes representing an initial thread grouping and data segmentation. We first compute the weighted communication edges among partitions from the individual edges between array referencing program statements and the array objects, observing the repetition factor from their enclosing looping structures. Then, the most-accessed array partition from a loop partition are assigned to the same detachment as the loop partition.

4.2.1 Building the Partition Communications Graph

Consider the example in Figure 4-3, whose left-hand column pictures the assumed input items: some partitioned 2-dimensional loops, $L1$ and $L2$, and data arrays, $D1$ and $D2$. Each has been partitioned into 6 blocks. For simplicity, we assume that the index expressions for the $D1$ and $D2$ references in $L1$ and $L2$ are all identity functions, meaning that $L1_{ij}$ writes $D1_{ij}$, which is then read by $L2_{ij}$ to compute $D2_{ij}$, for valid range of i and j.[3] To the right in the figure is a graph of the partition nodes of the original loops and arrays and the weighted communication edges among them induced by the identity array index. The nodes are numbered corresponding to the row-major-ordered partition numbers.

The communication edge weight is computed by projecting a loop iteration partition onto the array space. Since we use rectangular partitions, two coordinates are sufficient to characterize a partition under any linear transformation. Consider them the bounding boxes for the set of points in the rectangular region, R: $S = (\ldots, s_d, \ldots, s_0)$ and $E = (\ldots, e_d, \ldots, e_0)$, where s_d and e_d are respectively the minimum and maximum values along dimension d of all the points contained in R. A linear transformation t on S and E, $S' = t(S)$ and $E' = t(E)$, will preserve S' and E' as the bounding box for the set of points in $R' = t(R)$.

For each array index expression in the program, $[\ldots, f_c(X), \ldots, f_0(X)]$, where each $f_c(X)$ is of the form $x_d + k$ with constant k, we locate the array index space mapping for the S and

[3]This particular set of indices would not induce the partitioning outcome shown, but that is not our concern here.

partitions. They can easily be substituted in place of the supplied partitioning module (with a corresponding change in the code generator for space translations for non-rectangular partitions) to accommodate programs with more complex array index expressions. The present partitioner serves its purpose in completing and for validating the framework.

4.1.5 Partitioning without Sufficient Information

What if the loop or array bounds or the number of processors are not known statically, or that the compiler cannot resolve some of the array index expressions? The bottom-up partitioning strategy just described can still produce usable code. The compiler takes a default number (which may be user specified) of iterations to place into a partition. If the array index expressions are known, the partition shape ratio can be computed; otherwise, it defaults to a square. The follow-up procedures to align for detachments and serialize for brigades can be applied to these partitions, and the final code will simply incorporate expressions to determine dynamically the initial number of tasks to create.

Since the code for each partition already performs dynamic end-of-loop test, as described for the function of `doall-segment` in Section 4.1.3, generating enough initial tasks is all that is needed. If there are more processors than initial tasks, some of the tasks will be immediately broken up by the idle processors. If there are more initial tasks, then each processor will have several loaded in its task queue. The default partition size is chosen to yield a reasonable initial task granularity depending on whether it is better to err on having too fine-grained tasks or having to break up large tasks. We will suggest some policy heuristics later in Section 4.4.

4.2 Aligning Partitions

Alignment of loop and data partitions that communicate the most among themselves places them into the same detachment to minimize communication. There are numerous schemes to align loop and array partitions based on linear transformations of index expressions (*e.g.*, systolic machines [43], SIMDs [26]). These are precise alignments from which synchronized, global data movement instructions are then generated to cause the right data to land on the right processor at the right time for the designated operation to proceed. This degree of rigidity is not necessary for our scenario. We use an algorithm that provides more generality and permits more uncertainty; different sizes and

The minimum for this area expression is with

$$\frac{w_i}{w_j} = \frac{r_i}{r_j}$$

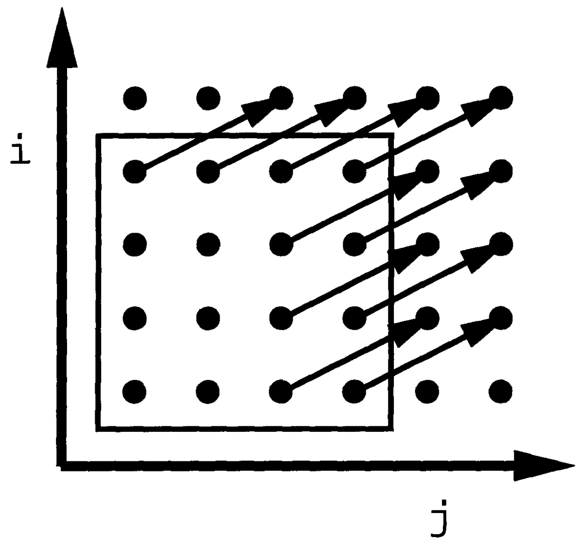


Figure 4-2: *Partition Boundary-Crossing Array Accesses*

Returning to the partitioning procedure, we collect for each loop the maximum and minimum access offsets for an array object to determine

$$r_i = |r_{i_{max}} - r_{i_{min}}|$$

and

$$r_j = |r_{j_{max}} - r_{j_{min}}|$$

because the array partitions can be conformally shifted relative to the iteration partitions. We sum up A_e from all the array objects a loop accesses to represent the total cost of boundary crossing accesses, and use this to determine its partition shape. The result is merely an annotation in the compiler's loop node structure of $(w_d, n_d, r_{d_{min}})$ for each dimension d. After all loops are partitioned, each array object is then partitioned similarly to one of the loops that accesses the array object.

This partitioning method admits array indices of the simple form, $v + k$, for any loop induction variable v and constant offset k. More sophisticated methods will accept indices of $\sum_i^n c_i v_i + k_i$, where v_i is any of the n extant loop induction variables and c_i and k_i are constants. These will cover any linear combinations of induction variables for each index coordinate and produce parallelogram

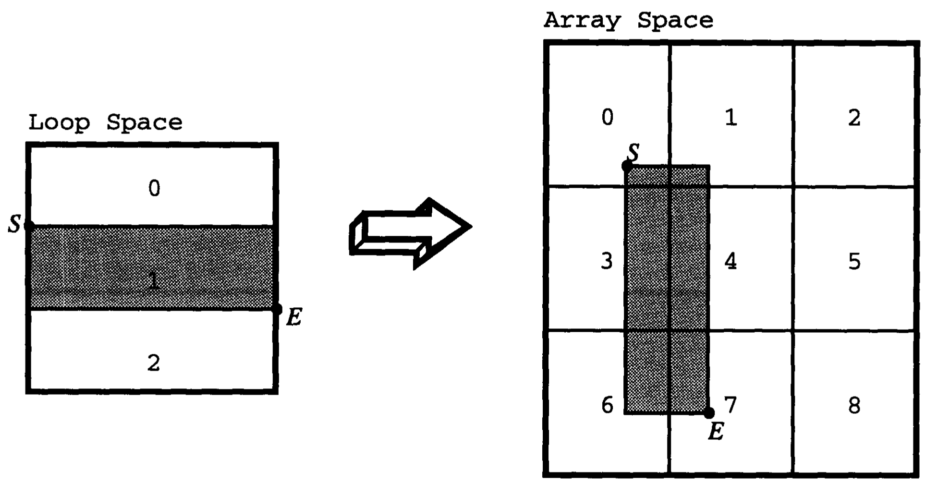


Figure 4-4: *Computing Communications between Partitions*

4.2.2 Computing Partition Alignment

From the constructed communication graph of loop and array partitions, we arrange an alignment of them to produce minimum amount of communication between unaligned partitions. A simple two-level cost structure is assumed: communication between two aligned nodes incurs no cost, while that between nonaligned nodes incurs a constant cost per unit data transferred, regardless of the actual distance between them. Several heuristics will approximate an optimized alignment of thread and data partitions. A simple 2-pass greedy algorithm assigns the first row loop nodes in Figure 4-3 distinct alignment numbers. On subsequent rows, each node is assigned as follows:

- On first pass, pick the node to which it is connected with the highest-weighted edge. If it has been assigned an alignment number that has not already been assigned to a node of the current row, then choose the same number.
- Otherwise, pick successively lower-weighted edges until an alignment number is found that is not already used in the current row. Assign the node this number.
- When the list of connected edges is exhausted, set the node aside for pass two, go to the next node in the row.
- On second pass, for each unassigned node, arbitrarily pick for it an unused alignment number. Leaving these nodes unassigned until the second pass improves alignment for the rest of the nodes of the row.

Using this strategy, the partitions of the example in the figure may be assigned to detachments thus: L1 (1 2 3 4 5 6), D1 (1 2 3 4 5 6), L2 (1 2 4 3 5 6), D2 (1 3 2 5 4 6). In the next chapter, the effectiveness of the alignment phase and its impact on running time will be shown for some programs.

4.2.3 Aligning Large Numbers of Partitions

With very large numbers of partitions, it becomes expensive and also often unnecessary to enumerate every partition. We may only need to align a subregion of the entire iteration or index space. Since all the partitions for a loop or array are identical in size and shape, (again, ignoring boundaries,) the spatial relationships of the partitions between loops and arrays recur along dimension d with a span equal to the least-common-multiple (LCM) of the dimension d width of the partitions, w_d. Consider the following example

```
(define a (array (1000 1000) 5))
(define c (array (1000 1000) ))

(doall (x 0 997 1)
  (doall (y 0 997 1)
    (set (aref c x y)
         (+ (aref a (+ x 2) y) (aref a x (+ y 1))))))
```

The loop and arrays are partitioned for 200 processors, thus

Name	Partition Size	Number of Partitions
Loop	(72×67)	(14×15)
a	(100×50)	(10×20)
c	(72×67)	(14×15)

The LCM blocks are (1800×3350), which unfortunately are even larger than the original dimensions of the items.

But large, relative prime partition widths can be padded out to the nearest simple fractions of each other to reduce their LCM block size. Reshaping the loop and c into (75×70), for example, will reduce the LCM blocks to (300×350). The number of partitions participating in the alignment is thus reduced from around 200 to around 20 for each partitioned item. Padding the partitions to powers of two, as we have already pointed out, also improves the efficiency of compiled code with regards to generating distributed array references. The alignment phase then is able to do its work on a subregion reduced to the LCM of the partitions of the loops and arrays to be aligned. The alignment pattern is then repeated to tile the actual spaces.

We are assuming here that the index expression involves short constant offsets. The LCM reduction becomes ineffective when the index offset is large or global in nature, or index expression has a periodicity of its own. As an example, consider a loop computing $a_i = b_i + b_{N-i}$. This accesses an array (of size N) from both ends; alignment on a reduced subregion will not yield the right results.

Now we show how `align` from Figure 3-7 accomplishes alignment of partitions. At runtime, `align` is given mt, the initial-task number corresponding to the detachment number. It returns a partition number, m, that, when applied to the coordinate translations compiled into the code body, will yield the right iterations to execute and the right array elements to access. The assignment numbers given at the end of last section must first be inversely-mapped, for they indicated in which detachment a partition is to be rather than which partition a detachment should assume. Once we have this list, it can be compiled as a static lookup table. Index into it with the detachment number, and the content gives the partition number. To accommodate LCM reduction, though, we would actually use $\text{mod}(mt, len)$, where len is the length of the mapping list, to index into the table, and add $mt - \text{mod}(mt, len)$ to the content. This simple operation allows mappings of the aligned LCM subregion to be extended by repetition to the whole iteration space at runtime.

4.3 Intra-detachment Thread Scheduling

The threads and partitions in a detachment are given a provisional execution order by the compiler in this phase. We use the lazy future to construct one or more brigades from the collection of threads and loop partitions assigned to a detachment. A lazy future provisionally in-lines its child thread by saving enough parent state and directly entering the child code, leaving the parent thread's continuation available for load stealing by idle processors. The initial cost of a lazy future's deferred task creation is much less than that of a regular future's immediate task creation, but much higher when that deferred task creation eventually becomes necessary.

An implication of using lazy futures is that there would be a top-level control thread that sequences through the threads of a detachment in forming each brigade, thus pre-determining the default serial order that these threads will execute on one processor in the absence of load stealing or synchronization fault. Lazy futures also demarcate the load-steal points along that control thread. Of runtime efficiency concern are the order in which the original threads are sequenced along the control thread and the placement of the load-steal points. In a sense, thread scheduling with lazy futures is akin to most of the "optimization for the common case" strategies[4] — it creates a scenario in which some static choices of thread ordering and granularity will most likely save more overhead from unnecessary task creation and synchronization, while other choices will likely result in orderings that cause more runtime synchronization faults or granularities that cause more mutual load stealings among processors, over some probability-weighted set of possible runtime conditions. The compiler management goal is to construct static code blocks that are sure to result in runtime occurrences matching those expected cases to dominate the exceptions significantly.

In this section, we will examine some detachment configurations appearing at the end of the partition and alignment phases, and propose ways to pre-schedule them statically. The primary concerns here are to control the default sequential thread ordering and the location of load-steal points. The former we consider more important than the latter when they cannot both be addressed because, again, the former occurs presumably as a part of the "common case", whereas the latter only occurs during an exception to the former. We rely as well on the original, programmer-specified sequence and on the single-assignment semantics of return values of `futures` and self-synchronizing data structures to predict the default thread ordering. The relative thread ordering specified in the program is assumed to be the one likely to cause the least number of synchronization faults. This is based on the observation that many Mul-T programs can be execute as serial programs free of deadlocks simply by removing all future declarations. When this order cannot be determined or honored, ordering inferred from reads and writes to single-assignment locations may be substituted. As we shall see, one such case arises from a need to expedite the scheduling of the loop unrolling critical path.

Before motivating the general intra-detachment thread scheduling problem, we first consider some commonly encountered cases. The following sections cover both independent and partially-dependent threads scheduling that arise from parallel loops. Finally, the general threads scheduling problem is examined and a particularly interesting instance of this scenario, the joining of parallel

[4]*E.g.*, branch prediction and processor register allocation gain from cost and usage frequency asymmetries. Branch instructions with symmetrical timing (whether taken or not) and heap-allocated procedure local variables — like the regular future — offer no such a way for a compiler to improve on their performance.

loops, is presented in depth.

4.3.1 Scheduling Independent Loop Iterations

The first, very common and relatively simple case is the scheduling of threads that originate from the same control point in the parent process (*e.g.*, successive future expressions or parallel loop iterations). Because they are from *adjacent* threads, where adjacency is defined as those futures that follow one another without any intervening operations[5], their serial (*i.e.*, thread spawning) order in the program may be freely interchanged. More importantly, it means that the threads assigned to different partitions from this control structure can all be all be started totally independently once the program control has reached the futures' point in the parent thread. It also means that the stolen threads can be started immediately without regards to the state of computation in the parent detachment. For this reason, this case offers the maximum parallelism unrolling of all the scenarios we will examine, and it is beneficial to cast program structures into this case. For the `doall` loop, in particular, the induction variable incrementing in series with each iteration invocation may be assumed absent (or internalized in the loop node) so that all the iterations may be considered adjacent — as long as each iteration eventually gets the proper induction variable binding value. We will see in the next section what to do when there is serial code between the iteration threads.

We consider the problem of building a lazy future tree to execute a given number of identical length adjacent threads. Depending on how much work each load-stealing operation is to take, we construct a tree to spawn out subtrees with the desired fractional divisions. A lazy thread fan-out tree is shown in Figure 4-5 with load-steal points arranged to allow each steal to take approximately $1 - n$ of the remaining work in the detachment (therefore, leaving n of the work behind, $0 < n < 1$), assuming the oldest-first load-stealing policy described in Section 2.2.4. The first steal thus takes $(1 - n)$ of the total threads assigned to the detachment; the second takes $n(1 - n)$ of the remaining n. The original task executing the detachment is then only left with n^2 of its assigned threads. In reality, the lazy future builds only binary trees, by forking a child thread from its parent continuation, so the resulting tree would be unbalanced for all $n \neq 0.5$. An actual lazy future tree with $n \approx 0.6$ is shown in Figure 4-6.

We now examine the implications of choosing n, the *loading ratio*, and propose a simple analytical model to obtain some insights into choosing n to minimize run time. We focus on the

[5]This idea will be refined later in discussion of the general thread scheduling of Section 4.3.3.

