Compiling for Parallel Multithreaded Computation on Symmetric Multiprocessors

by

Andrew Shaw

S.B., Massachusetts Institute of Technology, Cambridge, MA, 1989
S.M., Massachusetts Institute of Technology, Cambridge, MA, 1993

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

January, 1998

Signature of Author_____
Department of Electrical Engineering and Computer Science
January 22, 1998

Certified by_____
Arvind
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Committee on Graduate Students

Compiling for Parallel Multithreaded Computation on Symmetric Multiprocessors

by

Andrew Shaw

Submitted to the Department of Electrical Engineering and Computer Science
on January 22, 1998
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Shared-memory symmetric multiprocessors (SMP's) based on conventional microprocessors are by far the most common parallel architecture today, and will continue to be so for the forseeable future. This thesis describes techniques to compile and schedule Id-S, a dialect of the implicitly parallel language Id, for execution on SMP's.

We show that previous implementations of Id for conventional microprocessors incurred an overhead of at least 40-300% over an efficient sequential implementation of Id-S. We break down this overhead into various presence-tag checking and scheduling overheads. Given this overhead, we conclude that a fine-grained, element-wise synchronizing implementation of Id is not suitable for use on small-scale SMP's.

We then describe a parallelization technique for Id-S that discovers both DAG and loop parallelism. Our parallelization exploits Id-S's single-assignment semantics for data structures. We show that for many programs, our technique can discover ample parallelism, without need for Id's traditional non-strict, fine-grained, producer-consumer semantics. Because our parallelization eliminates the need for presence-tag checking and creates coarser-grained units of work, the parallelized codes only incur a small overhead versus sequential execution.

Finally, we describe code-generation and scheduling techniques which produce efficient parallel executables which we run on a Sun Ultra HPC 5000 SMP. We compare speedups of parallelized Id-S codes using two different schedulers: an SPMD scheduler, and a more general multithreaded scheduler. We describe the advantages and disadvantages of each scheduler, and quantify the limitations in speedups for each scheduler which are due to parallelization, code generation, and scheduling.

Thesis Supervisor: Arvind

Title: Professor of Computer Science and Engineering

Acknowledgements

It is an honor and a privilege to have been a member of the Computation Structures Group (CSG) at the MIT Laboratory for Computer Science. From the "old" generation, I'd like to thank Steve Heller, Ken Traub, and Greg Papadopoulos, who brought me into CSG. From the "next" generation, I'd like to thank Boon Ang, Derek Chiou, and Alex Caro for all of the battles we've fought together all these years. Thanks also to Xiaowei Shen for many late-night talks in the lab, and James Hoe and Mike Ehrlich for teaching me a little about hardware. Thanks to Shail Aditya for many long discussions about dataflow, functional languages, and advice about surviving grad school.

R. Paul Johnson and Andy Boughton have taken good care of us, through thick and thin. I have relied on your help and friendship too many times to count. Thank you.

I would like to thank my advisor, Arvind, who is often exasperatingly uncompromising, but who has taught me to think clearly, to focus on fundamentals and first principles, and to build a strong foundation for my arguments and ideas. I am proud to have been one of your students.

I'm very grateful to my readers, Rishiyur Nikhil and Saman Amarasinghe, who both read my thesis carefully, asked penetrating questions, pointed out holes and ambiguities, and helped me wrestle it into shape.

The work in this thesis was motivated by ideas from two research projects: TAM and Cilk. From the Berkeley TAM project, David Culler, Klaus Schauser, Seth Goldstein, and Thorsten von Eiken bridged the gap between dataflow and conventional architectures, and laid the groundwork for the partitioning and code generation approach used in this thesis. From Cilk and its predecessor, PCM, Yuli Zhou, Michael Halbherr and Chris Joerg showed me the importance of scheduling; my early code to implement Cilk on SMP's was the basis of the compiler used in this thesis. Special thanks to Chris Joerg, whose questions eventually led me to the work in this thesis, and who directly or indirectly helped me solve some of the hard problems, and who encouraged me in my work when I needed it.

I'd like to thank my family – my sister Nini, and my parents Julie Yueh-ho and Chong-Kuang Shaw who have supported me throughout my years at MIT. Everything I have, I owe to you.

Finally, thanks to my wife Atsuko, who brought true happiness to my life – your optimism and patience have pulled me through, and I am so lucky to have shared these years with you. I am also happy to have shared my final year at MIT with my daughter Marie, whose energy and thirst for knowledge inspire me anew.

For my parents, Julie Yueh-ho and Chong-Kuang Shaw

Contents

List of Figures

Chapter 1

Introduction

After decades of research and years of turmoil in industry, parallel computers have finally entered the mainstream in the form of relatively cheap, small-scale, shared-memory Symmetric Multiprocessors (SMP's). Every major computer manufacturer is now making and selling SMP's, but despite this proliferation of hardware, users cannot easily exploit this parallelism except for a subset of well-structured "scientific" applications and relatively coarse-grained throughput applications such as database servers and Web servers.

In this thesis, we describe an approach to exploiting parallelism on SMP's which has several advantages over previous approaches to parallel computing:

- It runs on stock hardware using a stock operating system.
- The programmer writes in a sequential style, leaving parallelization, work partitioning, synchronization, communication and scheduling to the compiler and run-time system.
- It handles both *structured* and *unstructured* parallelism, which requires solving problems at two different levels:
    1. Discovering both structured and unstructured parallelism in sequential programs.
    2. Scheduling possibly unstructured work on the processors with good load balance and low overhead.

This work bridges the gap between two very different approaches to parallel computing: Id90 on Monsoon and Fortran on SMP's. The Id90/Monsoon approach [62] [43] exposes implicit parallelism at the language level while requiring extensive hardware support for fine-grained synchronization and communication. Every level of parallelism is exploited, including instruction, procedure, loop, and

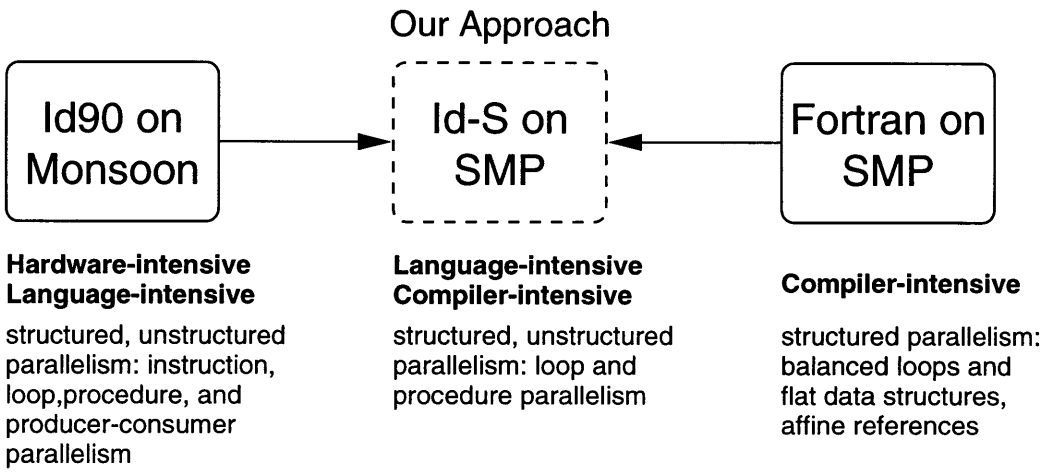

Figure 1.1: Our approach straddles two previous approaches to parallel computing: Id on Monsoon and Fortran on SMP's, attempting to take advantage of the strengths of each.

producer-consumer parallelism. The Id90/Monsoon approach can support both structured and unstructured parallelism at a significant cost in hardware and re-coding of applications. However, to the first degree, the programmer does not need to concern himself with explicit parallelization, work partitioning, synchronization, communication or scheduling.

The Fortran/SMP approach, exemplified by the SUIF [6] compiler, can take advantage of stock hardware and OS's for parallelizing many existing Fortran codes without programmer intervention. This approach exploits coarse-grained loop parallelism, where data structure accesses within the loop are affine functions over the iteration space and work is relatively well-balanced across the loop iterations. We call such parallelism *structured parallelism.* The Fortran/SMP approach emphasizes compiler analysis and transformations to parallelize codes and discover coarse-grained parallelism, while taking a semi-static approach to work scheduling and distribution.

Our approach, as shown in Figure 1.1, attempts to exploit structured and unstructured parallelism on stock hardware. We compile Id-S, a variant of the Id90 language, and perform interprocedural data dependence analysis to determine when we can exploit procedure parallelism and loop parallelism. Our analysis is greatly aided by the Id-S semantics, which require that each data structure element be written to at most once, thereby eliminating anti- and output- dependences at the source level. We then generate code which is scheduled dynamically on the SMP processors. A combination of compiler and run-time system support allows us to obtain a coarse-grain of parallelism at run-time. We have been able to compile almost all Id90 programs with minor changes, showing speedups on most codes.

1.1 Parallelism and efficiency

Our work builds on previous efforts to compile Id90 for conventional microprocessors, including TAM [25] [32] [72], P-RISC [56], and pHluid [18]. Our approach differs from those previous efforts in that we have changed the Id language semantics and opted to forgo fine-grained producer-consumer synchronization in order to improve the *efficiency* of the language implementation. *In effect, we are trading off parallelism for efficiency* – this tradeoff is necessary because we do not have hardware support for producer-consumer synchronization and we are targeting small machines.

1.1.1 Overheads in previous Id implementations

The only published parallel speedup data for an Id implementation targeted for "conventional" (i.e. non-dataflow) architectures are for TAM [72] running on the CM-5. These data include a linear speedup of 16 on 64 processors for the Simple benchmark. The linear speedup indicates that there was enough parallelism exposed to execute on 64 processors.

A further examination of the data from [72] reveals that the speedup numbers are relative to the same parallel executable running on 1 processor of the CM-5. However, this parallel executable is more than twice as slow as the same program compiled under the same TAM compiler for single-processor execution (as opposed to a 1-processor execution of a parallel-ready code), because of overheads introduced for real parallel execution – these overheads include support for remote memory references and parallel work distribution. In Chapter 3, we show that even the single-processor TAM implementation is twice as slow as it might be because of its support for fine-grained synchronization and scheduling.

In all, the linear speedup of 16 on 64 processors is actually a linear speedup of less than 4 when compared to an efficient sequential implementation. **In effect, the program is slowed down by a factor of 16, and then run on 64 processors to get a factor of 4 overall speedup.**

This is an extreme case, but the parallelism/efficiency tradeoff is a serious issue when considering implementation alternatives. Running on a relatively small SMP, we certainly cannot afford a factor of 16 overhead, even if it promises us abundant parallelism, but as the simple analysis in the next section shows, we probably cannot afford even a more realistic overhead of 2.5, except when there is relatively little parallelism to be exposed.

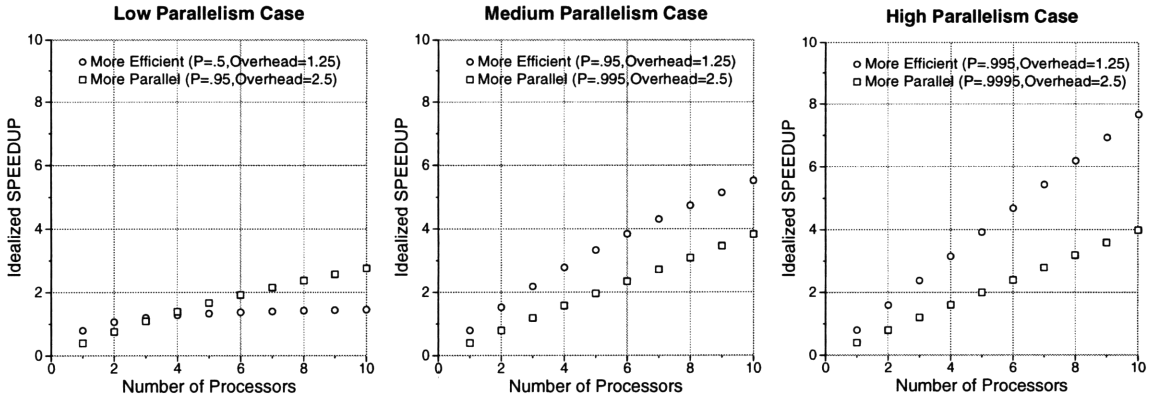

Figure 1.2: Assuming P is the fraction of a code which can be parallelized, and Overhead is the ratio of run-time versus sequential execution, these graphs show the speedups under two scenarios assuming the parallel part of the code speeds up perfectly. In general, efficiency is more important than parallelism for small numbers of processors, except when parallelism is low.

1.1.2 Analysis of parallelism/efficiency tradeoff

With a small number of processors and high synchronization overheads, additional parallelism may not make up for a loss in efficiency. Figure 1.2 shows speedups under a very simple analytical model. Let us assume that a code runs in time T in an efficient sequential implementation. In order to enable the code to run in parallel, assume there is an overhead factor OV such that a parallelized code running on one processor takes time $T \times OV$. Assume that the code consists of a perfectly parallelizable fraction P and a sequential fraction $(1 - P)$. The total run-time of a code running on n processors is then $T \times OV \times ((1 - P) + P/n)$, and the speedup on n processors is:

$$\text{Speedup} = \frac{1}{OV \times ((1 - P) + P/n)}$$

We then plug in values of OV and P to compare the speedups of a more efficient ($OV = 1.25, P = .5, .95, .995$) and a more parallel ($OV = 2.5, P = .95, .995, .9995$) version, where the more parallel version has a sequential section which is 10 times smaller than the more efficient version (i.e. has 10 times more parallelism). The factor of 2.5 overhead is a very conservative estimate of the fine-grained overhead introduced by a TAM-style Id implementation – this is analyzed in greater depth in Chapter 3.

The graphs show that for a small number of processors (≤ 10), the additional parallelism of the high overhead scenario does not make up for the additional overhead, except in the case of low parallelism, and then, only for more than 4 processors.

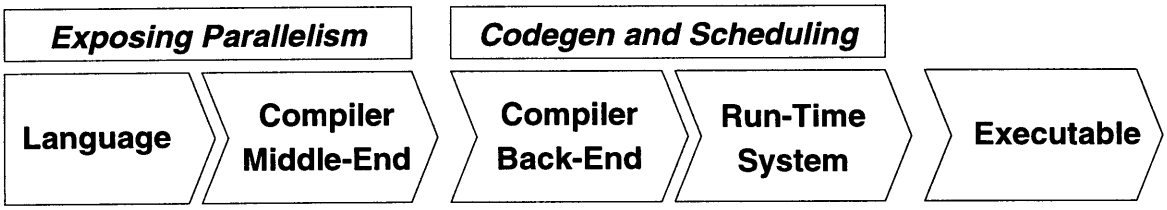

Figure 1.3: In exploiting structured and unstructured parallelism, there are two main, somewhat decoupled components: (1) exposing parallelism and (2) code generation and scheduling. We attack the first with a combination of language support and compiler analysis, and we handle the second with methods for efficient code generation and scheduling.

Of course, the tradeoffs depend significantly on the actual overhead parameters, and the difference in exploitable parallelism. In some cases the difference in overhead may be greater or smaller, and the difference in parallelism may be greater or smaller, but Figure 1.2 gives an intuitive feeling for the tradeoffs we are considering.

The overhead parameters we chose fit closely with the analysis in Chapter 3 comparing the more parallel Id90 with the more efficient Id-S. The parallelism regime in which we operate (i.e. low, medium or high) depends upon the effectiveness of our parallelization analysis and the characteristics of the codes we are considering. If the parallelization yields enough exploitable parallelism that we are operating in the medium or high range, then any additional parallelism we could have exploited with a more fine-grained approach would not be not necessary.

1.2 Structured versus unstructured parallelism

In our system, we would like to exploit structured as well as unstructured parallelism. *Structured parallelism* is loop parallelism where data structure accesses are affine functions of the iteration space, and where work is relatively well balanced across the loop iterations. *Unstructured parallelism* is all other parallelism, including procedure parallelism, and whole loop parallelism (e.g. two separate loops executing in parallel) – in unstructured parallelism, work may be unbalanced and the granularity of work may be unknown at compile time, or even by the programmer. We are not concerned with instruction-level parallelism, and leave that to the individual superscalar microprocessors in the SMP.

Exploiting parallelism requires solving two separate problems: (1) exposing parallelism and (2) code generation and scheduling. As shown in Figure 1.3, we address the first problem with the language

and compiler middle-end and the second problem with the compiler back-end and run-time system. The two problems are decoupled to the extent that different strategies for attacking them can be mixed and matched.

1.2.1 Exposing parallelism

Traditional Fortran loop parallelizing compilers expose parallelism using a combination of ad hoc nested loop data dependence analyses [48] [86] and interprocedural "region-based" and linear-inequality driven analyses and transformations [13] [65] [6] [38]. These approaches can work well for very structured Fortran programs where most of the work lies in nested loops, and where accesses to large, flat, static data structures in the loops are affine functions of the iteration space.

However, these Fortran techniques do not work as well for programs which have less structured accesses to data structures – for example, where most work does not lie in loops, or where accesses are not affine, or where nested data structures such as lists, trees and other non-array structures are used. Most of the research in unstructured parallelization has focused on pointer analysis in the context of C programs [84], but this work has had only limited success, largely due to the difficulty of handling C's very low-level semantics.

We take the approach of requiring some language support by having the user to program in a style which gives the compiler more information about data dependences. Because Id-S is a single-assignment language (i.e. each data structure element can only be written once), the compiler does not need to check for anti- or output- dependences, which can obscure some parallelism. Furthermore, the single-assignment semantics forces the user to use dynamic heap allocation, which allows the compiler to further distinguish between structure accesses, exposing more parallelism.

The net result is that we can use a very simple interprocedural data dependence analysis to find both loop and procedure parallelism. Fortran techniques would still be useful for a few loops that our approach cannot analyze, but our simple technique finds most of the loops Fortran techniques would find, and can also handle some loops with non-affine accesses or accesses to nested data structures, which Fortran techniques typically cannot handle. In general, our simple analysis is good enough that we have not bothered to implement the Fortran techniques.

The middle-end of the compiler is also responsible for giving a rough determination of when parallelism is worth exposing, and passing along this information to the back-end code generation phases.

We describe some simple, effective techniques to avoid exposing too much fine-grained parallelism. The work-estimation phase is not as important as it would be in other systems because we have taken great care in the code generation and run-time system to minimize synchronization and parallel bookkeeping overheads, and to increase work granularity at run-time.

1.2.2 Parallel execution model: SPMD vs. multithreading

Although traditional Fortran parallelizing loop compilers originally targeted vector supercomputers, more recent Fortran compilers [6] have targeted non-vector SMP's. These Fortran compilers generate mostly sequential code with parallelized outer loop iterations scheduled evenly across the processors. In this thesis, we call this execution model the *Single Program Multiple Data* (SPMD) execution model. SPMD works well for structured programs because work resides mostly in loops, and is well-balanced across loop iterations. Additionally, compiler-driven loop and data transformations can improve the locality of data references under SPMD execution [85].

The SPMD model has several drawbacks – it cannot effectively handle unbalanced loops, procedural parallelism, or nested parallel loops because loop iterations are distributed evenly across the processors, and only one level of loop parallelism is exploited. In contrast, the *multithreaded* execution model allows different processors to execute different procedure or loop activations, dynamically spawning, scheduling and synchronizing work. The multithreaded model is more general than the SPMD model, but may incur more overhead due to dynamic scheduling and synchronization, and poorer memory locality.

We use the terms "SPMD" and "multithreaded" very loosely – both terms can be used to describe a wide variety of execution models. Furthermore, we show that SPMD and multithreaded execution are two points on a continuum of execution models. We compile Id-S for *both* SPMD and multithreaded parallel execution, and compare their performance. For some highly structured applications, a more static, SPMD approach yields better performance, whereas for other, less structured, less loop-dominant applications, a more dynamic, multithreaded approach yields better performance.

In this thesis, we show how to structure multithreaded execution so that it can be performed efficiently, while being competitive with SPMD execution for many codes. Implementing multithreading efficiently requires efficient code generation techniques and adaptive, lightweight scheduling mechanisms.

1.3 Thesis contributions

In this thesis, we focus on implementing a parallel software system to exploit symmetric multiprocessors. Almost all parallel computers today are SMP's, and SMP's are becoming cheaper and more common on the low-end, and larger and more powerful on the high-end. SMP's provide shared memory support directly in hardware, easing the burden on the programmer and compiler. However, SMP's do not provide any hardware support for fine-grained synchronization or scheduling. Synchronization is usually performed through relatively heavyweight locks to shared memory locations, and scheduling must be done completely in software. Consequently, the granularity of parallelism must be coarse enough to overcome the overheads incurred for synchronization and scheduling.

The organization of the software is the key to exploiting parallelism effectively on SMP's. We describe an approach which includes language, compiler and run-time system working in concert to expose and exploit parallelism efficiently.

1.3.1 New language semantics

We introduce a variant of the Id90 language, which we call Id-S. Id-S has a sequential evaluation order like C or Fortran and single-assignment semantics like Id90. Id-S is not functional: data structures can be allocated, and reads and writes can be performed on the structures as in imperative languages, except that each structure element can only be written once. Id-S's sequential semantics do not preclude an implementation which exploits producer-consumer parallelism, and all legal Id-S programs are also legal Id90 programs which will execute, terminate and provide the same results if implemented on previous Id90 systems, including Monsoon.

We changed the Id semantics to provide a sequential evaluation order because the fine-grained, implicitly parallel dataflow semantics of Id90 are difficult to implement efficiently on non-dataflow hardware, and as we discussed in Section 1.1, the parallelism exposed by Id90 would not make up for the overhead incurred. The sequential semantics also give more information to the compiler about dependences caused by reads and writes.

We chose to compile Id-S rather than C or Fortran because the single-assignment semantics allowed us to both simplify parallelization analysis for structured loops, and also handle unstructured codes which would be difficult or impossible to parallelize under imperative semantics.

Performance comparison (seconds on SparcStation 10)				
	Input Size	strict TAM Id90	Id-S	C/FORTRAN
Gamteb	40,000	169.5	88.3	
MMT	500	61.0	22.2	19.8
Paraffins	19	2.2	1.6	
Quicksort	10,000	1.5	0.8	0.6
Simple	1 1 100	2.7	1.1	1.0
Speech	10240 30	0.6	0.16	0.12

Figure 1.4: Id-S is the fastest implementation of Id to date, and comparable in performance to C or Fortran.

Surprisingly, we found that almost all existing Id90 programs could be converted to Id-S programs, simply by reordering bindings within blocks. Almost all of the Id90 programs we found which we could not convert to Id-S were toy programs designed explicitly to exploit Id90's non-strict, implicitly parallel dataflow semantics.

1.3.2 New compiler parallelization and code-generation

The core of our research is built around a new Id compiler can work with varying language semantics, exploiting different levels of parallelization, and using different code-generation strategies.

Id-S's sequential evaluation order allows us to generate more efficient code for conventional microprocessors than for Id90, while the single-assignment semantics allows us to perform more parallelization than for Fortran or C. Our sequential implementation of Id-S is the fastest implementation of Id to date, with performance competitive with sequential Fortran or C. Figure 1.4 shows our performance relative to strict TAM Id90 and C or Fortran – the TAM performance numbers are from [72]. Identical Id programs were compiled for TAM and Id-S, and similar (or more efficient) algorithms were used for the C and Fortran programs. No C or Fortran programs using the same algorithms were available for Gamteb or paraffins. More details about the input sizes and structure of the programs are given in Sections 3.1 and 4.5.

1.3.3 Parallelism and efficiency tradeoff

Using our sequential Id-S implementation as a baseline, we perform a detailed evaluation of the overheads in the state-of-the-art implementations of Id90 for conventional microprocessors. Our results

show where improvements could be made in implementations with element-wise data structure synchronization, but also lead us to conclude that such an implementation would have overheads which are too high to overcome any benefits from additional parallelism.

1.3.4 Parallelizing Id-S

We give an algorithm for parallelizing Id-S which is based on existing interprocedural data dependence analyses. These analyses yield more results for Id-S than they do for imperative languages because of Id-S's single-assignment semantics and frequent heap allocations.

We instrument our compiled code to provide us with ideal parallelism numbers to give us an idea of how effective our parallelization is, and where it is weak. The ideal parallelism numbers allow us to separate the effectiveness of the parallelization from the details of the code generation and scheduling.

By disabling or modifying certain parallelization analyses, we also show the separate effects of procedure and loop parallelism, and how much parallelism we could discover if we do not assume single-assignment semantics.

1.3.5 Parallel code generation and run-time system

We describe our code generation schemas which are tightly linked with the run-time system. We show some control-flow style optimizations performed to make hooks into the run-time system cheaper.

We describe the structure of the run-time system, which increases work granularity at run-time using lazy work-stealing techniques. Our run-time system also handles loop and procedure parallelism uniformly, thereby simplifying our run-time system entry-points. Some simple compile-time work-estimation is performed to eliminate fine-grained parallelism which cannot be profitably exploited.

1.3.6 Parallel performance

Finally, we give performance numbers on an 8-processor Sun Microsystems Ultra HPC 5000 SMP, and analyze sources of lost performance on benchmarks which do not achieve perfect speedup. The potential sources of overhead we examine include:

- Parallel overheads due to code generation – although we attempt to keep overheads due to code generation down as much as possible, we still have some overhead compared to sequential execution.

- Insufficient parallelism – some codes do not parallelize well, because of limitations of the compiler. Some codes simply do not have much parallelism to exploit.
- Run-time system overheads – run-time system overheads include time to distribute work and synchronize on returning work. Run-time system overheads typically show up for codes which have a small work granularity and/or low parallelism. SPMD scheduling typically has less overhead than multithreaded scheduling, although for many codes, the difference is insignificant.
- Load balance – for some codes, a simple SPMD scheduling leads to load imbalance, whereas multithreading scheduling can handle more
- Memory system limitations – certain codes are fundamentally limited by the main memory bandwidth of the SMP. Some codes will not show good speedup unless heap memory is reclaimed because of poor cache behavior and memory bandwidth limitations.

We show some speedups for almost every code we ran, validating our approach.[1] Performance could be improved on the compiler side by improving code generation and parallelization. On the run-time system side, it could be improved by better (more robust) scheduling policies and a tighter implementation.

1.3.7 Issues not addressed in this thesis

Many issues remain unaddressed in this thesis because of a lack of time and resources. In a production Id system, garbage collection or some other form of automatic memory management [44] would be necessary to avoid running out of heap memory. Our system currently places this burden on the programmer, which is both tedious and error-prone. The effect of garbage collection on parallelism and performance might be somewhat different from our results.

Fortran techniques would be useful both for parallelizing loops which cannot be handled by our system, and for restructuring to improve usage of the memory hierarchy. Currently, we perform no locality optimizations in the compiler or run-time system. Many locality transformations are orthogonal to scheduling, and could provide improvements for multithreaded execution as well as SPMD execution.

Our compiler depends upon interprocedural analysis for parallelization. Currently, this is implemented by compiling the entire program at once, but interprocedural analysis can also be performed with separate compilation by maintaining some file information in a database, or by pushing some of the compilation to the linking phase.

[1] The Id-S benchmarks that we used to evaluate the language, compiler and run-time system can be retrieved at `http://www.csg.lcs.mit.edu/~shaw/shaw-phd-id.tar.gz`.

Year	System	Researchers and References
1987	Id / TTDA	Arvind, Nikhil, Iannucci, Traub, etc. [9] [10]
1989	Id / Monsoon	Papadopoulos and Culler [62], Hicks, Chiou, Ang, Arvind [43]
1988-1995	Id partitioning	Traub [81] [82], Schauser [72], Coorg [22]
1991	Id / TAM	Culler, Goldstein, Schauser, von Eiken [25] [32]
1992	Id / P-RISC	Nikhil [56]
1992	EM-C / EM-4	Sato, Sakai, etc. [66] [68]
1993	Id / StarT-88110MP	Carnevali, Shaw [55]
1994	Cilk / CM-5	Zhou, Halbherr, Joerg, etc. [37]
1994	Id / pHluid	Chiou, Nikhil [18]
1997	Id97 / SMP's	*This Work*

Figure 1.5: The Id97 system has dataflow roots but has moved towards implementation on more conventional architectures.

We only show performance results for relatively small ($\leq$ 8 processor) systems – for larger systems, locality, synchronization and communication become more of a concern, and the techniques we use may have to be modified or augmented.

1.4 Related work

The direct roots to the approach in this thesis have a long history, dating back to dataflow architectures and implicitly parallel functional programming languages. The general direction of this work has been from functional languages and heavy architectural support towards more conventional languages running on standard commercial hardware.

1.4.1 Direct ancestors to this research

Figure 1.5 shows some of the systems which are direct ancestors to the Id97 system. The original implementation of Id was on the Tagged-Token Dataflow Architecture (TTDA) [9] [10] [80], which was a simulated dataflow architecture. Each TTDA instruction synchronized on the arrival of its input values, and direct support for element-wise data structure synchronization existed in the form of I-structure boards.

Monsoon [62] was the direct successor of TTDA, and was realized in hardware. Monsoon had a explicit token matching store, and introduced the notion of activation frames as a synchronization namespace and temporary storage. Monsoon also had direct support for I-structures.

Ken Traub [81] developed the early theory of compiling the non-strict Id language into sequential threads to be executed on more conventional von Neumann architectures. The TAM compiler [25] [32] actually implemented some simple partitioning algorithms, and compiled Id for conventional microprocessors. Over the course of a few years, partitioning algorithms for non-strict Id improved with contributions from Traub [82], Klaus Schauser [72] and Satyan Coorg [22]

The P-RISC effort [56] was a combination of partitioning compiling technology and a few machine instructions to create a "Parallel RISC" instruction set. This eventually led to the StarT-88110MP machine, which was to be based on a modified Motorola 88110 processor. Derek Chiou and Rishiyur Nikhil continued on a P-RISC implementation of Id with the pHluid compiler [18], which was targeted towards sequential workstations and clusters of workstations.

I spent a year in Japan working on the EM-4 [66] project at the Electrotechnical Laboratory. EM-4 was a hybrid dataflow / von Neumann architecture, programmed using EM-C [68], a dialect of C with support for explicit threading and synchronization. This work bridged some of the gap for me between dataflow and von Neumann machines.

In 1994, Yuli Zhou developed a C pre-processor which he used to write explicitly parallel C programs for the CM-5 in a system which eventually became Cilk [37]. He and Michael Halbherr rediscovered the fact that much of the overhead of multithreading was actually in run-time scheduling, and that lazy task stealing could significantly lower communications and synchronization overhead, even for the CM-5 which had very expensive communication and no explicit support for fine-grained synchronization. Lazy work stealing was originally exploited in the Multilisp system [40] and a similar idea was exploited in lazy task creation [52] on Alewife [2].

The work in this thesis grew out of my frustrations with explicitly parallel languages, and the success of lazy work stealing in Cilk to reduce multithreading overheads. Although EM-C and Cilk were efficient, they were very difficult to use because parallelization and synchronization were left to the programmer. Id gave a much nicer programming model to the user, and had shown ample parallelism for a variety of applications, but seemed to incur too much overhead on conventional microprocessors. I had written the first SMP implementation of Cilk, and I knew work stealing could eliminate some of the overheads in scheduling Id programs. I began work to find the the overheads of implementing Id on SMP's which resulted in this thesis.

1.4.2 Related hardware

The underlying multithreaded execution model we implement in software has its roots in multithreaded hardware, including dataflow machines such as Monsoon [62], EM-4 [66], Sigma-1 [45] and the Manchester Machine [36]. Very similar in spirit to the dataflow machines are the HEP [76] and Tera [4]. Alewife [2] and the J-Machine [58] also provide support for multithreaded execution, although these machines are closer to conventional sequential machines.

1.4.3 Related languages

Sisal is a strict functional language [17] which has been compiled for parallel execution on a number of platforms, including sequential workstations, vector supercomputers, and SMP's [67]. Id97 differs from Sisal in that Id97 is not functional – empty objects can be allocated and reads and writes can occur to those objects. Cilk [16], EM-C [68], and Cid [57] are explicitly parallel dialects of C which have a multithreaded implementation.