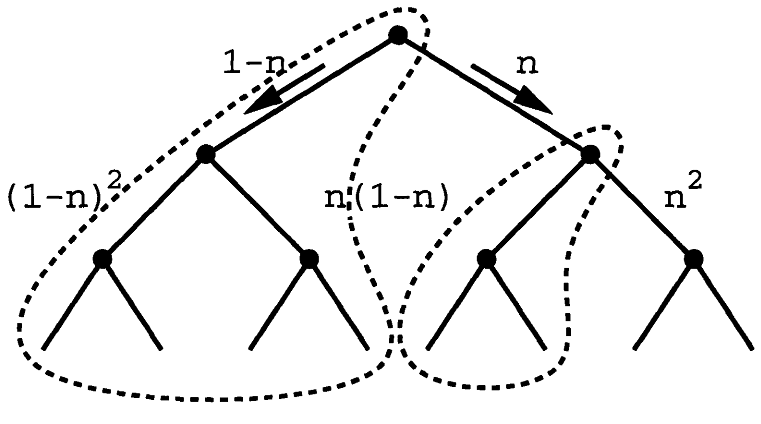


Figure 4-5: *Controlling the Loading Ratio of a Lazy Future Tree*

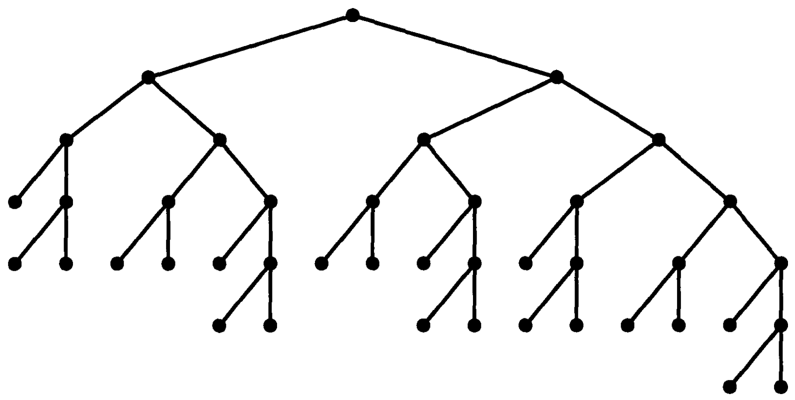


Figure 4-6: *Binary Lazy Future Fan-out Tree for* $n \approx 0.6$

possible outcomes at a single internal node in the tree, whose function is to split the amount of work given between its child and its continuation. The model uses these parameters:

T	$=$	total number of threads under the node
n	$=$	tree loading ratio (0.5 = balanced)
A	$=$	threads in the child branch $= nT$
B	$=$	threads in the continuation branch $= T - A$
c	$=$	time to steal a lazy future and create task
t	$=$	expected time between lazy future creation and its stealing

To limit complexity, the analysis assumes that A and B are not split further. At time τ a processor (PE#0) creates a lazy future and enters its child to compute the threads of A. Some t time later an idle processor (PE#1) comes to steal B, the continuation. After spending time c to migrate the context and set up a new task, it starts B at time $\tau + t + c$ (See Scenario #1 of Figure 4-7). The completion time is the later of the two processors to finish. We will therefore pick n to minimize

$$\max(A, t + c + B) = \max(nT, t + c + (1 - n)T)$$

which occurs when

$$nt = t + c + (1 - n)T$$

Moving terms to get

$$(2n - 1)T = t + c$$

and then

$$n = (t + c)/(2T) + 1/2$$

This is the first scenario, for which $t + c < A$ and the continuation is better stolen. In Scenario #2 of Figure 4-7, it takes too long for another processor to become idle and come steal a task, so that all the work is done by one processor. This happens when $t + c \geq A$ or really $t + c \geq T$ since A could be any fraction of T. The completion time for the node is therefore

$$\begin{aligned} P(T) &= \min(\text{Scenario \#1}, \text{Scenario \#2}) \\ &= \min((t + c + T)/2, T) \end{aligned}$$

Along with the earlier solution

$$n = \frac{t + c}{2T} + \frac{1}{2}$$

we see that $1/2 \leq n \leq 1$ and that n increases as T decreases to $t + c$, which is actually more likely to happen further down the tree.

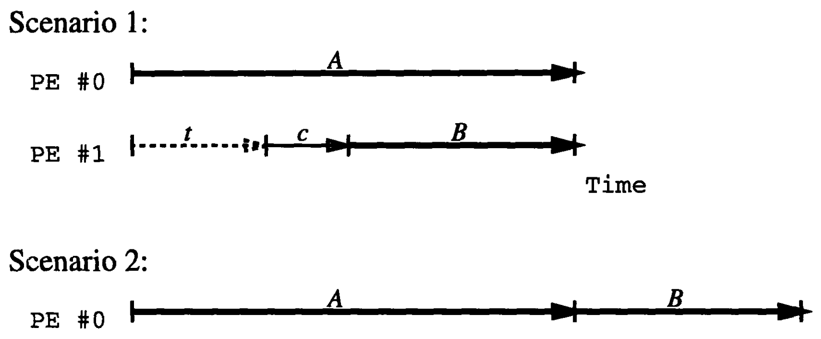


Figure 4-7: *Execution Time Schedule for Computing the Loading Ratio*

Figure 4-8 shows the load split ratio as a function of the amount of work (thus the number of threads to be scheduled) in units of $t + c$ at each node of the tree. In practice, both A and B

may be recursively broken down further by other stealing processors, and it is difficult to ascertain a value for t, since it is very dependent on when there are idle processors and how the runtime scheduler hunts for work. In light of these shortcomings, this model should only be considered as an exercise in gaining insights into the scheduling dynamics at each tree node and a stepping stone in building far more sophisticated stochastic models. Our implementation currently constructs trees with $n = 0.5$.

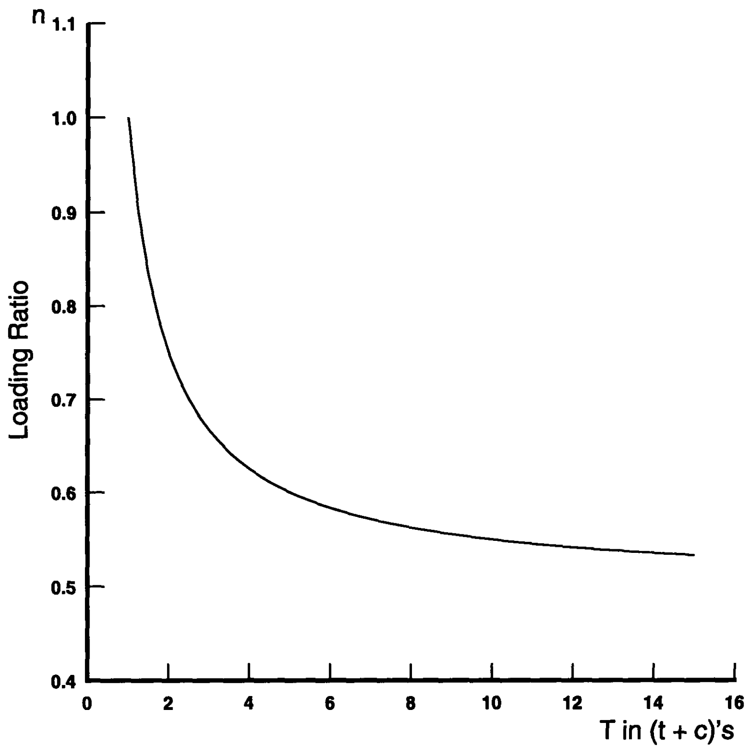


Figure 4-8: *Optimum Loading Ratio* vs. *Amount of Work*

In the context of the restructured example program of Figure 3-7, `doall-segment` would call `ftree`, the lazy future fan-out tree macro shown abstractly in Figure 4-9, using the induction variable name and the range of its values assigned to the partition. For multiple dimension loops the macro can be recursively nested. In actual implementation, the multiple dimension calls are collapsed somewhat and the induction variable values are computed more efficiently. Note that the induction variable values can be computed at each tree node independently of its preceding iterations because of the constant *start*, *limit*, and *stride* values. This constraint has allowed the substitution of a fan-out tree for the linear loop unrolling.

For more arbitrary loop controls, the paradigm to be described in the next section may still be applicable instead of resorting to the general procedure. Thread terminations are explicitly checked in this code, so that when the code returns, it signals the completion of all work below the current node. This signal can be waited on before starting computations following the loop, or collected in a list to be checked as part of program termination condition.

```
(define-local-syntax (ftree var start limit stride . body)
  `(iterate btree ((begin 0)
                   (end (/ (- ,limit ,start 1) ,stride)))
     (let ((range (- end begin)))
       (if (< range 2)                 ;; leaf node?

           (let* ((,var (+ ,start (* begin ,stride)))
                  (afut (future ,@body)))   ;; left thread
             (if (> range 0)
                 (let ((,var (+ ,var ,stride)))
                  ,@body))                  ;; right thread
             (touch afut))             ;; synchronize completion

           (let* ((center (+ begin (/ range 2)))
                  (afut (future (btree begin center)))) ;; left subtree
             (btree (+ center 1) end)                   ;; right subtree
             (touch afut)))            ;; synchronize subtree completion

       'done)))                   ;; acknowledge everything here is done
```

Figure 4-9: *Lazy Future Fan-out Tree Macro for Parallel Loop Iterations*

4.3.2 Scheduling Partially-dependent Loop Iterations

A somewhat more complex case arises when the threads to be serialized are not all from the same control point, but rather have small intervening sequential sections. This occurs, for example, in a C-style `for` loop with an iterated future body:

```
          e1           e2            e3
for ( p = list; p != NULL; p = p->next )
      future( ...  expression with p ...  );
```

where both $e2$, the termination check, and $e3$, the induction variable "incrementing function", really are intervening sequential loop sections, even when the body itself is an expression fully enclosed in a future. Instances fitting this scenario include the above example of dereferencing items from a list, each of which is then processed in parallel, or the calling of a pseudo-random number generator to produce a random value and a seed for the next iteration in a typical Monte Carlo simulation.

This case is our asynchronous analogue to the globally synchronous doacross construct [12] or the software iteration pipelining [29] in the sense that parallel tasks are initiated staggered in time to execute iterations of a partially-dependent loop on multiple processors. In a globally-synchronous system, a simple duration can be specified to offset the start of tasks on each successive processor that will insure that the corresponding instructions in successive iterations are delayed by the same initial offset to satisfy their dependence constraints. In an asynchronous system, however, the serial section must be executed before explicitly initiating a succeeding task. The thread structure of a generalized parallel `for` loop is illustrated in Figure 4-10. On the left, we see the dark line representing the sequential control thread of the iterating loop, along which parallel threads are spawned to execute the body of each iteration. Example partitions A and B are shown in shaded regions. A brigade, in other words a provisional serialization, must now be constructed to honor the partition specification without really creating individual tasks out of the threads at first.

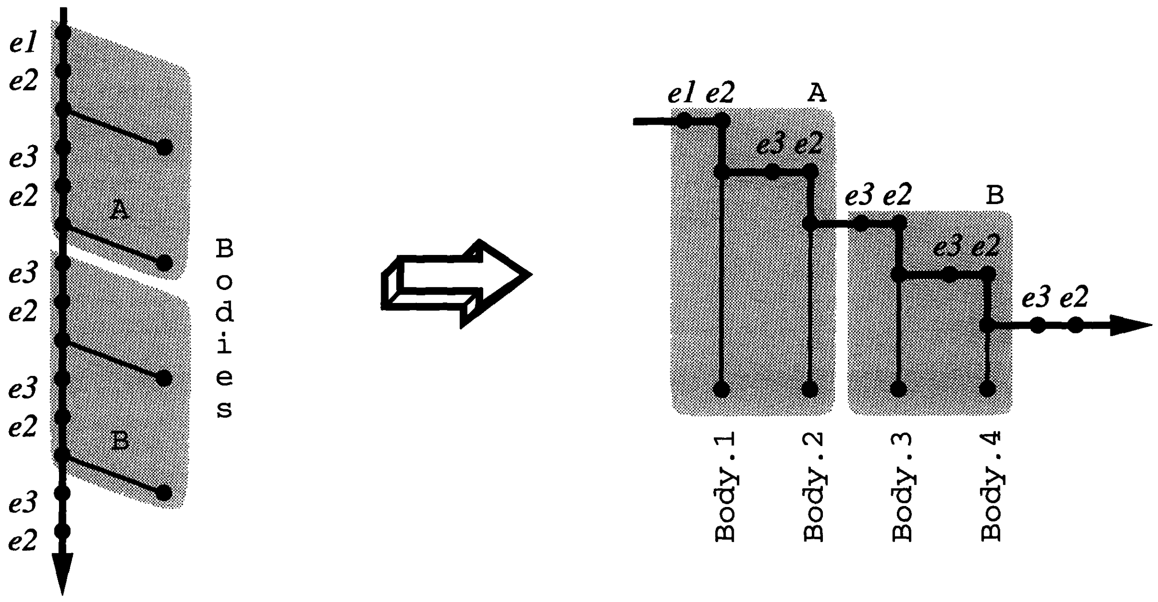


Figure 4-10: *Restructuring of Loop Iterations with Intervening Sequential Sections*

The ramifications of the relative scheduling order between the two thread branches exiting at a future fork were discussed in Section 2.2.3. Here we encounter another example[6] where the

[6]The earlier case being the doall, where the sequential unfolding of fully-parallel iterations are rearranged such that some are executed, "out of textual sequence", concurrently on distinct processors.

compiler explicitly manipulates the input code to control that execution order. Since a lazy future executes the child thread in-line first, the individual iteration threads in a brigade will be executed first by a task, only after which will the succeeding task be initiated. In order to expedite the path to start the next task over the lower priority execution of the iteration threads, the parent and child continuations are swapped in the diagram on the right. The unrolling branch is thus in line to start immediately — before work begins on any of the loop body iterations — until all iterations in the brigade have been exposed. At this time the unrolling branch is passed to the succeeding processor node, and execution of the iteration bodies begins on the "return" path. To describe the process in terms of the elements of the parallel `for` loop template, upon starting the loop, sequential statements $e3$ ($e1$ for the first iteration) and $e2$ are executed and a stealable return continuation is created for the loop body, as shown by the solid tracing arrow on the left side of Figure 4-11.

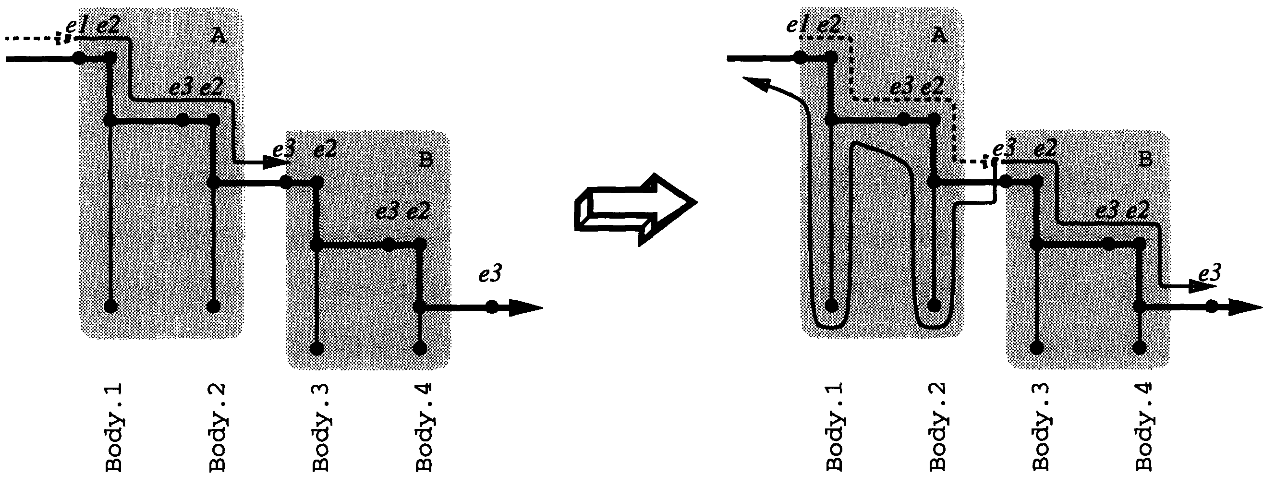


Figure 4-11: *Execution Schedule of a Partitioned Partially-dependent Loop*

This execution proceeds until a brigade crossing is reached, which is accomplished through a real task creation onto the next processor in succession. The succeeding processor, which is determined by the static alignment as before, then starts its loop task, while the first processor returns to evaluate the loop bodies it has left behind. These actions are shown by the solid tracing arrows in the diagram on the right. The abstract macro definition to implement the parallel `for` loop is shown in Figure 4-12. The construct `(future-on` *<node-id>* *<expression>*`)` enqueues a task on processor *<node-id>* to evaluate *<expression>*. Note that a task executes the loop bodies of, for instance, Brigade A of Figure 4-11 on its return continuation, so that `body.2` is scheduled before `body.1` (*i.e.*, in the reverse order from the way they are entered). The reversal would be entirely inconsequential if the loop bodies are truly independent of each other, as they should since any loop-carried dependencies should have been hoisted into the expression $e2$ or $e3$.

```
(define-local-syntax (facross var start limit stride  e1 e2 e3 . body)
  (iterate new-brigade ((partition 0) (begin ,start) (end ,limit))
    (future-on (align partition)
      (iterate loop ((,var begin))
        (if (= ,var ,start) ,e1
            ,e3)
        (if ,e2 (block
                 (future (if (< ,var end)
                             (loop (+ ,var ,stride))
                             (new-brigade (+ partition 1)
                                          (+ ,var ,stride)
                                          (+ end ,limit))))
                 ,@body))))))
```

Figure 4-12: *Macro for Unrolling Partitioned Partially-dependent Loop Iterations*

This scheme for accommodating partially-dependent loops has some incompatibilities with the current lazy future mechanism and the runtime system implementation described in Chapter 2. First, the in-line distribution of brigades requires the sequential section to be passed among the processors, therefore it becomes very important for the critical sequential path to unroll its way around at a high priority in order to distribute computation quickly. We have prioritized the static scheduling within a brigade, but the runtime scheduler should also cooperate. A simple modification is to provide a priority task queue, for this and other time-critical operations. But task dispatching can be even more expeditious if a processor's active task can be preempted by such a time-critical task. Current work on user-level IPI (interprocessor-interrupt) messaging and user-specifiable IPI message handler facilities [28] will eventually allow this idea to be implemented from within the compiler-generated code.

Second, it is not possible to restructure the steal points freely as before, so that the default oldest-first load-stealing policy now takes only a single iteration at a time. To overcome this setback, the lazy-future queue can be annotated for entries from the `facross` macro and the load-stealing code modified to extract the n-th oldest future in the queue, where n can be either a fixed number or determined as a fraction of the number of futures in the queue. These changes will complicate substantially what is now a very minimal runtime scheduler. Of course, the future queue size will grow linearly with the partition size instead of only logarithmically.

Lastly, because of the considerably longer time it takes for a loop to be distributed onto all the participating processors, there is a sufficient timing window — when starting on an idle machine — for heavy amounts of load stealings from those idle processors that will not be getting their turn for a while. To prevent the static partitions and alignments from being severely disturbed at runtime, load stealing should be turned off initially on all processors until a processor has had its turn to unroll its iterations. Yet another tack is to assign more iterations to the "earlier" partitions to compensate for losing their workload due to such early stealings, as well as for the fact that the "later" partitions will have less time to complete their work due to their later starting time. This tapering of the partition sizes is less necessary if there is much work pending before the loop and work awaiting after it, especially if what comes before or after is another `facross` macro-spawned loop that will mate well with the staggered starting or completion schedule of the loop. These are concerns that should be addressed as part of performance tuning parameters and heuristics.

4.3.3 Scheduling Random Parallel Threads

Now we consider the general case of forming a brigade from threads co-located in a detachment that are from random parts of the program. The procedure starts by constructing a serial thread that calls, through a future, the first thread, in program order, in the detachment. Thereafter, it calls, always through the revocable serialization of a future, successive threads in the detachment. When it encounters a recognized loop partition, the appropriate partition macro introduced in the last two sections is expanded to replicate instances of the loop body. When all the threads have been incorporated, the brigade ends. The brigade is then called — through a `future-on` or a gang task-spawning macro such as the `do-parallel-tasks` of Figure 3-7 — at the original call point of the first thread in the brigade by its parent thread. The serial thread chaining the threads of a brigade must preserve the original lexical environment and control order for each thread that it incorporates.

- Sometimes, in incorporating a thread, it is possible to conform to these requirements by inserting synchronizations to control explicitly the scheduled execution of the thread.

- In other cases, when a single thread is unable to reach all threads of a brigade, several separate serializing threads must be used to cover all of them, thus forcing a breakup into several brigades.

A task will be created at runtime for each of these serialized brigades. We will consider the lexical-environment and control-order preservation requirements separately.

Lexical-environment Preservation

Since the thread body of a future sees the variable bindings valid at the point of call, it is necessary in the restructured code to maintain the original lexical environment of each thread incorporated into a brigade. Since the brigade sees the variable bindings at the time of its calling, (*i.e.*, in the context of its first thread), a later thread can only be incorporated if it shares the same lexical context as the first thread, or that all the intervening context-defining code has been incorporated into the brigade. Since the compiler normally keeps track of the active context at all points in the program, it is easily decided if the condition is met to incorporate a given thread. If not, then a new brigade has to be started at that thread and continued forth. The example in Figure 3-7 showed the brigades being broken up midway by the evaluation of `(f 0)`, originally from the function `SetBounds`, which is part of the main thread and thus was not incorporated into any of the other brigades.

Control-order Preservation

The purpose of control-order preservation is to insure that the serial sections (*i.e.*, all computations not explicitly parallelized with future declarations) in the restructured code strictly obey the original program order, since Mul-T allows side effects. *Control order* is like that imposed by the serial program execution order — top to bottom, left to right, and, for function calls, depth first — in a sequential program, except that the future declarations create concurrent branching structures in the linear program order. Thus, in the example below,

```
...
ex-1
(future a)
(future b)
ex-2
(future c)
ex-3
...
```

the control order is

$$\ldots \rightarrow \texttt{ex-1} \rightarrow \left\{ \begin{array}{l} \texttt{(future a)} \\ \texttt{(future b)} \\ \texttt{ex-2} \rightarrow \left\{ \begin{array}{l} \texttt{(future c)} \\ \texttt{ex-3} \rightarrow \ldots \end{array} \right\} \end{array} \right\}$$

The two future expressions `a` and `b`, and the expressions in the continuation thread `ex-2 (future c) ex-3 ...` are mutually parallel. Threads are *adjacent* if they are parallel and share the same predecessor control node. So, `(future a)`, `(future b)`, and the continuation starting at `ex-2` are adjacent. And although `(future c)` is parallel with the other two futures, it is not adjacent to them. It is, however, adjacent to `ex-3`.

Figure 4-13 shows the thread diagram for some code fragments. The solid lines indicate the threads that have been assigned to the same brigade that is now our job to schedule. In Fragment A, functions `f` and `g` are in this brigade. Scheduling `f` and `g` into a brigade is trivial – just leave the code as is. It is similar for Fragment B, where `f` and `g` are in the same brigade but `p` is elsewhere. In Fragment C, the intervening future computation of `h`, which is to be executed from another brigade, is not a problem, either; scheduling of `f` followed by `g` and simply bypassing `h`:

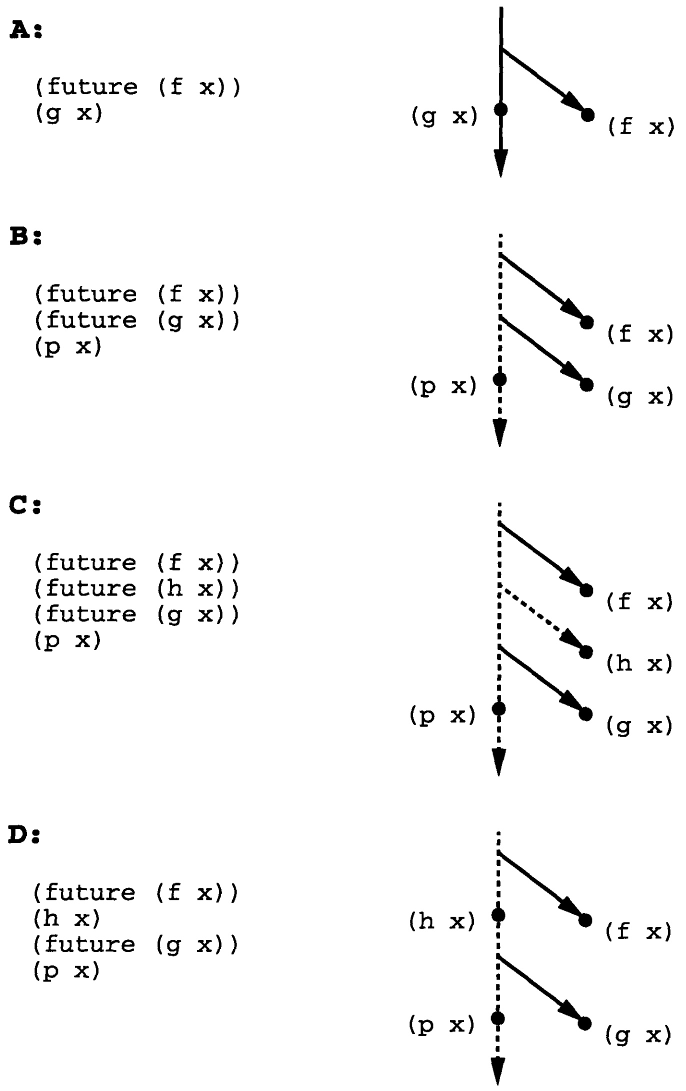
```
...
(future (f x))
(future (g x))
...
```

But Fragment D does present a problem in that `g` cannot start without waiting for `h` to have completed. Possible solutions include inserting explicit signaling code between brigades to synchronize the computations and forcing a break in the brigade (such that when `f` is completed, the task is terminated; a new task is then created to execute `g` when `h` finishes).[7]

In codifying rules for thread scheduling in these scenarios, we may directly chain threads via a future without violating control-order constraints when they meet one of the following conditions:

- A thread may be chained to its parent. This is just what the `future` declaration does.

[7]Another possible solution, which we have not yet explored, dispatches the compiler to analyze the side effects of `h`. If there are none, or none that is seen by `g`, then the brigade for `f` and `g` can be scheduled as if in Fragment C.



A:

```
(future (f x))
(g x)
```

B:

```
(future (f x))
(future (g x))
(p x)
```

C:

```
(future (f x))
(future (h x))
(future (g x))
(p x)
```

D:

```
(future (f x))
(h x)
(future (g x))
(p x)
```

Figure 4-13: *Scenarios for Thread Scheduling*

- A thread may be chained to its *adjacent* thread. The iterations of a doall, as seen in Section 4.3.1, is a special case of this, where all iterations are adjacent once the induction variable computation is abstracted into the loop node itself or pushed into the iterations.

These rules allow adjacent threads to be scheduled in any order amongst themselves, and allow any subset of them to be allocated to other processors, or stolen by them, without additional scheduling or synchronization considerations. Moreover, as the parallel threads in Mul-T do not in general mean dependence-free computation as they do in Parallel Fortran, we should avoid choosing a static ordering of adjacent threads where there would be compiler-detectable forward data dependencies involving synchronizing data references among these serialized threads — these dependencies are certain to incur expensive runtime synchronization faults. Such information can be obtained during compilation in the same way as standard data-flow analysis. A simple, blind heuristics suggested in Section 2.2.3 is to follow the serial program order for scheduling adjacent threads whenever there is no overriding reason otherwise.

When the listed conditions cannot be met, two threads can be chained in a brigade only after inserting proper synchronization to insure execution of the threads maintains the control order. We use the example code below to demonstrate the overall structure of a program transformed by generalized static thread scheduling.

```
(define (example)
  (let ((a (future (computeArray n))))
    (reduceArray a)
    (future (scaleArray a))))
```

Here `reduceArray` applies a serial reduction function over the elements of the array; `computeArray` and `scaleArray` are parallel loops to fill a j-structure array with initial values and to scale the element values based on the result[8] of the reduction function, respectively. Note that the futures share the same lexical environment, except for *a*, which is actually defined in the first future. Suppose that the functions called are in-line expanded for analysis and transformation purposes, that `reduceArray` is a sequential function executed in-line within the top-level thread, that the many parallel loop iterations in `computeArray` and `scaleArray` have been partitioned and aligned based on their high-volume communication through the intermediary array *a*, and that we wish to schedule the threads assigned to each detachment. The scenario is diagrammed at the top of Figure 4-14. Arcs show the arguments and results data exchange between the functional blocks. The return value from `scaleArray` is labeled as *d*. The partitions of `scaleArray` cannot be chained directly to the corresponding partitions of `computeArray` with which they have been co-located — the intervening code in `reduceArray` must be executed before `scaleArray` can begin. (Here, the result of this reduction function is stored in a global variable and used by the subsequent scaling function.)

Correct control order can be maintained, however, using explicit synchronization provided by inserting a flag-setting statement at the completion of `reduceArray` and modifying the partition

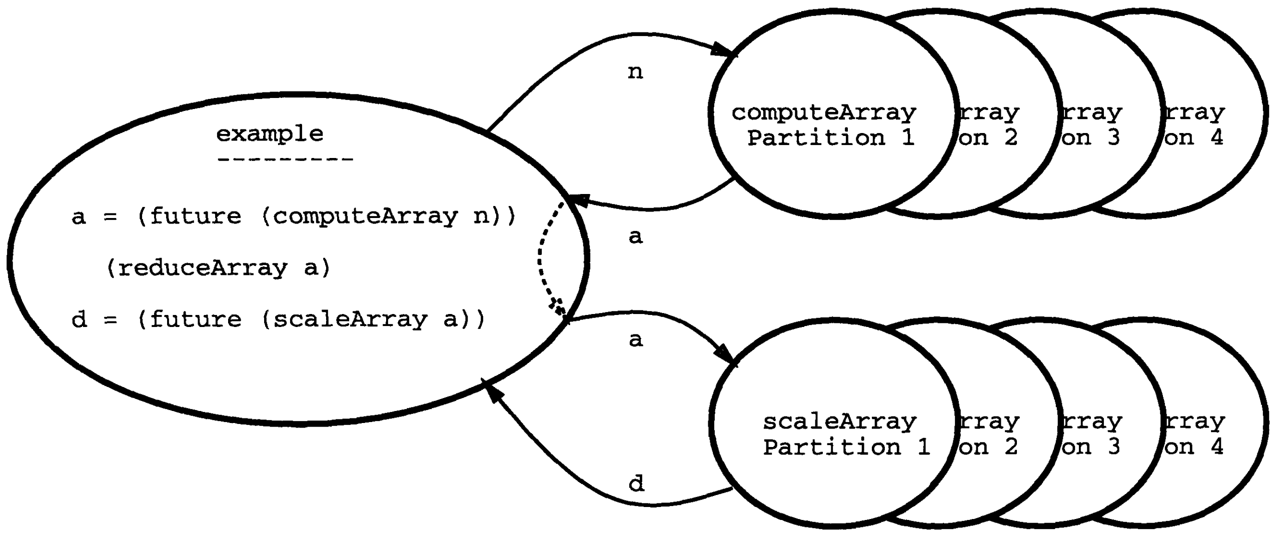
[8]Stored in some global variable.

joints to wait for the flag before proceeding into the `scaleArray` code. This merging transformation is shown in the lower half of Figure 4-14 as the added gating of control flow inside the brigades of `computeNscale`[9] created from chaining partitioned iterations of the original `computeArray` and `scaleArray` functions. Cache consistency protocol will have propagated all intervening store updates from `reduceArray` to the `scaleArray` partitions when they have detected the flag setting. The introduced synchronization likewise would prevent `scaleArray` from side-effecting the computation of `reduceArray` by delaying the former until the latter's completion. The example also shows how to rearrange the data value exchanges that occurred on the original future entries and exits. The standard future interface at `computeArray` is restructured to accommodate the return of the future value, a. The variable a is bound to a newly created placeholder before the function call. It is then passed into `computeNscale`, which will set the placeholder once its compute-array phase is completed. This provides `reduceArray` with its function argument before the actual future for `computeNscale` returns finally with the value of d. The top-level structure of the completed brigade is shown below.

```
...
(doall-segment ... )          ;; step through computeArray partition
(set-placeholder a local-a) ;; ''return'' a
(jref flag 0)                 ;; wait for reduceArray
(doall-segment ... )          ;; step through scaleArray partition
...
```

As in the previous case, the arrangement of the steal points here is also not ideal for the oldest-first load-stealing policy of the runtime scheduler, although it is much less problematic than that of the partially-dependent loop iterations. Note that while a processor is working through iterations in its assigned `computeArray` partition, the oldest continuation in its lazy future queue is the top-most pending recursive call of the `ftree`. Another processor stealing it would take the iterations in the second tree branch rooted at that node, along with the entire latter part of the brigade, including all of its assigned `scaleArray` partition. This, of course, is not the intended load-stealing behavior. Placing a future declaration, annotated so its future queue entry is marked as anchored "local", before entering the top-most `ftree` will force the stealer to take the stack frames between it and the next entry in the lazy future queue. The stealable continuation will then extend only up to the end of the `scaleArray` partition. The runtime scheduler's stealing function has to be modified

[9]This new function name is created for the purpose of exposition only.