1.4.4 Related compilers

The data dependence analysis for the Id97 compiler is inspired by Fortran loop parallelization techniques [48] [86] [13] [65]. Recent work by the SUIF project has extended this loop parallelization to including loops containing function calls, and loop restructuring for memory locality [6] [38]. The interprocedural analysis necessary for parallelizing Id97 is based on Banning's interprocedural alias analysis [15], and Cooper and Kennedy's advances [19] [20] on Banning's original algorithms.

1.4.5 Related multithreading run-time systems

The thread representation and scheduling work is a direct descendant of TAM [25], P-RISC [56] [18], and Cilk [37] [16] [46]. It is also closely related to Lazy Task Creation [52] and Lazy Threads [35] [34]. Lightweight thread representations and scheduling techniques have also been implemented in C-style thread libraries such as Filaments [29] [30] and other threads packages [47] [53] [21].

1.5 Outline of thesis

Chapter 2 provides an overview of our system, including the language, the compiler and the run-time system. Chapter 3 is a study of the overhead of element-wise data synchronization on conventional microprocessors. Readers may choose to skip Chapter 3 without loss of continuity, as the issues it addresses are orthogonal to the rest of the thesis. Chapter 4 describes the simple parallelization algorithm we use for Id-S, and its effectiveness. Chapter 5 explains how we generate multithreaded code, and a few optimizations we can perform which are unique to a multithreaded target. Chapter 6 describes the light-weight run-time system. We conclude with Chapter 7 summarizing the work, and providing some direction for future research.

Chapter 2

System Overview

In this chapter, we give an overview of our system and our study, because it involves many components which interact very closely and are difficult to understand individually without some understanding of the whole. Some of these components include the execution models, the compilers and the different languages they compile, and the basic structure of the run-time system. We conclude with some notes about implementation strategies we have have taken which have helped us to develop our system quickly.

2.1 Multithreaded execution model

Figure 2.1 shows a high-level view of the multithreaded execution model we are targeting – procedure activations and parallel loop activations are associated with *frames*, and those procedure and loop activations may refer to *global heap objects*. Procedure activations, such as **f**, may fork off other procedure activations, such as **g** and **h**. Procedures may also fork off loops which may execute in parallel, while forking off other parallel loops or procedure activations. All of the procedure and loop activations refer to objects residing in a global shared heap.

At this level of abstraction, the execution model is the same for Id90 executing on Monsoon, the TAM implementation of Id90 executing on conventional microprocessors, or for our parallelized Id-S implementation for SMP's. The parallelized Id-S implementation differs from the Monsoon and TAM implementations in that Id-S accesses to the heap objects are not synchronizing. In Monsoon and TAM, **loop1** may be writing to elements of the array which are being read by **loop2**, and both loops can execute in parallel with the knowledge that if **loop2** tries to reference an element before it is written, the **loop2** access suspends until that element is written by **loop1**. In compiling Id-S, we want to avoid the

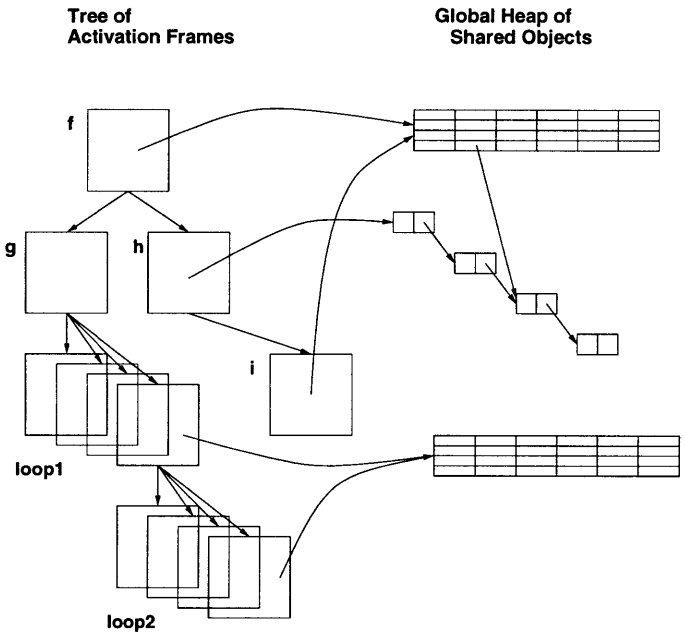

Figure 2.1: In the multithreaded execution model, procedure activations may execute in parallel, as may loop activations. Activation frames are analogous to stack frames, and all the procedure and loop activations may reference global heap objects.

overhead of run-time heap synchronizations, so the compiler analyzes the code and takes care to only fork off parallel work which will not require run-time heap synchronizations. This limits the amount of procedure and loop parallelism we can exploit relative to Monsoon and TAM, but it also reduces our heap synchronization overhead.

In Monsoon and TAM, *every* procedure and *every* loop can potentially execute in parallel, although more typically, the user specifies which loops he wishes to execute in parallel in order to control overhead from parallelism. In parallelized Id-S, we only execute procedures and loops in parallel which we can determine definitely do not have a dependence through a data structure element. Relating this to Figure 2.1, the activation frame tree for the same program implemented on Monsoon and TAM is likely to be much bushier and wide compared to the same activation frame tree for Id-S.

In the Monsoon, TAM and Id-S systems, activation frames are associated with processors, and work is distributed by distributing activation frames. Figure 2.1 shows the logical parallelism of a program at a moment during the execution – this logical parallelism is mapped to a physical machine by the run-time system. In the Monsoon and TAM implementations, frames are assigned to random processors to spread work randomly so that very likely, all the function activations and loop activations could be

on different processors. In Id-S, we use a work-stealing strategy where work is scheduled by default on the local processor unless another processor steals it – work-stealing gives much better locality and allows us to perform forking, returning, and synchronizations between parent and child frames cheaply without locking or message-passing. In practice, work-stealing also tends to provide very good load balance [16].

The multithreaded execution model is much more general than the SPMD-style execution used in the data-parallel or parallelized Fortran loop models. Multithreaded execution can take advantage of procedure or loop parallelism with an unstructured logical shape, but may require some additional overhead. We show a variety of methods in the compiler and run-time system which minimize this overhead, and also produce good speedup.

2.2 Id language and compilers

We use three main Id compilers in this study – their basic structure is shown in Figure 2.2. The three compilers share the front-end of the original Id compiler [80], which was used for the TTDA [10] and Monsoon [62] architecture. This front-end parses and desugars Id into a hierarchical dataflow graph intermediate representation, and many standard optimizations such as constant-folding, loop constant hoisting, common subexpression elimination, inlining (as directed by the programmer), strength reduction, algebraic simplification, etc. The front-end dumps a textual representation of the dataflow graph into a file, and each of the three back-ends begins work with identical program representations.

The three compilers have the following characteristics:

1. The fine-grained compiler is the equivalent of the TAM or pHluid Id compilers. It compiles Id assuming strict function calls, conditionals and loops, but with synchronizing heap accesses.
2. The sequential compiler is the equivalent of a sequential C or Fortran compiler. It compiles Id assuming sequential semantics in exactly the way a C or Fortran compiler works.
3. The parallelizing compiler performs procedure and loop parallelization assuming Id with sequential semantics.

To the extent possible, all three compilers generate C with identical schemas. The generated C uses some `gcc` extensions, and is compiled with the `gcc` compiler. The object files are then linked along with the respective run-time system to create a UNIX executable. The fine-grained and sequential

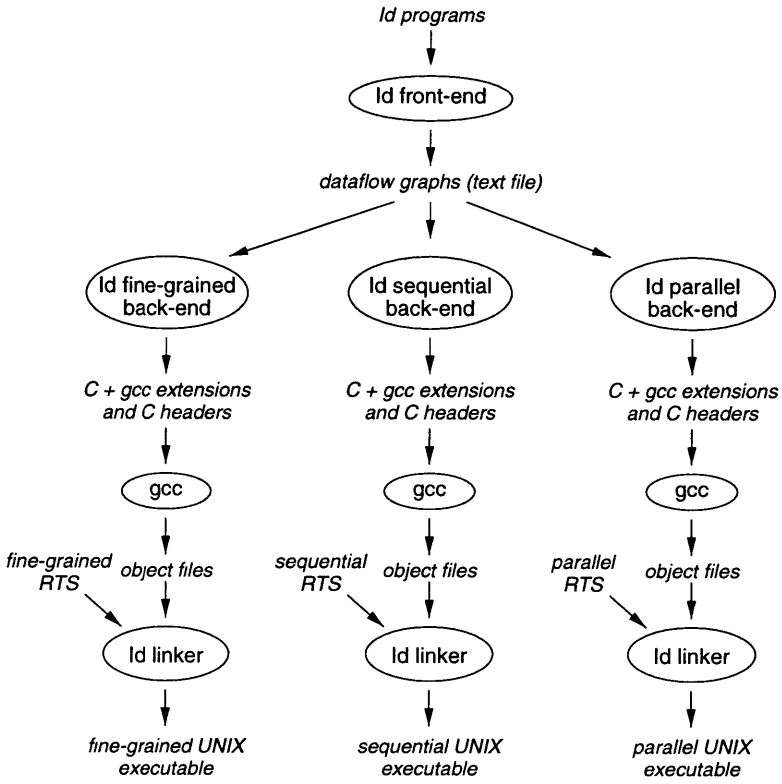

Figure 2.2: The three Id compilers: fine-grained, sequential and parallel.

executables run on one processor, and parallel executables can run on a variable number of processors which can be determined at run-time in an environment variable.

All three Id compilers generate very efficient code, given the assumptions they are working under. The fine-grained compiler generates code comparable in performance to that of the TAM compiler [72]. The sequential compiler generates code competitive with C and Fortran (as we showed in Figure 1.4), and significantly faster than the fine-grained compiler because it has no synchronization or multithreading overheads. The parallel compiler generates code with some small overhead relative to the sequential compiler, with varying speedup depending upon the characteristics of the codes.

The three Id compilers compile and run the largest set of Id programs ever assembled. Within each class (fine-grained, sequential, parallel), each compiler generates more efficient code (measured in machine instructions executed per program) than any previous implementation. The programs also run on faster hardware than any previous implementation, making these by some measures, the fastest Id implementations ever.

2.2.1 Performance impact of Id language features

In evaluating our results, it is important to understand what Id's performance is like relative to conventional imperative languages. If we assume a sequential semantics for Id, the primary performance-impacting language feature is Id's single-assignment restriction, which requires us to re-structure codes and algorithms to be single-assignment.

Aside from single-assignment, the two main languages features which impact Id's performance are the following:

- Polymorphic data structures – all structures and arrays have slots which are 64-bits wide, even for 32-bit elements such as integers, where we waste half of the memory. This is to handle Id's polymorphic type system, where polymorphic functions may access arrays of arbitrary type.
- Dynamic array extents – unlike C, Id arrays have extents which are defined when they are created. Furthermore, indices into the array may have an arbitrary base defined at creation-time. For example, a two-dimensional array may have an (i, j) base index pair which is $(-1, 3)$, rather than $(0, 0)$ which is the case in C. Address computation is therefore more complicated and expensive in general than for C.

Neither of these overheads is fundamental, and both are orthogonal to Id's suitability for parallel execution. Additional compiler analysis regarding usage of polymorphism could save data structure space – this analysis is used in other polymorphic languages implementations, including ML. The overhead from dynamic array extents can be largely optimized out in loops with loop constant hoisting.

Id contains many other interesting language features, most of which do not impact on performance, including pattern matching for function calls, case statements and bindings, array and list comprehensions, algebraic types, and a simple module system. Id also has support for curried and higher-order function applications, but we do not address applications which make heavy use of curried or higher-order function applications because their control flow is difficult to analyze – attempts to analyze the control flow of higher-order include work by Shivers [75] and Harrison [41].

2.2.2 Sequential Id compiler as baseline

We use the sequential Id implementation as a baseline to compare our other two Id implementations. The sequential implementation compiles Id as if it were a C or Fortran program with an Id syntax – it uses standard C calling conventions, and obtains similar performance to C or Fortran.

The sequential implementation of Id is made possible by creating a new dialect of Id, Id-S, which has a sequential evaluation order. Almost every Id program we found could run correctly with some minor modifications. These modifications were typically re-ordering of bindings within a block, or reordering of procedure calls. Only two non-trivial Id90 programs we found could not be modified in this way, and both were LUD linear system solvers which required parallel, fine-grained execution to terminate correctly.

The sequential Id compilation is a straightforward syntax-directed translation to equivalent C – Id function calls become C function calls, Id conditionals become C conditionals, and Id loops become C loops.

2.2.3 Compiler generated threaded code

The two compilers which generate multithreaded code are slightly more unusual, and we describe them in more detail in later chapters. Our scheme is quite similar to the TAM [25] and P-RISC [56] schemes – Goldstein [32] provides a particularly detailed description of the TAM implementation. Here, we provide an overview of the threaded code compilation process, and its relationship to the run-time system.

The Id compiler initially parses the program and creates a dataflow graph intermediate representation for each procedure in the program. The dataflow graph representation is similar to other compiler intermediate representations. The compiler then performs some optimizations and analysis, and eventually *partitions* the graph such that every graph node is a member of a partition. The partitioning algorithm is dependent upon the language semantics – for fine-grained execution, the partitions are likely to be much smaller, and for parallelized Id-S, the partitions are likely to be larger.

Given a partitioned graph, the compiler computes what state flows between partitions – this state is carried by the arcs which connect different partitions. Each partition must save the state that its children require into the activation frame, and child partitions are not enabled until all of their parent partitions have executed. In effect, after partitioning, we have a coarser-grained dataflow graph where multiple values may flow between the nodes, and multiple instructions are executed within each node.

For each partition, the compiler generates code to read incoming state from the activation frame. The partition executes the instructions within the partition, and then stores away state that its children need. Finally, the partition must perform some join-counter synchronizations to determine whether any

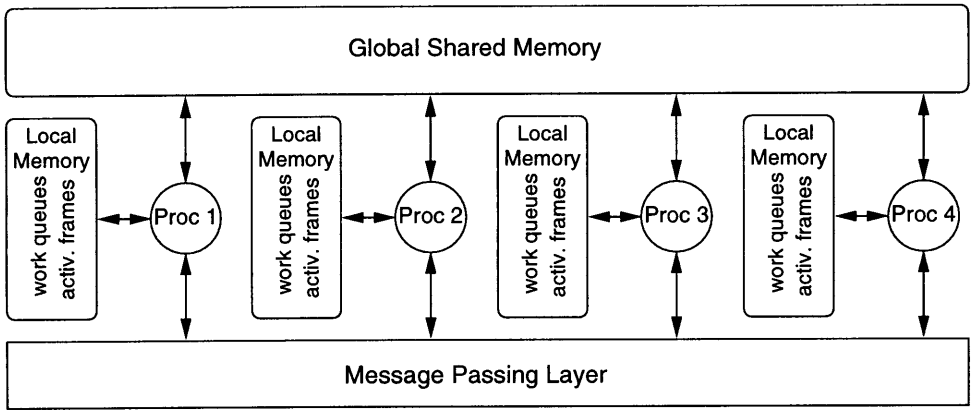

Figure 2.3: Run-time system structure.

of its children are ready to execute. Any ready work is then enqueued onto a work queue, and finally, the partition dequeues a piece of work from the work queue and executes it. Procedure calls and parallel loops are also enqueued on the work queue.

At the end of each partition, either a direct or conditional jump is made to another partition, or else a piece of work is dequeued from a work queue and executed. In general, we would prefer to execute a direct jump when we can, a conditional jump if possible, and only when we cannot resolve anything at compile time do we perform the general case of dequeuing work from the work queue.

The fine-grained compiler also inlines synchronization checks for heap accesses, and those synchronization checks may involve manipulating the work queues. In Chapter 3, we will examine the various overheads incurred by the fine-grained implementation including the synchronization check, the saving and restoring of thread state, thread synchronization, and thread scheduling overheads.

2.3 Run-time system

The run-time systems for Id perform a variety of tasks, including activation frame management, heap allocation, message-passing and scheduling. The run-time systems are written in C with some `gcc` extensions, and linked as a library with the object files generated by the Id compiler.

The run-time system has a *logical* view of the system as shown in Figure 2.3. There is a global shared memory which is accessible from all of the processors – the run-time system manages memory allocation for the shared memory. The processors communicate with each other through a message-

passing layer, which is written in an Active Message style [83]. By structuring communication as message-passing, the system can be easily ported to a parallel computer with support for both shared-memory and message-passing, such as Start-Voyager [8], Alewife [2], Flash [49], or to a cluster of workstations with a software distributed shared memory. In addition, message-passing simplified many of the protocols used for synchronization and work-distribution.

Activation frames reside in local memory, and are managed by the run-time system. Finally, local work queues for each processor reside in local memory. Physically, all of these structures are mapped onto an SMP – local memory is simply the main memory partitioned among the processors, and the message passing layer is implemented on top of shared memory.

Local work queues reduce contention and synchronization. Anderson, et.al.[7] provide a comparison of thread scheduling strategies for shared memory multiprocessors. The same approach was taken in the Multilisp [40] system, and for Cilk [16] – the parallel run-time system is a modification of the original SMP version of Cilk which we implemented.

The work queues are implemented as either LIFO or double-ended queues (deques), and each processor has several local work queues. Work which is ready to execute is enqueued on one of the work queues. If a processor runs out of work in its work queues, it sends a message to a random processor, requesting work. If that processor has some extra work it can share, it bundles up a descriptor of the work in a message, and sends the message. When the work is completed, the result is packaged up and sent back via another message.

Ready work can be represented simply as an instruction pointer, if the activation frame is implicit, or else as a continuation, consisting of an instruction pointer and a frame pointer. The compiler generates code which can handle both representations, depending upon the situation. The compiler also explicitly generates polling into the executable so that it can ensure that accesses to local memory are atomic. As in purely message-passing, distributed-memory machines, other processors may modify the local memory only through messages which are handled only when the local processor performs polling.

Enqueuing and dequeuing work using this representation simply becomes Enqueuing and dequeuing instruction pointers or pairs of instruction and frame pointers. Thread synchronization can be done without locking because of the message passing layer. The work-stealing policy tends to schedule large chunks of work for each processor, which do not require interprocessor synchronization or communication, and which do not have much overhead versus sequential execution because of the lightweight mechanisms for scheduling and synchronizing work.

2.4 Development hacks that made this study possible

The process of bringing up a system which is interesting enough to write about is usually a painful process, and most systems papers do not describe the non-research issues which came up in the development of their systems. Here, we describe some tools and strategies we used which made the development of this system possible by one person in about a year and a half.

2.4.1 Reusing code

By reusing the original Id front-end, we did not have to write and debug a front-end and scalar optimization phases. We estimate that there were about 250,000 lines in the original Id compiler, and we wrote about 100,000 lines for our new parallelization phases, back-ends, and run-time systems. There were plenty of bugs to catch in the new code, so it was a relief to have a front-end which was time-tested and stable. All of the Id compilers we used in this paper were written in Lisp, and we echo the recommendation of John Ellis in his PhD thesis [28] that if you don't know Lisp, learning Lisp is probably more useful than reading this thesis! 100,000 lines of Lisp is probably 200,000 lines of C or more.

2.4.2 Rapid prototyping

We originally generated Lisp rather than C or assembly – since we ran the compiler itself in a Lisp environment, the generated code could also be loaded directly into the Lisp interpreter, shortening the compile-debug-edit cycle. The Lisp interpreter retained enough source information that we could easily track down bugs in the partitioning or code-generation phases.

The performance of the interpreted Lisp we generated was not very good, but more than sufficient for prototyping, and fast enough to run all of our programs with small data sets. Once we decided that the compiler was stable enough to generate C, the process of generating C took only a few days. Most of the bugs in the various stages of the compiler were caught by the prototype compiler generating Lisp. The "extra" time spent in prototyping the compiler by generating Lisp probably saved months compared to directly generating and debugging C or assembly.

2.4.3 Test suites and automatic regression testing

Anytime we added a new feature or optimization, it would inevitably break the compiler. Tracking down these bugs on a large application code would be very difficult, but we had a test suite of small codes which we could send through the compiler. The tests were ordered by simplest to most complicated, with tests exercising increasingly complicated features grouped together. Typically, running the compiler through all of the test suites caught most of the bugs. Whenever we found a bug we didn't catch with the test suite, we added a new test which exercised the bug. Since we had hundreds of test programs, automatic testing was important because it would have been very tedious to run the compiler and check the results manually.

2.4.4 Generating C and gcc extensions

We generate non-standard C targeted at the `gcc` compiler [78]. The features we use are described in great detail in the paper about the implementation of the Mercury language [42]. Some of the extremely useful gcc extensions we used were pointers to labels, inlined assembly language, inlined functions, and the ability to assign global variables to specific registers. The ubiquity of the `gcc` compiler made our decision easier – we believe that generating C with gcc extensions is an ideal target for academic and prototype language and compiler projects.

There were some low-level optimizations we couldn't perform because we generated C, but the `gcc` compiler also performed some optimizations we did not have to do in our compiler. More importantly, generating C also allowed us to directly use some of the C infrastructure, such as the `gdb` debugger and the `gprof` profiler. Without a good debugger and profiler, implementing a new compiler would have been much more difficult.

Readers who are implementing parallel languages should note that the `gdb` documentation describes a technique for debugging multiple processes, which is incredibly useful when one of your child processes accesses an illegal memory location, a fairly common occurrence in developing a parallel language and compiler, and one which can be extremely difficult to track down without a debugger.

2.4.5 Graph viewer

To aid in compiler development, we wrote a program graph viewer with a window interface. This was quickly assembled with the `dot` graph layout tool, and the Tk [61] interface to it called `graphviz`

[27].

Some compiler bugs can easily be found by examining the compiler output, but others are more subtle, and require understanding entire procedures and how the compiler decided to compile them. Initially, we attempted to find these bugs by doing textual dumps of program graphs, but this was extremely time-consuming and error-prone. This was especially the case for debugging the partitioning phase of the compiler, where the graphs were well-formed, and the compiler was making errors in assigning nodes to partitions.

The program graph viewer allowed us to understand large program graphs in minutes, which would have otherwise take hours to track down from a text dump. The graph viewer is structured as a compiler phase which can be inserted at different points in the compiler. By using the Tk windowing interface, graphs which are larger than the screen can be displayed and scrolled, and multiple subgraphs can be displayed simultaneously. More detailed information about the graph, subgraphs and nodes can be annotated to the visual display.

2.4.6 Linker implemented in Perl

The Id linker must perform a little more work than the C linker because top-level constants must be initialized before we begin execution of the `main` procedure. To handle this, the Id linker generates a constant initialization procedure from the information it extracts from the Id object files which are being linked, compiles the constant initialization procedure and links it with the rest of the object files. The linker itself is written in Perl, which has good text processing and system integration support, allowing us to implement the linker in less than 150 lines of code, while giving us flexibility in easily adding functionality to the linker.

Chapter 3

Fine Grained Id Overheads

This chapter examines the overheads of a TAM-style [25] fine-grained implementation of Id on conventional microprocessors, and concludes that the overhead incurred is too high to make a fine-grained approach with element-wise heap synchronization useful for small-scale SMP's.

Element-wise heap synchronization in combination with Id90's non-strict semantics also provide a measure of expressiveness over sequential Id-S, but our experience indicates that programmers rarely use this additional expressiveness. Almost every non-contrived Id program we found ran correctly under sequential evaluation order, with minor changes in ordering of variable bindings, as we explained in Section 2.4 – this result concurs with Schauser and Goldstein's study of expressiveness and non-strictness [70]. Non-strictness (including function call non-strictness, conditional and loop non-strictness, as well as data structure non-strictness) is primarily useful as an implementation strategy to expose fine-grained parallelism, and is less useful in providing expressive power to the programmer.

3.1 Methodology

To determine the overheads of a fine-grained approach to compiling Id, we implemented a compiler very similar to the TAM compiler, using a similar partitioning algorithm [72] and code-generation schemas [32]. We make the optimistic assumption that strictness analysis of the type described by Schauser [72] and Coorg [22] can determine that every procedure call, conditional and loop can be executed strictly – this assumption minimizes the overhead of conditionals and parameter passing, which would otherwise require multiple entry-points for each procedure call and conditional, more partitions, and additional synchronization and threading overhead.

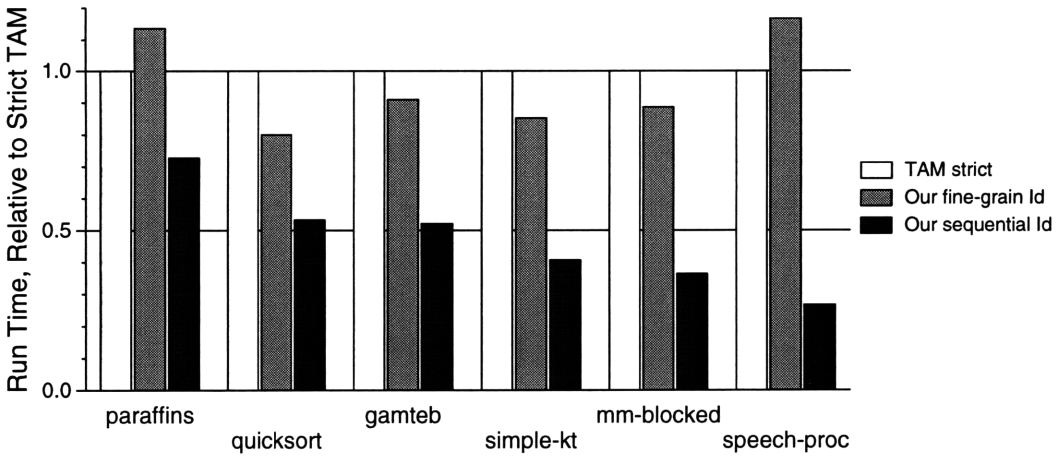

Figure 3.1: Normalized run-time of fine-grain and sequential Id-S compilers compared to the strict TAM compiler running on a Sparc 10.

Program	Input Size
paraffins	19
quicksort	10,000
gamteb	40,000
simple-kt	1 1 100
mm-blocked	500
speech-proc	10240 30

Figure 3.2: Program input sizes

3.1.1 Performance compared to strict TAM

To compare our times against those published by Schauser [72], we ran our test codes on one processor of an unloaded Sparcstation 10. Figure 3.1 shows the performance of our compiler relative to the strict TAM compiler, which is the best previous implementation of Id to our knowledge. Figure 3.2 shows the input sizes used for the programs.

Paraffins enumerates the structures of unsaturated hydrocarbons, and is a very unstructured code which uses many lists and other small data structures; the program generates all paraffins whose radius is less than or equal to the input size. Quicksort is an integer quicksort on a *list* of the input size. Gamteb is a photon-transport simulation which tracks photons moving through a cylinder and records a histogram of their behaviors; the input size is the number of photons. Simple-kt is a structured two-dimensional hydrodynamics simulation on a square grid; the input size describes the number of time steps taken, and

the lower and upper bounds of the square grid. MM-blocked is a 4×4 blocked double-precision matrix multiply on square matrices where the input size is the length of a side of the matrix. Speech-proc is a front-end code for a speech recognition system which has several very structured inner loops; the input size describes the number of samples and the size of a digital filter applied to the samples.

Not surprisingly, our implementation is comparable in speed to the strict TAM implementation, with performance within 20% of the strict TAM implementation. The sequential implementation, however, is a factor of 1.3 to 4 faster than the strict TAM implementation. For more symbolic, unstructured codes such as quicksort and paraffins, the penalty is less, but for more structured, scientific codes such as mm-blocked and speech, the sequential implementation is significantly faster.

This chapter focuses on determining the sources of overhead which cause the difference in performance between our fine-grained implementation and our sequential implementation of the same applications.

3.1.2 SimICS Sparc simulator

To determine the details of the fine-grained overheads, we use SimICS [50], an instruction-level Sparc simulator which can execute ordinary uninstrumented Sparc binaries. SimICS can provide instruction counts with both opcode *and* operand data, so that for example, we can determine how many loads or stores were performed using a given register and offset.

In addition to instruction counts, SimICS performs a first-level cache simulation of the SuperSparc (not UltraSparc) architecture with the following assumptions:

- 16 KB data cache (32-byte lines, 4-way set associative)
- 20 KB instruction cache (64-byte lines, 5-way set associative)

Because SimICS only provides instruction counts and miss ratios for the first-level cache, we cannot determine the effects of superscalar execution, second-level cache, TLB or paging, all of which may all have some impact on real observed performance.

3.1.3 Relationship between run-time and instruction counts

Figure 3.3 shows a comparison of the execution time ratios between the fine-grained and sequential versions, and the ratio of the instruction counts. The execution time ratios vary from about 1.5 to over 4.

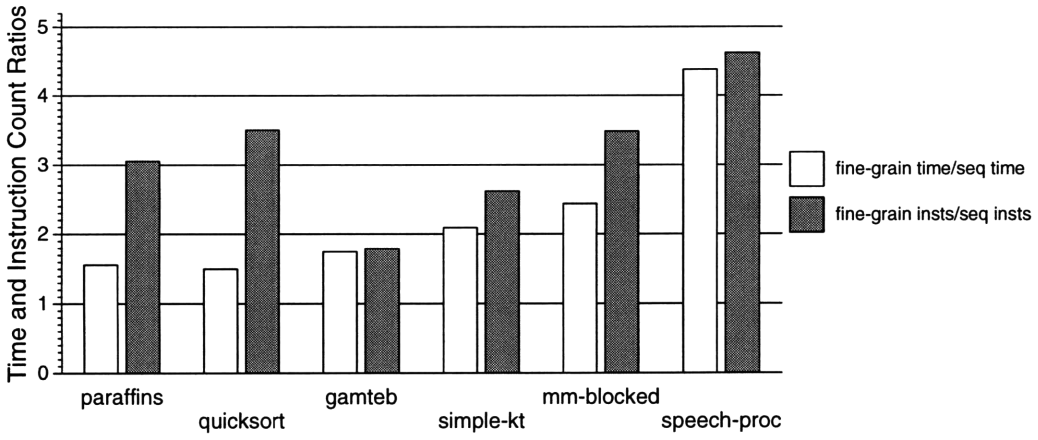

Figure 3.3: Execution time ratios compared to instruction count ratios. Much of the additional execution time for the strict versions is accounted for by additional instructions executed, although some of the additional instructions are masked by poor cache locality.

	Fine-Grain		Sequential	
Benchmark	# instructions	# dcache misses	# instructions	# dcache misses
paraffins	$4.7e+07$	$1.0e+06$	$1.5e+07$	$7.4e+05$
quicksort	$3.4e+07$	$4.2e+05$	$9.8e+06$	$3.5e+05$
gamteb	$4.1e+09$	$1.2e+07$	$2.3e+09$	$1.8e+07$
simple-kt	$7.7e+07$	$9.3e+05$	$2.9e+07$	$3.5e+05$
mm-blocked	$2.6e+09$	$2.6e+07$	$7.5e+08$	$1.3e+07$
speech-proc	$4.3e+07$	$4.4e+04$	$9.3e+06$	$1.6e+04$

Figure 3.4: Although the ratio of instructions between the fine-grained and sequential versions is between 2 and 5, the total difference in data cache misses between the versions is not as great – for codes with have a naturally high cache miss rate, a big reduction in instructions will not have as big a reduction in time because much of the time is spent handling cache misses.

However, many more instructions are executed for the fine-grained versions than could be determined from execution time alone, and the difference is much more notable for the less structured codes than for the structured codes.

One possible explanation is that caching effects mask some of the additional instructions executed in the fine-grained version, and that the worse cache hit ratio for the less structured codes mask the extra instructions even more than for the more structured codes. Figure 3.4 bears this out – in the codes with a naturally high data cache miss rate (paraffins, quicksort, and mm-blocked), the additional instructions added by the multithreaded version do not cause a lot more cache misses.

Figures 3.3 and 3.4 show that instruction counts, to the first degree, give a reasonable view of the

Presence-checking read of A[i]

```
if (((char *) A)[-i-1] == EMPTY)
  error("Read from empty location");
else
  dest = A[i];
```

Presence-checking write of A[i]

```
if (((char *) A)[-i-1] != EMPTY)
  error("Write to non-empty location");
else {
  ((char *) A)[-i-1] = FULL;
  A[i] = value;
}
```

Figure 3.5: The presence tags for a structure reside at negative offsets from the structure base, and are initialized to be empty. Each tag is a byte wide, to make reading and writing the tags cheaper.

eventual execution time, once cache effects are taken into account.

In the following sections, we examine the source of the additional instructions executed for the fine-grained version compared to the sequential version. We examine two sources of overhead in the fine-grained version: (1) presence tag checking and (2) threading overheads.

3.2 Presence tag checking overhead

To determine the overhead of presence tag checking in isolation from other threading overheads, we modified the sequential compiler to insert presence tag checking code used in the strict compiler for every memory reference. This checking, of course, is not necessary for the correct execution of the code, but it gives us a feeling for the overheads introduced by tag checking. This version of the compiler can also be used in development to check that the programmer is adhering to the single-assignment restriction.

3.2.1 Presence tag checking scheme

Figure 3.5 shows the code which is executed for a presence check during a read or a write. The presence tags reside at a negative offset from the structure base, and are initialized to be empty. On a read, an empty presence tag is an error for a sequential evaluation order, and on a write, a non-empty presence tag is an error. A write to a location updates the corresponding presence tag to be full. This presence tag scheme is very simple, and has the advantage of a small footprint for the presence tags and simple addressing.

Note that this presence checking scheme will only work for a single processor. For parallel execution, we must include some support for atomicity in referencing the presence tags, which will incur significant

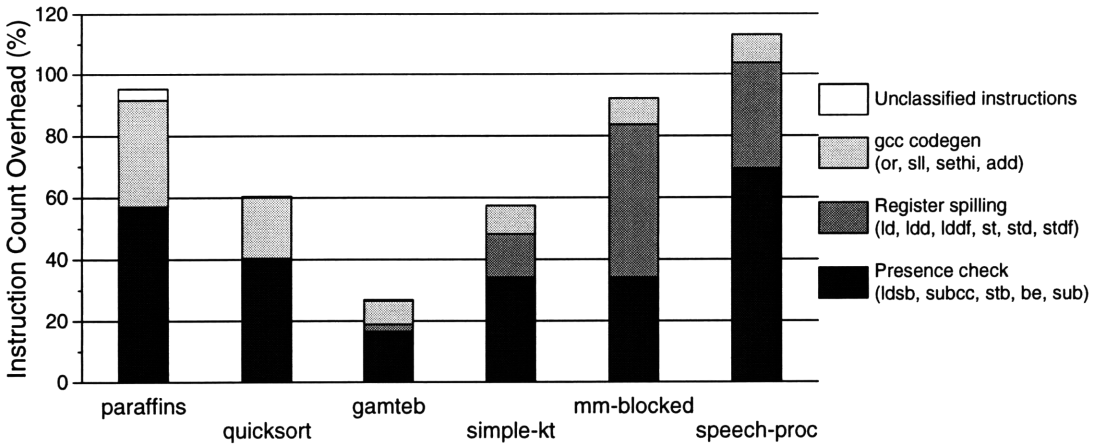

Figure 3.6: Presence tag overhead, classified into three main sources – instructions used in the presence check, instructions coming from register spilling, and instructions introduced by poor code generation by gcc.

additional overhead. In parallel versions of TAM, this overhead manifests itself indirectly through message-passing overheads – in parallel TAM codes for the CM-5, the single processor execution times are twice as long as those shown in Figure 3.1 because of message-passing overheads.

Our performance numbers are optimistic in two ways. First, we take advantage of operating system support to initialize the presence tags to zero (we allocate all presence tags from an `mmap` from `/dev/zero`). In a production system, the presence tags must be initialized to zero by the storage manager, although they can be done either 4 at a time (a 32-bit write) or 8 at a time (a 64-bit write). The second way in which these numbers are optimistic is that we do not account for atomicity overheads which are necessary in a real parallel environment – the presence tags operations must perform a read, modify, and write atomically. This can be done by locking every presence tag, or through some clever use of a load with reservation, store conditional instruction, which is not available on every microprocessor – it is not available on the Sparc, for instance.

3.2.2 Presence check performance results

Figure 3.6 shows the makeup of the additional instruction count overhead caused by tag checking. We categorized the instructions by determining the additional instructions executed by the presence tag checking version as compared to the regular sequential version. For example, instructions needed for the presence check are a subtraction (sub) to obtain the negative offset from the object bases, either a

byte load (ldsb) or store (stb), a comparison to a constant tag (subcc), and a branch (be). Instructions dedicated to register spilling include the various forms of loads (ld, ldd, lddf) and stores (st, std, stdf) – spilling is introduced when large basic blocks are chopped into multiple smaller basic blocks because of the additional branches introduced by presence checking for every heap memory reference.

In examining the assembly output from the gcc compiler, we also determined that gcc was not able to determine that constant offsets from object bases could be constant folded to yield a single load or store to access the presence tag. This arises more in codes which use small structures (such as list cons cells and algebraic types) rather than arrays. In these cases, the gcc compiler created a small literal, added it to 1 (add), and then performed the presence tag load/store. It used the same literal, now in a register, shifted it (sll) and performed the data load/store. Finally, the gcc compiler was not able to correctly insert the right instruction into the delay slot – instead of inserting an instruction from the branch taken destination, it inserted an instruction from the branch not taken destination, which is never taken. All of these overheads due to gcc could be eliminated by a native code generator, or some more trickery to fool gcc into performing constant folding and correct filling of the delay slot.

Depending on the application, the direct overhead from presence checking ranged from 20% to 65%. Register spilling was more of a significant problem in the structured scientific codes which had fairly large basic blocks which were benefitting from good register allocation. The overhead from gcc code generation could be eliminated by a marginally smarter compiler.

3.3 Threading overheads

It is a little bit more difficult to determine the overheads of multithreading, because the structure of the multithreaded code is quite different from the sequential and tag-checking sequential code. When we compared the sequential and tag-checking sequential code in Section 3.2, we made the assumption that most of the instructions generated by the compiler would be the same, except for the additional instructions inserted for presence tag checking. This assumption is probably well-founded, although we discovered that some extra instructions were being executed for register spilling and poor code generation by gcc.

To determine the sources of threading overhead, we make the assumption that most of the instructions executed in the tag-checking sequential versions are also executed in the fine-grained multithreaded version. We then subtract out the instructions executed by the tag-checking sequential version from the

fine-grained multithreaded version, and we attempt to classify those additional instructions.

3.3.1 Bookkeeping for threading overheads

Because we reserve a few Sparc registers for special purposes, we can account for instructions which use those registers. For instance, register `%l0` is the activation frame base – byte-wide loads and stores (ldub, stb) relative to this register are join counter loads and stores, because join counters are byte-wide slots in the activation frame. Other loads and stores relative to `%l0` are used for loading and storing thread state to the frame for thread switching. Likewise, registers `%l1` and `%l2` are reserved for work queue pointers, and references to those registers are used to push and pop work from the work queues – we can also find the add instructions which are used to increment and decrement the work queue pointers.

After we count the instructions we know are used for a specific purpose, we adjust the counts by adding instructions we know must be executed in addition to the classified instructions. For instance, if we count an instruction which is used to load from a join counter, we know that a subtract, a compare and a branch were also executed. Stores to the local queue are accompanied by two instructions, an "or", and a "sethi" in order to set a register to be a destination thread instruction pointer.

The categorizations of the different threading overheads were determined as follows:

- Thread state save/restore
  - All non-byte-sized loads from `%l0` offset
  - All non-byte-sized stores from `%l0` offset
- Join synchronization
  - All byte-sized loads from `%l0` offset, and an additional 3 instructions for each load (subtract, compare, branch)
  - All byte-sized stores from `%l0` offset
- Thread scheduling
  - All loads from `%l1` and `%l2` offsets
  - All stores to `%l1` offset, and an additional 2 instructions (or, sethi) for each store.
  - All stores to `%l2` offset
  - All adds and subtracts to `%l1` and `%l2` registers
  - Indirect jumps through register `%o0`
- Unclassified instructions
  - All remaining instructions

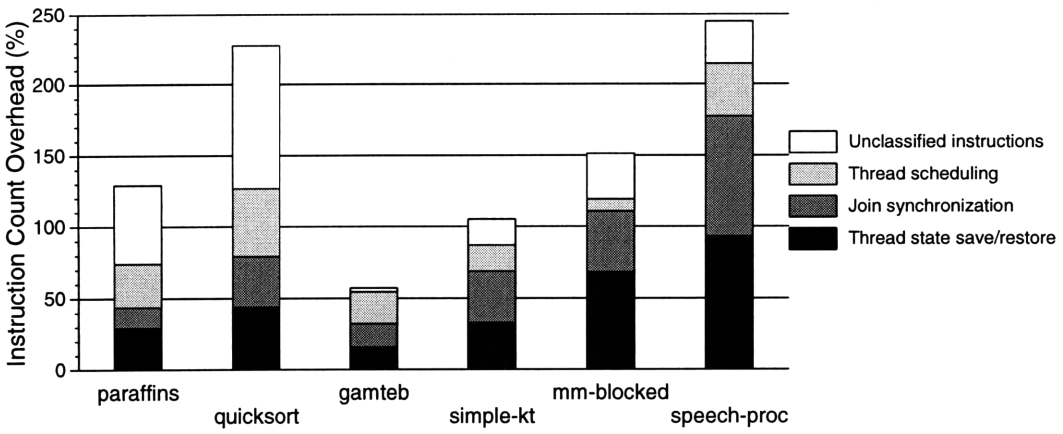

Figure 3.7: Threading overhead of strict version, relative to (non-tag-checking) sequential version.

Some of the unclassified instructions are performing tasks in one of the three classifications (thread state save and restore, join synchronization, thread scheduling) but we could not determine how to classify these instructions.

3.3.2 Threading overhead performance results

Figure 3.7 shows the results of the simulations and overhead bookkeeping. The sources of overhead are fairly evenly split between thread state save/restore, thread join synchronization and thread scheduling. The overheads are very significant, adding from 50% to almost 250% to the raw sequential instruction count, with unclassified instructions adding even more overhead for some applications.

To some extent, these overheads might be reduced by better compiler optimizations, but in general, we believe that the results indicate that the fine-grain multithreaded approach requires some hardware support, such as that provided in Monsoon [62], EM-4 [66] or Tera [4]. This support includes multiple contexts, hardware thread scheduling and support for thread synchronization.

These results, however, only apply to very fine-grained multithreading where synchronization is necessary on every memory reference. In the next section, we show that a coarser grain of multithreading without synchronization on memory references can be almost as efficient as sequential execution.

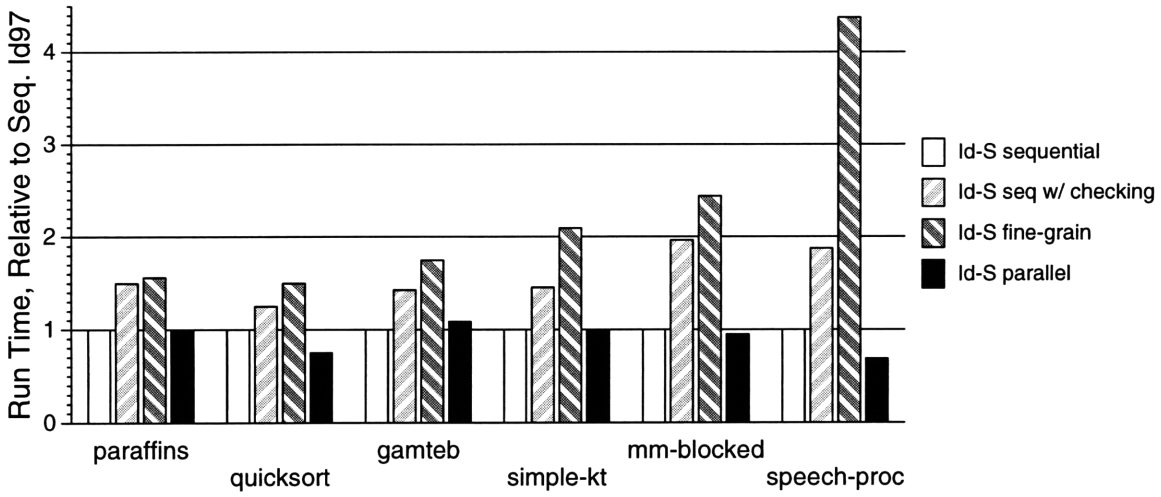

Figure 3.8: Run-time on a Sparc 10 performance of sequential, sequential with checking, fine-grained multithreading and parallel multithreading without element-wise synchronization.

3.4 Overhead summary and coarser-grained multithreading

To bring our performance evaluation back to real run-times, Figure 3.8 shows the normalized run-time performance of the sequential, sequential with checking, and the fine-grained multithreaded version. There is also a coarse-grained multithreaded version which we will discuss in more detail below.

About half of the overhead in time is accounted for by the tag-checking. The remaining overhead is accounted for by various multithreading overheads. As was shown in Figures 3.3 and 3.4, the run-time overhead is not directly proportional to the instruction-count overhead, especially in codes which have high cache miss rates.

Although these overheads are significant, they are due to the fine-grained synchronization associated with each memory reference. The last bar in Figure 3.8 shows the performance of our parallel implementation, to be discussed in greater detail in the following chapters, which only attempts to take advantage of parallelism at the function call and loop level, and which does not perform synchronization at every memory reference. Every function call in the parallel implementation is potentially parallel, and many of the loops are also potentially parallel.

In some cases, such as in quicksort, mm-blocked, and speech, the performance of the parallel version running on one processor is actually *faster* than the sequential version. This is because these codes have many recursive function calls, and in the sequential implementation, we use the standard C calling

convention on the Sparc, which uses register windows. Register windows incur significant overhead when they overflow, as the state of each window must be backed up to the stack. (This problem is especially serious in the SuperSparc processor used in the Sparc 10.) The calling convention in the parallel version does not use register windows, so it avoids these problems.

The run-time system for the fine-grained multithreaded and the parallel implementation are almost identical. Furthermore, many of the compiler modules, including the partitioning and code-generation modules, are the same. The primary difference is that the parallel implementation uses several additional modules to eliminate the need for element-wise presence-tag checking, and consequently, also produces much larger threads.

The timing results in Figure 3.8 will also look somewhat different from the graphs in the following chapters, because we ran these codes on a Sparc 10 which uses a SuperSparc processor, whereas in later chapters, we are using an UltraSparc server which has much higher performance. The problem sizes we are using for the runs in this chapter are also taken from previous studies, and are too small to produce accurate timing results on the UltraSparc, so we will be using larger problem sizes when running on the UltraSparc.

3.5 Related work

3.5.1 Software based fine-grain synchronization

Yeung and Agarwal performed a study of preconditioned conjugate gradient [87] to determine the importance of fine-grain synchronization. They concluded that fine-grained synchronization was important in exposing parallelism, but that the particular implementation of the fine-grained synchronization was less important, and that a software implementation could be almost as effective as a hardware implementation.

The application they chose required fine-grained synchronization to expose parallelism, and they chose a particularly small problem size ($16 \times 16 \times 16$) which could not be parallelized effectively in a coarse grained fashion. Their study assumed some hardware support for scheduling and thread synchronization, and although they varied the cost of heap memory synchronization, they did not take into account factors such as register spilling in a software based heap synchronization.

3.5.2 TAM

The work most closely related to our study of software-based fine-grained multithreading is the TAM compiler, which has been studied extensively [24] [25] [77] [32] [72].

Most of the performance studies of the TAM compiler are done on instruction mixes of the *abstract* TAM machine. However, the relationship of abstract machine instructions and actual native machine instructions is not one-to-one; for simple arithmetic or logical instructions, one TAM instruction may be compiled to one machine instruction, but for scheduling and communications instructions, often one TAM instruction compiles to several, or even tens of machine instructions. The TAM approach hides some effects such as that of register spilling shown in Figure 3.6, because the TAM compiler is not aware of register spilling introduced by the C back-end and cannot instrument the code to include those instructions.

The TAM performance studies are also not calibrated to an efficient sequential implementation, except in the case where they compared to C or Fortran implementations of the same code. Cross-language performance comparisons are difficult to perform because often, even slight changes in the algorithm can cause major differences in performance, and different languages encourage different programming styles. Because we are compiling the same source file and using much of the same code-generation and run-time system support, we can factor out the overheads which are due to the original code, the language semantics, and the run-time system (such as variable array-sizes, implementation of higher-order function calls, and the heap allocator).

3.5.3 Software distributed shared memory systems

The presence tag-checking approach we describe is similar to those used in software based distributed shared memory (software DSM) systems such as Shasta [69]. In the Shasta system, the compiler has the responsibility of reducing tag checking, which it can do because of the differing requirements of software DSM and synchronization. With aggressive compiler optimizations and clever tag representations and tag checking schema, the Shasta system has managed to maintain very low overhead over sequential execution.

However, our approach differs from the software DSM systems, because they can typically use a single tag for a cache line of many words, whereas we must typically require a tag for each memory word, unless the compiler can determine when multiple adjoining words are written together and then match

those writes with the consumers of those locations. This is an extremely difficult compiler problem, and not one that we expect to be solved in the near future.

3.6 Conclusion: fine-grain parallelism too expensive in software

In this chapter, we compared the performance of a state-of-the-art fine-grained multithreaded implementation of Id with an efficient sequential implementation of Id. The performance results indicate that a fine-grained implementation requires a factor of 2 to 4 times as many instructions as the sequential implementation. The additional instructions do not directly translate into additional run-time because of caching effects, but a fine-grained multithreaded implementation requires between 1.5 to 4 times as much wall clock time as a sequential implementation. Approximately half of the additional run-time can be attributed to presence tag checking and the other half can be attributed to other threading overheads.

Some of the presence tag overhead could be reduced by better code generation, as we discuss in Section 3.2. Compiler-directed data dependence analysis might also eliminate some redundant or unnecessary presence tag checks. In a local context, this analysis is similar to fetch-elimination, which obviates the need for this optimization. In a more global context, producer-consumer analysis is only plausible for codes which are amenable to Fortran-style index analysis, and for an Id with sequential semantics. Fully non-strict Id does not yield itself easily to data dependence analysis, and codes which are amenable to Fortran-style index analysis can probably be parallelized quite effectively without fine-grained I-structure style synchronization.

Aside from tag-checking, threading overheads are about equally divided into three sources: thread state save and restore, join synchronization and thread scheduling overheads. As shown by Figure 3.8, these overheads are not inherent to multithreaded execution – only to multithreaded execution at such a fine granularity.

We believe that given enough effort, many of the overheads due to fine-grain multithreaded execution could be lowered, but even optimistically assuming that half of the total overhead could be eliminated (not including atomicity overheads) the fine-grained version would still be about 1.25 to 2.5 times slower than the sequential implementation. For small-scale machines, this overhead is difficult to win back, as we showed in the simple analytical model in Section 1.1. This overhead is also unnecessary in medium or high-parallelism scenarios, where our compiler-directed parallelization can detect sufficient parallelism to keep the machine fairly busy. In many cases, fine-grained multithreading is simply

not necessary to attain good parallel performance on a small-scale machine.

A fine-grained multithreaded approach may require hardware support to be effective – this hardware support may include presence tags in memory, special read and write instructions which perform tag-checking, multiple hardware thread contexts, join synchronization support, and hardware thread scheduling. These features have been implemented in some experimental machines [62] [45] [66], but are unlikely to be included in commercial hardware in the near future. As such, we will describe an approach in the chapters which (1) parallelizes Id-S so that presence-tag checking is unnecessary and (2) generates thread lengths that are sufficiently long to amortize threading costs.

Chapter 4

Parallelizing Id-S

In this chapter, we describe a very simple approach to parallelizing Id-S which discovers both procedure-call *DAG* parallelism, as well as *loop* parallelism. Compiler-directed parallelization eliminates the need for the fine-grained synchronization checks used in traditional Id implementations. We then evaluate the effectiveness of the compiler parallelization, giving both a quantitative description of the *idealized parallelism* discovered, as well as a qualitative description of the strengths and weaknesses of this approach to parallelization.

Our approach to parallelization works because of Id-S's *single-assignment semantics* for data structures – each element of a data structure may be written only once, but may be read many times. Single-assignment semantics eliminate output-dependences and anti-dependences, so that parallel evaluation is only restricted by true data dependences. As in functional languages, Id-S's lack of true side-effects forces the user to frequently perform heap allocation, which disambiguates some data dependences for the compiler.

The effect of Id-S's single-assignment semantics apply to scalar variables as well as to data structures – scalar variables in Id may only be "assigned" once, and "updates" to scalar variables are handled by creating a new variable.

We begin with a motivating example, describing our approach in the context of discovering DAG parallelism. Discovering loop parallelism is a simple extension to handling DAG parallelism.

```
def foo A B C i j k =
  {
    C[k] = A[i] + B[j] + 1;
  };

def bar A B C l m n =
  {
    B[l] = A[m];
    C[n] = A[n];
  };

def baz A B C i j k l m n =
  {
    A = i_array(1,10);
    B = i_array(1,10);
    C = i_array(1,10);

    # some intervening code ...

    _ = foo A B C i j k;
    _ = bar A B C l m n;
  };
```

Figure 4.1: In this code fragment, the calls to `foo` and `bar` within `baz` can occur in parallel, because there is no data dependency between the calls to `foo` and `bar`.

4.1 Simple parallelization example

Figure 4.1 shows an example of sequential Id-S code. Can the procedure calls to `foo` and `bar` in the procedure `baz` be executed in parallel? This is the question at the heart of our parallelization – if we can answer this question, we can discover other places where we can perform procedure calls in parallel, and a simple extension allows us to determine when we can execute loop iterations in parallel.

Arrays `A`, `B`, and `C` are allocated in function `baz`, and they are referenced in the calls to `foo` and `bar`. This code fragment shows how Id-S is different from functional languages such as Haskell or Sisal, where object elements may only be defined when the object is created – in Id-S, objects may be created, then updated, but each element of the object may only be updated once. Functional constructs in Id-S such as array and list comprehensions, functional conses, functional tuples and functional structures are de-sugared into a heap allocation and then fill-in.

Because of the array references in the program, we cannot determine whether the two procedure calls can be executed in parallel without *interprocedural analysis* which determines which objects can be *aliased* (either via procedure arguments or return values) and what *side-effects* are performed to which objects in each procedure call.

By inspection, we can determine that there is no aliasing due to procedure argument passing or return values in the above program fragment. An interprocedural side-effect analysis indicates the following:

- procedure `foo` reads `A`, `B` ; writes `C`
- procedure `bar` reads `A` ; writes `B`, `C`
- procedure `baz` reads `A`, `B`, etc. ; writes `B`, `C`, etc.

If we don't know the values of the array indices (`i`, `j`, `k`, `l`, `m`, `n`) are related to each other, we must conservatively assume that they may overlap. Is there a dependence due to the side effects to the arrays in the calls to `foo` and `bar`?

If we examine the references, `A` is read by both `foo` and `bar`, but that does not cause a dependence. `B` is read by `foo` and written by `bar`, but because of the sequential single-assignment semantics, we know that they must be referencing different slots of `B` because the call to `foo` occurs sequentially before the call to `bar`, so any elements referenced by `foo` must have already been defined before `foo` was called. Furthermore, if those elements had been defined, they may not be re-defined, because of the single-assignment semantics. For imperative languages, the references to `B` by `foo` and `bar` would cause an "anti-dependence", because `bar` might over-write a location referenced by `foo`. Id-S does not have anti-dependences because of its single-assignment semantics.

Finally, `C` is written by both `foo` and `bar`. Again, because of the single-assignment semantics, we can infer that they must be writing to different elements of `C` – otherwise, the program would be incorrect. Because they are writing to different elements of `C`, there is no dependency. For imperative languages, these references would cause an "output-dependence" because the writes to `C` from `bar` might overwrite the writes to `C` by `foo`, and later code might reference those over-written elements. Id-S does not have output-dependences because of its single-assignment semantics.

Because there are no dependences due to direct scalar dependences between `foo` and `bar` (for example, if `bar` used a value returned by `foo`) and there are no dependences through data structures between the two function calls, they may be executed safely in parallel.

4.2 Commentary on parallelization

From the example in the previous section, several characteristics of our parallelization were introduced or hinted at, which we briefly discuss here before we go into more detail about how the parallelization

analysis is performed.

4.2.1 Role of single-assignment semantics

Id-S's single-assignment semantics allows our analysis to ignore output-dependences or anti-dependences, because they are not legal. If the program in Figure 4.1 were written in an imperative language such as C or Fortran, the compiler would either give up when it discovered an output-dependence or anti-dependence, or else attempt to disambiguate the reads and writes which caused the output-dependence and anti-dependence, to show that they do not overlap.

Traditionally, this further disambiguation of read and writes is done with *index analysis*, although it could also be done with other techniques such as strong typing or run-time checks.

4.2.2 No index analysis

In the example code, index analysis may be very difficult or impossible to do because the values of the indices are passed through procedure calls, and may be calculated at run-time. Index analysis is most powerful when used for nested loops where the indices are affine functions. The compiler can then perform some automated proofs that the indices will not overlap, usually by constructing and solving sets of linear inequalities.

Our approach does not do any index analysis, at all. The analysis traces potential reads and writes to structures, without bothering to keep track of the indices of those reads and writes. As we discuss later, we could discover more parallelism with index analysis, but we have found a significant amount of parallelism, even without index analysis.

4.2.3 Whole program analysis

Although not strictly necessary, our approach works best when we can examine the entire program. This should be clear from the example, because if we did not have access to the functions `foo` or `bar`, we could not determine whether we could parallelize the two calls to them in `baz`. On the other hand, we could determine that those calls could be executed in parallel, even without looking at the code between the structure allocations and the two function calls in `baz`. In this thesis, all of our results are with whole program analysis.

We do not handle higher-order function calls, because in general, it is difficult or impossible to determine the possible flow of control with higher-order functions, making it difficult or impossible to determine the side-effects caused by individual function calls. This is not a fundamental limitation – any control-flow analysis which works in the context of higher-order functions [41] [75] would be easily incorporated into our parallelization framework.

We handle closures by using the machinery in the Id compiler front end which converts all closures into lambda-lifted top-level procedures, with closure variables explicitly passed in through procedure variables.

In effect, the language we are parallelizing is a sequential, single-assignment, first-order language, and all of our results assume that we have access to the entire program.

4.2.4 Extension to loop parallelization

There is nothing fundamentally different about the discovery of loop as opposed to DAG parallelism – our analysis is essentially the same as we sketched out in the previous section. In parallelizing loops, consider each loop body as a separate procedure, and the iterations of the loop as separate calls to the loop body.

If the separate calls to the loop body do not have any potential true data dependences, then we parallelize the loop, just as we parallelize the two procedure calls when we find DAG parallelism.

4.2.5 Parallelization is decoupled from scheduling

As in other compiler-driven parallelization approaches, the parallelization analysis is somewhat decoupled from the run-time scheduling and also code generation. Unlike compiling Id90 for sequential processors, when we parallelize Id-S, we can choose whether or not to generate parallel code for instances where we find that we can legally execute loops or procedure calls in parallel. Our choice of whether or not to generate parallel code may be driven by the run-time system we choose, or guesses as to whether the code is worthwhile executing in parallel or not.

The run-time scheduling is decoupled from the parallelization to a degree, in that the parallelization does not determine which processors procedure calls or loop iterations are run on, or whether a lazy or eager approach to forking parallel work is taken. Parallelization only determines the legality of parallel execution.

Compiler Parallelization Stages		
1	Find procedure local mod and ref sets	417 lines
2	Find interprocedural mod and ref sets	345 lines
3	Find procedure argument aliases	268 lines
4	Find procedure return value aliases	247 lines
5	Incorporate interprocedural mod, ref & alias information locally	162 lines
6	Add data structure dependence arcs	383 lines
7	Classify loop induction variables	254 lines
8	Mark parallel loops	100 lines
	Total Line Count	2,176 lines

Figure 4.2: Compiler parallelization stages and line counts of each stage, including comments and debugging code. The compiler was written in Lisp, increasing code density, but the compiler stages are still extremely short and simple.

4.3 Approach to parallelization

From the example in Section 4.1, it is clear that *interprocedural side-effect analysis* can tell us when we can safely execute two procedure calls in parallel. Interprocedural side-effect analysis is a fundamental and well-studied problem, and is used in sequential compilers for many optimizations, including code motion, common subexpression elimination, and loop constant hoisting. A simple treatment of this analysis is given in Aho, Sethi and Ullman's introductory compiler text [3]. Our approach to interprocedural side-effect analysis is not significantly different, except in our treatment of heap-allocated data structures, and the way that we use the side-effect information.

Figure 4.2 shows the parallelization analysis stages in the compiler, and their line counts including comments and debugging code. The parallelization stages are a small fraction of the Id-S compiler, and these analysis phases already exist in most optimizing compilers for conventional languages. Note that by comparison, just the integer programming core of the Omega test [65], a popular Fortran-style index analysis algorithm, requires over 4,000 lines of C – not including the data dependence analysis required to set up the integer programming problems. The DAG and loop parallelization analysis for Id-S is considerably simpler than the loop parallelization analysis for Fortran.

Stages 1-5 perform the interprocedural side-effect analysis, which computes the data structures each program graph node potentially references (reads or writes). The set of data structures potentially read by a node is called the *ref set*, and the set of data structures potentially written by a node is called the *mod set*. This analysis must trace objects through function calls, conditionals and loops, and resolve

aliasing which could arise from either parameter passing or return values. Our interprocedural side-effect analysis is based on Cooper and Kennedy's [19] [20] refinement of Banning's original solution [15] to finding interprocedural side-effects.

Stage 6 uses the mod/ref information to insert new data dependence arcs into the program graph. After this stage, any two nodes which do not have a dependence between them in the program graph may be executed in parallel. All of the DAG parallelism is exposed after this step.

Stages 7 and 8 expose loop parallelism using the side-effect information. Each loop is analyzed to see if it can be parallelized, and each loop which can be parallelized is marked for special treatment in later compiler phases. Not every loop which can be parallelized is parallelized – some loops do not perform enough work to warrant parallelization.

The following sections describe the compiler stages in more detail. Readers who are familiar with interprocedural side-effect analysis may choose to skip Sections 4.3.2 to 4.3.6.

4.3.1 Parallelization example 2

Figure 4.3 shows an Id-S program which calculates $\sum_1^n fib(n)$ where $fib(n)$ is the nth fibonacci number, using a very indirect algorithm. The main function creates an array of size n, and fills the $i'th$ element with a binary "fibonacci" tree, where each node of the binary tree either is a leaf node containing a value of '0' or '1', or else an internal node containing two fibonacci subtrees. Finally, the sum of the leaves of the fibonacci trees is taken in `sum_fib_array`.

The "`type fibtree`" declaration at the top of the program is a declaration of a new algebraic type, which is a tagged union and struct. The `sumtree` function uses pattern matching to conditionally branch on the tag of the algebraic type. This gets desugared by the Id front-end into code which looks more like Figure 4.4. Figure 4.4 has more explicit information about the sequential ordering of the procedure calls and data structure reads and writes – Figure 4.4 is not a legal Id source program because it is not type correct, but it is a good representation of what the compiler sees when it performs the parallelization analysis.