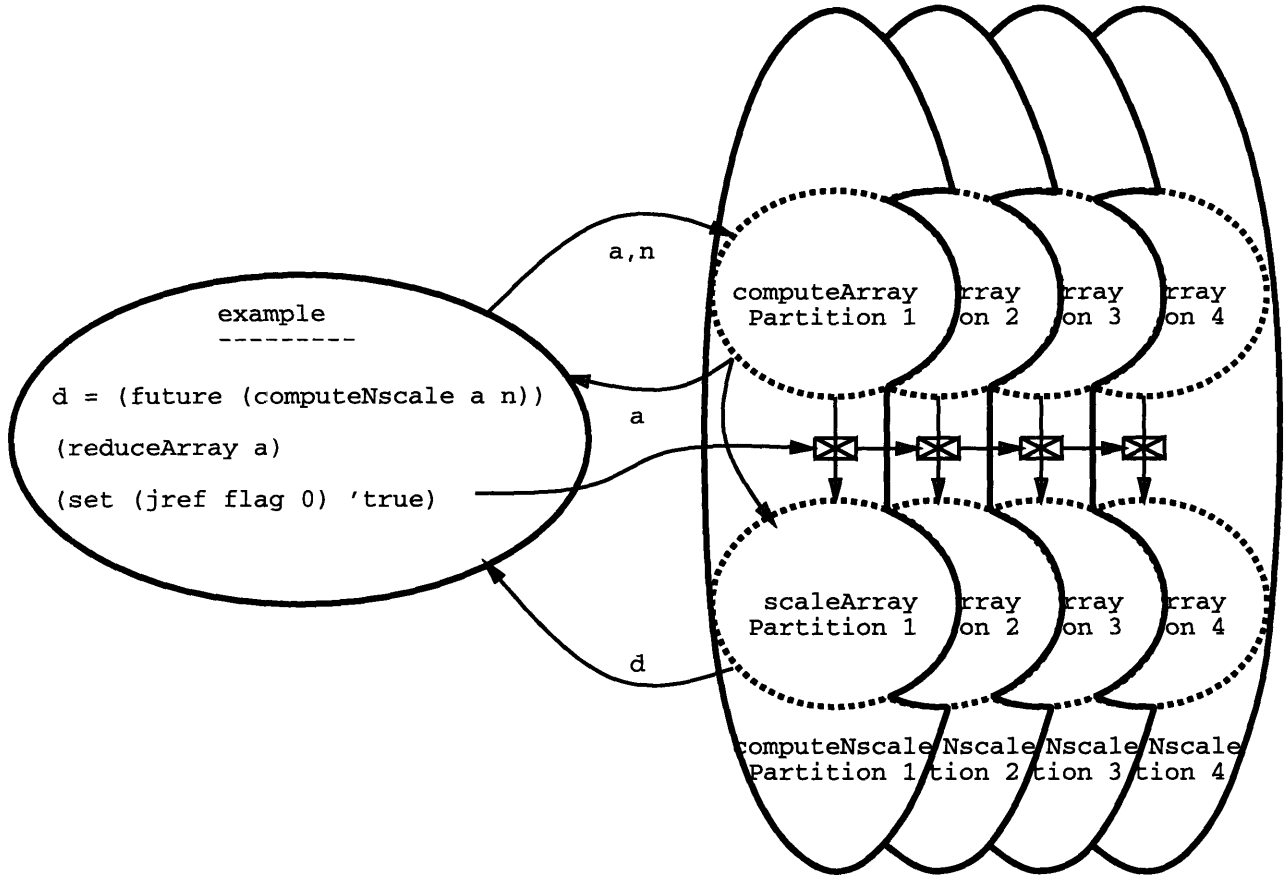


Figure 4-14: *Proper Synchronization in Chaining Data-parallel Loops*

to recognize the queue entry annotation and to skip those that are restricted to be the local stealing points.

The static scheduling to chain multiple partitions of threads into brigades does not in general always increase the run length[10]. At worst, the brigades will all block at each synchronization point and later resumed, resulting in run lengths just as long as if individual tasks are created for each of the loop partitions. Under other conditions, some task scheduling overhead is avoided. But the formation of larger brigade has several more advantages:

- Independent tasks need not be created for the `scaleArray` partitions, so they will not appear in the scheduler task queues before the corresponding a elements are actually computed by `computeArray`. Since the runtime scheduler does not know about the control and data dependencies between tasks, if those tasks were in the task queues, they may be randomly scheduled out of that order. In addition to the task scheduler thrashing that this may cause, early start of a task not ready for execution may also cause resources to be allocated long before they are actually required. Example initializations that `scaleArray` may do include memory allocations and the spawning of yet more tasks.

- Locality of reference can be better maintained within each task when loop partitions operating on the same data are chained into brigades. When a task running a brigade is migrated, the compiler-imposed partition alignment is maintained through the migration. Since the producer and the consumer of a data array segment, as well as the allocation of that segment, can all be relocated together by any such global load-balancing operation, locality is much better preserved.

- When there is no intervening code between the partitioned loops to be chained, a direct chaining, without any explicit or implicit synchronization at their boundary, will suffice. The run length is then effectively increased by the formation of these brigades.

The intra-detachment thread scheduling transformations of the types proposed here are certainly too complex to perform when using the runtime mechanisms of the lazy future alone, which is only able to affect thread integration locally at each fork-join interface. Only compiler-directed scheduling can coordinate these transformations effectively.

[10]The number of instructions executed between synchronization points.

4.4 Policies for Loop and Array Partitioning

We identified three major places in the proposed compilation framework where some external policy decisions are needed to proceed: the granularity of static partitions the compiler assumes and the amount and which parcel of work each load stealing from the detachment will take. We will focus here on the primary issue of determining the appropriate static partition granularity. The other issues have already been taken up in discussions on static thread scheduling.

The independent policy for controlling granularity of static partitions is a set of heuristics that allows the choice to be progressively "more optimal" when more runtime information is known at compile time (*e.g.*, system and problem size). Throughout, a minimum task granularity should be maintained based on the cost structure of communication and the cost of process creation. The following suggests some heuristics, in increasing complexity order:

- Each partition consists of some small fixed number of iterations. This enforces the minimum task granularity requirement in the absence of more useful information.
- Each partition consists of k/p-th of the iteration space of a loop, where $k =$ some fractional constant and $p =$ the number of processors in the system. Partition size is variable as a function of the loop bounds and the number of processors. Initial partitions can be either distributed around the system or left as stealable tasks on key processor nodes. With $k = 1/n$, n initial partitions are generated initially for each processor nominally. The current implementation uses this method, choosing $k = 1$.
- As above, but compile several partition sizes; let the runtime system choose the best matching case for a given runtime condition.
- Instead of assuming each loop to be distributed across all processors in the system, construct a static global schedule and use it as a basis for a top-down static partitioning algorithm such as [41] when all partitioner-relevant program and timing parameters are available to the compiler. If not, statistically-derived estimates of these runtime parameters (such a tack was used by [46]) might be substituted. With it we determine statically which loops are possibly concurrent and thus obtain an estimate for a smaller subset of processors to which those loops are distributed. The larger, fewer partitions improve locality and reduce the amount and possibly the distance of the inter-node communication. Feed these partition sizes to the

static partitioner of Section 4.1.4. The compiler partitioned code may synchronize at loop boundaries using the completion signal mentioned in Section 4.3.1 and generated as shown in Figure 4-9 to preserve the global static schedule.

The threads of the partitions chosen are, as usual, serialized via lazy futures as given by the various scenarios described in this chapter, so that they are subject to load stealing by a remote or the local processor.

4.5 Summary

In this chapter we presented the implementation strategies and the mechanisms for realizing the compiler phases of the Revocable Thread Serialization framework. Fine-grained parallel programs are analyzed and statically partitioned, aligned, and scheduled to optimize their data-parallel computations. This process allows compiler-based analysis and optimization strategies developed for a parallel Fortran-like programming system to be introduced in an asynchronous, applicative function-based parallel system. The performance results of this implementation will be presented in the next chapter.

Chapter 5

Performance Characterization and Benchmark Programs

Thus far we have introduced a framework for integrating compiler analyses and optimizations and the runtime scheduler to manage the behavior of parallel programs. In the last chapter we saw how the compiler transforms the input program into explicit packages of code and data allocations, called brigades, that the runtime scheduler can deposit onto processors, migrate about them, and even take apart later when there is not enough work for the idle processors. In this chapter we discuss some performance issues. We will examine the effectiveness of some of the key components of this framework, realized as described in the last chapter, as compared to existing alternative strategies. Of the framework's components, the program analysis front-end and the loop/array partitioning are the most tentative and most highly borrowed from existing work. They exist to complete the processing chain and can be updated with any suitable state-of-the-art techniques and implementations. The focus presently is thus on the performance of the serialization and alignment processes.

We devise simple test programs to test particular aspects of the proposed system independently and to contrast their behaviors with those of some alternative schemes. First, a comparison of several thread serialization strategies is seen, followed by an examination of the effects of the alignment process, and then by a demonstration of the impact of the compiler's mistakenly serializing threads that results in a runtime synchronization fault. Finally, a fairly large scientific benchmark program is shown to run on this implementation. Again, alternative serialization strategies are tried on it, and the success of alignment on the large number of loops and arrays are analyzed. These results

Strategy	Partition	Alignment	Serialization
Eager Future	no	no	no
Self-schedule	no	no	no
Lazy Future, Linear	no	no	soft
Lazy Future, Tree	runtime-decomposable tree	no	soft
Static-block	compiled blocks	no	hard
Aligned Static-block	compiled blocks	yes	hard
Guided Self-schedule	varying runtime blocks	no	hard
RTS	compiled decomposable blocks	yes	soft

Table 5.1: *Taxonomy of Some Named Parallelism-management Strategies*

serve to validate the initial performance claims made in Chapter 1, as well as the applicability of the proposed framework to real parallel programming scenarios. The simulation results used in this chapter are all obtained from a very detailed software simulator [6] that models the precise, cycle-by-cycle behavior of the processor, coherent cache, network router, and memory controller that are in the Alewife hardware system. Throughout, power-of-two system sizes and array and loop sizes are chosen so that the much cheaper bit shifts and masks can be substituted for multiplication, division, and modulo operations (see Section 4.1.3); there is no reason otherwise to insist on them.

Since we will refer to alternative strategies in the literature, and use comparisons with them in validating the effectiveness of RTS, it is useful to systematize their functions according some set of bases. Table 5.1 gives a taxonomy of some of them in terms of their particular policy as mapped onto each of the RTS transformation phases. Under serialization, *soft* means provisional and revocable, while *hard* means the iteration threads, once scheduled at compile-time or runtime, are irrevocable.

5.1 Performance of the Revocable Thread Serialization

Section 1.3 outlined the particular intentions, structures, and performance characteristics of the proposed RTS framework. We will verify here one of the principal characteristics of RTS — its tolerance to variations in execution time and deviations from the schedule the compiler forecast. Further, it needs to be shown that this is accomplished without incurring a high cost relative to other simpler schemes even within their optimal operating regions. A synthetic-load program is used in an experiment to generate a scenario similar to that in Figure 1-4. We need a simple loop that can be partitioned and whose iteration body execution time can be varied explicitly over a controlled range. It is then compiled and run using different parallelism management strategies. The test loop in Figure 5-1 has a sequential counting loop, `dotimes`, inside its parallel relaxation loop. Its count is determined by the elements of array *len*, whose values come from `(nrandom-array` *nPEs* m σ`)`, which returns an *nPEs*-element array of normally-distributed random numbers with mean m and standard deviation σ. *tid* is a compiler-generated detachment identification.

```
(define a (make-array 2048))
(define len (nrandom-array 16 200.0 50.0))   ;; nPEs mean sigma

(doall (j 0 2047 1)
  (set (aref a j) (+ j j j)))      ;; initialize array

(doall (i 1 2046 1)
  (dotimes (k (aref len tid))      ;; run relaxation
    (set (aref a i) (/ (+ (aref a (+ i 1)) (aref a (- i 1))) 2)))))
```

Figure 5-1: *Load-balancing Test Program with Variable Loop Iteration Length*

Table 5.2 contains 16- and 64-processor simulation results. With $m = 200$, each relaxation loop iteration thread averages 9975 cycles in duration. Due to the random numbers involved in the experiment setup, multiple trials are run with each parameter tuple and their results averaged in the graphs of Figure 5-2[1]. Among the strategies tested, the *static-block* has compiler-partitioned blocks of arrays and iterations that are in-line serialized. The *partitioned lazy linear* and *RTS*, which in this context is better described as *partitioned lazy tree*, have compiler-partitioned blocks of arrays and iterations that are unrolled linearly and branched out in a tree, respectively, through lazy futures. The *lazy linear* and *lazy tree* have unpartitioned loops that are unrolled linearly and branched out in a tree, respectively, through lazy futures. Lazy linear is what a standard loop with future body generates in the unaugmented Mul-T system. Being an order of magnitude greater than the rest, its curve is well outside of the range of the graph.

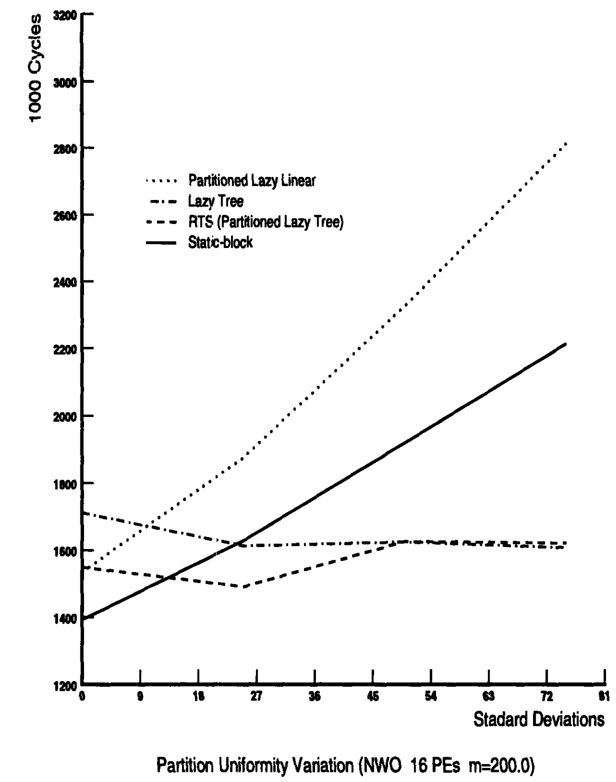
It can be observed that on 16 processors, a very small system, the various strategies, except lazy linear, all started fairly evenly, the differences due mostly to those overhead costs of declaring futures and traversing the trees. As expected, static-block has the lowest runtime overhead. With

[1]Note that the vertical axes do not start from 0.

$m = 200$		16 Processors			64 Processors		
Strategy	σ	Execution Times			Execution Times		
Static-block	0	1392017	—	—	483617	—	—
	25	1595034	1670762	1619741	508909	623087	557020
	50	1854188	2009874	1893601	596825	764937	684470
	75	2120295	2345412	2171146	687211	898652	810777
Partitioned Lazy Linear	0	1534614	—	—	704468	—	—
	25	1778823	1931938	1915576	766668	668648	707176
	50	2131180	2515769	2331488	705700	789501	888280
	75	2533539	3056381	2843841	846121	982112	1022862
RTS (Partitioned Lazy Tree)	0	1548936	—	—	519066	—	—
	25	1501748	1426756	1545170	502611	525719	554917
	50	1546579	1580752	1749395	555750	556129	553783
	75	1448073	1520807	1892771	718742	610831	710175
Lazy Linear	0	19229731	—	—	7841420	—	—
	25	18472668	18298400	20066170	7759946	7539042	7855315
	50	17404961	16968201	20530348	7541988	7171750	7786662
	75	16264287	15335747	20566577	7334738	6611877	7410600
Lazy Tree	0	1711577	—	—	749606	—	—
	25	1538486	1594866	1703929	838091	751480	730029
	50	1491956	1540442	1839930	724190	695636	727814
	75	1442679	1455353	1923084	709193	767791	783361

Table 5.2: *Simulation Results: Varying σ of Thread Size, $m = 200$*

the introduction of thread-size variations, however, the completion time of static-block becomes directly dependent on the slowest (*i.e.*, the largest) partition, which is exactly linearly proportional to the standard deviation of the normally-distributed thread size random variable. The graph shows a nearly linear curve for it. Lazy linear fares similarly poorly as thread sizes vary and load stealings increase, since each steal takes the continuation to all the iterations remaining in the partition — not a very efficient load-redistribution scheme. Both lazy tree and RTS show good immunity to thread-size variation, while lazy linear has the worst results throughout. Switching to the larger 64 processor system, the lazy tree shows a marked lag behind RTS, because it is initially an unpartitioned tree. The partitioned brigades initially distribute work more efficiently and evenly. This is much more critical on a larger system where the communication delays and, actually even more important in this case, the runtime manager cost in searching for available work, is much greater. This is because the runtime manager on each idle processor blindly polls the task queue of every processor, in round-robin order. A runtime manager that searches tasks hierarchically, such as proposed in [35], might yield a better asymptotic behavior at the cost of complicating the manager



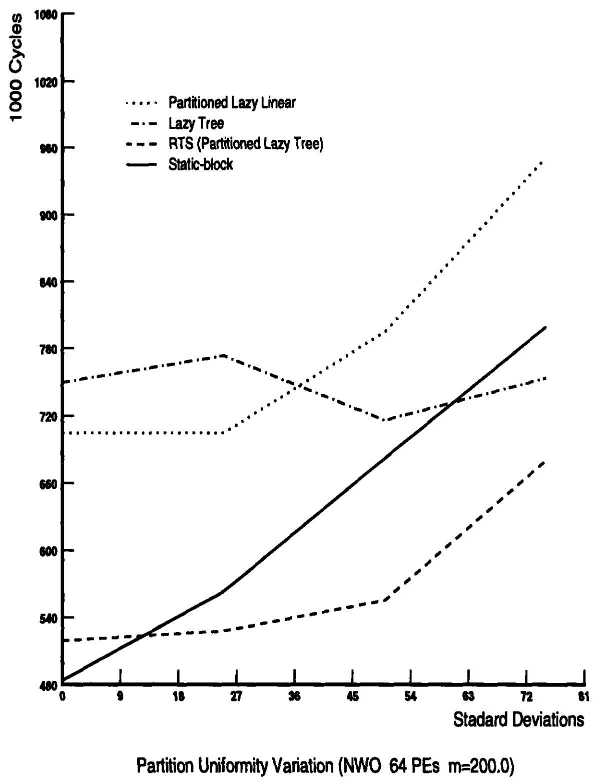


Figure 5-2: *Effects of Varying σ of Thread Size*, $m = 200$

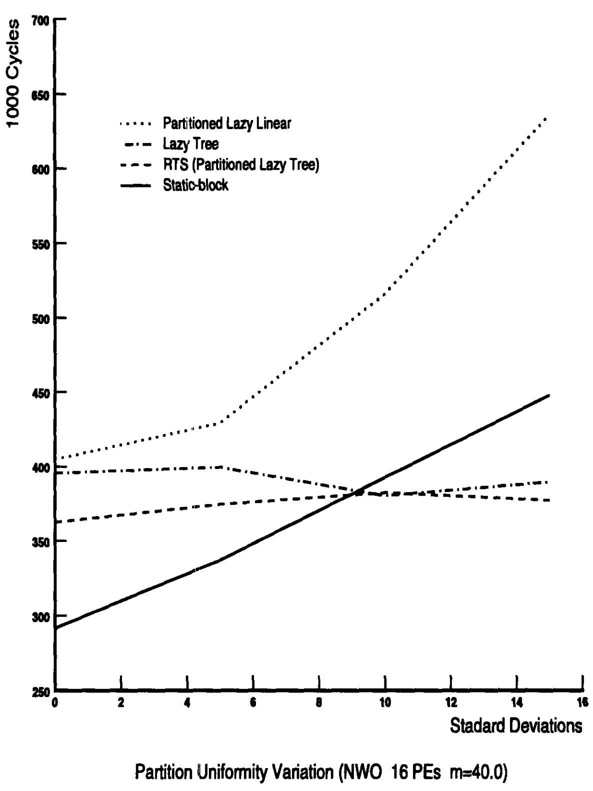
operation. Such a scheme consolidates multiple task searchers in the system through a hierarchy tree of middle-level managers to avoid duplication of efforts and to improve the worst-case time to find work anywhere in the system from a linear search to a logarithm of the number of processor nodes. Both graphs incidentally show that RTS starts to outperform static-block when σ is above the teens. This, of course, is dependent on a host of factors such as task granularity and runtime cost of each strategy.