We will use the version of the program in Figure 4.4 to explain most of the stages of the parallelization shown in Figure 4.2.

```
type fibtree = fibleaf I | fibnode fibtree fibtree;

def make_fibtree n =
  if (n < 2) then
    fibleaf n
  else
    fibnode (make_fibtree (n-1)) (make_fibtree (n-2));

def sumtree (fibleaf value) = value
  | sumtree (fibnode l r) = (sumtree l) + (sumtree r);

def fill_fib_array_slot a i =
  { a[i] = make_fibtree i; };

def evenp n =
  (logand n 1) == 1;

def fill_fib_array_even a n =
  { for i <- 1 to n do
      if (evenp i) then
        fill_fib_array_slot a i;};

def fill_fib_array_odd a n =
  { for i <- 1 to n do
      if (not (evenp i)) then
        fill_fib_array_slot a i;};

def sum_fib_array a n =
  { sum = 0;
    in
      { for i <- 1 to n do
         next sum = sum + sumtree a[i];
        finally sum}};

def main n =
  { a = i_array (1,n);
    _ = fill_fib_array_odd a n;
    _ = fill_fib_array_even a n;
    sum = sum_fib_array a n;
    in
      sum};
```

Figure 4.3: Example program to calculate $\sum_1^n fib(n)$, where $fib(n)$ is the nth fibonacci number.

4.3.2 Stage 1: find local mod and ref sets

Within each procedure, we first determine the mod and ref sets for each program graph node. This is a straightforward data flow problem – for each node which potentially accesses an object, we follow arcs until we find the definition point of the object. Nodes which may potentially access objects include heap reads and writes, and procedure calls. At conditional or loop boundaries, we conservatively take the union of the sets from the two paths, because we are doing a *flow insensitive* analysis. We also conservatively assume that each static occurrence of an object definition may refer to the same object dynamically, which is not always the case – for example, and object allocated within a loop body has only one static occurrence within the intermediate form, but dynamically has one for each loop iteration.

An object definition point is one of the following nodes:

- An object allocation

```
 1: def make_fibtree n =
 2:   if (n < 2) then
 3:     { leaf = i_structure (0,1);
 4:       leaf[0] = leaf_tag;
 5:       leaf[1] = n;
 6:       in
 7:         leaf}
 8:   else
 9:     { node = i_structure (0,2);
10:       left_tree = (make_fibtree (n-1));
11:       right_tree = (make_fibtree (n-2));
12:       node[0] = node_tag;
13:       node[1] = left_tree;
14:       node[2] = right_tree;
15:       in
16:         node};
17:
18: def sumtree fibtree =
19:   if (fibtree[0] == leaf_tag) then
20:     fibtree[1];
21:   else
22:     { left_tree =  fibtree[1];
23:       left_sum = sumtree left_tree;
24:       right_tree = fibtree[2];
25:       right_sum = sumtree right_tree;
26:       sum = left_sum + right_sum;
27:       in
28:          sum};
29:
30: def fill_fib_array_slot a1 i =
31:   { fibi = make_fibtree i;
32:     a1[i] = fibi; };
33:
34: def evenp n =
35:   { lowbit = (logand n 1);
36:     pred = (lowbit == 1);
37:     in
38:       pred };
39:
40: def fill_fib_array_even a2 n1 =
41:   { for i1 <- 1 to n1 do
42:      if (evenp i1) then
43:        fill_fib_array_slot a2 i1;};
44:
45: def fill_fib_array_odd a3 n2 =
46:   { for i2 <- 1 to n2 do
47:      if (not (evenp i2)) then
48:        fill_fib_array_slot a3 i2;};
49:
50: def sum_fib_array a4 n3 =
51:   { sum = 0;
52:     in
53:       {for i3 <- 1 to n3 do
54:          next sum = sum + sumtree a4[i3];
55:        finally sum};
56:
57: def main n4 =
58:   { a5 = i_array (1,n4);
59:     _ = fill_fib_array_odd a5 n4;
60:     _ = fill_fib_array_even a5 n4;
61:     sum = sum_fib_array a5 n4;
62:     in
63:       sum};
```

Figure 4.4: Desugaring of `make_fibtree` and `sumtree` providing a unique variable name to each object.

Procedure	Reference (line no.)	Variable	Object Source (line no.)
make_fibtree	write (4)	leaf	object alloc (3)
	write (5)	leaf	object alloc (3)
	write (12)	node	object alloc (9)
	write (13)	node	object alloc (9)
	write (14)	node	object alloc (9)
sumtree	read (19)	fibtree	argument (18)
	read (20)	fibtree	argument (18)
	read (22)	fibtree	argument (18)
	call (23)	left_tree	object reference (22)
	read (24)	fibtree	argument (18)
	call (25)	right_tree	object reference (24)
fill_fib_array_slot	write (32)	a1	procedure argument (30)
fill_fib_array_even	call (43)	a2	procedure argument (40)
fill_fib_array_odd	call (48)	a3	procedure argument (45)
sum_fib_array	call (53)	a4	procedure argument (50)
main	call (59)	a5	object alloc (58)
	call (60)	a5	object alloc (58)
	call (61)	a5	object alloc (58)

Figure 4.5: After Stage 1, all of the reads, writes and procedure calls are labelled with the object or objects which they reference. Each object is labelled with its source.

- One of the current procedure's arguments
- A return value from a procedure call
- A reference from an object (e.g. the cdr of a cons cell). This could be sharpened with type information, but we do not bother to do so.
- A global constant

References traced to new object allocations, procedure arguments, or procedure return values are included in the mod and ref sets. These mod and ref sets may be later augmented with alias information – for example, procedure arguments may be aliased to each other if the current procedure is called with two arguments being the same object, and return values from procedure calls could potentially be aliased to any other object ($\top$).

References to objects which are traced back to references to other objects are assigned to $\top$, meaning that they are potentially aliased to any other object. This happens for nested objects, such as trees, lists or tuples of objects. In some cases, references to $\top$ can severely limit parallelism we can detect – we will give some examples of this in Section 4.4. However, single-assignment semantics allow us to parallelize some codes even when there are references to $\top$ because we are only concerned with real data dependences, not output- or anti- dependences. Additionally, $\top$ is sometimes too conservative an

estimate, and we also show some simple ways to back off from this conservative estimate to obtain more precise information about objects which are read and written.

We have chosen to ignore global constant objects in our parallelization – we evaluate all constants in a sequential initialization phase, and assume that they are immutable thereafter, so there are no dependencies arising from references through global constant objects. Our parallelization can be modified to handle data dependences through global constant objects, but for the applications we have studied, assuming global constants are initialized sequentially and immutable thereafter does not seriously affect either performance or expressiveness.

After this first stage is completed, each read, write or procedure call in the procedure is labelled with the object or objects which it references. Figure 4.5 shows this analysis for the fibtree program. Note that each reference in our example references a single object, but that in general, the reference may be to a set of objects. This occurs for references of objects which are the result of conditionals. For instance, in the following procedure, the read in line 4 refers to both of the objects `a` and `b`, which are both arguments of the procedure foo.

```
def foo n a b =
  { c = if (n > 0) then a else b;
    in
      c[0] };
```

While we compute the objects which are referenced by each read, write and procedure call, we also calculate the direct mod and ref sets of each procedure as a whole. The direct mod and ref sets are the set of objects which are externally visible to the procedure which may be read or written within the procedure. Note that objects which are externally visible do not include objects which are allocated within the current procedure, because those objects are only visible externally once the procedure has executed, and by that point, all of the reads and writes are complete. As we note above, we also ignore reads and writes to global constants, so the only references which are externally visible are to $\top$, one of the procedure arguments, or to a return value from another procedure call.

Figure 4.6 shows the direct mod and ref sets of the fibtree example. The only *direct* write in the program is to the `a1` argument of fill_fib_array_slot; the only *direct* read in the program is to the fibtree argument in `sumtree`. There are some indirect reads and writes – for example, both fill_fib_array_even and fill_fib_array_odd perform writes to the array passed to them, but they do it by doing a procedure call to fill_fib_array_slot. In order to find the indirect reads and writes, we must perform some interprocedural analysis, as we do in Stage 2.

Procedure	direct mod set	direct ref set
make_fibtree	{ }	{ }
sumtree	{ }	{ fibtree }
fill_fib_array_slot	{ a1 }	{ }
evenp	{ }	{ }
fill_fib_array_even	{ }	{ }
fill_fib_array_odd	{ }	{ }
sum_fib_array	{ }	{ }
main	{ }	{ }

Figure 4.6: The direct mod and ref sets of each procedure are only to objects which are $\top$, procedure arguments, or return values of calls – references to objects allocated within the procedure are not visible externally. In our fibtree example, the direct mod and ref sets are all to procedure arguments.

Propagate mod/ref information interprocedurally

1. Create a binding multi-graph, representing interactions between formal parameters
2. Eliminate cycles by finding strongly connected components (SCC's), and substituting each SCC with a supernode
3. Propagate mod-ref information up through the supernode DAG, bottom up
4. Propagate information back from supernodes to nodes

Figure 4.7: Procedure for propagating mod/ref information interprocedurally.

4.3.3 Stage 2: find interprocedural mod and ref sets

To propagate mod/ref information interprocedurally, we use Cooper and Kennedy's algorithm [19], which we found to be simpler than Banning's original algorithm [15], or the algorithm in Aho, Sethi and Ullman [3].

Cooper and Kennedy's algorithm uses a data structure called the *binding multi-graph*, instead of a normal call graph. The binding multi-graph, $\beta = (N_\beta, E_\beta)$, represents interactions between procedural formal parameters (arguments). Each node in the multi-graph represents a formal parameter of a procedure, and each edge in the multi-graph connects a formal parameter which is itself used as an argument in a procedure call.

Let the nth argument (formal parameter) to procedure p be called fp_p^n. If, fp_p^3 is used in a procedure call as the second argument to procedure q, fp_q^2, then there is an edge in the binding multigraph between node fp_p^3 and node fp_q^2. This structure is a multi-graph because there may be multiple edges between

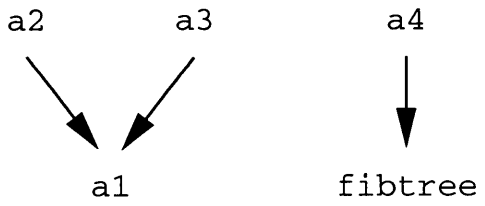

Figure 4.8: Binding multi-graph for fibtree example. We know from Stage 1 that a1 is written, so we determine that a2 and a3 are also written. We know from Stage 1 that fibtree is read, so we determine that a4 is also read.

pairs of nodes in the graph. Each node in the multi-graph has two flags which indicate whether that node is referenced or modified – if we know that the procedure associated with that node directly references or modifies that formal parameter (which we calculated in Stage 1) then the appropriate flag is set to true, otherwise, it is initialized to false. If the procedure directly modifies or references $\top$, then we set the appropriate flags to $\top$.

The binding multi-graph for the fibtree example is shown in Figure 4.8. Each node is a procedure argument which may be a data structure, and each edge is the potential use of an argument passed to another argument. Note that make_fibtree has only a scalar (integer) argument, so its argument is not in the graph. The recursive function sumtree has a data structure argument (fibtree), but that argument is not passed to itself in the recursive call, so there is no cycle in the binding multi-graph. Finally, the data structure a5 (line 58) is passed to fill_fib_array_odd's argument a2, and fill_fib_array_even's argument a3, but a5 itself is not an argument, so it is not in the binding multi-graph.

The binding multi-graph may have cycles if the program contains recursive calls which bind arguments to call-sites in a cyclic fashion. To eliminate any cycles, we calculate the strongly connected components (SCC) of the multi-graph, and substitute each SCC with a supernode, while maintaining the edges between SCC's for each supernode. Any cycle in the original multi-graph indicates that all the nodes in the cycle should have the same mod/ref sets, so the mod/ref sets of each supernode is representative of the all of the nodes in the cycle. Each supernode should be initialized mod/ref flags with the "logical or" of all of the nodes in the SCC, while maintaining and direct reads and writes to $\top$. Once the supernode graph is constructed from the SCC, the graph is a directed acyclic graph (DAG).

Performing a bottom-up traversal of the graph from its leaves to its roots, we set the mod/ref flags for each node to be the "logical or" of the original flag setting and the settings of its descendants. The logical or is used because if any of the nodes children read the structure, then we want to mark the node

Resolving Procedure Argument Aliases

1. Create a pair binding multi-graph, where each node represents a potentially aliased procedure argument pair
2. Initialize all nodes to *nil*, except those which we determine are directly aliased within the procedure
3. Push alias pairs through the multi-graph

Figure 4.9: Procedure for resolving aliases from procedure arguments.

itself as being read, if it isn't already marked as being read from Stage 1. Likewise, if any of the node's children are marked as being written, then we mark the node as being written, , if it isn't already marked as being written from Stage 1.

In the fibtree example, the leaf nodes `a1` and `fibtree` are written and read, respectively. When we percolate this up the graph, we mark `a2` and `a3` as written, and `a4` as read. In the fibtree example, each node has at most one child, but in general, a procedure argument may be passed to several different procedures, each of which may read or write it.

Finally, if we have supernodes due to cycles in the original binding multi-graph, we propagate this information back to the component nodes of each supernode. We keep this information for Stage 5, where we will combine it with interprocedural alias information of Stage 3 and Stage 4, and relate it back to the local context of each procedure.

Readers interested in more details about the correctness and construction of this algorithm are referred to [19].

4.3.4 Stage 3: find procedure argument aliases

Although Id does not have arbitrary pointer aliasing of the type found in C, aliasing can still occur through two mechanisms: procedure argument passing, and return value. Procedure argument aliasing can occur when the same object is passed to two different arguments of the same procedure. Within that procedure, one must assume that all of the arguments may potentially be the same object unless interprocedural analysis proves that there are no call sites in the entire program which call the procedure with multiple identical objects.

We use Cooper and Kennedy's [20] refinement of Banning's original interprocedural argument alias

analysis [15]. We give a sketch in Figure 4.9. The analysis uses a interprocedural data structure which is different from the binding multi-graph, called the *pair binding multi-graph*, $\pi = (N_\pi, E_\pi)$. In this graph, each node is a pair of formal parameters to the same procedure representing a possible mapping of an alias pair in one procedure to an alias pair of another procedure. For example, if the first and third formal parameters of procedure p (fp_p^1,fp_p^3) are passed as the fourth and second arguments of procedure q (fp_q^4,fp_q^2), then there is an edge between the two nodes – note that each node in the pair binding multi-graph represents a *pair* of formal parameters.

Each node in the multi-graph has a flag determining whether the formal parameter pair is aliased or not. This flag is initialized from procedure local information to be false unless we know from local information that the object sets associated with two formal parameters intersect, causing a potential alias within the callee.

We then push the flags for each aliased node to its children and their descendants. At this point, all potential aliased formal parameter pairs have been marked, and we keep this information for Stage 5, where we related it back to the local context for each procedure.

None of the functions in the fibtree example have procedures which have more than one argument which is a data structure, so there are no opportunities for procedure argument aliasing in that program. However, consider the following three procedures:

```
def write_a a =
  { a[1] = 1; };

def read_b b =
  b[1];

def call_two_functions a1 b1 =
  { _ = write_a a1;
    n = read_b b1;
    in
      n };
```

Can the two procedure calls in `call_two_functions` be executed in parallel? It depends upon whether arguments `a1` and `b1` are the same or not. If the rest of the program was the following main function, then the two functions cannot be called in parallel, because the write in the first function causes a dependency to the read in the second function.

```
def main _ =
  { a2 = i_array (1,1);
    n1 = call_two_functions a2 a2;
    in
       n1 };
```

If, however, the rest of the program was the following main function, then the two functions in `call_two_functions` can be called in parallel, because the write in the first function is to a different data structure than the read in the second function.

```
def main _ =
  { a3 = i_array (1,1);
    b3 = i_array (1,1);
    b3[1] = 2;
    n2 = call_two_functions a3 b3;
    in
       n2 };
```

Aliasing through arguments can happen often when $\top$ is passed as one of the arguments. If there are any other arguments which are potentially data structures, we must assume that those arguments may be aliased to $\top$.

This approach to alias analysis is conservative, because if any call-site happens to cause a potential alias in a called procedure, then we assume that that alias can always occur. It is also possible to have the compiler clone a procedure such that call-sites which use aliased arguments call one version, while call-sites which don't use aliased arguments call a separate, more parallel version. Cloning to provide specialized, optimized or parallelized versions of procedures is discussed in Hall's PhD thesis [39]. Our analysis does not perform cloning.

4.3.5 Stage 4: find return value aliases

For Id, we also have to find the aliases which may be due to procedure return values – the value that is returned by a procedure may be either aliased through one of two ways:

- It may be aliased to one of the arguments that is passed to the procedure in the case that the procedure returns a value which was passed to it – for example, the identity function returns a value which is aliased to its formal parameter.
- It may be aliased to $\top$, in the case that it is returning a referenced value which is an object.

Our algorithm for resolving aliasing through return values is shown in Figure 4.10. Again, this procedure is extremely simple and conservative, but sufficient.

We make the simplifying assumption that all return values are either *new*, i.e. objects allocated within the context of the procedure call, or else $\top$. Although there are some times when this procedure would label a return-value as $\top$, when a more precise approach would determine it was aliased to a

Resolving Return Value Aliases

1. Create a "return value" node for each procedure which may return a data structure.
2. Initialize each rv-node to *nil*, unless you know it is a new node (allocated in the current procedure) or you know it is $\top$.
3. Wire up rv-nodes to each other in a doubly-linked fashion if the return value of one procedure is the return value of another.
4. Propagate $\top$ and *new* to your children, and remove the arc to the child. If $\top$ is propagated, the child also becomes of type $\top$. If *new* is propagated, then the child becomes of type *new* if there are no more parent arcs. Children whose types are resolved to be either $\top$ or *new* propagate their types to their children.
5. When there are no more nodes to work on, label any remaining *nil* nodes to be of type $\top$.

Figure 4.10: Procedure for resolving aliases from return values.

procedure argument or even *new*, it does not seem to impede our parallelization for most of the programs we are testing.

We have found that most objects which are returned from a procedure call are *new*, perhaps because Id programmers tend to write in a functional programming style. Of course, procedures can also return scalar or null values, which do not affect alias analysis. Fairly rarely, a procedures returns one of the values it was passed in an argument (e.g. the identity function) which is what makes our simplifying assumption reasonable.

The only procedure in the fibtree example which returns a data structure is the `make_fibtree` function, which returns either `leaf` or `node`, both of which are *new*, and therefore we mark the return value of `make_fibtree` as returning `new`.

We keep the return value alias information for the next stage, where it is incorporated with information gathered from Stage 2 and 3 to give us a procedure local view of mod-ref sets.

4.3.6 Stage 5: incorporate interprocedural mod/ref and alias information locally

Once interprocedural alias and mod-ref information is gathered, we update the local mod and ref sets. This is necessary because we now know that some of the objects passed into procedure calls may be modified or referenced, or that a procedure call may modify or reference an arbitrary object. From the

Incorporating interprocedural alias and mod/ref information locally

1. Recursively update mod/ref information through conditional and loop encapsulators with new mod/ref information for each call-site
2. Propagate new reads and write to $\top$ back to callers using the binding multigraph
3. For each procedure, create an "alias map", mapping local object names to the set of objects that they may possibly be aliased to.
4. Update each node's mod/ref sets by augmenting the sets with information from the procedure's alias map.

Figure 4.11: Incorporating interprocedural information locally.

Augmenting local mod/ref sets with interprocedural information			
Procedure	**call-site (line no.)**	**Objects Reads**	**Objects Written**
make_fibtree	make_fibtree (10)	*none*	*none*
	make_fibtree (11)	*none*	*none*
sumtree	sumtree (23)	$\top$	*none*
	sumtree (25)	$\top$	*none*
fill_fib_array_slot	make_fibtree (31)	*none*	*none*
fill_fib_array_even	fill_fib_array_slot (43)	*none*	a2
fill_fib_array_odd	fill_fib_array_slot (48)	*none*	a3
sum_fib_array	sumtree (54)	$\top$	*none*
main	fill_fib_array_odd (59)	*none*	a5
	fill_fib_array_even (60)	*none*	a5
	sum_fib_array (61)	$\top$, a5	*none*

Figure 4.12: In Stage 5, we augment each of the calls with the objects which are potentially read and written, and propagate this information through conditionals and loops, and back through the call tree, if necessary.

interprocedural aliasing information, we also may discover that a store or fetch to one object may also reference another object.

As proven by Banning [15], this can be performed in two steps – first by incorporating mod/ref information due to procedure calls, and second by augmenting local mod/ref information with alias information. The first step is done by recursively updating mod/ref sets through conditional and loop encapsulators for each call-site using the information gathered in Stage 2. During the recursive descent, we also update the mod/ref sets of each conditional and loop with the union of the mod/ref sets of the nodes internal to each conditional and loop.

In essence, the first step replaces call of the individual procedure call references in Figure 4.5 with

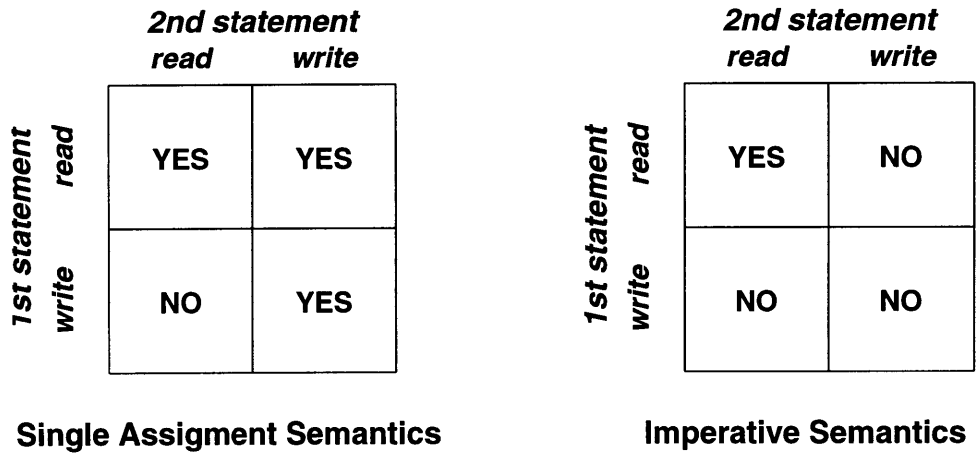

Figure 4.13: Dependence matrices which determine when nodes can execute in parallel. Single-assignment semantics allow more nodes to be executed in parallel than imperative semantics.

the reads and writes which are performed when those procedure calls are performed.

If there are any new references to $\top$ caused by this step, this information is propagated back through the call tree.

Once the mod/ref information is updated to include the effect of procedure calls, we augment the mod/ref sets with alias information. First, we create an "alias map" for each procedure, using the alias information we calculated in Stages 3 and 4. Then, for each program graph node in the procedure, we supplement the mod and ref sets with the objects which may be aliased to the objects in the current mod and ref sets.

4.3.7 Stage 6: add data structure dependences

Once we determine what objects are modified and referenced by each conditional, loop, store, fetch, and procedure call, we recursively consider each encapsulator surface. For each potentially side-effecting node in a surface, we compare it with every other node within the same surface and apply one of the the matrices shown in Figure 4.13.

When we assume single-assignment semantics, we consider the mod set of the first node and the ref set of the second node – if they intersect, then we must insert a dependence arc between the two nodes to respect the potential dependence. When we assume imperative semantics, the only case in which we do not insert a dependence between two nodes is when the only intersections between the mod/ref sets

Adding dependence arcs				
Procedure	**First ref (line no.)**	**Second ref**	**Read after Write?**	**1st mod set ∩ 2nd ref set**
make_fibtree	write leaf (4)	write leaf (4)	no	
	write node (12)	write node (13)	no	
	write node (12)	write node (14)	no	
	write node (13)	write node (14)	no	
sumtree	read fibtree (22)	read ⊤ (23)	no	
	read fibtree (22)	read fibtree (24)	no	
	read fibtree (22)	read ⊤ (25)	no	
	read ⊤ (23)	read fibtree (24)	no	
	read ⊤ (23)	read ⊤ (25)	no	
	read fibtree (24)	read ⊤ (25)	no	
fill_fib_array_slot	*none*	*none*		
evenp	*none*	*none*		
fill_fib_array_even	*none*	*none*		
fill_fib_array_odd	*none*	*none*		
sum_fib_array	*none*	*none*		
main	write a5 (59)	write a5 (60)	no	
	write a5 (59)	read ⊤ (61)	yes	{ ⊤ }
	write a5 (60)	read ⊤ (61)	yes	{ ⊤ }

Figure 4.14: In Stage 6, we decide whether to add dependencies between references, using the dependence matrix for single-assignment semantics.

of the two nodes is between the ref sets.

The two matrices in Figure 4.13 are different because with single-assignment semantics, only true data dependencies must be respected. There are no output dependences or anti-dependences, which are represented by the additional two "NO" entries in the imperative matrix.

For our fibtree example, the dependence arcs are added only between the call to `fill_fib_array_odd` and the call to `sum_fib_array`, and the the call to `fill_fib_array_even` and the call to `sum_fib_array`, as shown in Figure 4.14. For the functions `fill_fib_array_slot`, `fill_fib_array_even`, `fill_fib_array_odd`, and `sum_fib_array`, there is only one static reference in each function, so no additional dependence arcs need to be added.

For `make_fibtree`, there are only write after write pairs, which would be considered output dependences in an imperative language, but which are not output dependences in a single-assignment semantics. Note that we only compare references which occur within the same control block – references in one arm of the conditional are not compared for potential dependences with references in the other arm, because within a particular function invocation, they cannot be simultaneously active.

Furthermore, we are only comparing direct structure writes in `make_fibtree`, because our in-

terprocedural analysis determined that the two recursive procedure calls do not cause any visible references. We need to compare direct reads and writes for potential dependences, because there is no sequentialization within the dataflow graph representation, except indirectly in the node numbering.

For `sumtree`, there are only read after read pairs, which do not cause dependences, even though there are references to $\top$. Here, we compare direct reads and procedure calls.

Only within the `main` procedure do we need to insert dependences, and even the two calls to `fill_fib_array_odd` and `fill_fib_array_even` can proceed in parallel, because the only visible side-effects which `main` can see are writes. Because of the single-assignment semantics, the writes may occur in parallel to the same data structure.

However, the writes caused by `fill_fib_array_odd` and `fill_fib_array_even` do occur before the reads caused by `sum_fib_array`, so we need to check for those reference pairs whether the mod sets of the first nodes intersect with the ref set of `sum_fib_array`. In fact, because `sum_fib_array` references $\top$, no write may execute in parallel with it, because $\top$ intersects with any non-empty ref set. Because of this, we insert a two dependence arcs in the `main` procedure, one between the call to `fill_fib_array_odd` and `sum_fib_array` and one between `fill_fib_array_even` and `sum_fib_array`.

Once this phase has completed, the code generator can generate multithreaded code which takes advantage of DAG-style parallelism – two procedures which do not have a data dependency between them can execute in parallel. However, have not yet discovered loop-level parallelism, although the analysis we have performed thus far can be used to trivially parallelize loops, as we explain in the next section.

4.3.8 Stage 7: characterizing loop induction variables

As we noted in Section 4.2, there is no fundamental difference between discovering DAG parallelism and loop parallelism. Our analysis attempts to parallelize loops which have no loop-carried dependencies – that is, those loops for which we can execute every iteration in parallel, without some run-time synchronization between concurrent iterations.

We initially check a loop for its suitability for parallelization by analyzing each induction variable, and determining whether it is a *constant-incrementing* variable, or a *reduction variable*. A constant-incrementing variable is a variable which increments by a constant amount on each iteration – this

Procedure and line no. of loop	Induction variable	Type of induction variable
fill_fib_array_even (41)	i1	constant-incrementing (+1, initial: 1, final: n1)
fill_fib_array_odd (46)	i2	constant-incrementing (+1, initial: 1, final: n2)
sum_fib_array (53)	i3	constant-incrementing (+1, initial: 1, final: n3)
	sum	reduction (integer +)

Figure 4.15: In Stage 7, we characterize all loop induction variables as being either *constant-incrementing*, *reduction*, or *other*. Loops which have any induction variables which are classified as *other* we do not parallelize

constant may be a literal or a loop constant. Constant-incrementing variables also require initial and final values which are either literals or loop constant.

A reduction variable is a variable which computes a reduction using an associative and commutative operator, where the running value is not used within the loop – the most common example of a reduction variable is a sum.

We characterize loop induction variables with a simple pattern matching approach, augmented by some normalizing transformations. We do not attempt to parallelize loops which contain induction variables that do not fall into one of these two categories.

In the fibtree example, there are three loops, one each in `fill_fib_array_even`, `fill_fib_array_odd`, and `sum_fib_array`. We show all of their induction variables and the characterization of each variable in Figure 4.15.

`fill_fib_array_even` and `fill_fib_array_odd` only have the main induction variable, which increments by 1 on each iteration. The initial value for both loops is a literal, 1, and the final value is a loop constant, `n1` and `n2`, respectively. Because we know the initial, final and increment value of the induction variables for these two loops, there are no loop carried dependencies caused by the induction variables. There may be some loop carried dependencies caused by references to data structures, which we will check in Stage 8.

For `sum_fib_array`, there is the main induction variable, `i3`, and one more induction variable, `sum`. The `i3` induction variable is a constant-incrementing variable like the previous two. The `sum` induction variable is a reduction – it computes the integer sum of the return values from the calls to `sumtree`, and integer addition is both commutative and associative. Furthermore, the intermediate values of `sum` are not used within the loop body. Because `sum` is a reduction variable, we can parallelize its computation such that the loop iterations can go on in parallel, while the reduction (the sum) can be

computed in a tree-like fashion, instead of iteratively.

Note that although for fibsum, the constant-incrementing induction variables have an initial value and increment value which are literals, those could also be loop variables. For example, suppose that we decide to replace `fill_fib_array_even` and `fill_fib_array_odd` with a single procedure which takes the initial and increment values as procedure arguments, as follows:

```
def fill_fib_array_from_by a n start inc =
  { for i <- start to n by inc do
      fill_fib_array_slot a i;};
```

We can then pass in the arguments `start` and `inc` as 0 and 2 to fill in the even elements, and 1 and 2 to fill in the odd elements. The main induction variable is still constant-incrementing, because `start` and `inc` are loop constants.

For reduction induction variables, it is important that the running value is not used within the loop body, because that makes it difficult to parallelize the computation of that variable. For instance, suppose that the procedure `sum_fib_array` was modified to fill in the running sum $\sum_{i=1}^{n-1} fib(i)$ into an array `b` into the nth slot of `b`, as in the following procedure.

```
def running_sum_fib_array a b n =
  { sum = 0;
    in
      { for i <- 1 to n do
          b[i] = sum;
          next sum = sum + sumtree a[i];
          }};
```

In this case, although the induction variable `sum` is being computed with an associative and commutative operator, because the running value is used in the loop body, we choose not to parallelize this loop. This induction variable would be classified as *other*.

4.3.9 Stage 8: marking parallel loops

Once we determine which loops have induction variables which are all either *constant-incrementing* or *reduction* variables, we then check the loop's mod and ref sets. Only loops which have disjoint mod and ref sets are marked to be parallelized. If a loop may reference and modify the same object, then we make the conservative assumption that the loop has some loop-carried dependences.

Figure 4.16 shows how we mark the loops in the fibtree example. As we saw in Stage 7, none of the loops have any loop carried dependences due to the induction variables. When we examine the mod

Procedure and line no. of loop	Induct. var. dependence?	mod set	ref set	Parallelizable?
fill_fib_array_even (41)	no	{ a2 }	{ }	yes
fill_fib_array_odd (46)	no	{ a3 }	{ }	yes
sum_fib_array (53)	no	{ }	{ ⊤ }	yes

Figure 4.16: In Stage 8, we first check if there are any loop carried dependences due to induction variables, and then we check whether there is any intersection between the mod and ref sets of the loop. If there are no loop carried dependences in the induction variables and the intersection of the mod and ref sets is the empty set, then we can parallelize the loop

and ref sets of each loop, we see that there is no intersection for each of the three loops, so they are all parallelizable.

In the fibtree program, all of the loops either had mod sets or ref sets which were empty. In fact, many loop have non-empty mod and ref sets, but are still parallelizable because their intersection is empty. A simple example is the following function, which copies the contents of array `a` into array `b`:

```
def copy_array a b n =
  { for i <- 1 to n do
      b[i] = a[i];};
```

The mod set of the loop is { `b` } and the ref set is { `a` }, but the two sets don't intersect, so the loop is parallelizable. Note that the analysis never looked at the loop indices. We could just as easily parallelize the following loop, without considering what `any_function` does.

```
def permute_array a b n =
  { for i <- 1 to n do
      b[(any_function i)] = a[i];};
```

We depend on the programmer to adhere to the single-assignment semantics, ensuring that no element of `b` gets written more than once.