With $m = 40$, each relaxation loop iteration thread averages only 1970 cycles in duration. In comparison, once a stealable task is located, an interprocessor load steal takes approximately 150 to 350 cycles for the futures in this program. The same experiment is repeated with the new setting; Table 5.3 and Figure 5-3 plots the results. Similar trends in the data as before is seen. The crossover between RTS and static-block now occurs at approximately 9 on 16 processors but not in the range tested at all on 64 processors. The thread granularity is simply too fine here to overcome the baseline runtime cost of dynamic load balancing on 64 processors for these test parameters. Again, the lazy linear results, ranging 2918137 – 3568461 on 64 processors and 4122152 – 5117163 on 16 processors, are an order magnitude slower than the rest. So its curves do not show up inside the graphs.

$m = 40$		16 Processors			64 Processors		
Strategy	σ	Execution Times			Execution Times		
Static-block	0	291628	—	—	118813	—	—
	5	331185	342927	337096	123343	125598	136205
	10	380166	410773	386520	142634	145326	152955
	15	429481	472220	441617	161736	165292	170543
Partitioned Lazy Linear	0	404659	—	—	175462	—	—
	5	475176	412714	399923	183910	179971	229439
	10	535960	505328	506342	217523	184165	199653
	15	565279	684341	657973	250904	256444	232079
RTS (Partitioned Lazy Tree)	0	362460	—	—	176491	—	—
	5	361999	366300	395155	161895	170955	176555
	10	373855	362098	410969	182663	165579	172445
	15	343891	363800	423812	186988	187065	173854
Lazy Linear	0	4891687	—	—	3073282	—	—
	5	4706215	4681352	5053293	3093825	3649347	3060321
	10	4581033	4402304	5065143	3045613	3017529	3085214
	15	4275738	4122152	5117163	2985252	2918137	3568461
Lazy Tree	0	395455	—	—	197560	—	—
	5	409379	383936	404764	210570	220494	258367
	10	355863	368598	416430	184389	222366	228942
	15	359647	364301	444615	213125	217313	223890

Table 5.3: *Simulation Results: Varying σ of Thread Size,* $m = 40$

Chapter 1 mentioned that dynamic load variations can come from asynchronous hardware operation, cache or page misses, or network routing. Another source of dynamic load variation is purely algorithm-induced. We examined GAMTEB, a photon-neutron transport program using the Monte Carlo technique. In it, each particle is fired randomly down a cylinder and its interaction with the field traced and its final exit condition recorded. The variations in compute cycles spent for each particle of a run on a uniprocessor are summarized in the table below. The standard deviation, σ, of the samples is normalized such that the mean, m, is 200, for comparison with the earlier test suite. When the particles are evenly distributed on the nodes of a multiprocessor, the dynamic load variations among the nodes will gradually diminish when more particles are allocated to each processor. This behavior is deducible from the central limit theorem [17], which states that the sum of a series of independent random variables has a decreasing variance when an increasing number of random variables in the series are summed. So we can expect the dynamic load variation



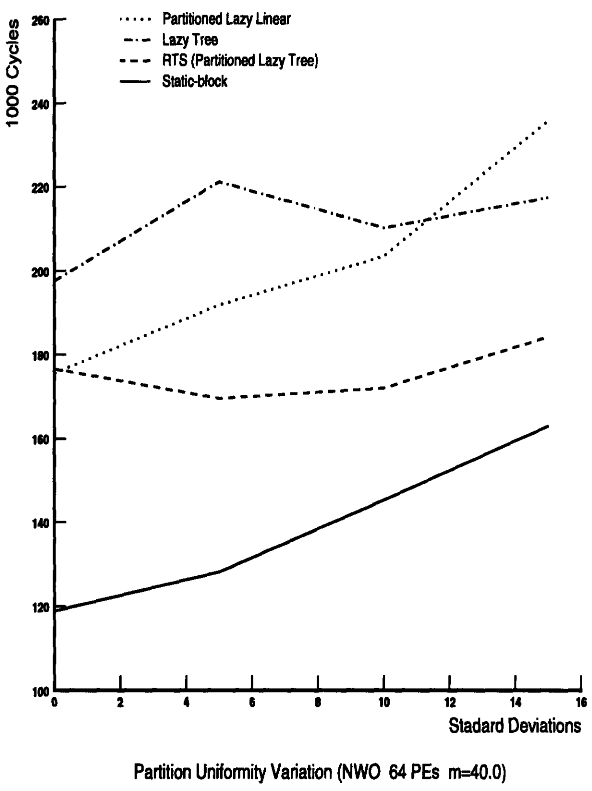


Figure 5-3: *Effects of Varying σ of Thread Size, m = 40*

induced purely by the random nature of the GAMTEB program algorithm to be less than the nominal single-particle-per-processor statistical variations cite when many more particles are simulated on relatively few processors.

GAMTEB(n=32)	: min 22.1	max 98.6	m 200.0	σ 90.0
GAMTEB(n=128)	: min 23.0	max 127.8	m 200.0	σ 87.5

We see from this set of experiments that the proposed RTS has performance that is competitive with alternative strategies. Even compared to the statically partitioned and aligned blocks, which has very low runtime overhead, under uniform runtime loads, RTS ranks a fairly close second with only fractionally worse speed. Under a wide variety of system sizes and runtime load variations, RTS has the distinction of having consistently performed very well against the others.

5.2 Impact of Partition Alignment

We set forth here to examine the contribution of the alignment phase to the performance of the compiled code. Alignment of task and data has shown in cases to be an important gain in SIMD and systolic systems [26, 43] where the computation at the nodes is fairly lean in comparison with the internode data movement costs. The close lockstep operation also makes impromptu remote fetches expensive since they hold up all the nodes. On the other hand, when most nodes fetch during the cycle, very bursty network activity results, causing excess congestion and delay. In a shared-memory MIMD system, the cost structures and functionalities are significantly different to require a re-assessment. The outcome is useful for deciding whether a more complex compile-time strategy is warranted.

The code in Figure 5-4 transposes array *a* into *b*, four elements at a time. The alignment process places partitions of the loop and array *b*, and the transpose-located partitions of array *a* into the same detachment, so that the array copying is entirely internal to each node. When it is bypassed, the default detachments are formed from the naturally-overlaid (*i.e.*, non-transposed) partitions of the loop and arrays, thus causing interprocessor data movement during the array copy. Each iteration copies four elements because of the four-word cache line size; thus an average of one extra network data fetch per iteration is required when alignment is bypassed. In either case the detachments are then serialized using compile-time static-block and RTS, or the runtime self-scheduling methods commonly seen in SPMD systems.

```
(define a (array (1024 256) 9))
(define b (array (256 1024) 0))

(doall (x 0 255 2)            ;; loop for tiles of 2
  (doall (y 0 1023 2)         ;;   by 2
    (set (aref b x y) (aref a y x))
    (set (aref b (+ x 1) y) (aref a y (+ x 1)))
    (set (aref b x (+ y 1)) (aref a (+ y 1) x))
    (set (aref b (+ x 1) (+ y 1)) (aref a (+ y 1) (+ x 1)))))
```

Figure 5-4: *Alignment Test Program*

Table 5.4 contains 16- and 64-processor simulation results on the effect of alignment. To distinguish better the contributions from a multiple of underlying factors, a set of runs are done in which the simulator is put into a "perfect memory" (PM) mode. In it, all memory fetches, whether local or remote, hit in the cache and thus take exactly one cycle to complete. It represents an absolute lower-bound on program execution time attainable by any alignment. In fact, it is even better than the best partition alignment since an optimal alignment puts all data required by a node into its local memory, which still takes 11 more cycles to access than a hit in the cache. But as we shall see, the PM data still reveals much. Data for compile-time aligned and unaligned runs are tabulated along with their ratios to the PM data. The ratio of unaligned to aligned runs are tabulated in the last column. The runtime thread scheduling methods are not amenable to alignment. The independent loop iterations are dispatched from a central site one at a time to requesting processors under self-scheduling [49], or a block of iterations at a time under guided self-scheduling [40], whose iteration block size is a constant fraction of the number of iterations remaining to be dispatched. Plots of the data are in Figure 5-5.

16 Processors						
Strategy	Perfect Mem.	Aligned	% of PM	Non-aligned	% of PM	% of Aligned
Static-block	809109	918105	88.1	1070887	75.6	85.7
RTS	1092877	1420555	76.9	1503399	72.7	94.5
Self-schedule	865458			7942880	10.9	
Guided S-S	870111			1702002	51.1	

64 Processors						
Strategy	Perfect Mem.	Aligned	% of PM	Non-aligned	% of PM	% of Aligned
Static-block	203745	216606	94.1	301643	67.5	71.8
RTS	283447	337924	83.9	398227	71.2	84.9
Self-schedule	309258			27004075	1.1	
Guided S-S	221733			2552842	8.7	

Table 5.4: *Simulation Results: Loop/Array Partition Alignment*

We observe that alignment is more effective (*i.e.*, the aligned partitions achieve a higher percentage of PM's performance) with the static-block than with RTS, in spite of the higher baseline overhead of RTS that tends to reduce the numerical percentage of similar effects. This is due to the interference of the runtime load stealing on the optimally aligned detachments. In the non-aligned runs, static-block shows a similar percentage reduction as RTS from that of PM's performance, because for both cases almost every array a fetch missing in the cache is remote. The greater performance degradation of non-aligned runs from the aligned runs (*i.e.*, "% of aligned") for static-block *vis.* RTS shows that it is more sensitive to, and also benefits more from, proper alignment. But the most notable observation is that, even in this very memory-intensive loop, the difference between good and poor alignment is not manifold, but rather a small fraction. This conclusion, drawn from performance comparisons of PM to the non-aligned, should not be casually generalized beyond

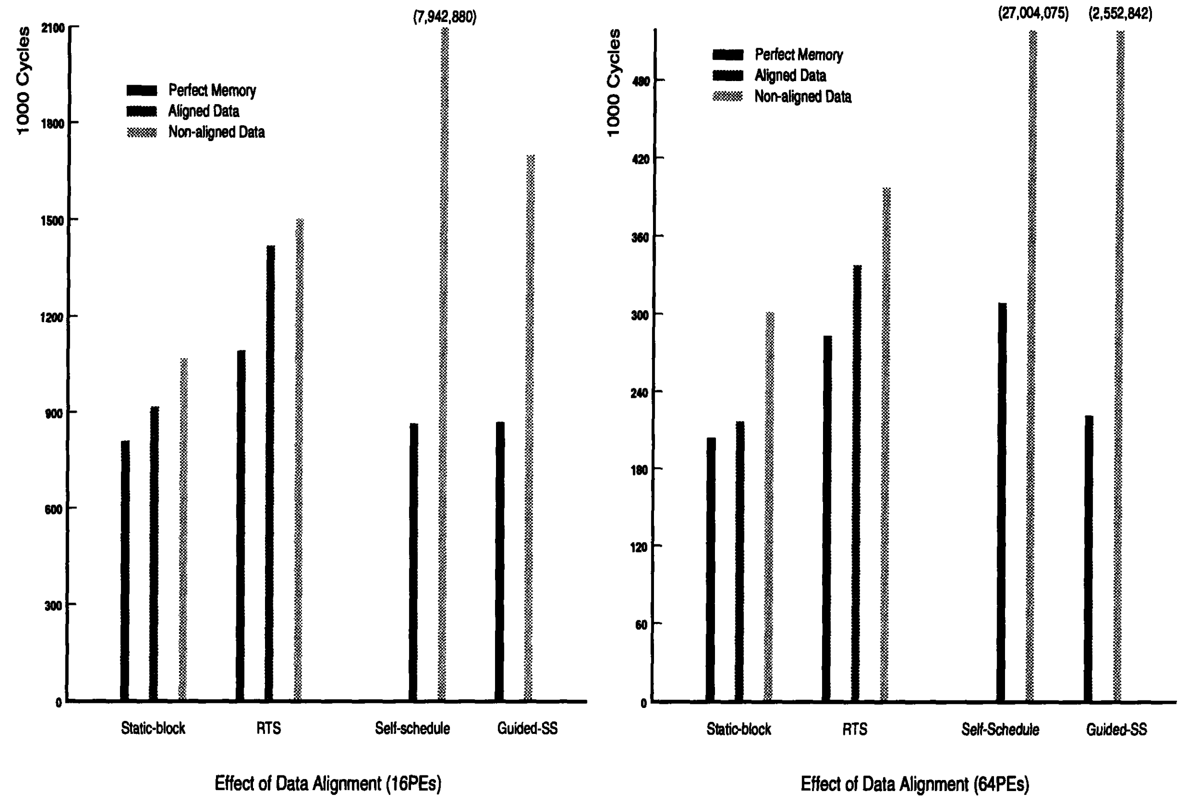


Figure 5-5: *Effects of Loop/Array Partition Alignment*

the 16- to 64-node Alewife machine. In Alewife, the computation and supporting (*e.g.*, coherent caching and network) operations on each node typically is much more complex than those of a SIMD machine, such that the internode data movement does not add a significant cost. Other overheads contributing to the high baseline cost per iteration include the full index coordinate translation that each array reference now undergoes, as per Section 4.1.3. The randomly staggered network accesses from relaxing the lockstep SIMD operation further alleviates network congestion and reduces the delay of network-bound memory accesses, as mentioned earlier.

The runtime strategies of self-scheduling and guided self-scheduling already show a high-degree of congestion at the central iteration dispatch site with 16 processors. With the expansion from 16 to 64 processors, they both exhibit a marked slow down in completion time. In order for these runtime schedulers to function properly in the Alewife system, either the thread granularity must be increased much more than we use here or the thread dispatch site must be distributed much more finely, thus incurring another layer of load-balancing protocol between the dispatching sites. In the extreme, a dispatch site is set up at each processor node and the load-balancing protocol may consist of an initial partition and alignment by the compiler as we have described and some kind of runtime negotiated trading of a single or a block of iteration indices between the sites. In other words, we have a strategy that is just like the RTS except for its lack of a fully revocable serialization: a thread is still not allowed to block in midstream and another one started. But this last point can be overcome with a conventional multi-tasking scheduler. The next section will focus on the runtime behavior for revoking intractable thread serializations.

5.3 Cost of Undoing Intractable Serialization

We demonstrate here another claim made in Section 1.3, that the system performance gradually degrades when the compiler has mistakenly serialized parallel threads against actual data dependencies. Unlike some of the other strategies used in our comparisons, it will neither deadlock nor produce the wrong results. Since unsynchronized data accesses between parallel threads are not guaranteed to occur in any specified order by the language semantics, they never raise any problems. The concern lies only in the synchronized data accesses (*e.g.*, J-structure references or future touches) in which a synchronization fault causes the task to be suspended and another task started that will eventually cause the original blocking condition to be removed.

The code in Figure 5-6 simulates such a scenario. Here, the i-th loop iteration sets the J-structure element a_i and reads either a_i, which succeeds, or a_{i+1}, which fails until iteration $i+1$ has completed. The decision function looks up a list to determine if the current iteration should break. It is produced by `(nrandom-lists` *index-range*`)`, which returns a list[2] of randomly-generated iterations each processor will break on. The thread serialization applied is RTS.