Note that even if a loop references and modifies the same object, the loop may be parallelized using traditional loop dependence analyses to determine whether the references to the objects incur loop-carried dependences, or whether those dependences could be eliminated with some loop transformations. For example, consider the following loop which copies one row of a matrix into the following row:

```
def copy_row a n j =
  { for i <- 1 to n do
      a[i,j+1] = a[i,j];};
```

Our parallelization will see that the mod and ref sets of the loop overlap, so we conservatively assume there is some loop carried data dependence. In fact, the loop can be parallelized using traditional Fortran compiler index analysis, which would determine that the read and write indices into the matrix never overlap. Other loops which benefit from index analysis include those in wavefront computations, and LU or cholesky decomposition.

4.4 Limitations and improvements to the parallelization

In this section, we describe some of the limitations of our parallelization approach, as described so far, and some of the ways we address the limitations.

4.4.1 False dependencies due to $\top$

When we call a procedure within a loop, that procedure may reference $\top$, which causes the loop itself to have $\top$ within its reference set. If the same loop is attempting to store values into an object allocated within the procedure, then our simple approach will assume that the store to the local object and the reference to $\top$ intersect.

However, $\top$ is just an approximation – it cannot really be aliased to every object in the system. In particular, if we trace the lifetime of the objects allocated within the current procedure, we can determine that those objects cannot be stored via a function call, if those objects do not escape the local procedure context. The two main methods by which an object may escape are by being stored into a data structure which is passed to a procedure, or else being passed to a procedure directly.

We trace all created objects to determine if they escape, and if they do not escape, then we can determine that they cannot be referenced by a procedure call.

4.4.2 False dependencies due conservative object labelling

Our local object labelling algorithm (Stage 1) labels each textual occurrence of an object definition, but does not distinguish between occurrences which are textually identical, but not the same object. For instance, an object definition which occurs within a loop will be labelled the same, regardless of which iteration of the loop it is allocated in.

This could be improved by labelling object definitions differently for different iterations. A simple, conservative approach would be to have two labels for each object – one of which corresponded to the current iteration, and one which corresponded to any other iteration.

4.4.3 Room for improvement in index analysis

As we discussed earlier, our analysis does not detect some parallel loops which might be detected using traditional parallelizing loop optimizations and transformations. However, our compiler framework does not preclude using those techniques, and in fact, because those techniques are typically more expensive, our simple analysis could be viewed as a pre-test to screen out the majority of loops which do not require a more sophisticated analysis [51] [65] [14] [5] [31].

This approach has also been taken in the parallelizing loop analysis by Maydan, et.al. [51] and Goff, et.al. [31] – a tree of dependence tests is performed, where the fastest and simplest tests are performed first, while slower and more complicated tests are performed only if necessary, with the order and type of test performed determined by the results of the initial tests.

4.5 Parallelization results

In this section, we show some results which indicate how well the compiler was able to detect parallelism for a set of Id-S program. We use the measurement of *idealized parallelism* to isolate results of the compiler parallelization from machine-specific issues such as cache organization, memory bandwidth limitations, as well as the effect of run-time scheduling policies. Real speedup numbers will be shown in a later chapter.

Idealized parallelism gives us an upper bound on the parallel speedup we can expect on a real machine (modulo superlinear effects due to increased cache capacity in a parallel machine) by assuming speedup with an infinite number of processors, perfect scheduling, and no communication or synchronization costs. We can only measure idealized parallelism for a certain program input – not surprisingly, the same program can have a different idealized parallelism for a different program input.

We use idealized parallelism to measure the effectiveness of our parallelization rather than a common alternative measurement called *parallel coverage* [6] – parallel coverage gives the percentage of time a program spends in parallelized sections of code. Parallel coverage can give a lower bound on

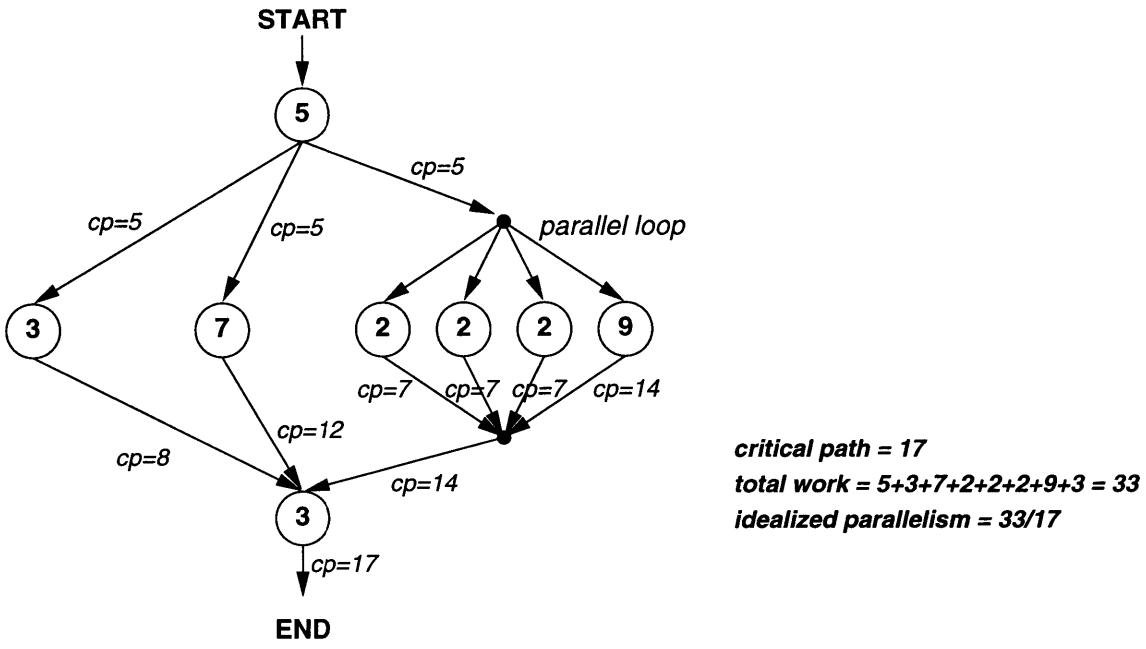

Figure 4.17: Example of idealized parallelism calculation. The critical path and total work is calculated, and the idealized parallelism is the ratio of the total work to critical path.

the sequential section of a program run, but does not relate to real parallel speedup. In the case where the entire program fits in one outer loop which is parallelized, but which only has one iteration, there is 100% parallel coverage, but no idealized speedup. Parallel coverage also masks the effects of parallelization which is not at the outer loop, and unbalanced work within loop iterations, both of which have impact on real speedup.

4.5.1 Methodology for measuring idealized parallelism

We use the same strategy as the Cilk system [16] to measure idealized parallelism. We instrument the parallelized code to count the *total work* performed, and the *critical path* through the code. The total work is equal to the time the code would take to run on 1 processor, and the the critical path is the time the code would take to run on an infinite number of processors, where all work is scheduled perfectly. *The idealized parallelism is the ratio of the total work to the critical path.* A simple example of idealized parallelism is shown in Figure 4.17, with each thread represented by a node in the execution DAG.

The total work is measured simply by inserting a global counter increment for every operation the compiler generates. At the end of program execution, the total work is the value of the global counter.

In Figure 4.17 is the sum of all the work in all the threads, or 33 units.

The critical path is measured by "timestamping" threads – when two threads are forked off, we timestamp both threads, and when two threads join, we take the maximum timestamp of the two threads. For parallelized loops, we assume all iterations begin at the same time, and the finish time for a parallel loop is the start time plus the time taken by the longest running iteration. In Figure 4.17, the critical path of the program execution is shown to be also the height of the execution DAG, or 17 units. The idealized parallelism of the program execution in Figure 4.17 is therefore 33/17.

The idealized parallelism for a program execution is affected by many factors, including:

1. The inherent parallelism in the program – some algorithms simply do not have very much parallelism.
2. The parallel execution model – we are only taking advantage of procedure and loop parallelism. Our idealized parallelism numbers do not include producer-consumer or instruction-level parallelism [12].
3. The effectiveness of the parallelization – the compiler may miss some opportunities for parallelizing loops or procedure calls.
4. The program input – the parallelism is likely to be less for a small problem size than a large one.

Our parallelization results measure the first three factors. We provide some commentary on the characteristics of the program, the sources of potential parallelism, and the effectiveness of the parallelization. We also separate out the effects of procedural (or DAG) parallelism from loop parallelism to gain more insight on the sources of parallelism and the effectiveness of the compiler. Our program inputs are somewhat arbitrary, and are set to be large enough to allow us to obtain good wall clock timings for Chapters 5 and 6.

Note that idealized parallelism is not the same as the eventual speedup we will see on a real parallel machine. The idealized parallelism shows how much of the inherent parallelism in the algorithm the compiler could discover. Some of this parallelism may be too fine-grained to take advantage of profitably on an SMP, even though we are avoiding the type of producer-consumer parallelism used in Id90. Furthermore, idealized parallelism does not measure the effects of the memory system, and takes an idealized approach to scheduling and work distribution overheads. That being said, idealized parallelism is useful in that it gives us an upper bound on the parallelism we can expect to exploit, and provides some insight into the interaction of DAG and loop parallelism.

Program	Arguments	DAG only	Loop only	DAG & Loop
eigen-jacobi	50	2.05	23.11	82.80
knapsack-dp	100 10000	1.00	1.00	1.00
mm-kes	500	68.45	192.97	18157.99
mm-kes	700	95.53	269.45	35532.30
mm	500	1.00	308979.23	308979.23
nas-multigrid	6 5	1.14	1529.22	2175.44
poly-mult	13000	1.00	13922.50	13922.50
precond-conj-grad	300	1.25	4476.74	5195.32
region-labelling	500	1.52	46001.40	66167.64
relax	1000 10	1.01	14574.52	43274.73
simple-kt	10 200	1.12	3.47	4.89
simple-new	10 200	1.93	24.25	24.37
simplex	70	1.05	1.17	1.19
warshall	300	1.00	20524.01	20541.52

Figure 4.18: Idealized parallelism for structured codes, showing the effects of DAG parallelism, loop parallelism, and both types of parallelism.

4.5.2 Parallelism in structured codes

We first discuss the parallelization results for structured codes, which are more amenable to Fortran-style analysis. Figure 4.18 shows the idealized parallelism for a set of structured codes. We show separate numbers for DAG and loop parallelism, as well as combined DAG and loop parallelism. These numbers were obtained by selectively turning off analysis phases in the compiler which discover loop and DAG parallelism, generating codes which only exploit one type of parallelism.

The parallelism numbers do not make a distinction between vector-style inner-loop parallelism and coarser-grained outer loop parallelism – that is, those numbers incorporate vector-style loop parallelism as well as coarse-grained outer-loop parallelism and DAG parallelism. In Chapter 5, we will revisit these parallelism numbers in the context of generating efficient parallel code, which will entail a tradeoff of fine-grained vector-style parallelism for efficiency.

Not surprisingly, most of the structured programs (except for mm-kes) exhibit mostly loop parallelism, and relatively little DAG parallelism. Because there is such a large amount of loop parallelism being exposed, even a little bit of DAG parallelism seems to have a large effect when the two are combined – for example, in the relax program, a miniscule amount of DAG parallelism seems to triple the ideal parallelism between loop and "DAG & loop". This effect is not important for us, because we are targeting small machines, where the additional parallelism is not necessary.

Here, we explain in more detail the parallelism results in Figure 4.18.

eigen-jacobi is a Jacobi eigensolver, which finds the eigenvalues of a symmetric matrix which has a width and height the size of the input argument. There is a moderate amount of loop parallelism, and a small amount of DAG parallelism. The combined parallelism is greater than the product of the individual parallelism numbers, a phenomenon we explain in greater detail in Section 4.6.

knapsack-dp solves an integer knapsack problem using a dynamic programming algorithm. The first argument is the number of items we have, and the second argument is the capacity of the knapsack. The program determines the set of items with the maximum value which weigh less than the capacity of the knapsack.

The inner loops of the program fill in a 2D array with wavefront dependencies. Our loop parallelization analysis cannot determine that the loops can be parallelized because we do not perform index analysis.

Index analysis might determine that the loops should be transformed so that the iteration space is traversed along diagonal elements of the 2D array, yielding a parallelizable inner loop – however, this inner loop parallelism is extremely fine-grained, and would probably only yield at most 100 idealized parallelism (because a diagonal of length 100 could be calculated in parallel at any time).

mm-kes is a blocked 4×4 double-precision matrix multiply which does a decomposition to 4×4 blocks recursively, then computes the values of the 4×4 block iteratively. The argument is the length of a side of the square matrices which are multiplied. mm-kes shows both DAG and loop parallelism.

mm is a standard triply-nested matrix multiply, which parallelizes well. The argument is the length of a side of the square matrices which are multiplied.

nas-multigrid is the 3D multigrid benchmark from the NAS suite [59], which parallelizes well. The first argument is the number of levels that the multigrid uses, and the second argument is the number of iterations of performed. The size of the grid at maximum resolution is $64 \times 64 \times 64$.

poly-mult performs a polynomial multiplication, where the polynomials are represented as arrays of coefficients. Poly-mult parallelizes well. The argument represents the degree of the input polynomial which is multiplied by itself.

precond-conj-grad is the core of an ocean model which solves a sparse, banded system linear of equations [74] which parallelizes well. The argument is the size of the square input matrices which describe the 2D pressure equation solved by a preconditioned conjugate gradient algorithm.

region-labelling is an image processing algorithm which labels "regions" of the same color with unique identifiers. The argument is the size of the square "image" which is labelled. Region-labelling parallelizes well.

relax is a simple 2D Gaussian relaxation problem which parallelizes well. The first argument is the size of the input matrix, and the second is the number of iterations.

simple-kt is a 2D hydrodynamics simulation – the first argument is the number of iterations, and the second is the size of a side of the square grid area.

Simple-kt which parallelizes poorly because it makes heavy use of tuples of state arrays to pass results between loop iterations. Our analysis loses track of arrays once they are stored and read from tuples, and assumes that those arrays could be aliased to $\top$, which makes parallelization rather poor.

simple-new is a re-write of simple-kt which does not use tuples to pass results, and consequently achieves better parallelization. Several procedures in simple-new still do not parallelize, leading to a critical path which is about 4% of the run-time.

simplex is an optimization algorithm for a linearly constrained system, which has an LUD decomposition at its core. The linear constraint is of the size of the argument.

The LUD decomposition cannot be parallelized because the inner loops read and write the same LUD array – Fortran-style index analysis might be able to analyze the data dependencies, and restructure the loops to allow them to execute in parallel.

Program	Arguments	DAG only	Loop only	DAG & Loop
btree	5000	63.99	1.00	63.99
fft-1d	65536	6.80	9.12	13939.13
fib	36	1254426.01	1.03	1254426.01
gamteb	40000	1.18	1.00	1.18
gamteb-new	40000	1.21	333.61	339.47
knapsack-bb	100 10000	2484.17	1.00	2484.17
mm-sparse	300	1.00	197.71	380.26
nas-cg	300	1.00	532.51	539.09
nqueens	12	15796.00	1.00	15796.00
paraffins	20	1.28	1.74	2.53
pic	64 40000 100	1.01	2498.57	3830.77
qs	10000	3.81	1.00	3.81
ray-tracer	500	1.06	7659.55	8272.00
speech-dtw	350	69.72	85.55	86.75
speech-proc	10240 30	9.54	13.57	16.66
speech-proc	2000000 40	8.17	13.19	15.90
tree	23	501995.42	1.00	501995.42

Figure 4.19: Idealized parallelism for unstructured codes.

warshall is Warshall's all-pairs shortest paths algorithm, where the argument is the number of cities. Warshall parallelizes well.

4.5.3 Parallelism in unstructured codes

Figure 4.19 shows the idealized parallelism numbers for some codes which are unstructured – these codes probably would not parallelize well using conventional Fortran parallelizing loop compiler technology for a variety of reasons. These codes make greater use of DAG style parallelism than the structured codes, but many still rely primarily on loop parallelism.

Even the codes which rely primarily on loop parallelism are not structured codes because the pattern of memory accesses within the loops are not amenable to traditional index analysis.

btree converts a list of integers into a btree structure, and shows moderate speedup. The argument is the size of the list. This code has some fundamental sequential sections which limit idealized speedup.

fft-1d uses a recursive divide-and-conquer 1D FFT algorithm with a base case of 4 elements. The argument is the size of the array on which the FFT is performed. The combine is implemented as a

single loop. Although DAG and loop idealized parallelism numbers are moderate in isolation, fft-1d shows good idealized parallelism when both are used. This phenomenon is explained in Section 4.6.2.

fib is a doubly recursive fibonacci program which speeds up well. The argument is the nth fibonacci number.

gamteb is a Monte Carlo photon transport simulation code which creates particles of random energy and velocity, and tracks their fate through a cylindrical geometry. The argument is the number of particles which are simulated.

The main loop of gamteb performs a vector-style sum of a tuple of integers on each iteration, which causes a loop-carried dependency. Although our compiler can detect reductions of scalars, it cannot detect reductions on arrays or tuples, and thus cannot parallelize the main loop.

gamteb-new is a re-write of gamteb which is approximately twice as fast on a single processor, and which does not have a loop-carried dependency in the main loop. Gamteb-new parallelizes well.

knapsack-bb is a knapsack problem which uses a branch and bound search algorithm instead of the more structured dynamic programming algorithm explained above. The first argument is the number of items we have, and the second argument is the capacity of the knapsack. The program determines the set of items with the maximum value which weigh less than the capacity of the knapsack. Knapsack-bb parallelizes well.

mm-sparse performs a sparse matrix multiply using an list-based matrix representation. The size of the argument is the length of the side of the square input matrix. Mm-sparse parallelizes well.

nas-cg is a sparse conjugate gradient problem from the NAS benchmark suite. The input size is the square root of the side of the input matrix.

Nas-cg differs from precond-conj-grad in that it can solve any linear system, whereas precond-conj-grad is specialized to solve only a particular banded linear system.

nqueens computes the number of solutions to the N Queens problem, and shows good ideal parallelism. The input size is the number of queens and the size of the side of the chess-board on which the queens are placed.

paraffins enumerates the unsaturated hydrocarbons up to a certain diameter. Paraffins is an unusual case in that we are able to parallelize the main outer loop which calculates the paraffins of diameter 1 to n, where n is the input size. For each diameter of paraffin, we can take advantage of DAG parallelism to calculate two different types of paraffins of the same diameter in parallel. However, we cannot parallelize the inner-most loops which are triply-nested loops constructing a *list* of paraffins. The list construction causes a loop-carried dependency which we cannot break. Because the lengths of the paraffins lists increase exponentially with the diameter, we are constrained by the calculation of the list of the paraffins of the largest diameter, which must be done sequentially.

Paraffins cannot be parallelized effectively unless the compiler can parallelize loops which simultaneously traverse and construct lists. This is an application which is better handled for now by the more dynamic approach of Id running on Monsoon.

pic is a 2D particle in cell simulation where particles are followed around in rectangular cells on a grid. The first argument is the number of cells on the side of an $N \times N$ grid, the second argument is the number of particles simulated, and the third argument is the number of time steps simulated.

The code consists of alternating phases which are particle-centric and cell-centric. Depending on the motion of the particles, the loops in the code may be extremely unbalanced, because certain iterations of important outer loops may handle thousands of particles, while other iterations handle none. Pic parallelizes well.

qs is a quicksort on lists, where the input size is the length of the list to be sorted.

Qs parallelizes poorly. Although the compiler can detect the DAG-style parallelism and fork off two child quicksorts in parallel, the divide and merge steps are both inherently sequential, because they traverse lists. Even on the extremely idealized TTDA dataflow architecture, this code achieves limited idealized parallelism (about 50, including instruction-level and producer-consumer parallelism) due to inherent sequentiality in the divide and merge steps.

ray-tracer is a ray-tracer which traces a square scene consisting of a few spheres. The input argument is the number of pixels on the side of the scene.

Ray-tracer shows good idealized parallelism because the outer loops enumerating the field are parallelized, even though the code creates and consumes many lists and tuples.

speech-dtw is a kernel from a speech recognition code which performs dynamic time warping. The argument is the size of the side of the square reference template,

The code performs many doubly recursive decompositions with iterative base cases, resulting in both DAG and loop parallelism. Speech shows good DAG and loop parallelism, but does not show significant improvement when both are used together. This is explained in Section 4.6

speech-proc is another kernel from a speech recognition code which performs pre-processing of raw speech samples. The first argument is the number of samples, and the second is the size of the window over the samples.

Like speech-dtw, it performs many doubly recursive decompositions with iterative base cases, but it has a unparallelized wavefront computation which takes up a relatively small portion of the sequential execution, but which limits idealized parallelism.

tree constructs a balanced binary tree and then recursively computes the sum of the leaves. The input argument is the depth of the tree. Tree shows good idealized parallelism.

4.5.4 Effect of single-assignment semantics

Much of the parallelism we are detecting is due to the single-assignment semantics, but some of the parallelism we could have detected even assuming imperative semantics, where object elements may be written more than once. Figure 4.13 shows the idealized parallelism under imperative and single-assignment semantics, while exploiting both DAG and loop parallelism.

Most of the structured codes do not parallelize well without single-assignment semantics, using our parallelization approach. Only mm-kes and mm showed some speedup, and that is for the inner-most loop of the matrix multiply, which is an inner product – the loop can be parallelized because it only

Structured codes			
Program	Arguments	Imperative	Single-Assignment
eigen-jacobi	75	5.50	82.80
knapsack-dp	100 10000	1.00	1.00
mm-kes	500	105.91	18157.99
mm-kes	700	147.85	35532.30
mm	500	214.45	308979.23
nas-multigrid	6 5	1.02	2175.44
poly-mult	13000	2.14	13922.50
precond-conj-grad	300	1.52	5195.32
region-labelling	500	1.53	66167.64
relax	1000 10	1.01	43274.73
simple-kt	10 200	1.08	4.89
simple-new	10 200	1.61	24.37
simplex	70	1.12	1.19
warshall	300	1.00	20541.52
Unstructured codes			
Program	Arguments	Imperative	Single-Assignment
btree	5000	63.74	63.99
fft-1d	65536	6.14	13939.13
fib	36	1254426.01	1254426.01
gamteb	40000	1.17	1.18
gamteb-new	40000	336.21	339.47
knapsack-bb	100 10000	2484.17	2484.17
nas-cg	300	7.51	539.09
nqueens	12	15796.00	15796.00
paraffins	1 20	1.28	2.53
pic	64 40000 100	1.60	3830.77
qs	10000	1.66	3.81
ray-tracer	500	1.04	8272.00
speech-dtw	350	53.63	86.75
speech-proc	10240 30	13.40	16.66
speech-proc	2000000 40	13.19	15.90
tree	23	493065.88	501995.42

Figure 4.20: Idealized parallelism assuming imperative and single-assignment semantics. Single-assignment has a greater effect on structured codes – for unstructured codes, the functional style of programming forced by single-assignment semantics is more important.

references matrices, and computes a running sum on the inner product. This level parallelism can be exploited on a vector machine, but is not easily exploited on an SMP, because it is very fine-grained.

As a simple example of why these structured codes do not lend themselves to our simple parallelization approach, consider the following Id-S daxpy function, which writes the solution to the daxpy computation into array `z`. Without the single-assignment semantics, even this simple loop cannot be parallelized without index analysis, because the compiler cannot determine whether there are output dependences to the writes to `z` within the loop.

```
def daxpy z a x y size =
  {for i <- 1 to size do
     z[i] = a * x[i] + y[i]};
```

Of course, the index analysis for daxpy is quite straightforward, and we believe that many of the structured applications could be parallelized effectively with Fortran-style index analysis. However, the fact that our approach could not parallelize them indicates that these programs exhibit a significant amount of output- and anti- dependences.

For the unstructured codes, single-assignment semantics allow the compiler to find more parallelism than imperative semantics, but the compiler can still find a significant amount of parallelism even when it must assume imperative semantics. The reason for this is that single-assignment semantics encourage the programmer to use a *functional style* of programming, where new heap objects are allocated frequently. When a function is called which creates a new object and fills it in, all writes to the new object are masked from the callee, therefore eliminating dependencies, including output-, anti-, and data-dependences.

As an example, consider the recursive buildtree function, which constructs a balanced binary tree of depth `depth`.

```
type  Ttree =  LEAF I | NODE Ttree Ttree;

def buildtree depth =
  if (depth == 0) then
    LEAF 1
  else
    NODE (buildtree (depth-1)) (buildtree (depth-1));
```

When the `depth` argument is 0, the function returns a `LEAF` structure with a value of 1. For `depth` greater than 0, the function makes two recursive calls to itself, and stores the left and right subtrees into a `NODE` structure. Although the `buildtree` function performs some side-effects, they are to structures which are allocated in scope of the function, which does not get reflected into the scope of the callee, allowing the two recursive calls to be safely executed in parallel.

Although many of the unstructured codes exhibit good idealized speedup even with imperative semantics, some of the programs would not necessarily show such good speedup if written in a Fortran-style or C-style, because of the those languages do not encourage programmers to frequently use heap allocation – data structures are typically static, and programmers are forced to use and re-use them. C programs also contain arbitrary pointers and type-casting, which make side-effect analysis difficult even when the programmer frequently uses heap allocation. However, many of the unstructured programs could be written in Lisp or Java, and our parallelization techniques could discover a fair amount of parallelism.

4.6 Interaction of DAG and loop parallelism

As can be seen from the results in Figure 4.19, the effects of DAG and loop parallelism can vary widely. Some codes only have DAG parallelism, such as fib, and some codes only have loop parallelism, such as nas-cg. These codes are fairly straightforward to understand. However, some codes show *both* DAG and loop parallelism, and the interaction of the two is sometimes somewhat unintuitive.

The combination of DAG and loop parallelism can produce significantly more parallelism than either DAG or loop parallelism does individually, or about the same, depending upon characteristics of the application and parallelization.

In this section, we try to provide some intuition for the interaction of DAG and loop parallelism in determining overall parallelism.

4.6.1 Example to illustrate DAG and loop parallelism interaction

Consider the simple example in Figure 4.21. Let us assume that most of the time is spent in the the constant-time sequential function `slow_seq_function`, such that procedure call overhead and loop execution overhead are negligible. If we assign the value of 1 for the time taken to execute `slow_seq_function`, then the total time taken to sequentially execute functions `two_par_loops` and `one_par_one_seq` is 8 apiece.

However, these two functions parallelize differently, leading to different idealized parallelism numbers. Figure 4.22 shows graphically the effect of different parallelizations of `two_par_loops`. If we only take advantage of DAG parallelism, then the two calls to `parallelizable_loop` can execute

```
def parallelizable_loop a =
  {for i <- 1 to 4 do
     a[i] = slow_seq_function 0;};

def sequential_loop b n =
  {for i <- 1 to 4 do
     b[i] = b[i+n] + slow_seq_function 0;};

def two_par_loops c d =
  { _ = parallel_loop c;
    _ = parallel_loop d; };

def one_par_one_seq e f m =
  { _ = parallel_loop e;
    _ = sequential_loop f m; };
```

Figure 4.21: Example program to illustrate interaction of DAG and loop parallelism. The function `slow_seq_function` is very slow, performs no side effects, and takes a constant amount of time.

in parallel, and we halve our critical path, giving us an ideal parallelism of 2. If we only take advantage of loop parallelism, then the two calls to `parallelizable_loop` must be executed sequentially, but the loops may be executed in parallel, leaving us with a critical path of 2, and an idealized parallelism of 4. If we take advantage of both DAG and loop parallelism, then both the calls can occur in parallel, and the loops can also execute in parallel, giving us a critical path of 1, and an ideal parallelism of 8.

Now consider the other function, `one_par_one_seq`, whose parallelization we show in Figure 4.23. The white loop is inherently sequential, whereas the shaded loop is parallel. This function takes the same amount of time sequentially, and for DAG parallelization. However, since only one of the loops can be parallelized, the critical path is 5, and the idealized parallelism is $8/5$. This function can exploit both DAG *and* loop parallelism, but when both are combined, the critical path is not any better than when we just exploited DAG parallelism, because the critical path is determined by the sequential loop.

These two programs are fairly easy to understand using the graphical representation in Figure 4.22 and Figure 4.23, and they show that the DAG and loop parallelism sometimes complement each other, and sometimes do not. We see examples of both cases in the codes that we have parallelized.

However, sometimes, the overall parallelism is much higher than either of the individual DAG or loop parallelism numbers, such as for FFT. This phenomenon is best explained with a complexity analysis of the critical path under the different parallelization assumptions, as we do in the following section.

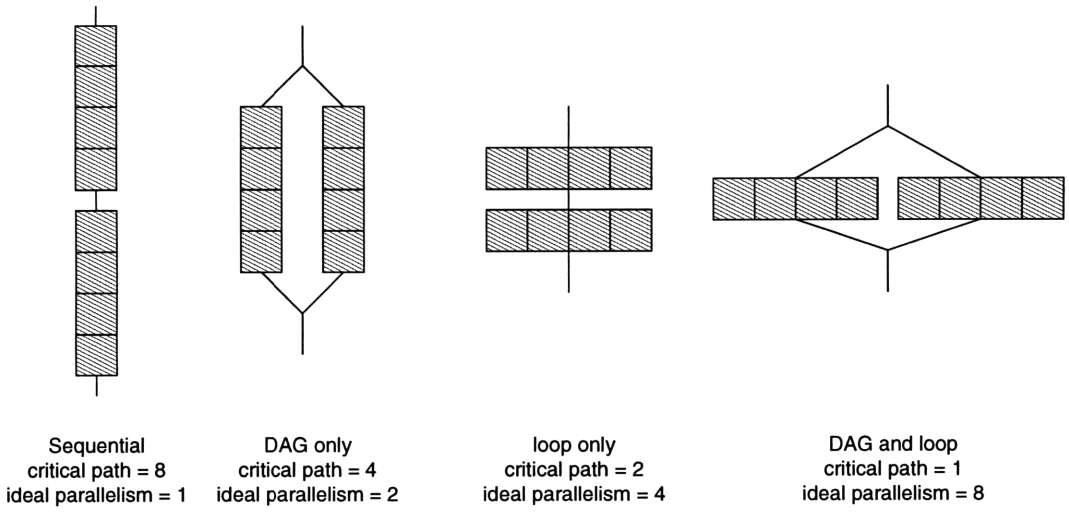

Figure 4.22: The parallelism of `two_par_loops` under different assumptions about parallelization. Exploiting both DAG and loop parallelism is better than exploiting just one or the other.

4.6.2 Analyzing DAG and loop parallelism interaction for FFT

The FFT (Fast Fourier Transform) program, shown in Figure 4.24, performs a recursive divide-and-conquer in procedure `fft`, until it reaches the base case of size 4. The function `shuffle` divides the problem into two subproblems, and the function `combine` takes the two subsolutions and combines them into one solution. FFT is typical of divide-and-conquer algorithms.

When we try to parallelize this program, the DAG parallel portions are the recursive calls to `fft`, and the loop parallel portions are the divide (`shuffle`) and join (`combine`). For all three versions, the total work is $O(n \ln n)$. However, the critical paths of the three versions are derived in three quite different ways.

We perform a complexity analysis next to explain the DAG and loop parallelism interaction, and we also show some data collected from instrumented FFT executables in Figure 4.24 which confirm the analysis. The units of the measurements are machine operations, as estimated by the compiler.

For the DAG-parallel version, the two recursive calls can be done in parallel, but the divide and combine must be done sequentially. The time to solve a problem of size n can then be done in time:

$$T(n) = T(n/2) + O(n) \tag{4.1}$$

Using the master theorem described in Chapter 4 of [23], we can solve this recurrence, and determine

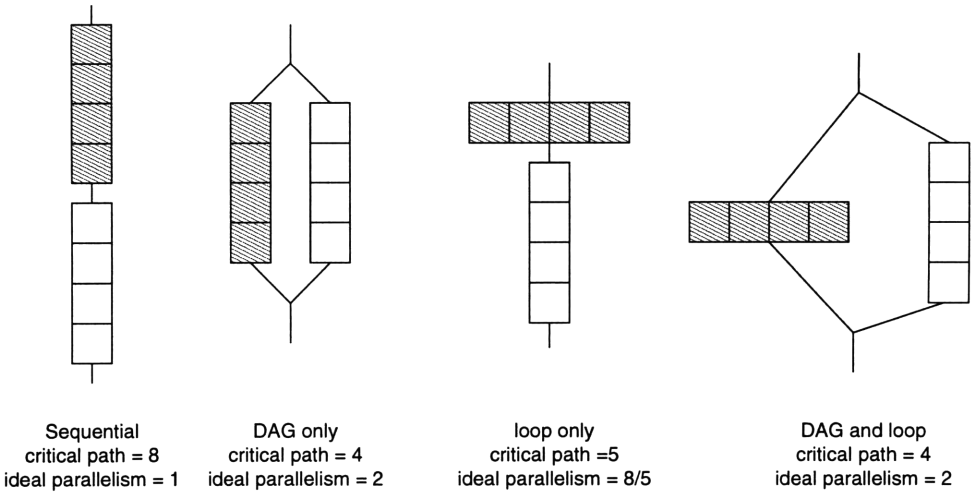

Figure 4.23: The parallelism of `one_par_one_seq` under different assumptions about parallelization. The shaded loop is parallelizable, whereas the unshaded loop is unparallelizable. Exploiting both DAG and loop does not provide any incremental improvement over just DAG parallelism, because the critical path does not change.

that the critical path of the DAG parallel version scales as $O(n)$. This is confirmed in Figure 4.24, where we run the DAG parallel version under increasing problem sizes, with the critical path scaling with the problem size.

For the loop-parallel version, the two recursive calls must be done sequentially, but both the divide and combine can be done in parallel. The time to solve a problem of size n can then be done in time:

$$T(n) = 2T(n/2) + O(1) \tag{4.2}$$

Using the master theorem, the critical path of the loop-parallel version also scales as $O(n)$. This is also confirmed with our instrumented executable in Figure 4.24.

Finally, using both DAG and loop parallelism, the recursive calls can be done in parallel, and the divide and combine can be done in parallel. The time to solve a problem of size n can then be done in time:

$$T(n) = T(n/2) + O(1) \tag{4.3}$$

Using the master theorem, the critical path of the DAG and loop parallel version also scale as $O(\lg n)$, which is also confirmed by the instrumented executable. This critical path is significantly shorter than either the critical paths of the DAG-parallel or loop-parallel versions.

```
defsubst shuffle v =
  { (_, size) = bounds v;
    m = div size 2;
   in
     { 2_arrays (1, m) of
      [i] = v[ i*2 - 1 ] , v[ i*2 ] || i <- 1 to m}};

def fft v roU =
  { (_, size) = bounds v ;
   in
     if (size == 4) then
       { l1 =   complex_add v[1] v[3];
         l2 =   complex_sub v[1] v[3];
         r1 =   complex_add v[2] v[4];
         r2 =   complex_sub v[2] v[4];
        in
          { array (1, size) of
           [1] = complex_add l1 r1
           | [2] = addflip_c l2 r2
           | [3] = complex_sub l1 r1
           | [4] = subflip_c l2 r2}}
   else
     { (left_v, right_v) = shuffle v ;
       fft_left  = fft left_v roU;
       fft_right = fft right_v roU;
      in
        combine fft_left fft_right roU}};

def combine u v roU =
  {(_,m) = bounds u;
   (_,n) = bounds roU;
   index = div n m;
   prod  = { array (1, m) of
             [i] = complex_mul roU[((i-1)*index)+ 1] v[i]
                   || i <- 1 to m};
   in
     { array (1, 2*m) of
        [i]   = complex_add u[i] prod[i] || i <- 1 to m
      | [m+i] = complex_sub u[i] prod[i] || i <- 1 to m}};
```

	Total Work			Critical Path			Idealized Parallelism		
n	DAG	Loop	Both	DAG	Loop	Both	DAG	Loop	Both
128	2.8e4	2.8e4	2.8e4	9.9e3	7.0e3	957	2.8	4.0	29.1
256	6.4e4	6.4e4	6.4e4	2.0e4	1.4e4	1115	3.3	4.5	57.3
512	1.4e5	1.4e5	1.4e5	3.9e4	2.8e4	1273	3.7	5.1	113.4
1024	3.2e5	3.2e5	3.2e5	7.7e4	5.6e4	1431	4.2	5.7	224.7
	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$

Figure 4.24: The FFT program illustrates the effect of exponential total parallelism in comparison to individual DAG and loop parallelism. The "Total Work", "Critical Path" and "Ideal Parallelism" data are collected by instrumenting the FFT program and running under different problem sizes.

Since the total work for all three versions is equal, and scales as $O(n \lg n)$, the idealized parallelism for the DAG-only and loop-only versions is $O(\lg n)$, whereas the idealized parallelism for the combined DAG and loop version is $O(\lg n)$, which is significantly higher for even moderate problem sizes such as $n = 1024$.

4.7 Id-S parallelization conclusions

In this chapter, we have described a simple parallelization technique for Id-S programs which uses existing interprocedural side-effect analyses. Our technique detects procedural DAG parallelism, as well as loop parallelism. We measured the effectiveness of our approach by instrumenting parallelized codes to give us *idealized parallelism.*

For structured codes, we primarily discovered loop parallelism, much of which might have been discovered using conventional Fortran-style index analysis even if the language semantics were imperative, instead of single-assignment. However, single-assignment semantics allow us to discover loop parallelism without complicated and expensive index analysis, although there are some cases (such as for wavefront or LUD decomposition) where our technique could be profitably augmented with index analysis.

For unstructured codes, we discovered both significant DAG and loop parallelism, even for codes which do not have "structured" loops amenable to traditional index analysis. Much of this parallelism could have been discovered even if we assumed imperative semantics, using the same side-effect analysis, because of the *functional style* of programming encouraged by the single-assignment semantics. Many side-effects were masked by heap allocations occurring within a function call.

These parallelization results suggest that a reasonable parallelization might be performed on an imperative language without arbitrary pointers and type-casting (i.e. C), by using a combination of Fortran-style index analysis for structured codes, and relying on the programmer to use a functional programming style with unstructured codes, or perhaps compiler transformations to transform less functional codes into a more functional style.

DAG and loop parallelism sometimes interact in unintuitive ways – the parallelism which can be exploited when using both DAG and loop parallelism is sometimes much less than the product of the individual DAG and loop parallelism numbers, and sometimes much more than the product of the individual numbers. This phenomenon is the result of the effect of parallelization on the *complexity* of the

critical path.

In the following chapters, we build on our parallelization with code generation and run-time system techniques for obtaining meaningful real-time speedups for these codes versus their efficient sequential implementation.

Chapter 5

Code Generation

Good parallel code generation is the first line of defense in attaining parallel speedups relative to efficient sequential execution. Any overheads that we incur in code generation will carry over into our parallel speedup numbers – once we generate parallel-ready code, the best we can usually expect is a perfect speedup versus the parallel code. In practice, other overheads will further reduce speedups, including lack of parallelism, scheduling overhead, communication and synchronization overhead, and memory system limitations such as cache interference and main memory bandwidth. However, these overheads mostly come into play *after* we have already generated parallel code.

In Chapter 3, we showed that an ultra-fine grained multithreading approach with synchronization on every memory reference incurred too much overhead to be exploited profitably on a small-scale SMP. In this chapter, we show that the coarser grained parallelism we have detected in Chapter 4 has a chance of being exploited profitably if care is taken at the code generation phase.

Our code generation schemas are based on the TAM approach [32], which is similar to P-RISC [56]. Goldstein's [32] detailed explanation of the TAM code generation approach is highly recommended. Nonetheless, the only TAM parallel speedup numbers we are aware of are for the CM-5, where they achieve linear speedup up to 64 processors on two benchmarks (Gamteb and Simple) by (1) linearly increasing the problem size, and then (2) paying about an $10\times$ (i.e. 900%) overhead for Gamteb and about a $16\times$ (i.e. 1500%) overhead for Simple versus efficient sequential execution. In other words, they achieved a real speedup versus an efficient sequential implementation of Id of $6\times$ for Gamteb and $4\times$ for Simple on a 64 processor CM-5. Much of that is due to architectural overheads, but some of it is due to fine-grain execution (which we showed in Chapter 3 was a factor of 1.5-4.0) and some of it is due to code generation and run-time scheduling.

We begin our description of our code generation strategy with the partitioning algorithm, and then continue with the calling convention, run-time system interface, control optimizations, and parallel loops. Although code generation is the meat of our compiler, in practice, most of the time in compilation is spent by the C compiler and the assembler.

5.1 Partitioning

We will only briefly discuss partitioning because it is not as critical to our implementation as it is for Id90 [81] [82] [72] [22], where thread length was severely limited by non-strictness (function call, conditional and data structure) and element-wise synchronization of data structures. *For our implementation, the primary determinator of thread length is the number of procedure calls and parallelized loops, because we only attempt to take advantage of procedure-style DAG and loop parallelism.*

Partitioning is also not as important for Id as for Sisal [67] because we maintain procedures as a natural unit of execution, whereas most Sisal compilers inline all procedures and perform partitioning to minimizing communication and synchronization. We depend upon our work stealing scheduling to lower our communication and synchronization costs, which puts less emphasis on the compiler, and more on the run-time system – this is described in more detail in Chapter 6.

Before the partitioning phase, parallelized loops are spliced out of their procedures and are handled as if they were separate procedures. The parallelization phase inserted enough dependency arcs such that any remaining reads, writes and procedure calls can execute in parallel without any Id90-style run-time presence-tag checks – we therefore translate all reads and writes to heap as C memory reads and writes. Furthermore, all ALU operations (adds, subtracts, etc.) are simply translated to their C equivalent. By this stage in the compiler, all while loops and for loops have been translated into conditionals and do-until loops – any remaining do-until loops we encounter are compiled sequentially, since all parallelized loops have been spliced out.

To the partitioner, each procedure consists of four types of "nodes" which are illustrated in Figure 5.1: (1) atomic operations such as ALU operations, literals or memory accesses, (2) procedure calls, (3) do-until loops, and (4) conditionals. Conditionals and do-until loops are frames (which we call *encapsulators*) for subgraphs of nodes (which we call *surfaces*). In the partitioner, we treat all encapsulators identically. Encapsulators are augmented with pseudo-nodes at their entry and exit points – these nodes are simply place holders which assist in the partitioning phase and later code generation.

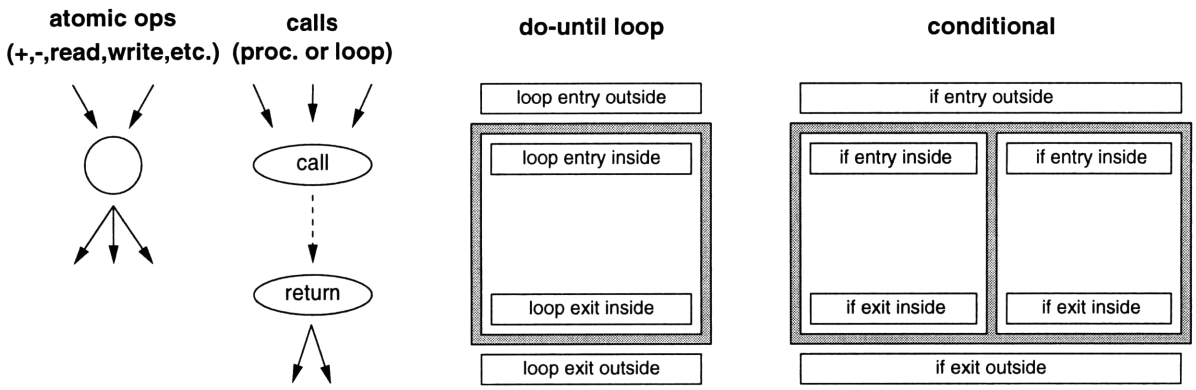

Figure 5.1: Dataflow graph nodes

Partitions are groups of nodes within a procedure which can be executed together. We would like have as few partitions as possible within a procedure in order to maximize thread length, and minimize threading overheads such as state transfer, synchronization and scheduling. Any partitioning we choose must fit the following constraints:

1. No partitions may cross encapsulator boundaries. This simplifies our partitioning, and we can increase thread length by gluing together partitions across encapsulator boundaries at a later stage of code generation.

2. Matching *call* and *return* nodes must reside in different partitions, as must any ancestor nodes of the *call* and descendant nodes of the *return*.

3. Matching "outside" pseudo-nodes for encapsulators must reside in different partitions, as must any ancestor nodes of the *entry* and descendant nodes of the *exit*.

4. Only one *return* may be in a partition. Note, however, that multiple outside *exit* pseudo-nodes may be in the same partition.

5. No descendant node may reside in an ancestor partition.

Restriction 1 is particularly useful, because it allows us to treat each encapsulator surface independently of any other nodes in the procedure. Conversely, when we encounter an encapsulator during partitioning, we can treat it as a black box and ignore any nodes within it until a later stage. The remaining restrictions guide our partitioning as we see below.

Pre-partitioning to make encapsulators atomic

1. Perform a recursive inside-out traversal of the encapsulators in the program graph
2. Mark the encapsulator as "atomic" if all nodes, including encapsulators, are atomic – i.e. are not call nodes.

Figure 5.2: Pre-partitioning to make encapsulators atomic

5.1.1 Pre-partitioning

Before we perform partitioning, we perform a stage of pre-partitioning to make some encapsulators atomic nodes – encapsulators which contain no calls to procedures or parallel loops can be executed atomically. The algorithm for pre-partitioning is shown in Figure 5.2.

5.1.2 Partitioning algorithm

The partitioning algorithm we use is Schauser's algorithm for partitioning non-strict Id [72], which for intraprocedural partitioning is identical to Coorg's [22] algorithm. This algorithm is overly general for our problem because our language is strict, but we used the same algorithm because we had already implemented it for our fine-grained compiler. A simpler algorithm with a lower algorithmic complexity would probably be suitable for our strict language – for instance, the demand and dependence algorithms described initially by Traub [81] [82].

Readers interested in the correctness of Schauser's algorithm are referred to [71]. The general outline of the algorithm is shown in Figure 5.3. This algorithm has complexity of $O(n^3)$, which results from $O(n^2)$ possible partition merges, each of which requires $O(n)$ to check whether the merge is possible and $O(n)$ to update separation constraints.

Although this computational complexity is unacceptable for a production compiler, our problem size is limited by the fact that we check each surface independently, and we perform another pre-partitioning step which does a fast, naive, demand-set based partitioning where nodes with identical ancestor sets are grouped into partitions. Furthermore, we use a bit-vector set representation which makes set operations much cheaper. In practice, partitioning is quite fast for all of the programs we are compiling.

Separation Constraint Partitioning

1. Examine each encapsulator surface separately, including the main procedure surface, because they are all independent.
2. Assign a *separation constraint* between each matching pair of *call* and *return*, and every matching pair of encapsulator outside pseudo-nodes.
3. Calculate all of the ancestor nodes and descendant nodes of every node.
4. Assign a *separation constraint* between every descendant node of a matching pair.
5. Assign each node its own partition.
6. For each pair of partitions, merge them under the following conditions:
   - There is no separation constraint
   - A maximum of one of them has a *return* node.
   - If one of them contains a *return* node, the other partition's ancestors must be a subset of the first's ancestors.
7. Update separation constraints on each merge.
8. Stop when there are no further possible merges.

Figure 5.3: Separation Constraint Partitioning

5.1.3 Peephole fixups

Although separation constraint partitioning guarantees maximal partitions, it does not necessarily produce optimal partitions [71]. What this means is that separation constraint partitioning produces the minimal number of partitions, but that the number of arcs between partitions is not minimized.

After partitioning, we do some peephole "fixups" to reduce the overhead of state transfer between threads. There is an analogous problem in compiling sequential languages in determining what operations should be in the basic block before and after a function call. Some nodes which are on the border of a partition may be moved to the other side of the border, reducing a state transfer – for instance, if a literal node is in a parent partition, but all of its uses are in a child partition, we move the literal to the child (or children) to avoid the cost of saving and restoring that value across the border. In some cases, we must clone the literal so that it can be in the parent and child, or in multiple children.

A similar fixup is possible with binary operators in a child whose operands come from the same parent, and whose operands have no other destination within the child. In this case, it is better to move

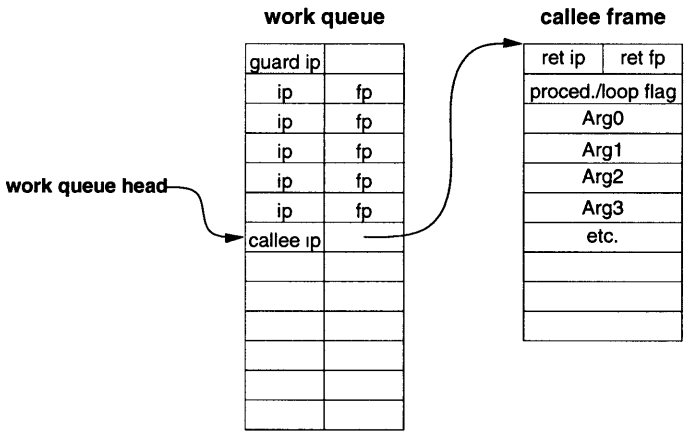

Figure 5.4: The calling convention pushes procedure calls onto the main work queue, in ip/fp *continuations*. The fp points to a frame, which contains the procedure arguments, as well as the return continuation, and a flag to tell whether the frame is associated with a procedure call or parallel loop call.

the binary operator into the parent to save the transfer of one value.

5.2 Calling convention

We must use a non-standard calling convention because procedure calls may execute in parallel. Our calling convention is similar to one used in the original Cilk system [16], which is similar to lazy task creation [52], lazy threads [35], and the hybrid calling convention used in the Illinois Concert system [64] [63]. We use an adaptation of the procedure calling convention for parallel loops, which we describe in Section 5.4.

Procedure calls occur when we have ready work – our calling convention represents this ready work in such a way that it can be transferred to another processor to be executed, if necessary. However, we make the assumption that most likely, it will be executed on the local processor.

Figure 5.4 shows the calling convention we use – first, a frame is allocated for the new procedure activation, and the return instruction pointer (i.e. the partition which will receive the procedure return value) and current frame pointer are written into the base of the frame. For scheduling purposes, we need a flag within the frame to determine whether the frame is associated with a procedure activation or loop activation – by default, the frame allocator gives us a frame with a procedure flag, so we do not need to write the frame flag. We write the arguments for the procedure into the frame, and then enqueue

the callee instruction pointer and the frame pointer onto the work queue. The instruction and frame pointer pair is called a *continuation*. Once we enqueue the procedure call onto the work queue, we can continue executing the current thread until it is completed. The procedure is eventually executed when it is dequeued from the work queue, either by the current processor, or by a steal request from another processor.

When a procedure has finished executing and is ready to return a value to its caller, it simply writes its return value into a global register, deallocates its own frame, sets the current frame to be the return frame, and jumps to the return instruction pointer. We explain in Chapter 6 what happens when the return value must be sent to another processor, but the return sequence is identical whether the caller activation is local or remote.

This calling convention has two main overheads compared to a typical sequential calling convention. First, instead of allocating and deallocating a stack frame on a contiguous stack, we allocate and deallocate activation frames in a heap-like manner, because at any point in time, we have a tree of frames which may be growing and shrinking in different areas simultaneously. To make this allocation fast, all frames are local to a processor, and frame allocation is done using a local free list, and all frames are the same size. We determine the default frame size at link time, by taking the maximum required frame size of all the procedures in our program. Finally, the free list base resides in a machine register rather than a memory location, to eliminate one more literal instruction (on the Sparc) and one memory reference. A slightly more sophisticated frame allocation algorithm would provide multiple frame sizes to increase frame memory utilization and improve cache performance. However, we found our simple, fixed-size frame allocator to be adequate for the programs we tested.

The second main overhead is that the arguments are written into the frame, and then read out, rather than being passed through registers. We believe this is the most significant overhead, although if the frame remains in the first-level cache, this should only double the cost of argument passing.

Some other overheads include enqueuing and dequeuing the continuation from the work queue, performing an indirect jump rather than a direct jump to enter a procedure, and some loss of temporal locality by postponing the procedure call rather than executing it immediately.

Figure 5.5 shows the overhead which is directly due to our parallel calling convention. To measure the overhead from the parallel calling convention, we have turned off all of the parallelization phases so that we do not take advantage of DAG or loop parallelism, and we have also removed all message polling overheads. We executing the code using the parallel calling convention on one processor, and

Parallel Calling Convention Overhead vs. Sequential Time				
Structured Codes				
Program	Arguments	Seq. Time	Cycles/Call	Calling Overhead
eigen-jacobi	50	10.8	$9.1e+04$	+1.9%
knapsack-dp	100 10000	0.4	$1.7e+07$	+0.0%
mm-kes	500	7.0	$7.4e+04$	+0.0%
mm-kes	700	20.1	$1.1e+05$	+2.0%
mm	500	39.3	$1.7e+09$	+0.3%
nas-multigrid	6 5	17.9	$3.6e+06$	+3.9%
poly-mult	13000	22.0	$1.2e+09$	+0.5%
precond-conj-grad	300	19.5	$1.1e+07$	+1.0%
region-labelling	500	16.5	$4.7e+07$	+2.4%
relax	1000 10	19.8	$2.2e+08$	+2.0%
simple-kt	10 200	13.9	$2.0e+03$	+7.2%
simple-new	15 300	39.3	969.1	+8.9%
simplex	70	3.3	704.6	+9.1%
warshall	300	16.2	$9.0e+06$	+0.0%
Unstructured Codes				
Program	Arguments	Seq. Time	Cycles/Call	Calling Overhead
btree	5000	29.1	94.0	−29.6%
fft-1d	262144	10.4	$8.9e+03$	−12.5%
fib	36	9.4	32.7	+48.9%
gamteb	40000	30.4	747.3	+4.3%
gamteb-new	80000	30.9	597.6	+9.1%
knapsack-bb	120 100000	36.4	486.0	+16.5%
mm-sparse	350	44.5	345.8	−35.5%
nas-cg	600	28.2	$2.5e+07$	+27.7%
nqueens	14	13.4	83.4	+58.2%
paraffins	1 22	4.8	$2.4e+04$	+4.2%
pic	64 40000 100	21.7	$1.7e+06$	+26.3%
qs	10000	0.2	560.0	+0.0%
ray-tracer	500	15.7	155.4	+2.5%
speech-dtw	350	19.9	$1.2e+06$	+1.0%
speech-proc	2000000 40	18.1	$3.1e+03$	−2.2%
tree	23	16.8	84.1	+22.0%

Figure 5.5: The parallel calling convention overhead is more noticeable for unstructured codes, because those codes tend to execute more function calls, as evidenced by the processor cycles between calls. The UltraSparc processors run at 168 MHz. Because of register windows, some codes actually run faster with the parallel calling convention.

compare it to the same code using the sequential calling convention.

As could be expected, the overhead for the structured codes is not very high, because these codes do not rely heavily on procedure calls – most of the work in these codes lies in loops.

The unstructured codes show a wide range of overheads, and in some cases, our parallel calling convention is actually *faster* than the sequential C calling convention, because of inefficiencies due to the Sparc register windows. Register windows are effective on codes which have a shallow call depth, but can cause thrashing when the call depth changes dramatically. In mm-sparse, the inner product is written as a tail-recursive function, and the call depth is dependent upon the data, but is typically more than the number of register windows available on the UltraSparc, causing significant thrashing.

For other codes, the additional overhead of storing and reading the procedure arguments from memory are too high to overcome, especially in codes with small procedure bodies that perform many procedure calls, such as fib, nqueens, and tree.

5.3 Join synchronization

The calling convention we described, in combination with the data dependence analysis we performed for parallelization, allows us to take advantage of DAG style parallelism. Once multiple procedure calls are forked off, however, they must be synchronized on their return. To synchronize threads, we use *join counters*, coupled with a *local work queue*. Join counters are also used in TAM [32], P-RISC [56], and Cilk [16].

Using control flow information, we perform some compiler optimizations to reduce the cost of join synchronization, including tail optimization, and join sorting.

5.3.1 Join counters and local work queue

Join synchronization is necessary when a partition has multiple parents which may return asynchronously. The partition can only execute when all of its parent partitions have returned. Some partitions do not require synchronization, because they receive return values from procedure calls – because of our calling convention, these partitions execute immediately. Also, a partition does not need to synchronize on the value of all of its parents, just the parents which do not depend on any of the other parents.

Assigning join counters

1. Calculate each partitions ancestors
2. For each partition, examine its parents. If the parent is not the ancestor of any of the other parents, then it is one of the *synchronizing parents* for the partition.
3. The *join count* for each partition is the number of synchronizing parents.

Figure 5.6: Join counter assignment.

Figure 5.6 sketches the procedure for determining the synchronization count for each partition. The join counts are based in the frame, and the join count frame slots are initialized when a procedure enters its first partition. When a synchronizing parent terminates, it decrements the join count for all of its children, and if the join count reaches zero, then that child is enabled.

We enable a child by enqueuing it on a *local work queue* – unlike the regular work queue, the local work queue only has instruction pointers, because it will only be used on threads from the same activation frame. We enable work by pushing an instruction pointer onto the queue (which is a stack) and we schedule work by popping off an instruction pointer and jumping to it. The top of the local work queue is pointed to by the local work queue pointer, which resides in a machine register, to make enqueuing and dequeuing fast. The bottom of the work queue is a *guard slot* which points to a code fragment which dequeues work from the main work queue, setting the frame appropriately.

At the end of each thread, we dequeue the next ready thread from the local queue and jump to it. Because the frame resides in local memory, and we know that no other processor can touch this state, all of the operations can be performed without shared memory synchronization.

5.3.2 Control optimizations for join synchronizations

The scheme we outlined above is very general, but we can perform some optimizations because we have some control information. For example, the typical thread will perform one or more join synchronizations, and then dequeue a thread off of the local queue and jump to it. This is redundant if we push some work onto the queue and then pop it off immediately – in this case, the last join synchronization in a thread is modified such that if it succeeds, it will jump directly to the thread. In the case the synchronization fails, we pop work from the local queue and jump to it, as we usually do.

Sometimes, we know that there is no work on the local queue, because we were either the first thread

in the procedure or a receiving a return value from a procedure call, and we know we didn't put any work on the local queue because we didn't perform any join synchronizations. In that case, we avoid executing the guard code fragment from the bottom of the local work queue, and directly dequeue the next ready thread from the main work queue and execute it

5.3.3 Optimizing for binary joins

Almost all of the joins which are executed in practice are binary joins, where the join counter is initialized to "2". In these cases, we do not need to perform a subtraction of the join counter – we just need to read the value, compare it to 1, and store 1 into the join counter if the value is not 1. This eliminates a few instructions in the join synchronization.

5.3.4 Join synchronization overhead

Figure 5.7 shows some of the statistics for join synchronizations. To give an idea of the relationship between joins and procedure calls, we give the ratio of joins to calls, which typically is less than 1 – for example, in the fib program, almost every procedure call except the original call to fib must perform a join synchronization, resulting in a ratio of 1.0. However, the nqueens program shows that the number of joins per call may be more than one – in this case, the procedure call is wrapped in a conditional, and may or may not be called. In either case, the thread containing the conditional terminates and causes a join synchronization. For every procedure call taken in nqueens, one is not taken which still requires a join synchronization.

The fact that there are usually less joins than calls indicates that joins are much less important to optimize than calls. Furthermore, the cost of joins is only significant when there are a large number of them – of our codes, only the ones which are heavily doubly recursive ones (btree, fib, knapsack-bb, nqueens, and tree) have a relatively frequent occurrence of joins. As expected, the structured codes show much fewer joins than the unstructured codes, with a few structured codes having no joins at all.

Our control optimizations are also able to turn the vast majority of joins into binary immediate joins, eliminating many indirect jumps, join counter arithmetic, and work queue manipulation. All in all, join synchronizations add a small overhead in addition to the parallel calling call overhead.

Note, however, that more join synchronizations will be necessary when we attempt to take advantage of parallel loops because parallel loops are implemented as separate functions. In the next sections, we

Join Synchronization Overhead					
Structured Codes					
Program	Arguments	Joins/Call	Cycles/Join	% Binary Immd.	Overhead
eigen-jacobi	50	0.66	$1.4e+05$	100.0	-0.9%
knapsack-dp	100 10000	0.50	$3.4e+07$	100.0	$+0.0\%$
mm-kes	500	0.02	$4.6e+06$	98.8	$+4.3\%$
mm-kes	700	0.01	$9.8e+06$	99.1	$+4.4\%$
mm	500	*no joins*			
nas-multigrid	6 5	0.84	$4.4e+06$	0.28	$+0.0\%$
poly-mult	13000	*no joins*			
precond-conj-grad	300	0.40	$2.8e+07$	3.4	-5.1%
region-labelling	500	0.61	$7.9e+07$	100.0	-0.6%
relax	1000 10	0.13	$1.7e+09$	100.0	$+1.5\%$
simple-kt	10 200	0.00	$7.0e+07$	55.6	$+2.0\%$
simple-new	15 300	0.99	$1.1e+03$	0.0	$+0.2\%$
simplex	70	0.00	$2.1e+06$	100.0	-2.8%
warshall	300	*no joins*			
Unstructured Codes					
Program	Arguments	Joins/Call	Cycles/Join	% Binary Immd.	Overhead
btree	5000	1.00	66.4	100.0	$+0.0\%$
fft-1d	262144	0.67	$1.2e+04$	100.0	$+5.5\%$
fib	36	1.00	51.1	100.0	$+10.7\%$
gamteb	40000	0.08	$1.0e+04$	100.0	-3.8%
gamteb-new	80000	0.12	$5.4e+03$	100.0	$+1.5\%$
knapsack-bb	120 100000	1.00	566.1	100.0	-8.7%
mm-sparse	350	0.00	$2.4e+09$	100.0	$+0.7\%$
nas-cg	600	0.33	$9.8e+07$	100.0	$+0.3\%$
nqueens	14	2.00	66.0	50.0	$+0.5\%$
paraffins	1 22	0.00	$1.9e+07$	100.0	$+0.0\%$
pic	64 40000 100	0.10	$2.3e+07$	100.0	$+4.4\%$
qs	10000	0.33	$1.7e+03$	100.0	$+0.0\%$
ray-tracer	500	0.06	$2.5e+03$	100.0	-6.2%
speech-dtw	350	0.75	$1.6e+06$	100.0	-1.0%
speech-proc	2000000 40	0.43	$7.0e+03$	100.0	$+0.6\%$
tree	23	1.00	102.6	100.0	$+5.9\%$

Figure 5.7: Join synchronization typically occurs less often than procedure calls, and is only frequent for programs which make a lot of procedure calls. Many of the join synchronizations can be optimized into binary immediate joins, and join synchronization does not add significant code generation overhead.

will show the effect of parallel loop codes.

5.4 Parallel Loop Code

Thus far, we have described the implementation of the parallel calling convention, join synchronization and join synchronization optimizations, which coupled with a parallel run-time system allows us to take advantage of DAG-style parallelism. As we saw in Chapter 4, many codes depend upon parallel loops for most of their parallelism.

Our loop code generation strategy is structured such the work queues are used in much the same way that they are for procedures. When work is dequeued from the main work queue, the same instructions are executed: the frame pointer is set, and we jump to the instruction pointer. Since we would like the loop to be executed in parallel, we do not execute all of the iterations on each scheduling of the loop. Rather, we execute a *chunk* of iterations at once, and leave the rest to be executed later – the specification of the iterations which are left to be executed are stored in the frame. The size of the *chunk* is determined by the compiler, and set to be large enough to minimize loop scheduling overhead while small enough to expose parallelism in the loop.