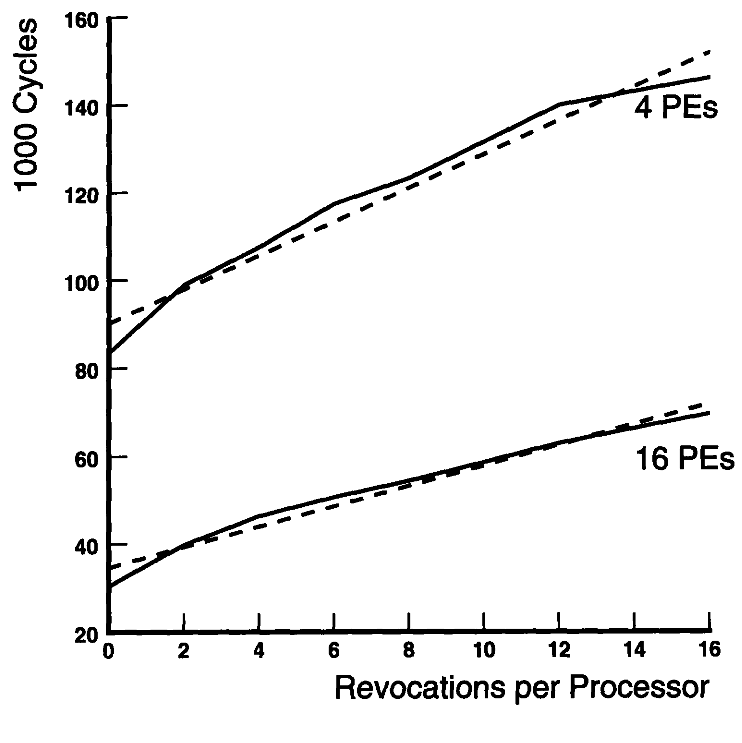
```
(define breakons (nrandom-lists 1024))   ;; index-range
(define a (jstruct (1024)))
(doall (i 0 1023 1)
  (set (jref a i) 0)
  (jref a (if (break? i breakons tid) (+ i 1) i))))
```

Figure 5-6: *Runtime Serialization Revocation Test Loop*

Figure 5-7 shows run time with the test loop partitioned for four and 16 processors. Dashed linear regression lines are plotted for each, which show close fit. The axis intercept is the non-blocking

[2]Actually a processor ID-indexed array of lists — to eliminate list-reading contention among the processors.

completion time, where the compiler has serialized the parallel threads in an order consistent with the runtime dependence relations, while the slope is the penalty for each required runtime revocation. The four-processor runs have intercept at 90220 and slope of 3867; the 16-processor runs have intercept at 34618 and slope of 2337. These figures are, of course, dependent on the system configurations, program structures, and runtime states. This scenario represents a concern from those trying to serialize parallel programs: without the revocability of serialization, parallel threads that are not dependence-free must be executed as independent tasks. But this revocability is also useful for parallelizing iterations that are mostly independent except at a few unforeseeable points. Revocation allows the runtime scheduler to bypass the threads blocked by dependence constraints without incurring the cost of requiring all threads to become independently scheduled tasks.



Runtime Serialization Revocation

Figure 5-7: *Incremental Cost of Revoking Compile-time Serialization*

5.4 The SIMPLE Benchmark Program

We present a fairly large scientific benchmark program, SIMPLE, and describe its compilation and running results on this implementation, along with results using a few other alternative schemes. The

SIMPLE code [9] solves a two-dimensional hydrodynamics and heat conduction problem. It has become one of the standards for benchmarking high-performance computers, so performance data from many commercial and research systems exist. Its top-level structure, major subcomponents, and dataflow dependencies are depicted in Figure 5-8.

With the exception of procedures `world` and `generate`, which set up the program constants and the initial conditions, the entire program is iterated once for each simulated time step. Each iteration starts by calculating the positions of the border zones surrounding the grid boundary by reflecting the adjacent interior point across the boundary, and by calculating their physical attributes (*i.e.*, mass density, viscosity, and pressure.) The problem is then divided into a hydrodynamics and a heat conduction phase. During the first phase, the velocity and position of the nodes are incremented based on the acceleration vector at each node. Then, new values of area, volume, and density at the new positions of the nodes are computed along with their intermediate values of pressure and temperature. The second phase transfers energy between adjacent zones to account for the heat conduction. This involves solving a system of linear difference equations by the *alternating direction implicit* (ADI) method to determine the final pressure and temperature of each zone.

For a grid size of $n \times n$, both of the major phases consist of some doubly nested loops generating $O(n^2)$ work for each simulated time step. But many others are one dimensional, such as the code for grid boundary handling, and the ADI code, whose inner loop is sequential; these cannot scale beyond n. Some of the loop iteration bodies perform table lookups that have loops with data-dependent termination, making vectorization and SIMD execution difficult. Scepter gives up on determining repetition counts for them; rather, it relies on runtime load balancing if needed.

We also encounter several reduction functions (*e.g.*, computing the extremum or summation over an array). Their fully associative and commutative properties could in principle be exploited in automating the partitioning so that their partial results are computed on each processor, before the final value is assembled via a fan-in tree among the processors. In our version, atomic updates to designated "accumulator" locations are used to compute these functions. Table 5.5 summarizes the looping and procedure call structures of SIMPLE. The "Sizes" field indicates the loop iteration count when running SIMPLE ($k \times l$), specifying the grid size of the program.

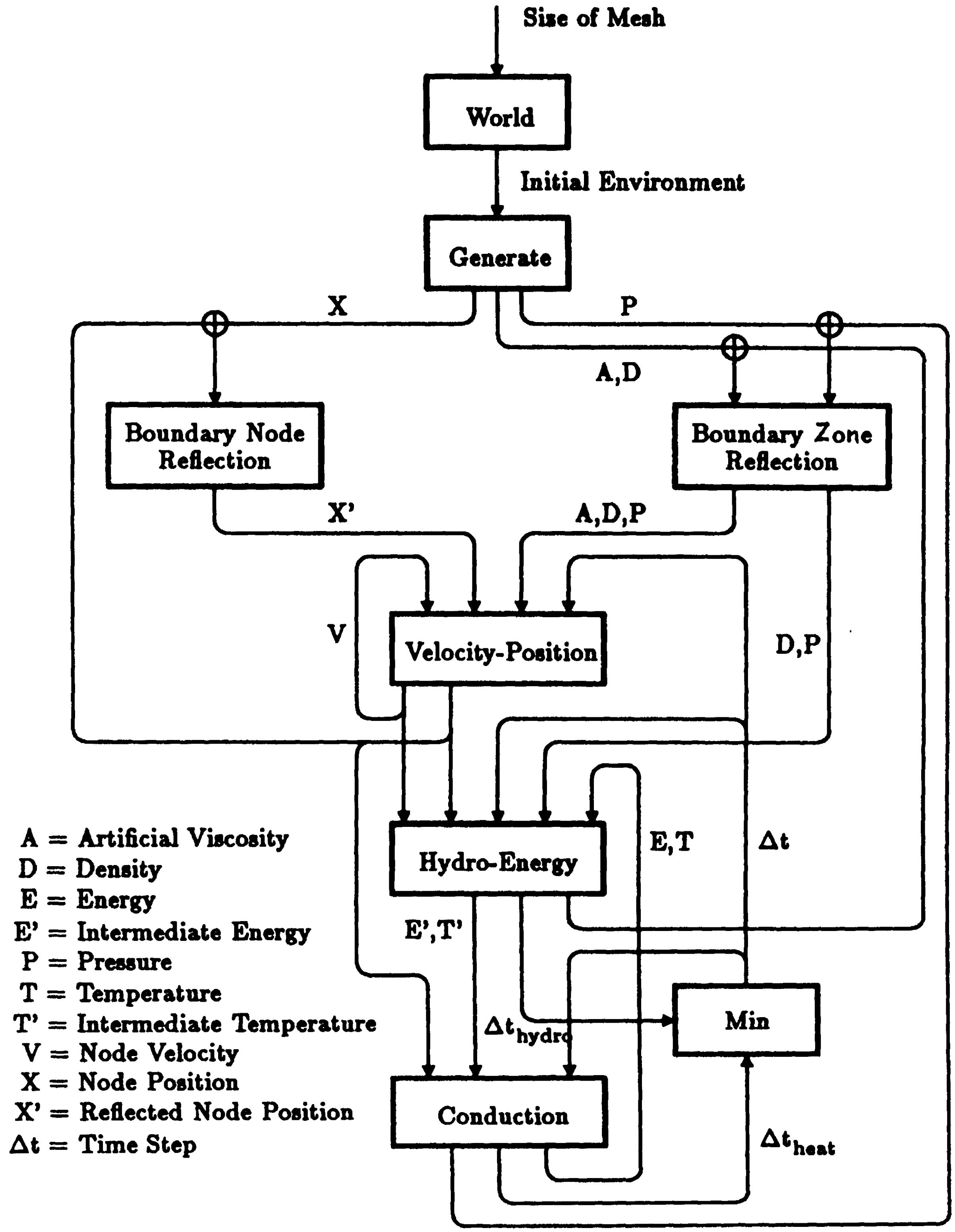


Figure 5-8: *Block diagram of SIMPLE*

5.4.1 Static Alignment of Loop/Array Partitions in SIMPLE

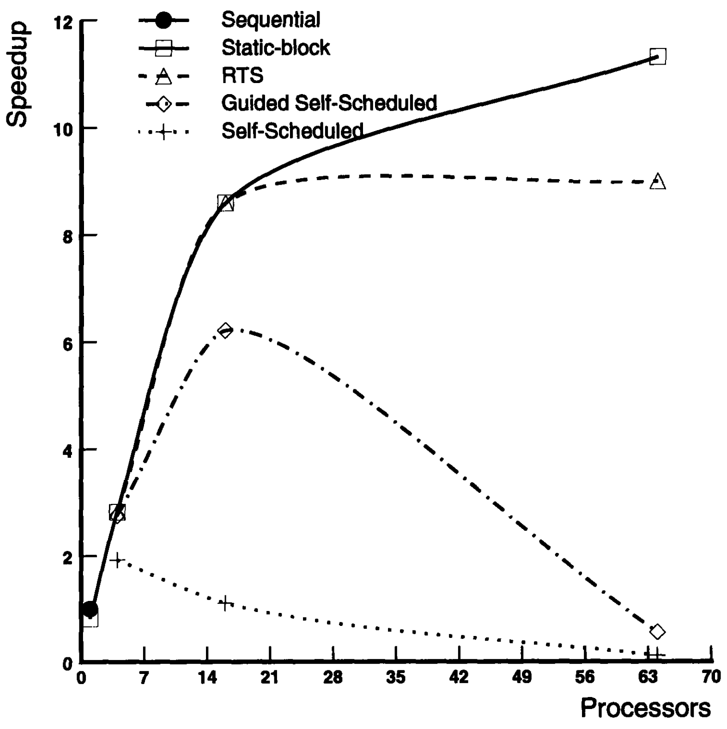
Since a loop partition communicates only with data partitions and vice versa, by examining the number of edges (see Section 4.2) emanating from each loop partition and the fraction of those that enter data partitions that are aligned to the loop partition, we can assess the effectiveness of the alignment phase in localizing memory accesses. Table 5.6 displays the figures, as recorded by the compiler, for each partitioned loop of SIMPLE (64×64). When partitioning for increasing numbers of processors, we observe that the rate of successfully localizing accesses decreases as expected. The total figures are tabulated at the bottom. The lower-than-expected alignment rate is due to the partitions being chosen at first, with the intent of optimizing cache hit rate and without concern for the later alignment stage. So, in attempting to align square and strip partitions, there is a certain failure rate in localizing data accesses that increases when the system is scaled. But as we have already noted, the actual performance impact of aligning partitions seem be relatively small on the current version of Alewife system under construction.[3]

5.4.2 Speedup of SIMPLE

Table 5.7 and Figure 5-9 show the execution time of SIMPLE (64×64), for one time step iteration, on varying numbers of processors and under several parallelism management strategies. The sequential run is a completely serial version, while the single-processor static-block has the loops and arrays "partitioned" for one processor, *i.e.*, with the loop/array address translations of Section 4.1.3. They all use the same baseline system: Scepter and Orbit compiler, Alewife machine, and runtime library. The speedup figures are all normalized to the fully serial version. The static-block and RTS show good gains up to 16 processors, but quickly level off beyond that machine size. The serial sections and many one-dimensional loops do not provide enough parallelism to keep the machine busy with the chosen grid size — at around 2 million cycles for a 16 processor run using static-block or RTS, the run time on a real processor is about 1/20 of a second. RTS operates with the runtime scheduler, while static-block has its runtime scheduler turned off. But as we had encountered earlier in the synthetic benchmark analyses, this scheduler, because of its round-robin task querying, becomes less effective, and more a burden, with increasing number of processors. We note once again that the

[3]Initial tests of the recently assembled working hardware prototype showed the network has several cycles higher latency than the expected delay assumed in the simulator. The effect of alignment is therefore underestimated in our simulation results.

entirely runtime managed self-scheduling shows a high-degree of congestion at the central iteration dispatch site with only 16 processors, and that even the guided self-scheduling loses much of its performance beyond 16 processors.



Speedup of Parallel SIMPLE(64 x 64)

Figure 5-9: *Speedup Curve of SIMPLE* (64×64)

5.5 Summary

Through the set of synthetic benchmark programs and a scientific application, we tested the characteristic behaviors of the RTS framework integrating compile-time and runtime parallelism management. These results validate the expected behaviors initially described only qualitatively in Chapter 1. We saw that by paying a small additional overhead to the static-block, RTS is able to achieve flexible runtime load balance and deadlock avoidance, and that by admitting compile-time pre-partitioning, pre-alignment, and pre-serialization, it is able to accommodate massively data-parallel computations that often overwhelm the runtime-managed self-scheduling, guided self-scheduling methods, and the future-based Mul-T programming paradigm.

In Procedure	Loops	Calls	Remarks
Simple	t	BoundaryNodeReflection, BoundaryZoneReflection, VelocityPosition, HydroWorkOnBoundary, HydroEnergy, Conduction	Sequential loop: time steps
BoundaryNodeReflection	k	ReflectNode	Boundary top/bottom rows
	l	ReflectNode	Boundary left/right columns
ReflectNode			
BoundaryZoneReflection	k		Boundary top/bottom rows
	l		Boundary left/right columns
VelocityPosition	$k \times l$		Allocate matrices
	$k \times l$		Over all points in parallel
HydroWorkOnBoundary	k		Boundary top/bottom rows
	l		Boundary left/right columns
	k		Sequential sum reduction
HydroEnergy	$k \times l$		Allocate matrices
	$k \times l$	ArtificialViscosity, TableLookup InvertEnergyTheta, PressurePoly	Over all points in parallel
	k		Sequential reductions
ArtificialViscosity			
Conduction	$k \times l$		Allocate matrices
	k		Initialize boundary columns in parallel
	$k \times l$		Over all points in parallel
	$k \times l$		Over all points in parallel
	l		Initialize boundary rows in parallel
	k		Initialize boundary columns in parallel
	$k \times l$		Parallel columns, sequential rows
	$k \times l$		Parallel columns, sequential rows backwards
	l		Initialize boundary rows in parallel
	$k \times l$		Parallel rows, sequential columns
	$k \times l$		Parallel rows, sequential columns backwards
	$k \times l$	TableLookup, EnergyPoly, PressurePoly	Over all points in parallel
	k		Sequential reductions
	l		Sequential reductions
InvertEnergyTheta	"while"	TableLookup, EnergyPoly	Data dependent iteration count
EnergyPoly			
PressurePoly			
TableLookup	"while"		Data dependent iteration count
Generate	$k \times l$		Array initializations in parallel
	$k \times l$		Array initializations in parallel
	$k \times l$		Array initializations in parallel

Table 5.5: *Loop and Call Structure in SIMPLE*

		Memory Accesses Made Local through Alignment		
In Procedure	Loop	4 PEs	16 PEs	64 PEs
BoundaryNodeReflection	k	548/1220	272/1220	132/1220
	l	548/1220	162/1220	80/1220
BoundaryZoneReflection	k	1200/1200	1200/1200	1180/1200
	l	290/1200	50/1200	20/1200
VelocityPosition	$k \times l$	104874/128374	91654/128374	82170/128374
HydroWorkOnBoundary	k	722/2190	462/2190	168/2190
HydroEnergy	$k \times l$	100849/113400	93346/113400	87200/113400
Conduction	k	30/60	15/60	8/60
	$k \times l$	19204/19470	18679/19470	17653/19470
	$k \times l$	19204/19470	18679/19470	17653/19470
	l	59/120	19/120	12/120
	k	62/90	46/90	38/90
	$k \times l$	5220/12600	2700/12600	1260/12600
	$k \times l$	5520/7200	4560/7200	4080/7200
	l	102/150	82/150	72/150
	$k \times l$	960/12600	2460/12600	1440/12600
	$k \times l$	3660/7200	1740/7200	960/7200
	$k \times l$	27893/34200	24746/34200	23192/34200
Generate	$k \times l$	5581/5581	5581/5581	5581/5581
	$k \times l$	47878/52200	44656/52200	40968/52200
	$k \times l$	1800/1800	1800/1800	1800/1800
Total		346285/422205	312909/422205	285667/422205
		82.01%	74.11%	67.66%

Table 5.6: *Localizing Memory Accesses in the Parallel Loops of SIMPLE* (64 $\times$ 64) *through Alignment*

	Processors (Time & Speedup)							
Strategies	1		4		16		64	
Sequential	18944862	1.000						
Static-block	23205369	0.816	6699269	2.828	2207309	8.583	1673348	11.322
RTS			6705150	2.825	2258634	8.388	2106327	8.994
Guided Self-schedule			6866618	2.759	3051180	6.209	34563411	0.548
Self-schedule			9848378	1.9236	17200908	1.101	168002657	0.1127

Table 5.7: *Speedup of SIMPLE* (64 $\times$ 64)

Chapter 6

Conclusion

We have presented here work on the design and construction of a framework to integrate compile-time and runtime parallelism management for MIMD multiprocessors. Under the Revocable Thread Serialization (RTS) framework, the compiler is given the duty to partition, align, and then provisionally serialize the fine-grained parallel threads specified in the input program to run on a known number of processor nodes. Under an ideal scenario, all these decisions will, in fact, be correct and will lead to an optimized execution as does a statically-scheduled system with a perfect compile-time accounting of all runtime timings. There would only be the addition of the small overhead of the provisional serialization cost in the RTS runs. But when the number of runtime processors is changed from the expected number, when the amount of work assigned to each processor is changed due to data-dependent conditionals, when the timing of various machine operations are changed by unanticipated network congestions or cache activities, or when the scheduled thread execution order is inconsistent with the actual data dependencies, the RTS framework allows the runtime scheduler to re-balance the processor loads or to rectify the impending deadlock by migrating or reordering the threads.