If no work is ever stolen from the parallel loop, we execute all of the loop iterations in order on the same processor. If work is stolen from the loop, the loop is roughly executed in a divide-and-conquer fashion as a binary tree.

5.4.1 Parallel loop calling convention

The parallel loop calling convention is outlined in Figure 5.8. As for procedure calls, parallel loops are called by enqueuing a continuation onto the main work queue. However, loop frames contain more information than procedure frames, in order to handle parallel loop bookkeeping. For example, the frame contains information about how many times work was stolen from this loop activation, what the final iteration and current iteration are, as well as the loop induction variables and loop constants.

As we described in Chapter 4, we only parallelize loops where all of the induction variables are either constantly incrementing variables, or reduction variables. These induction loop variables are maintained in the loop frame, with the incrementing variables kept at their correct state for the number of iterations executed. The reduction variables keep the running sum of the iterations calculated thus far.

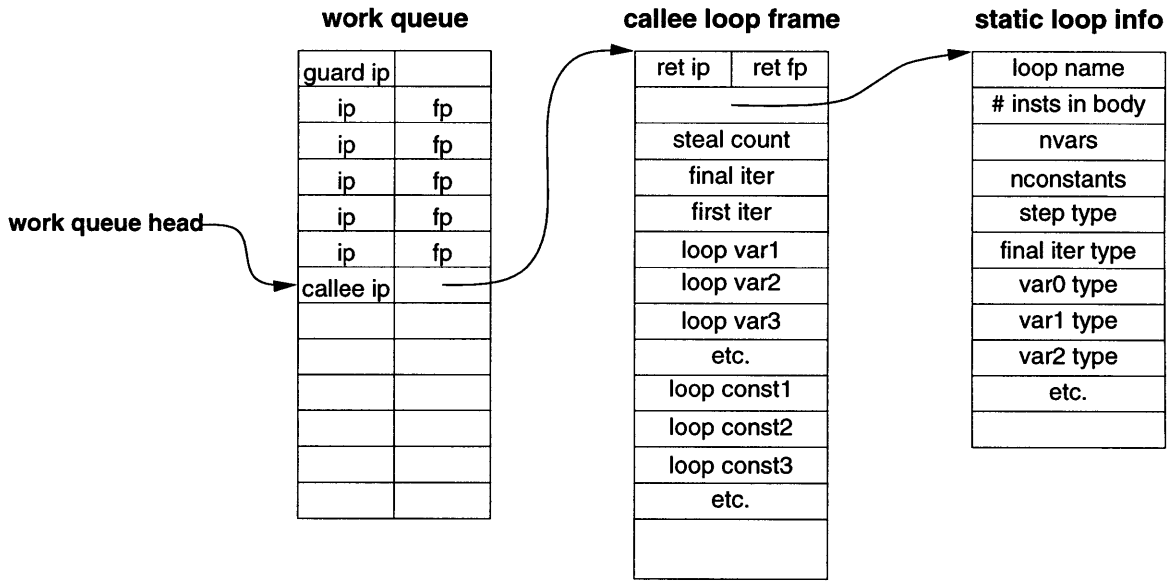

Figure 5.8: Parallel loop calling convention

The number and type of loop induction variables and loop constants is kept in an auxiliary static structure called the "loop info" which is pointed at by the second slot in the frame – for procedures, this same slot is filled with zero. The run-time system, when it needs to distinguish between a procedure and loop, can then check this flag to see whether it is a procedure or loop frame. Loop induction variables are defined as follows:

- Incrementing variables
  - incremented by a literal, literal value
  - incremented by a constant, constant frame slot offset
- Reduction variables
  - Boolean and, or
  - Integer addition, minimum, maximum, logical and, logical or, multiplication
  - Floating point addition, multiplication, minimum, maximum

These loop variable specifications are stored in the loop info, so that the run-time system can figure out how to split up a loop to execute the two parts in parallel, and how to reconcile the two parts once they are each finished. Given the first and last iterations as specified in the loop frame, and the loop info, we can calculate the incrementing variables, and we can initialize the reduction variables with their identity values.

Loop chunking protocol

1. Loop prelude
   (a) read the first and last iterations
   (b) determine the number of iterations to execute – either chunk size n or the number of iterations left in the loop, whichever is smaller.
   (c) update the first iteration in the frame
   (d) increment the steal count
   (e) read each constant incrementing loop variable, and increment it.
2. Loop body, executed n times as a do-while loop, where n is a compile-time constant.
3. Loop postlude
   (a) decrement the steal count
   (b) if there are more iterations left, push loop continuation onto main work queue
   (c) otherwise if the steal count is greater than 0, then dequeue and execute next continuation on main work queue
   (d) otherwise, return the loop variables on the local work queue, and jump to the parent return partition.

Figure 5.9: The loop chunking protocol is used to "strip mine" loop iterations and reduce the overhead of parallel loop execution, while exposing loop parallelism.

During local execution, we execute a set number of iterations at a time, using the loop chunking protocol described in the following section.

5.4.2 Loop chunking protocol

The loop chunking protocol is described in Figure 5.9 – on every entry to the parallel loop, a set number of iterations are executed at a time, and the loop prelude and postlude perform the bookkeeping and synchronization to ensure that any iterations which are stolen are properly accounted for.

The loop chunk, as shown in Step 2 of Figure 5.9 is potentially different for each loop. A larger loop chunk will create a coarser grain of execution – we would like to use large loop chunks for loops with small loop bodies, because we need to overcome the parallel call overhead. We would like to use a small loop chunk for loops which have a large loop body, because we can afford to execute the parallel loop protocol many times, and because we want to expose as much parallelism as possible for other processors to steal. The loop chunk is not unrollable because we may have a different number of

Setting default loop chunks

1. If parallel loop nest is 1 or 2 then:
   (a) If loop body is greater than or equal to 30 nodes, then set loop chunk to 1
   (b) If loop body is less than 30 nodes, then set loop chunk to ∞
2. If parallel loop nest is greater than 2, don't parallelize the loop

Figure 5.10: Simple minded algorithm to set the parallel loop chunk.

iterations for the tail of the loop execution if the number of iterations is not a multiple of the loop chunk, which it usually is not.

Although the loop chunking protocol is fairly complicated, in practice, we can amortize its overhead effectively by being judicious about which loops we parallelize, and by setting the loop chunk size using compile time information about the size of the loop bodies, which we describe in the next section.

5.4.3 Work estimation

Although we have tried to keep the loop parallelization overhead as low as possible, essentially, we are adding a procedure-call overhead for each parallel loop, as well as some bookkeeping overhead for each loop chunk schedule. For some loops, this overhead is insignificant, because the loop executes for such a long time – for other loops, this overhead is very significant, because the loops execute for a short time, and have a small loop body.

We face two questions in parallelizing loops – first, whether to parallelize the loop at all, and secondly, if we decide to parallelize the loop, what chunk size we should use. Both of these questions relate to the amount of work in a loop body. If a loop body contains a lot of work, then we should parallelize the loop and set a small chunk size to allow maximum parallelization of the loop. If the loop body contains less work, then may still want to parallelize the loop, but set a large chunk size. If the loop body contains very little work, we may not want to parallelize the loop at all, because it does not contain enough work to warrant parallelization.

To determine the amount of work performed in a loop body, we perform a simple intraprocedural analysis, determining the loop nesting depth and counting the number of nodes within a loop body, with procedure calls estimated to be 250 nodes, and loops within any loop bodies estimated to execute 10 iterations.

The loop chunk is then set using the procedure shown in Figure 5.10. This procedure could probably be tuned using by more precise work estimates and more information about tradeoffs due to the number of loop variables and constants, and interprocedural information about loop body work. We don't even bother to set the loop chunk to anything other than 1 or ∞ – note that setting the loop chunk to ∞ is not equivalent to sequentializing the loop, because it is still possible for loop iterations to be stolen when the loop chunk is ∞ if the loop is not scheduled before a steal request is handled by the local processor.

In practice, even this simple minded procedure does a good job in keeping parallel loop overhead down, while exposing enough parallelism to provide reasonable speedups on most of the codes. In the next section, we examine the cumulative effect of overheads resulting from the calling convention, join synchronization, and parallel loops with chunks set by the procedure in Figure 5.10.

5.4.4 Cumulative overheads from calling convention, join synchronization, and parallel loops

Figures 5.11 and 5.12 show the cumulative effect of the code generation overheads versus the efficient sequential implementation, when the codes are run on a single processor. We perform no message polling for these runs – polling overhead is characterized in the next chapter. For the structured codes, there is little overhead incurred, even with the parallel loop schema described in the previous section.

For the unstructured codes, the overhead varies widely, with some codes actually running faster (btree, mm-sparse and ray-tracer) because of the overhead of register windows in the sequential version. Some codes with small procedure bodies and many procedure calls (fib, nqueens, and tree) show high overheads due to the procedure calls.

In general, the join synchronization overhead is only significant for speech-proc – the other codes do not show a significant jump in overhead from the overhead introduced by the calling convention. The join synchronization overhead is low because many of the codes do not perform a significant number of join synchronizations, and the ones which do are typically amenable to the control optimizations we described in Section 5.3.

Finally, the parallel loop overhead is only significant for nas-cg, after our work estimation weeded out loops which were not worth parallelizing, and setting loop chunks to minimize parallel loop overhead while exposing coarse-grained work.

The overhead numbers in the rightmost bars of the graphs in Figure 5.11 and 5.12 show our starting

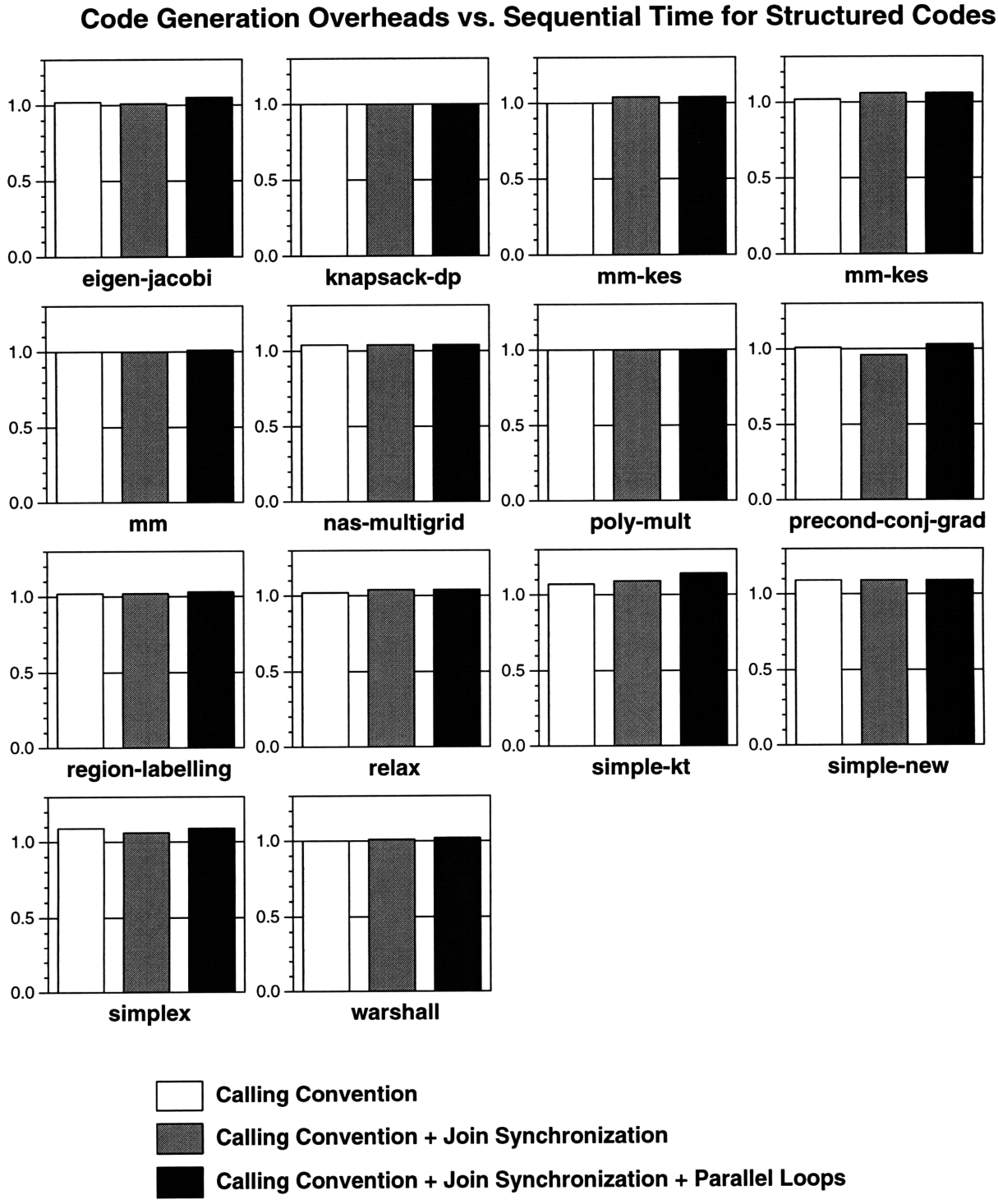

Figure 5.11: Cumulative overheads from the calling convention, join synchronization and parallel loops for the structured codes. The rightmost bars are our starting points in parallel execution.

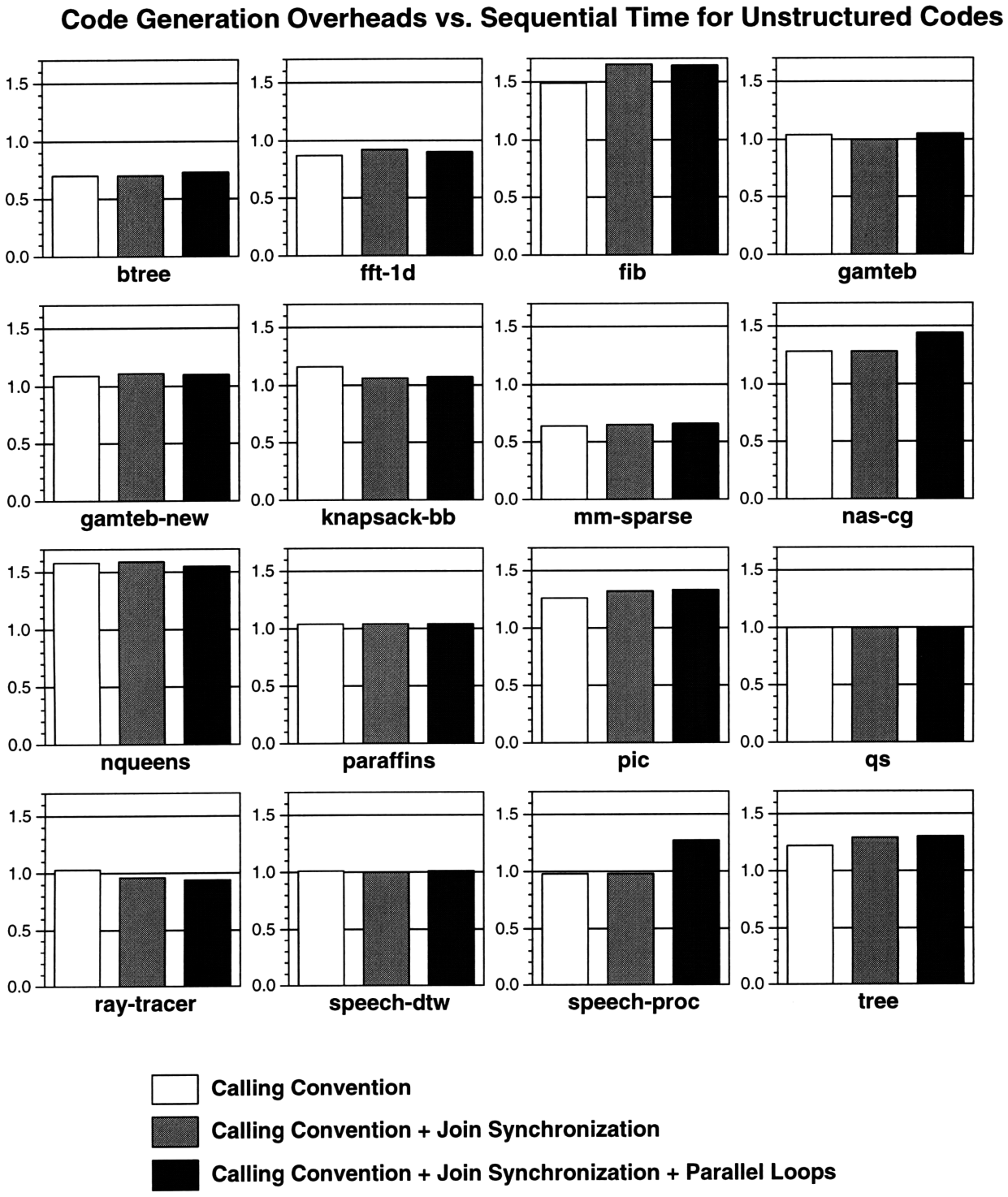

Figure 5.12: Cumulative overheads from the calling convention, join synchronization and parallel loops for the unstructured codes. The rightmost bars are our starting points in parallel execution.

point in parallelization. The parallel codes incorporate this base overhead, and our speedup numbers relative to the efficient sequential codes are limited by it. We have incurred all of the overheads of code generation which will ready us for parallel execution – most of the additional overheads we face in parallel execution are due to the run-time system and limitations of the architecture.

Chapter 6

Run-Time System

In the previous chapters, we have described how the compiler discovers DAG and loop parallelism, how it generates parallel-ready code to exploit that parallelism, and how much overhead is incurred on a single-processor execution of the parallel-ready code. In this chapter, we show how to schedule this code in parallel with relatively low overhead, achieving good speedups on our SMP configurations.

The run-time system communicates using a *active message-passing layer* [83] written on top of the hardware supported shared memory. The message-passing layer allows us to introduce modifications to processor-local state only when we are prepared to, thereby reducing synchronization overheads. Messages are handled only when the local processor polls its network input queues, and we reduce the cost of the polling so that it can be done frequently without adding additional overhead.

Our run-time system uses a *random work stealing* scheduling policy. Work is performed locally by each processor, and if a processor has no work to do, it sends *steal requests* for work to random processors. This policy has been shown to be efficient in Cilk [16], lazy task creation [52], Multilisp [40], and lazy threads [33]. We extend the policy to work with parallel loops, using the parallel loop protocol described in the previous chapter, and describe some alternatives for work stealing which we compare.

Finally, we implement an SPMD-style scheduler which takes parallelized loops and divides their iteration space equally among the processors, while exploiting no DAG parallelism. The structure of the SPMD implementation is quite similar to the work-stealing compiler and run-time system – the SPMD scheduler and compilation can be considered a subset of the multithreaded scheduler and compilation. We compare performance results from SPMD scheduling with work-stealing multithreaded scheduling.

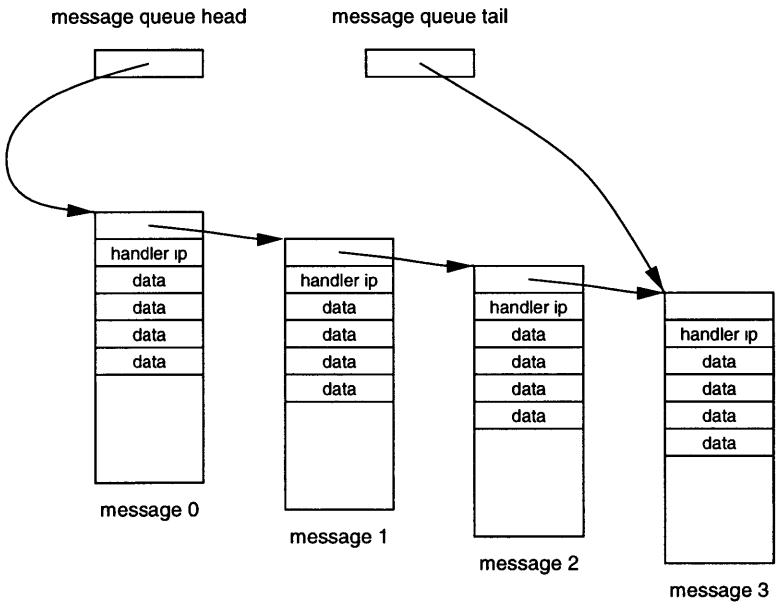

Figure 6.1: The structure of the active-message passing layer on shared memory. Each processor has a message queue head, tail and lock, all of which reside in shared memory. Messages are linked fixed-sized blocks of shared memory.

6.1 Message passing layer

Although our target architecture is shared-memory SMP's, we implement communications within the run-time system using message-passing because it allows us to manipulate processor local state without locking – this processor state includes the main work queues and the join synchronization counters. The message-passing layer is structured as an *active message* passing layer [83].

Figure 6.1 shows the structure of the active message layer within a processor. Each processor has a message head, message tail, and message lock for its own local message queue. The message queue heads, tails and locks all reside in shared memory, accessible to all processors.

When a processor sends a message to another processor, it must perform the following actions:

1. Allocate a message (a fixed-size block of shared memory) from a quick list, and compose the message, filling in the message handler instruction pointer, and the data in the message
2. Acquire the lock for the destination processor's message queue
3. Splice the message onto the end of the message queue's linked list structure
4. Release the message queue lock

Polling Overhead vs. Non-polling Code			
Structured Codes		*Unstructured Codes*	
Program	Polling Overhead	Program	Polling Overhead
eigen-jacobi	+0.9%	btree	+2.4%
knapsack-dp	+0.0%	fft-1d	−2.1%
mm-kes	+0.0%	fib	+5.2%
mm-kes	+0.5%	gamteb	−4.7%
mm	+0.0%	gamteb-new	−2.9%
nas-multigrid	−0.5%	knapsack-bb	+2.8%
poly-mult	−0.5%	mm-sparse	+0.3%
precond-conj-grad	+0.5%	nas-cg	+2.2%
region-labelling	+0.0%	nqueens	+7.2%
relax	+0.0%	paraffins	+2.0%
simple-kt	−0.6%	pic	+0.3%
simple-new	−1.9%	qs	+0.0%
simplex	+2.8%	ray-tracer	+2.0%
warshall	+1.2%	speech-dtw	+1.0%
		speech-proc	+0.0%
		tree	+2.8%

Figure 6.2: Polling overheads shown by running identical parallel-ready codes with polling and without polling on one processor. Polling is done before every procedure entry or loop chunk execution. Polling does not show a significant overhead, except for procedure-call intensive codes such as fib, nqueens and tree.

When a processor is ready to check its message queues (i.e. "poll") it merely does an unlocked read of the message queue head – if the head is 0, then the queue is empty. We reserve a register to point to the head of the message queue so that polling consists of a memory read, a compare and branch, where the vast majority of the time, there is no message in the message queue.

When a processor poll finds that there is at least one message in the queue, it does the following:

1. Acquire its own message queue lock
2. Splice out the linked list of messages, setting the head and tail of the queue to 0
3. Release the message queue lock
4. Execute the handler for each message, using the message itself as an argument – the handler themselves are responsible for interpreting the data in the message

Although the overhead of message passing can be relatively high, we are not as concerned about message passing overhead as much as we are attempting to avoid overhead in the common case. Message passing allows us to avoid locking when we modify local state such as work queues and join

synchronizations, and the cheap polling allows us to avoid most of the message passing overheads in the usual case when we have no messages to handle.

Each processor polls the network when there is no longer any work in the local work queue, before it pulls work from the main work queue. This policy is frequent enough to gain good response, but infrequent enough that polling does not add significantly to the overhead. Figure 6.2 shows the overhead of polling relative to the identical code which doesn't perform polling, running on a single processor. The overhead due to polling is not high, except for codes which perform a lot of procedure calls, such as fib, nqueens, and tree, where more polling is done than is necessary.

6.2 Work stealing policy

Work is distributed in our system via *randomized work stealing*. Idle processors are responsible for finding work – they do that by picking a random processor, sending a request for work to that processor, and waiting for a response. During the wait for the response, the processor continues handling messages from other processors, in order to avoid deadlock. If the response is negative (i.e. no work) then the processor picks another random processor and tries until it finds some work. If the response is positive, enough information is included in the response to start work.

Figure 6.3 shows the basic steal protocol. When a processor becomes idle, it sends a steal request to a random "victim" processor. If the victim processor has surplus work on its main work queue, then it sends the work in the form of a continuation to the idle processor, which then executes the work. When the work is complete, the result is sent back to the victim processor, which immediately returns it to the original caller.

We use *return* and *receive stubs* so that we can use the ordinary calling convention described in 5.2, without checking whether we need to send the return value of a procedure to a different processor, and without checking whether we need to receive the return value of a procedure call from a different processor, even though any procedure call may be potentially executed on a different processor than the callee.

6.2.1 Return and receive stubs

When a continuation is stolen from a victim's work queue, a "receive stub" is created which keeps track of the continuation's own return continuation. The receive stub is a frame/ip pair where the return

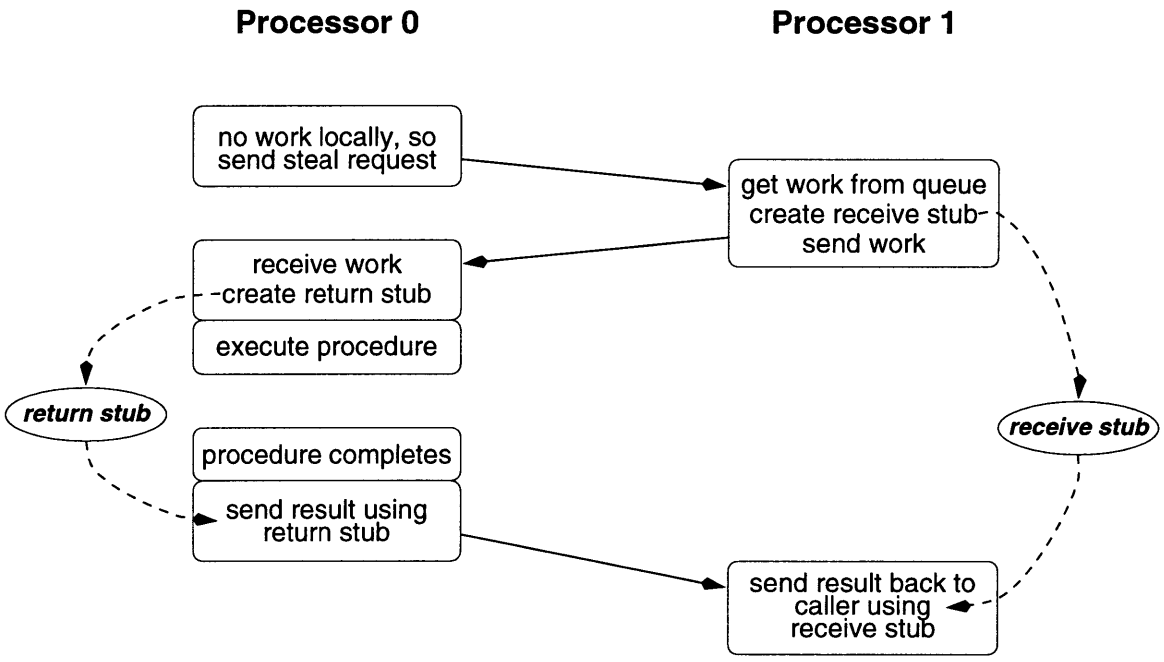

Figure 6.3: The basic steal protocol – idle processors send requests to random processor for work. If the request is satisfied, the idle processor executes the work, and when done, sends back the result.

continuation is saved in the frame, and a pointer to a code sequence which receives the return value from a message, and then immediately passes it to the return continuation. A pointer to the receive stub frame is passed along with the work continuation and the processor number to the requesting processor.

When the requesting processor receives the work continuation, it stores away the victim processor number along with the pointer to the receive stub frame in its own "return stub" frame. The return stub consists of a frame/ip pair with the aforementioned frame, and a pointer to a code sequence which receives the return value from the work continuation (once it has completed) and packages it up along with a pointer to the receive stub frame, and sends it back to the victim processor.

With these two stubs, the compiler only generates procedure calls and returns which match each other, and are optimized for sequential execution on a single processor. The stubs themselves are much slower than the normal calling sequence because they must marshal and unmarshal values to and from messages, but are used infrequently because most procedure calls will execute sequentially.

Return and receive stubs allow us to make the tradeoff of optimizing the common case (sequential call and return) while handling the uncommon case (parallel call and return). We have extended the basic protocol for stealing procedures to also handle stealing of loop iterations.

6.2.2 Stealing loop iterations

The parallel loop calling convention described in Section 5.4 was designed so that parallel loops are handled in the same way as procedure calls from the local point of view, with enough hooks so that the run-time system could distribute iterations of the loop when a steal request is received. When a steal request is received the victim processor checks the bottom of its work queue, looking at the loop/procedure flag as shown in Figure 5.4.

If the flag indicates that the frame (and continuation) is a procedure call, then the standard steal protocol is executed as just described. If the flag indicates that the frame (and continuation) is a parallel loop call, then the victim processor updates the loop frame by bumping the steal counter, removing half of the iterations and packaging them up to send to the requesting processor. The loop/procedure flag also is a pointer to a static *loop info* structure, which is generated by the compiler, and which describes the structure of the loop induction variables, loop constants, and how these are mapped into the loop frame. This loop info pointer is stored in the receive stub, and also sent along with the loop iterations to the requesting processor.

The loop info pointer can be used by the requesting processor to interpret the message sent by the victim, and create a new loop frame, which is treated in the same way as any other parallel loop frame – other processors may also steal iterations from this loop as well. The stolen loop iterations are chunked in the same way as the original loop, and when the iterations are complete, the loop returns control to the return stub, which using the static loop info, determines which induction variables are reduction variables, and need to be returned to the victim processor to merge with the original loop frame.

These reduction variables are marshaled into a message along with the pointer to the loop info structure and a pointer to the receive stub frame. The receive stub interprets the loop info, and then performs the merge of the reduction variables into the parent loop frame. After the merge, the steal count is decremented, and if the local loop is complete and the steal count is 0, then the loop returns to its caller.

6.3 SPMD scheduling

Our compilation approach to this point has been extremely flexible, which allows us to experiment with different run-time policies. One alternative policy to the multithreaded work-stealing strategy is the Single Program Multiple Data (SPMD) policy, where loops are divided up evenly among the processors.

With very minor changes to the run-time system and code-generation, we can also generate codes which are scheduled in an SPMD fashion. This is only useful for codes which rely mostly on loop parallelism, including all of the structured codes, and a few of the unstructured codes.

The code generation is slightly different for the SPMD scheduling than for multithreaded scheduling – most obviously, we turn off all DAG parallelism, as we did in Chapter 4, because SPMD scheduling does not exploit DAG parallelism. Furthermore, we only try to exploit one level of loop parallelism, so we do not parallelize inner loops. However, in general, we do not know whether a parallelizable loop will or will not be executed within the context of another parallel loop, since the two loops may be in different procedures, so we also generate code for each parallel loop which dynamically checks a processor local flag which is set when we enter a parallel loop, in order to avoid forking off additional parallel loops. Some of these checks could be eliminated by better interprocedural analysis, but the check itself is so cheap relative to forking the parallel loop that this analysis would not improve performance noticeably.

At the run-time system level, we use a slightly modified scheduler which never sends steal requests. Processor "0" computes sequentially until it reaches a parallel loop, and then distributes the iterations of the loop evenly to the other processors via message passing. The same handlers are used to distribute loop iterations in the multithreaded and SPMD versions, since the multithreaded versions must be general enough to handle an arbitrary number of iterations. When a loop terminates, it sends a message with the loop return values to processor 0.

We use the same message handlers for the loop distribution as we do for the loop iteration stealing, and we use the same "loop victim stubs". In general, the loop schema used by the compiler differs from the work-stealing version in only a few lines of code for the loop iteration distribution. The two run-time systems differ only in that for the SPMD version, the steal code is commented out.

This approach allows us to compare the relative merits of a semi-static, single-threaded work distribution policy with a dynamic, work-stealing multithreaded policy, using the same language and same compiler.