We saw evidence in Chapter 5 that the individual characteristics of RTS, verified through various synthetic programs and finally through an actual scientific benchmark program, appears as claimed in comparison with alternative strategies. The central observation is its unique ability to adapt to various runtime uncertainties through gradual performance degradation, starting from a point very close to the optimally statically-scheduled management strategy with a totally-anticipated runtime scenario. We also saw the effects and the effectiveness of thread and data alignment in the context of the Alewife machine.

Functionally, the work on RTS provides a seamless way to merge the programming paradigms and the operating mechanisms that support efficient control-parallel *and* data-parallel computations. Programmers do not need to resort to different programming models and methods to write these distinct classes of application programs.

6.1 Future Work

There are several major areas of immediate extensions to the current work: adding a sequential language front-end to Scepter, updating the analysis and transformation modules with better or newer techniques, implementing and introducing controlling policies for the RTS framework, and improving the scalability of the runtime manager.

Scepter accepts parallel programs in Mul-T as input. It should be somewhat trivial adapting it for other parallel programming languages. An interesting proposition is to target Mul-T as the output language of a parallelizing compiler for a sequential language such as Fortran. The parallelizing compiler would be charged with the task of discovering as much parallelism as possible, while Scepter would fold the excess parallelism onto available processors.

The analysis modules in place now are adequate for handling the input programs we have used. Their repertoire of functions can be enlarged by substituting newer or more complex methods for any of the modules. Of the transformation modules described in Chapter 4, the partitioning module is the most studied in the literature and it would be of interest to adapt one of the more general schemes (the existing one produces rectangular partitions) and examine the implications to code generation and to compare the performances. Other refinements such as decomposing reduction functions along partitions are useful. This occurs when, for example, in summing the elements of a partitioned array. When the array is partitioned, an accumulator should be established on each processor for the values in the subregion of the array mapped to the processor. Only when the final sum for each accumulator is obtained should interprocessor communication be used to merge the total value from among the processors. This two-level distributed reduction function optimization cannot be programmed from the input program and should therefore be generated by the compiler.

The currently implemented controlling policy for determining the static partitioning granularity is a very simple one — all loops and arrays are distributed evenly onto all the configured processors. There are several other alternative policies described in Section 4.4; some can make use of partition sizes determined by a top-down partitioner such as [41]. Feeding from such a program a select subset of processors to use for a particular loop or array object into Scepter's partitioning phase will direct it to allocate only those processors for the object, leaving the rest for concurrent execution of other loops. This space-sharing arrangement moves the operating point further to the left on the speedup curve for each of the loops to improve overall efficiency as explained in Section 1.1. The RTS framework is thus able to incorporate the decision choices from a hierarchical static partitioning and scheduling algorithm.

The current version of the runtime scheduler has seen its limitation on performance scalability in those results in Chapter 5 where it was used. A more scalable way to search for and distribute work such as [35] may be substituted. There are also ready reasons, given in Sections 4.3.2 and 4.3.3 to support the thread scheduling scenarios described therein, for using multiple priority-levels of task and future queues for the runtime scheduler. If the machine is partitioned for space sharing as in the above paragraph, the runtime scheduler may do well to steal preferentially from the processors in its own machine partition, before going outside it for work.

The use of the lazy future to serialize threads provisionally serves two main purposes: for load balancing and for resolving deadlocks. When the deadlocking possibility can be fully disproven by the compiler, the load balancing effect can still be achieved if very small threads are joined together through hard in-lining. This may require parallel loops be unrolled several iterations first, for example, before the expanded body is turned into a future. This transformation helps to amortize the cost of very fine-grained future calls.

Finally, the alignment results from simulation need to be re-evaluated on the Alewife hardware prototype and on other machine platforms. As we have pointed out, initial tests on the hardware prototype showed the network to be slower than had been assumed by the simulator. This has the effect of understating the performance impact of aligning threads and data.

Appendix A

Program Analyses in Scepter

Scepter is a source-to-source parallelism restructuring package that implements the compiler phases of the Revocable Thread Serialization (RTS) framework. Scepter (structural diagram in Figure 3-1) generates and maintains an internal hierarchical graphical representation of the source Mul-T program called WAIF, a recursive acronym for *WAIF is Alewife's Intermediate Form.* WAIF has two abstraction levels: a program graph with data access edges (WAIF-PG) and, overlaid on it, an abstracted thread/data communication graph (WAIF-CG). WAIF-PG is a custom version of a conventional program dependence graph. From a series of scalar, alias, dynamic, index, and escape analyses on WAIF-PG, Scepter produces WAIF-CG to summarize the runtime interaction of threads and data objects implied by the underlying program graph. This appendix briefly explains the approaches taken in these program analyses to produce the WAIF-CG, whose purpose is to support the later transformations for thread/data partitioning, alignment, and serialization.

From a Mul-T source program, alpha conversions assign bound variables unique names and expands constants and macros. The de-syntax-sugared code, consisting of only around a dozen different forms, is translated into a WAIF-PG graph, with each node representing an expression list. In addition, in the WAIF-CG graph, a thread node is created to represent each future, and a data node for each data object. After the conversion and the initial WAIF node generation, two classes of analysis and transformation modules are successively applied to the WAIF graph. Class One modules analyze WAIF-PG to discover and refine communication relationships implied in the program and build the WAIF-CG to store this information:

- Static scalar alias and dataflow analyses on variable references provide the static data dependence edges for the program graph and communication edges for the thread/data communication graph.

- Dynamic analysis differentiates invocations of the same textual code. This treatment applies for loops, recursions, function calls, *etc.*, and data structures dynamically allocated inside these replicating program structures. Thread and data nodes are annotated with nesting-level information when available. Non-inlined calls (*i.e.*, calls that have a symbol referring to an external function, instead of a lambda form at its first, functional, position) may cause subgraphs containing dynamic sections to be replicated.

- Index analysis refines dependence relations on data nodes down to *index subranges* or individual elements of arrays. Here each initial data node is split into as many subnodes as it has distinct subranges.

- Escaping references analysis detects, for unknown calls, references passed into or out of an analyzed region. The unknown calls arise from cases such as incremental compilation or indirect calls. References escaping from a region is assumed to be used somewhere outside. A data array that has been partitioned must have its address translation function passed out at runtime, and references to elements of an externally-defined array must check for such an addressing function in case it has been partitioned.

Class Two modules analyze the WAIF-CG graph produced by Class One modules and, along with some system specifications, restructure the WAIF-PG graph. These are described in depth in Chapters 3 and 4. Finally, the code generator translates the restructured WAIF program graph into a restructured Mul-T program.

A.1 Static Scalar Alias and Dataflow Analyses

This section describes the static scalar dataflow analysis that traverses a WAIF-PG graph and deduces the possible communication relationships. *Static* signifies that the bodies of loops (and recursions) are not unrolled or otherwise considered to be multiple-instanced, and the branches of conditionals are both considered to be executed. *Scalar* implies that the analysis does not distinguish dependences and aliases beyond their object identity. All data structures are treated as atomic entities (*i.e.*, references to distinct elements of an array are only recognized as references to the same array; distinguishing indices values in these cases will be addressed by the index analyzer). The determination of these relationships is conservative. Some of the communications identified here will in fact not be necessary. These may be identified and refined or removed later by more specific analyses. In the rest of this section, the notion of *freevars* of an expression is introduced first, followed by the propagation of freevar values in constructing alias sets of variables. Finally, we show how communication edges are inferred from the collected information.

A.1.1 The Freevars

Associated with each WAIF-PG node is a list of *freevars*,[1] which are the variables referenced within an expression but not bound in it. These freevars are significant to our analyses because they constitute the external communications required to evaluate the expression. Each entry in the freevars list keeps a record of the number and type of references made of the variable by the node and its descendants. Variable references can only appear in a `ref` or a `set!` node, which reads or writes the variable value, respectively. Based on the type of access of these nodes, we further make a distinction between referencing the variable value itself (*i.e.*, a scalar or an entire data structure) and referencing through the variable value (*i.e.*, for an element of an array or a selector, such as the `car`, of a cons or a structure). There are thus four access types: read, write, read dereferenced, and write dereferenced. At each internal (non-leaf) WAIF node, the freevars list is computed as a union of the freevars of all its descendant nodes, with the freevar access counters summed whenever the freevars merged in the union refer to the same variable. At we traverse up the WAIF-PG, all those freevars which become bound at a node are dropped from the list propagated furthur upwards, but their freevar access counters are first summed into the bound variable's access counters.

A different freevar propagation procedure applies in regards to future nodes — the parent node of a future should not see any references made by the body of the `future` expression, since that is not a part of its thread. So the freevars of a future node, as seen by its parent node, is always null; although the freevars list kept at the node still reflects the references external to the future, plus the references needed to create the future closure.

[1]The definition of the structures of WAIF nodes are documented in [32].

A.1.2 Constructing Alias Sets

Concurrently to recording and propagating the freevars, we also record variable definitions and assignments to track the propagation of values. This would not be of immediate interest in a static analysis, but since data structures are passed and assigned by reference in Mul-T, several different names could refer to the same array or structure. By noting the transfer of variables' values, which occurs at a `set!` or a `call` node where formals are bound to actuals, a complete set of alias names to a given data structure-containing variable can be traced. In the absence of further information, we must assume that an access dereferenced through any one of these variables constitutes a possible communication link with accesses dereferenced through all other variables in its *alias set*.

We cross record for each variable binding or assignment the left (destination) and right (source) variables involved in their respective raliases and laliases WAIF-PG node fields. For non-inlined function calls, the same is done for bindings of actuals to formals; however, if the function symbol's value cannot be found statically (due to separate compilation, runtime binding, *etc.*), then all the actual arguments are marked as escaping values and later handled by the escape analysis.

The *alias set*, $\mathcal{A}(v)$, for variable v can be computed from the simple raliases and laliases relations by forming a set, $\mathcal{R}(v)$, of an raliases-transitive closure on the variable, and then forming the set $\mathcal{A}(v) = \mathcal{L}(\mathcal{R}(v))$, of laliases-transitive closure over the set $\mathcal{R}$. Consider the set of bindings in the following code fragment:

```
   ...
     (f t)
   ...
       (f s)
   ...

(define (f v)
   ...
    (g v)
   ...

(define (g u)
   ...
```

Figure A-1 illustrates the alias set for v in an alias-relation lattice as the shaded region. The alias set of a variable cannot be computed incrementally from its lalias or ralias neighbors'. The alias set of s does not include t, for example, but may contain others that are not in v's alias set. In this construction, all rightmost variable nodes in the figure are associated with a data node, since that must be where the data values are either created or their references first passed into the code fragment under analysis. In particular, these include all places where calls such as `make-array` are made. The data nodes a and b are thus the only data structures that an write dereferenced through v can possibly alter. We shall refer to this set of data nodes as the *reachable set* of variable v. Note that we do assume these rightmost nodes are not aliased further, unless told explicitly otherwise. Also, v in fact is never bound to both the values of s and t; this is only an artifact of the static analysis. The dynamic analyzer has an opportunity to rectify this, for example, by replicating[2] (*i.e.*, inlining) the function body of `f`.

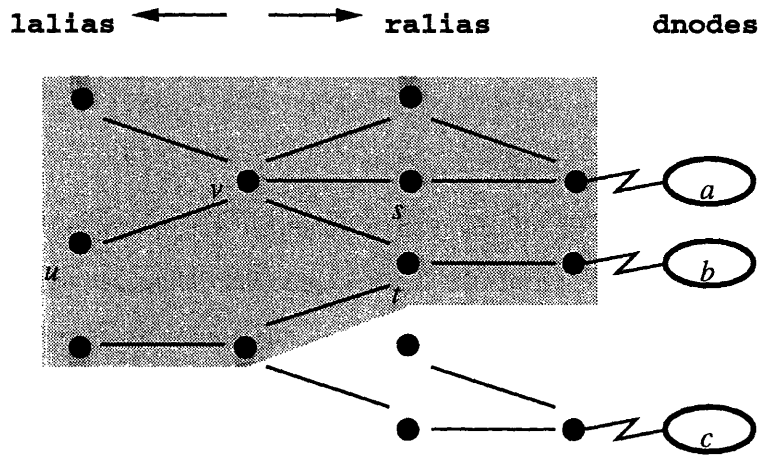


Figure A-1: *The Alias Set of Variable v within the Alias-relation Lattice*

A.1.3 Deriving Communication Edges

The static scalar communication edges can now be computed from the freevars list and the alias set given above. We create, for each thread node, a communication edge for each of the following:

- Freevars of the future expression of the thread (to and from the thread node one of whose lambdas binds the freevar),
- Indirect reference through the freevars of the future expression of the thread (to and from each data node in the reachable set of the freevar),

[2]This replication may be either physical or just logically annotated to distinguish the actual instances.

- Freevars of each non-inlined lambdas called by the future expression (to and from the thread node one of whose lambdas binds the freevar),
- Indirect references through the freevars of each non-inlined lambda called by the future expression (to and from each data node in the reachable set of the freevar),
- Return values of the future expression (to and from the thread node which called it),
- Indirect references through the bound variables of each lambda (inlined and non-inlined) called by the future expression (to and from each data node in the reachable set of those bound variables).

In the first and third conditions above, we implicitly assumed that lambda-bound variables (*i.e.*, the "local stack frame") are allocated on the processor executing the lambda[3]. The width of each edge corresponding to each variable reference is the sum of all the freevar counters found in the above search for that variable[4]. This should be scaled by the size of each variable access, which is otherwise assumed to be the normalized unity.

A.2 Dynamic Analysis

One textual piece of code may in fact be executed a number of times or perhaps not at all, as dictated by the control structures of the program: non-inlined calls, conditionals, loops, and recursions. The dynamic analysis highlights the multiple-instance aspects of some subgraphs. As we shall see, WAIF graph components can be annotated to indicate their instance space if they occur in a regular structure (*e.g.*, simple loops), or entire subgraphs can be replicated if there are a few sufficiently irregular instances (*e.g.*, for non-inlined calls).

A.2.1 Annotating Dynamic Instances

We use annotations to denote multiple-instantiations of a subgraph when the control structure is simple and regular, such as for loops with constant-stride induction variables. This is logically equivalent to an actual replication of the graph and may be viewed as a short-hand notation for it.

[3]This is consistent with the Mul-T compiler's code generator.

[4]Except indirect structure references are kept as individual edges, because they may really be for distinct locations, once the index analyzer examines them.

The annotation syntax offers an extra benefit over replication in that the number of instances need not be a constant determinable at compile-time.

The thread node's `enclosing-loops` field contains a list of (logically) enclosing loop nodes, derived from a traversal of the call graph and the program graph. From this information the `instances` list is created by visiting each loop node, extracting its induction variable ($ivar$), range (low and $high$), and $stride$ specifications. The permissible range and stride values are expressions consisting of constants and basic arithmetic functions of other variables. When they are all reducible to constants at compile-time, an exact number of instances for this subgraph can be determined. Analogously, when a data structure is dynamically allocated within an iterated code block, we can identify the distinct instances of the data node, all of which are textually represented by the same identifier, by similarly annotating its `enclosing-loops` and `instances` fields. With multiple-instanced thread and data nodes come the multiple-instanced edges. The `instances` list on an edge restricts the validity of the edge to those instances of the thread node.

This allows instances of threads with slightly different communication patterns to be jointly represented in our notation. The `instance-map` function specifies the mapping from the instance tuple of the `from` object to that of the `to` object:

$$(\mathrm{ivar}_1^{\mathrm{t}}, \mathrm{ivar}_2^{\mathrm{t}}, \ldots, \mathrm{ivar}_{n_{\mathrm{t}}}^{\mathrm{t}}) = \texttt{instance-map}\ \ (\mathrm{ivar}_1^{\mathrm{f}}, \mathrm{ivar}_2^{\mathrm{f}}, \ldots, \mathrm{ivar}_{n_{\mathrm{f}}}^{\mathrm{f}})$$

for a communication edge from the n_{f}-tupled instance of the WAIF-CG `from` node to the n_{t}-tupled instance of the `to` node.

The importance of the extra information gained from the dynamic analysis is seen in the example code below, in which we shall concentrate on the inter-iteration data dependences of the outermost loop.[5] The code computes $tsteps$ of the relaxation

$$a_{i,j} = (a_{i,j} + a_{i,j-1} + a_{i,j+1} + a_{i-1,j} + a_{i+1,j})/5$$

For now ignore the array indices and simply treat them as scalar references. We shall take up analysis and treatment of the array indices in the two inner loops in the next section.

[5]The loop is expressed as a tail recursive call

```
(define (loop t a)
  (if (< t tsteps)
      (let ((newa (make-jstruct n n)))
        (future
          (doall (i 1 n 1)
            (doall (j 1 n 1)
              (set (aref a i j)
                   (/ (+ (aref a i j) (aref newa i (- j 1))
                         (aref a i (+ j 1)) (aref newa (- i 1) j)
                         (aref a (+ i 1) j))
                      5))))))
        (loop (+ t 1) newa))
      newa))

(loop 0 a0)
```

There are in fact *tsteps* logical copies of the $n \times n$ array, each with a lifetime lasting two consecutive outer iterations. Dynamic analysis labels each reference through a in the innermost loop as an access of the data node of the `make-jstruct` from iteration $t - 1$[6], and each reference through *newa* as an access to the data node of the current outer iteration's `make-jstruct`. For our example, the two communication edges in WAIF-CG corresponding to each a reference would be

> [from: a0, to: thread#foo, instances: (), instance-map: (0, i, j), count: 1, . . .][7]

and

> [from: newa, to: thread#foo, instances: (t, 1, tsteps - 1, 1), instance-map: (t + 1, i, j), count: tstep - 1, . . .].

Knowing this is useful for the domain decomposition process, as well as for future compile-time storage management and reclamation. In contrast, without the analysis, we would still have a valid, albeit less exact, graph in which all a's and *newa*'s are aliased to the single data node created for the single textual `make-jstruct` found by the static analyzer. The simpler representation would force a single partitioning strategy on all instances of the data structure and preclude a lifetime analysis to reuse storage.

Our notation breaks down with call graphs that are not trees, *e.g.*, if the thread is inside a loop which is called from a few locations. This may cause the thread to have several different

[6]Except for those in the first iteration, which access the data node of the external variable $a0$.

[7]thread#foo is the thread-id assigned to the innermost loop's body.

enclosing-loops lists, one for each path up the call and program graphs. There are two ready solutions, one of which is to permit a regular-expression-like syntax for enclosing-loops and instances. At a branch along the path, alternates ([A | B]) can be specified. The Kleene star notation can be modified to specify complex enclosing loops (*i.e.*, A * n for n iterations of path A). We have opted to adopt the simpler solution of replicating the subgraph and permitting only simple enclosing-loops lists.

A.2.2 Replicating Subgraphs of Dynamic Instances

When the total number of instances is known, replicating subgraphs is the most general way to represent dynamic instances[8]. Whole loops and recursions may be "unrolled" this way; but the much more likely use for this procedure is for non-inlined calls. As mentioned above, the complications resulting from having to maintain loop nesting structure information across non-inlined calls probably justifies the brute force in graph replication. Replicating a subgraph is straightforward, and requires no special support or modifications in the WAIF specification.

A.3 Index Analysis

Index analysis refines the communication relationship involving arrays discovered earlier in scalar analysis. For the WAIF graph, it resolves references of an array data node to *index subranges*[9] or individual elements of that node. Given an un-indexed edge of an array data node produced by the scalar analyzer, which would be interpreted as meaning a dependence on the whole array itself, the index analyzer annotates it with the most-specific indices range it could find. This may then result in a splitting of the data node itself, which we will describe later. The functional mapping of induction variables to array indices extracted from the WAIF-PG is kept in the WAIF-CG edges. It appears in the form of a multi-valued function:

$$(index_1, index_2, \ldots, index_d) = \texttt{indices-map}\ (ivar_1, ivar_2, \ldots, ivar_n)$$

describes a set of edges denoting a d-dimension array accessed from an nth-level loop, where the $index$'s are ordered as the indices field of the corresponding data node and the $ivars$'s as the

[8]Again, ignoring space considerations for now.

[9]In current implementation specification, an index subrange for an n-dimensional array reference is a subset of the valid indices that can be denoted by the linear expression $c_0 + c_1 i_1 + c_2 i_2 + \ldots + c_n i_n$, where $i_k \mid 1 \leq k \leq n$ are the n indices and $c_k \mid 0 \leq k \leq n$ are integers.

`instances` field of the thread node. The illustration in Figure A-2 shows the instance space to indices space mapping for the inner loops of the example in the last section:

```
(doall (i 1 n 1)
  (doall (j 1 n 1)
    (set (aref a i j)
         (/ (+ (aref a i j) (aref newa i (- j 1))
               (aref a i (+ j 1)) (aref newa (- i 1) j)
               (aref a (+ i 1) j))
            5))))
```

Each of the five edges in the figure is actually an $(i \times j)$-instanced object, as are the data nodes and the thread node. The edges are labeled with their 2-valued `indices-map` functions. The actual WAIF-CG graph is shown just below.

A.3.1 Splitting Data Nodes

A common operation on the WAIF-CG data nodes is, what are all the communication edges with respect to a particular element, or a subrange of indices, of an array? To permit efficient lookup of such queries, a hierarchical structure is used to store interval information. When the index analyzer determines that two edges of a data node have indices ranges that never overlap each other, the data node is split into two subnodes covering those ranges.[10] The edges are then moved to the subnodes to indicate their relative independence. An edge whose indices range spans across both subnodes will point to the parent node. As more edges are added, the hierarchy expands in depth and breadth. Any internal node of the hierarchy not having an edge is redundant and removed, therefore the resulting tree is neither binary nor balanced. This construction preserves the *inclusion* property on the hierarchy: the indices range of a parent node always covers the indices ranges of all of its children nodes. To find all the edges which may reference a certain subrange, we look for edges of the leaf node(s) covering that subrange and all those of their ancestral nodes.

A.4 Escaping References Analysis

The only contribution of the escaping references analyzer to the definition of WAIF is the data node's `escaped?` field. This field is set if a reference to the data structure is ever passed into

[10]These non-overlapping subnodes must completely cover the indices range of the original node.

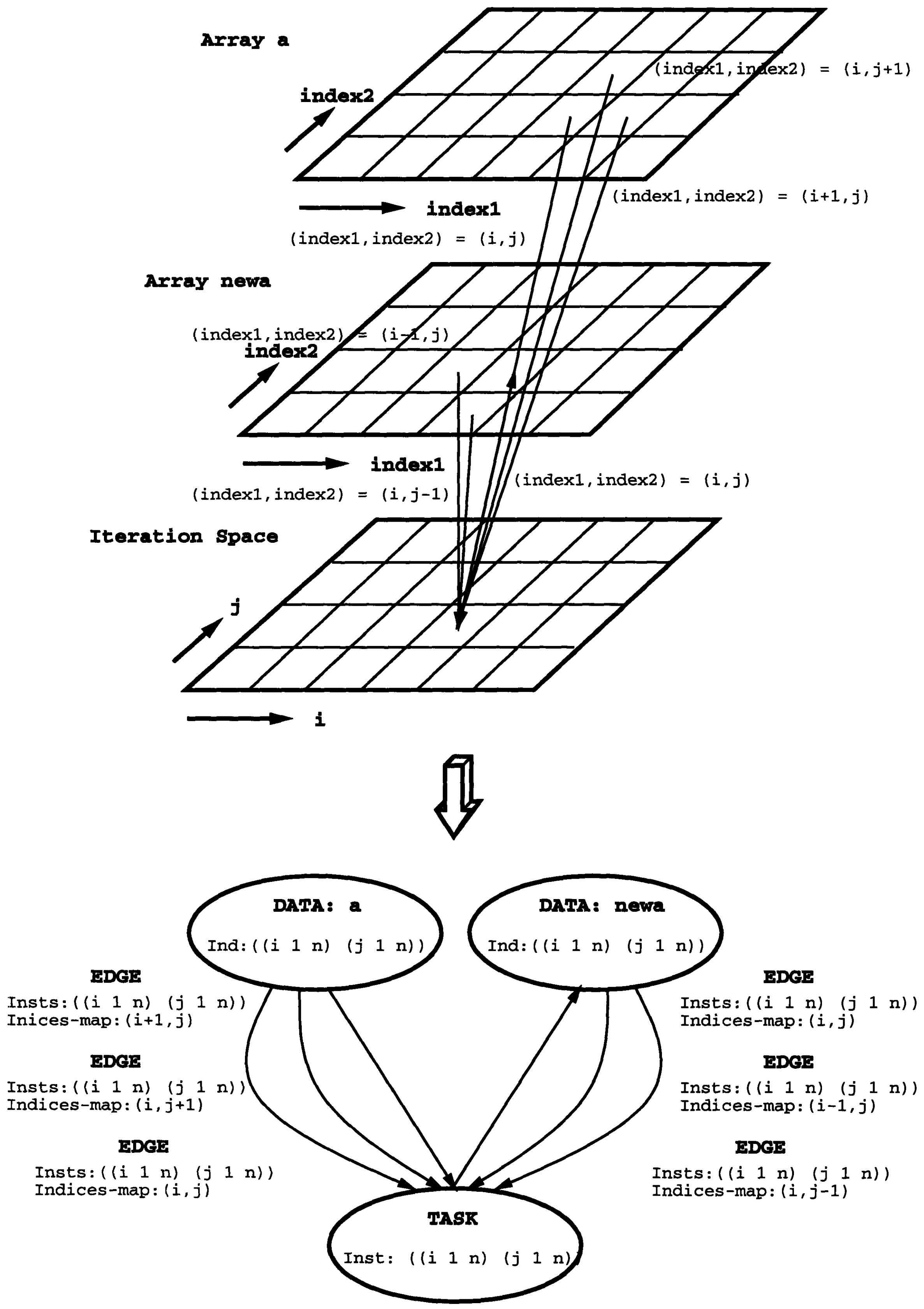


Figure A-2: *Representing Instance–Indices Mapping in WAIF-CG* or out of the scope of Scepter's analyses. When an array is distributed on to multiple processors by Scepter, the escaping data handling described in Section 4.1.3 is needed to insure the data's accessibility for both the program been analyzed and the external functions uniformly.

Bibliography

[1] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. To appear.

[2] Anant Agarwal, John Guttag, and Marios Papaefthymiou. Memory Assignment for Multiprocessor Caches through Graph Coloring. Technical Report Tech. Memo MIT-LCS-TM-468, MIT Lab for Computer Science, May 1992.

[3] Arvind and D. E. Culler. *Dataflow Architectures*, pages 225–253. Annual Reviews, Inc., Palo Alto, CA, 1986.

[4] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, September 1986.

[5] J. Beethem, M. Denneau, and D. Weingarten. The GF11 Supercomputer. In *Proceedings of the 12th Annual Int. Symposium on Computer Architecture*, pages 108–115, 1985.

[6] David Chaiken and Kirk Johnson. NWO User's Manual. ALEWIFE Memo No. 36, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1993.

[7] Marina C. Chen. Can Parallel Machines be Made Easy to Program? A Data-parallel Model for Functional Languages. Technical Report YALEU/DCS/RR-556, Yale University, August 1987.

[8] *The Connection Machine CM-5: Technical Summary*. Thinking Machines Corporation, November 1992.

[9] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The Simple Code. Technical Report UCID-17715, Lawrence Livermore Laboratory, February 1978.

[10] D. E. Culler. Resource Management for the Tagged-token Dataflow Architecture. Technical Report TR-332, MIT LCS, January 1985.

[11] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the International Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[12] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *Proc. of the Intl. Conf. Parallel Processing*, pages 836–844, August 1986.

[13] D. Kranz and R. Halstead and E. Mohr. Mul-T, A High-Performance Parallel Lisp. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[14] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *IFIP Congress*, 1989.

[15] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. Technical Report RC 11552 (55212), IBM T. J. Watson Research Center, Yorktown Heights, November 1986.

[16] E. Mohr and D. Kranz and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 185–197, June 1990.

[17] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1, chapter 10. John Wiley, Palo Alto, CA, 1950.

[18] C. M. Flaig. VLSI Mesh Routing Systems. Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[19] R. P. Gabriel and J. McCarthy. Queue-based Multi-processing Lisp. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[20] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[21] J. Gurd, W. Bohm, and Y. M. Teo. Performance Issues in Dataflow Machines. *Future Generations Computer Systems*, 3:285–297, 1987.

[22] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[23] S. Hiranandani, K. Kennedy, and C. W. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. Technical Report Tech Report CRPC-TR91162, Rice University, April 1991.

[24] D. E. Hudak and S. G. Abraham. Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops. In *ACM International Conf. on Supercomputing*, pages 187–200, 1990.

[25] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proc. of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[26] K. Knobe, J. D. Lukas, and G. L. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[27] J. Kubiatowicz. User's Manual for the A-1000 Communications and Memory Management Unit. ALEWIFE Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[28] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *ACM International Conf. on Supercomputing*, 1993.

[29] M. Lam. A Systolic Array Optimizing Compiler. Technical Report CMU-CS-87-187, Carnegie Mellon University, May 1987.

[30] V. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M Mohamed, and J. Telle. OREGAMI: Software Tools for Mapping Parallel Computations to Parallel Architectures. Technical Report CIS-TR-89-18, Dept. of Computer and Information Science, University of Oregon, January 1990.

[31] G. K. Maa. Code Mapping Policies for the Tagged Token Dataflow Architecture. Technical Report MIT/LCS/TR-425, Laboratory for Computer Science, MIT, February 1988.

[32] G. K. Maa. The WAIF Intermediate Graphical Form. ALEWIFE Memo No. 23, Labfor Computer Science, Massachusetts Institute of Technology, July 1991.

[33] Samuel P. Midkiff. Automatic Generation of Synchronizations for High Speed Multiprocessors. Master's thesis, University of Illinois at Urbana-Champaign, November 1986.

[34] D. Nussbaum. Alewife Runtime Scheduler Interface. Personal Communication, 1990.

[35] D. S. Nussbaum. *Run-Time Thread Management for Large-scale Multiprocessors*. PhD thesis, MIT, February 1994.

[36] Dan Nussbaum. Hypertrees Achieve $\Theta(1)$-Competitive Thread Search Time on k-ary n-Dimensional Mesh Networks. ALEWIFE Memo No. 32, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1992.

[37] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the International Conf. on Parallel Processing*, pages 764–771, August 1985.

[38] G. F. Pfister and V. A. Norton. 'Hotspot' Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10), October 1985.

[39] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C.L. Lee, and B. Leung. Parafrase 2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proc. of the Intl. Conf. on Parallel Processing*, 1989.

[40] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[41] G.N.Srinivasa Prasanna. Structure Driven Multiprocessor Compilation of Numeric Problems. Technical Report MIT/LCS/TR-502, Laboratory for Computer Science, MIT, April 1991.

[42] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[43] Hudson B. Ribas. Automatic Generation of Systolic Programs from Nested Loops. Technical Report CMU-CS-90-143, Carnegie Mellon University, June 1990.

[44] Anne Rogers and Keshav Pingali. Process Decomposition through Locality of Reference. In *SIGPLAN '89, Conf. on Programming Language Design and Implementation*, June 1989.

[45] C. A. Ruggiero and J. Sargeant. Hardware and Software Mechanisms for Control of Parallelism. University of Manchester Department of Computer Science Internal Report, April 1985.

[46] Vivek Sarkar and John L. Hennessy. Compile-Time Partitioning and Scheduling of Parallel Programs. In *SIGPLAN '86, Conference on Compiler Construction*, June 1986.

[47] S. Slade. *The T Programming Language: A Dialect of LISP*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[48] G. Steele Jr. *Common LISP: The Language*, pages 87–89. Digital Press, 1984.

[49] P. Tang and P. C. Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proc. of the Intl. Conf. on Parallel Processing*, August 1986.

[50] K. R. Traub. Compilation as Partitioning: A new Approach to Compiling Non-strict Functional Languages. In *Proceedings of the Conf. on Functional Programming Languages and Computer Architecture*, October 1989.

[51] S. Ward. The NuMesh: A Scalable, Modular 3D Interconnect. NuMesh Memo No. 0.5, Lab. for Computer Science, Massachusetts Institute of Technology, January 1991.

[52] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *SIGPLAN Conf. on Programming Language Design and Implementation*, June 1991.