6.4 Performance and speedups

Figures 6.4 and 6.5 show the speedup numbers on 4 processors for the structured and unstructured codes, under the work-stealing multithreaded and SPMD schedulers. The outer bars of the graphs show the limitations due to lack of parallelism – for example, we were not able to parallelize knapsack-dp and

simplex because of limitations in our compiler parallelization, so those codes are limited by a lack of parallelism. We simply took the maximum of the number of processors (4) and the idealized parallelism we measured in Chapter 4. For the multithreaded case, we included both loop and DAG parallelism, whereas for the SPMD case, we only included loop parallelization.

Note that this bar is optimistic – in fact, the real exploitable parallelism could be less than shown in the graphs for a number of reasons. For example, suppose that the idealized parallelism is 5, but that there is a sequential section which accounts for 1/5 of the running time of the code. In this case, if the rest of the code is sped up by a factor of 4, the speedup is still only $\frac{1}{1/5+4/5\times1/4} = 5/2 = 2.5$. In other cases, it might be possible to speed up the code by a factor of 4, even if the idealized parallelism is 5 – it depends how the program is parallelized. In Figures 6.4 and 6.5, we optimistically assume that we can achieve a speedup of 4, even for a code which has an idealized parallelism of 5.

The second and fifth bars on the graphs show any further limits due to code generation – since the parallel versions of the code are usually slower than the sequential ones when both are run on a single processor. We call this code generation overhead, and this shows the effect of the parallel calling convention, join synchronization and parallel loop schema. In general, the SPMD versions show less code generation overhead than the multithreaded versions because we are not taking advantage of DAG parallelism, so we don't have join synchronizations – furthermore, we choose to parallelize fewer loops in the SPMD version, because we can only exploit a single level of loop parallelism.

The third and fourth bars on the graphs show the actual measured speedups on four processors versus the efficient sequential version of the same code. Depending upon the application, the multithreaded or SPMD version may be faster. As expected, the SPMD scheduler is better for the structured codes, where only loop parallelism is exploited, and where loops and data structures are very regular.

6.4.1 Speedups limited by lack of parallelism

As we explained above, some codes are limited by a lack of parallelism found by the compiler. Some of the unstructured codes do not exploit loop parallelism, such as btree, fib, nqueens, qs, tree) and so are not amenable to SPMD scheduling.

Some of the idealized parallelism we measured is too fine-grain to be exploited effectively on an SMP. This is the case for codes like eigen-jacobi, simple-kt, and simple-new, which contain fine-grained loop parallelism which is either eliminated by the compiler, or not exploited by the run-time system when it realizes that the loop is too fine-grained.

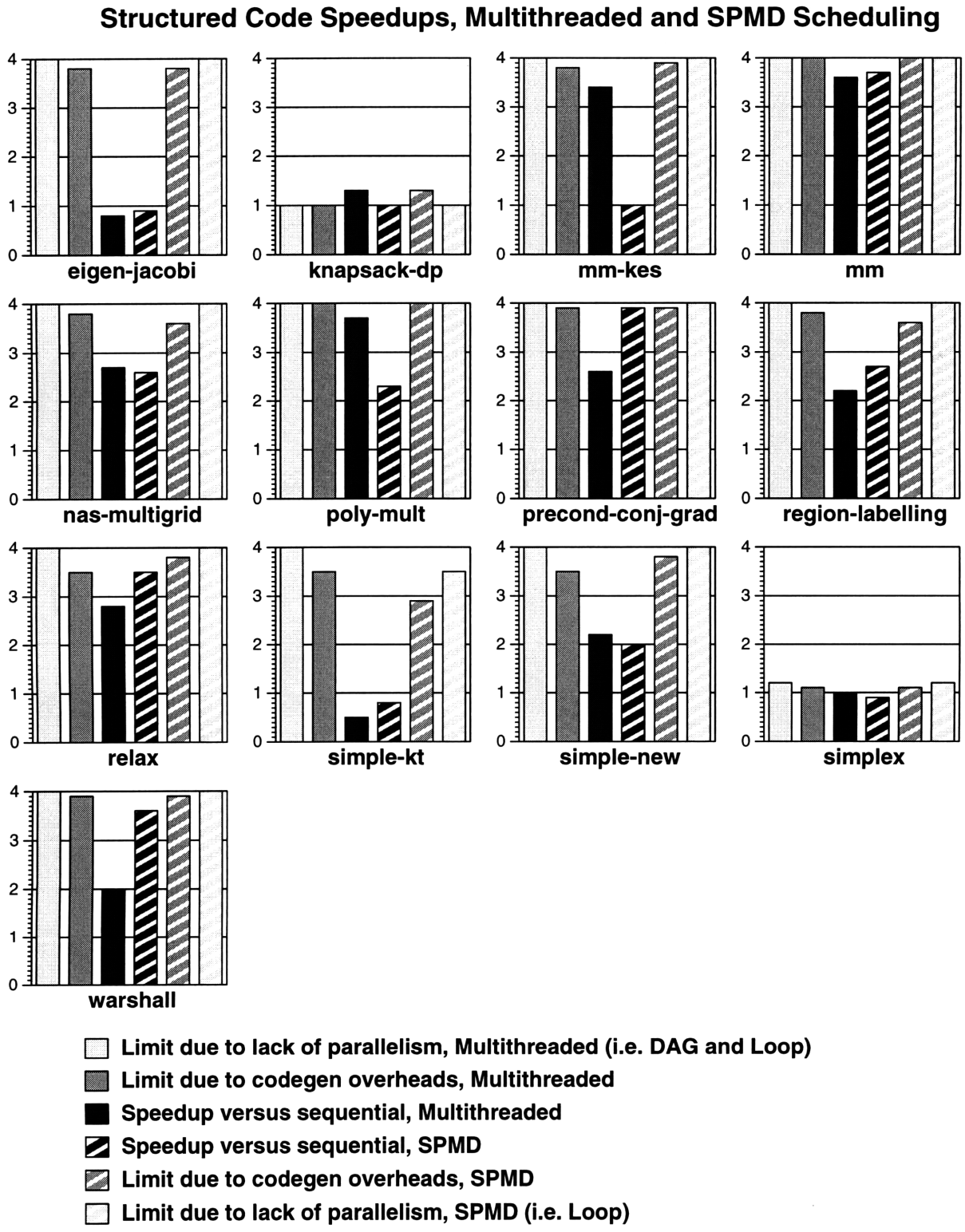

Figure 6.4: Speedups on 4 processors for structured applications under work-stealing multithreaded and SPMD scheduling. The outer bars show the limitations due to lack of parallelism and code generation overheads

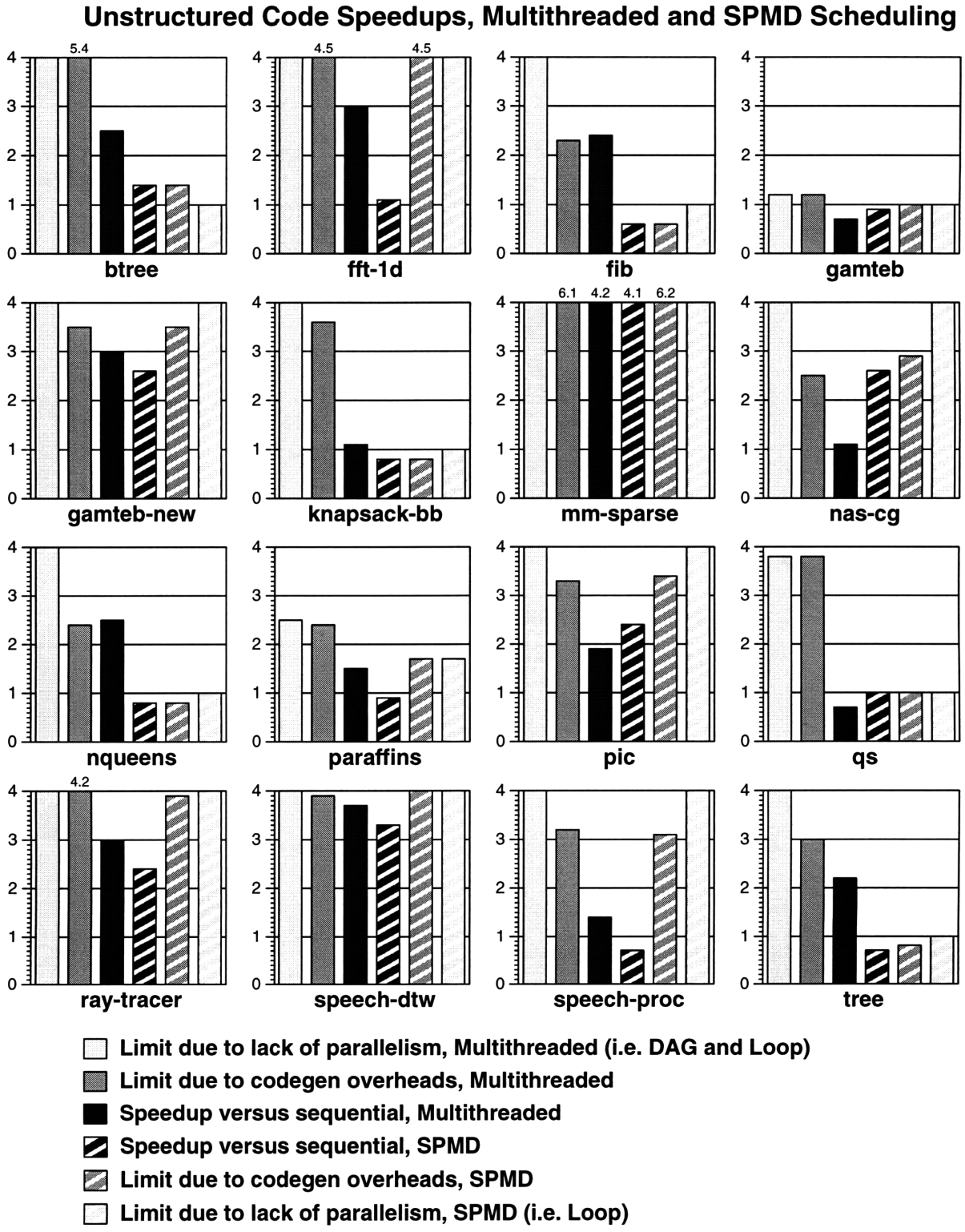

Figure 6.5: Speedups on 4 processors for structured applications under work-stealing multithreaded and SPMD scheduling. The outer bars show the limitations due to lack of parallelism and code generation overheads

For codes which exploit both loop and DAG parallelism such as mm-kes, fft-1d, and speech-proc the DAG parallelism is coarser-grained than the loop parallelism, and is more easily exploited by the multithreaded scheduler.

6.4.2 Speedups limited by code generation

Some codes have a decent amount of parallelism, but are hit by poor code generation. As we explained in Chapter 5, these are mostly codes which are procedure-call intensive, such as fib,nqueens, and tree, although some codes actually get faster when we use the parallel calling convention because of effects of the register windows on the sequential execution.

Some of these code generation overheads might be alleviated by using a more efficient parallel calling convention, such as that described by Mohr [52], Goldstein [33] or Plevyak [64].

6.4.3 Run-time system overheads

Overheads due to the run-time system are evident in many of the structured codes, where the work-stealing multithreaded run-time system is shown to be less efficient than a simpler SPMD scheduler. The overheads introduced by the multithreaded scheduler include overheads from excessive work stealing, poor load distribution, and suboptimal utilization of the memory system, because work is not scheduled identically across the processors as it is in the SPMD code.

Some unstructured codes which contain sequential sections suffer from excessive steal requests, such as btree and speech-proc. We try to alleviate these effects with backoff requests, but there is still some performance degradation.

6.4.4 Load imbalances

Certain codes which are loop-dominated show better performance under the multithreaded scheduler than the SPMD scheduler – for example, poly-mult, gamteb-new, and ray-tracer. This is due primarily to load imbalance across the loop, which is handled by the work-stealing policy of the multithreaded scheduler. Although for the data set we use as input for mm-sparse shows good speedup under both multithreaded and SPMD scheduling, other data sets could be arbitrarily bad for SPMD scheduling.

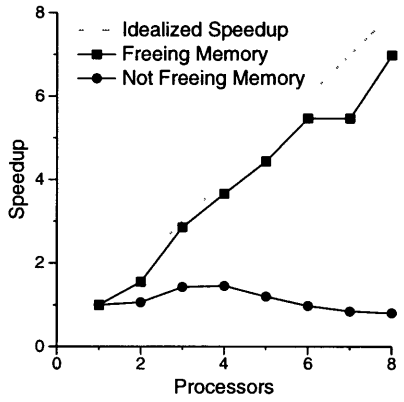

Figure 6.6: Speedups for "warshall 400" when freeing memory and not freeing memory, running under the SPMD scheduler. Note that the sequential times for this were 37.7 seconds when freeing memory and 55.2 when not freeing memory.

6.4.5 Effects of the memory system and freeing memory

When we do not free memory, we are forced to perform writes to newly allocated memory, which does not reside in the processor caches. A very extreme example of what can happen when there is no memory reclamation is shown in Figure 6.6, where we run the warshall code using SPMD scheduling on a problem size of 400, which creates 400 400×400 matrices. Without freeing the memory, all of the matrices are allocated from fresh memory, and with freeing memory, only two matrices are allocated.

First of all, not surprisingly, the sequential code runs faster when memory is reclaimed – 37.7 seconds versus 55.2 seconds. However, even scaling speedups from a slower sequential program, the non-freeing version shows very poor speedup because all writes must go to main memory. On our bus-based SMP, the memory bandwidth is shared among the processors, and so we hit a limit and even see some performance degradation. This is an extreme example – for many codes, we can have some speedup even when we do not free memory, but freeing memory keeps our working set size smaller, and puts less stress on the memory system.

Some of the poor speedup for the unstructured codes is due to this phenomenon – we do not have a garbage collector, and not as much memory is reclaimed by the programmer as in the structured codes, because it is more difficult to manually determine the safety of freeing memory, especially in a parallel environment.

For shared-bus SMP's, it is important to manage memory carefully, not just for increased sequential performance, but to allow for good parallel speedups as well. By limiting the effect of cache write-

through and cache thrashing each processor places less stress on the fixed resource of the shared-bus memory bandwidth.

For multithreaded execution and structured codes, the effect of memory management may not be as significant, unless the memory management is more closely tied with the scheduler.

6.5 Multithreaded versus SPMD scheduling

It should be clearer that multithreaded and SPMD scheduling are two points along a continuum: SPMD scheduling has proven efficient for many structured, Fortran-style codes, which we also see in our performance data. However, a more dynamic multithreaded scheduling also allows us to exploit a wider variety of programs, without sacrificing significant performance given a careful implementation.

For some very regular, structured applications, SPMD is better than multithreaded scheduling because the loop iteration load balance is regular, and the memory system can take advantage of locality. For these situations, the programmer could have the option of linking in the SPMD run-time system as a library, and using SPMD flags for the compiler. If the program changes at a later stage to be less structured, a simple re-compile and re-link could allow it to use the dynamic multithreaded scheduling. Alternatively, the run-time system and compiler could take an adaptive approach which determines at run-time which scheduling approach would be more effective.

Compiler analysis for parallelization and code generation does not differ significantly for multithreaded and SPMD execution if parallelization and code generation are designed with both in mind at the beginning. Id-S provides the programmer a high-level programming model such that he does not need to be concerned, to the first degree, with how the program is parallelized and scheduled.

Chapter 7

Conclusions

This thesis addresses many practical issues in exploiting structured and unstructured parallelism on small-scale shared-memory SMP's. The results fall into four main categories: language, parallelization, code generation, and scheduling. To an extent, the results from the different categories can be considered independently of each other. Although we have taken a particular path in terms of language semantics, approach to parallelization, code generation and scheduling, any of these components may be useful in and of themselves in a different context.

7.1 Non-strictness isn't used to write more expressive programs and non-strictness adds too much execution overhead

We have two major results regarding language semantics – the first regards *expressiveness* and the second regards *performance*.

Almost every Id program that we could find executed correctly (with very minor modifications) using a sequential top-down left-to-right evaluation order such as that used in C or Fortran. This indicates that non-strict eager language semantics do not provide significant *expressiveness* to programmers. It is fairly difficult to construct example programs which actually take advantage of non-strictness, and most of the programs we found which did take advantage of non-strictness were formulated explicitly to test non-strictness. This result concurs with Schauser and Goldstein's [70] analysis of how non-strictness is used in lenient programs – our methodology is more rigorous than theirs.

Using the best known partitioning algorithms and code-generation techniques developed for eager non-strict languages [72] [32], we showed that there is an instruction count overhead of 40-300%, not

counting support for atomicity in parallel execution. The overhead for non-strict execution is about evenly divided between presence checking for non-strict data structures, and multithreading overheads due to scheduling very short threads.

This overhead introduced by non-strictness makes it less attractive for exploiting parallelism on small-scale machines. Non-strictness is primarily needed for parallelism, but we describe methods of detecting DAG and loop parallelism from sequential programs which incur much less overhead. Non-strictness may be useful given hardware support for presence checking and fine-grained multithreading. Non-strictness may also be useful for certain codes which are difficult to parallelize, when a large machine size can overcome some of the execution overheads.

7.2 Sequential single-assignment simplifies parallelization

Imposing a sequential evaluation order on Id allows us to have a fast implementation, competitive with conventional imperative sequential languages such as C or Fortran. The *single-assignment* semantics for data structures allow us perform a very simple, traditional interprocedural side-effect analysis in order to parallelize loops without index analysis and to find opportunities for procedure-level parallelism. Single-assignment semantics eliminate output-dependences and anti-dependences at the source level. For most of our programs, we found ample idealized parallelism, although some index analysis would allow us to find some more.

By selectively parallelizing loops and procedures, we found the amount of idealized DAG parallelism, loop parallelism, and parallelism when both loop and DAG parallelism are exploited. Not surprisingly, programs which we categorized as "structured" contained primarily loop parallelism, while "unstructured" programs contained a mix of loop and/or DAG parallelism.

The interaction of loop and DAG parallelism is sometimes unintuitive – combining loop and DAG parallelism can sometimes provide an exponential amount more idealized parallelism than simply using loop or DAG parallelism in isolation.

7.3 Generating efficient multithreaded parallel code for SMP's

Once parallelization is complete, we generate parallel-ready code which contains hooks for parallel execution. These hooks include parallel procedure calls, join synchronization, and parallel loops. We

tested the incremental cost of each, and for the codes that we tested, most of the overhead was incurred for parallel procedure calls, but almost only for codes which were very procedure-call intensive. Join synchronization and parallel loops did not incur significant overheads due to code generation.

Our parallel loop chunking schema allows for dynamic multithreaded scheduling of loop iterations without the large number of activation frames which would be used in a straightforward recursive divide-and-conquer implementation.

7.4 Multithreaded scheduling can be as efficient as SPMD scheduling for structured codes, while handling DAG parallelism and unbalanced loop parallelism better than SPMD

Using the similar parallelization and code-generation techniques, we were able to generate code which could be scheduled under SPMD and work-stealing multithreaded scheduling. SPMD scheduling proved to be more efficient for very structured "Fortran-style" codes, because of caching effects and good load balancing. However, even for codes which are extremely well-suited for SPMD, the dynamically scheduled multithreaded implementation typically performed within 30% of the SPMD implementation, and often at almost the same performance.

SPMD is not useful for codes which primarily have DAG parallelism, and multithreaded execution also performs better for codes which are loop-dominated, but where the work is not evenly distributed across the loop iterations.

The SPMD scheduler is a slight modification of the multithreaded scheduler and the SPMD code generation is also only slightly different from the multithreaded code generation.

For certain applications, a pure SPMD implementation may be more efficient, but our work shows that a multithreaded implementation is more versatile. Compiler and run-time infrastructure can be shared, and the programmer can be free to experiment with either policy simply by changing some compiler and run-time library options.

7.5 Memory management increases single-processor performance and increases speedups

For implementations on bus-based SMP's, memory management is important in obtaining good speedups because some programs become memory-bus limited if they are always writing to "new" memory. Our results show that not only is single-processor performance better when memory is reclaimed, but those programs also show much better speedups, even relative to a faster single-processor performance.

7.6 Future work

We had to make many engineering decisions which allowed us to answer the questions we were interested within given time constraints. Some obvious improvements to the system would include the following:

- Support for interprocedural analysis with separate compilation.
- Garbage collection [1] and/or compiler-directed memory deallocation [44].
- More Fortran-style index analysis and loop transformations.
- A lighter-weight parallel calling convention.
- A debugger for parallel multithreaded execution.
- A profiler for parallel multithreaded execution.

In addition, some future work which is related to, but not directly based on the language/compiler/run-time system we describe in this thesis include the following.

7.6.1 Parallelizing conventional languages

Although we learned a significant amount from parallelizing a single-assignment language, parallelizing conventional imperative languages would have a much larger impact. Our parallelism studies show that many of the structured programs are amenable to Fortran-style parallelization, while many of the unstructured programs have significant parallelism even when the compiler assumes imperative semantics because of the *functional programming style* which is used, where frequent heap allocations mask side-effects which would otherwise limit parallelization.

Java is a language which may encourage programmers to use a *functional style* because of its language support for heap allocations and garbage collection. Furthermore, Java does not have arbitrary pointer aliasing as C does, making it a more attractive target for parallelization. Java's strong typing may also disambiguate the effects of some side-effects, and compiler transformations which turn an imperative program into a single-assignment program may also reveal more parallelism.

7.6.2 Integration with software distributed shared memory

Our system was developed for a hardware-supported shared memory computer, but it would be relatively straightforward to re-target it for a software-based distributed shared memory system [69] [73]. Such a system might have much simpler cache-coherence protocols than for a general imperative language, because the protocols could take advantage of the single-assignment semantics [11].

7.6.3 Single-chip SMP as an alternative to wider superscalars and VLIW's

Although the machines that we ran on for this thesis are extremely expensive, high-end servers, much cheaper SMP's are also available. Furthermore, the most promising direction may be in *single-chip SMP's* [79] [54] [60]. By having the SMP on a single-chip, system design costs could be significantly lowered, truly bringing parallel computing to the desktop for the mass of users.

A single-chip SMP, coupled with parallelizing compiler technology, could prove to be a realistic alternative to higher performance to wider superscalars or VLIW's. A single-chip SMP could be used for running multiple jobs in parallel, or a single parallel job, or even multiple parallel jobs, given some more progress in operating systems and run-time systems. Rather than an alternative to VLIW, a single-chip SMP might have multiple narrower VLIW's, rather than a single wide VLIW. Superscalar architectures are beginning to reach their limits in complexity, and the additional payoff in functional units is not necessarily worth the chip real estate.

Many of the programs which are amenable to VLIW compiler techniques such as software pipelining can also be handled with SMP-style architectures and parallelizing compilers. DAG-parallelism is difficult or impossible to exploit on VLIW-style processors.

The technology is available today to implement 2- or 4-processor single-chip SMP's today, and the design issues on the hardware side are fairly straightforward. The enabling technologies for this approach are the compiler and run-time system which we have addressed in this thesis.

Bibliography

[1] Shail Aditya, Christine H. Flood, and James E. Hicks. Garbage collection for a strongly-typed languages using run-time type reconstruction. In *Lisp and functional Programming*, June 1994.

[2] Agarwal, Anant and Bianchini, Ricardo and Chaiken, David and Johnson, Kirk and Kranz, David and Kubiatowicz, John and Lim, Beng-Hong and Mackenzie, Ken and Yeung, Donald. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of ISCA*, 1995.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, pages 1–6, 1990.

[5] Saman Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford, 1997.

[6] Amarasinghe, S.P. and Anderson, J.M. and Lam, M.S. and Tseng, C.W. The SUIF Compiler for Scalable Parallel Machines. In *Proceedings of the Seventh SAM Conference on Parallel Processing for Scientific Computing*, 1995.

[7] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[8] Boon s. Ang and Derek Chiou. StarT-Voyager: Hardware Engineering Specification. Technical Report CSG Memo 385, MIT Laboratory for Computer Science, 1997.

[9] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 25-29 1987. TTDA, parallel processing, synchronization, latency, also CSG Memo 226-6.

[10] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[11] Arvind and Xiaowei Shen. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. Technical Report CSG Memo 398, MIT Computation Structures Group, 1997.

[12] Arvind, G. K. Maa and D. E. Culler. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 1987.

[13] Uptal Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.

[14] Utpal Banerjee. *Loop Transformations for Restructuring Compiler*. Kluwer Academic Publishers, 1993.

[15] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth POPL*, January 1979.

[16] Blumofe, Robert D. and Joerg, Christopher F. and Kuszmaul, Bradley C. and Leiserson, Charles E. and Randall, Keith H. and Zhou, Yuli. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of PPoPP '95*, 1995.

[17] Cann, David. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1990.

[18] Chiou, Derek. Using GCC as an Efficient, Portable Back-End. In *Proceedings of the MIT Student Workshop for Scalable Computing*, 1995. http://www.csg.lcs.mit.edu:8001/Users/derek/.

[19] K. D. Cooper and K. Kennedy. Interprocedural Side-Effect Analysis in Linear Time. In *PLDI*, pages 57–66, July 1988. http://softlib.rice.edu/MSCP/publications.html.

[20] K. D. Cooper and K. Kennedy. Fast Interprocedural Alias Analysis. In *POPL*, pages 49–59, January 1989. http://softlib.rice.edu/MSCP/publications.html.

[21] Cooper, Eric C. and Draves, Richard P. C Threads. Technical report, Carnegie Mellon University, September 11 1990.

[22] Coorg, Satyan. Partitioning Non-strict Languages for Multi-threaded Code Generation. Master's thesis, MIT, May 1994.

[23] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[24] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of ASPLOS*, pages 164–175, Santa Clara, California, 1991.

[25] D.E. Culler, S.C. Goldstein, K.E. Schauser, and T. von Eiken. TAM – a Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *POPL*, pages 25–35, 1989.

[27] John Elison, Eleftherios Koutsofios, and Stephen North. graphviz – tools for viewing and interacting with graph diagrams. http://www.research.att.com/sw/tools/graphviz/.

[28] J. R. Ellis. *Bulldog: a compiler for VLIW architectures*. MIT Press, 1985.

[29] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Shared Filaments: Efficient support for fine-grain parallelism on shared-memory multiprocessors. Technical Report TR 93-13, University of Arizona, April 1993.

[30] Vincent W. Freeh, David K. Lownthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstation. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, November 1994.

[31] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. In *PLDI*, pages 15–29, 1991.

[32] Seth Copen Goldstein. The Implementation of a Threaded Abstract machine. Master's thesis, UC Berkeley, 1994.

[33] Seth Copen Goldstein. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 1996.

[34] Seth Copen Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, UC Berkeley, 1997.

[35] Seth Copen Goldstein, Klaus Eric Schauser, and David Culler. Lazy threads, stacklets, and synchronization. In *Proceedings of POOMA '94*, 1994.

[36] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[37] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994. To appear.

[38] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. Liao, and M. S. Lam. Interprocedural Analysis for Parallelization. In *Proceedings of Supercomputing '95*, December 1995.

[39] Mary Wolcott Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, April 1991.

[40] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[41] W. L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, October 1989.

[42] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*, 1995. also http://www.cs.mu.oz.au/mercury/papers.html.

[43] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. *Journal of Parallel and Distributed Computing*, 18(3):273–300, 1993.

[44] James E. Hicks Jr. *Compiler-directed Storage Reclamation Using Object Lifetime Analysis*. PhD thesis, MIT, 1992. MIT/LCS/TR-555.

[45] Kei Hiraki, Kenji Nishida, Satoshi Sekiguchi, Toshio Shimada, and Toshitsugu Yuba. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal of Information Processing*, 10(4):219–226, 1987.

[46] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, MIT Laboratory for Computer Science, January 1996.

[47] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE93-05-06, University of Washington, 1993.

[48] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, 1978.

[49] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford flash multiprocessor. In *ISCA*, pages 302–313, April 1994.

[50] Peter S. Magnusson. Simulation of Parallel Hardware. In *Proceedings of MASCOTS*, 1993. http://www.sics.se./simics/.

[51] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and Exact Data Dependence Analysis. In *PLDI*, 1991.

[52] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy task creation a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Computing*, 2(3):264–80, July 1991.

[53] F. Mueller. A library implementation of posix threads under unix. In *Proceedings of USENIX Conference, Winter '93*, pages 29–41, 1993.

[54] Basem A. Nayfeh and Kunle Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. 21st Annual Symposium on Computer Architecture*, pages 166–175, May 1994.

[55] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

[56] R.S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proc. 6th. Ann. Wkshp. on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993. Springer-Verlag. also http://www.research.digital.com/CRL/personal/nikhil/pHluid/home.html.

[57] R.S. Nikhil. Cid: A Parallel "Shared-memory" C for distributed Memory Machines. In *7th Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994. http://www.research.digital.com/CRL/personal/nikhil/cid/home.html.

[58] Michael D. Noakes, Deborah Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of ISCA*, 1993. ftp://ftp.ai.mit.edu/pub/cva/isca.ps.Z.

[59] Numerical Aerospace Simulation Facility. NAS Parallel Benchmarks. Web Site. http://science.nas.nasa.gov/Software/NPB/.

[60] Kunle Olukotun, Jules Bergmann, Kun-Yung Chang, and Basem Nayfeh. Rationale, design and performance of the hydra multiprocessor. Technical Report STAN//CSL-TR-94-645, Stanford University, Computer Systems Laboratory, November 1994. [Adminitrivia V1/Prg/19950209].

[61] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[62] Gregory Michael Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. MIT Press, 1989.

[63] John Plevyak. *Optimization of Object-Oriented and Concurrent Programming*. PhD thesis, UIUC, 1996. also: http://www-csag.cs.uiuc.edu/papers/jplevyak-thesis.ps.

[64] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew A. Chien. A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers. In *Supercomputing*, 1995. also: http://www-csag.cs.uiuc.edu/papers/stack-sc95.ps.

[65] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 1992. also: ftp://ftp.cs.umd.edu/pub/omega/techReports/non-TRs/omega/.

[66] S. Sakai, Y Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of ISCA*, pages 46–53, 1989.

[67] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

[68] Mitsuhisa Sato, Yuetzu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, and Yasuhito Komura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proceedings of the 19th ISCA*, pages 146–155, Gold Coast, Australia, May 1992.

[69] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach to supporting fine-grain shared memory. In *ASPLOS VII*, pages 174–185, October 1996.

[70] Klaus E. Schauser and Seth C. Goldstein. How Much Non-strictness do Lenient Programs Need? In *Proceedings of FPCA*, June 1995.

[71] Schauser, Klaus E. *Compiling Lenient Languages for Parallel Asynchronous Execution*. PhD thesis, UC Berkeley, 1994.

[72] Schauser, Klaus E. and Culler, David, E. and Goldstein, Seth C. Separation Constraint Partitioning – A New Algorithm for Partitioning Non-strict Programs into Sequential Threads. In *POPL '95*, 1995.

[73] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS VI*, pages 297–306, October 1994.

[74] Andrew Shaw. Performance Tuning Scientific Codes for Dataflow Execution. In *PACT*, 1996.

[75] Olin Shivers. Control-flow analysis in scheme. In *PLDI*, June 1988.

[76] B. J. Smith. Architecture and applications of the HEP multiprocessor computer. In *Real-Time signal processing IV*, volume 298, pages 241–248, 1981.

[77] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser95a, Thorsten von Eiken, David Culler, and William J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proceedings of ISCA*, 1993.

[78] Richard Stallman. Using and porting the GNU CC compiler, 1989-1995. The Free Software Foundation.

[79] Masafumi Takahashi, Hiroyuki Takano, Emi Kaneko, and Seigo Suzuki. A Shared-bus Control Mechanism and a Cache Coherence Protocol for a High-performance On-chip Multiprocessor. In *HPCA*, pages 314–322, 1996.

[80] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[81] Kenneth R. Traub. *Sequential Implementation of Lenient Programming Languages*. MIT Press, 1988.

[82] Kenneth R. Traub. Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Functional Programming Languages and Computer Architectures*, pages 75–88. ACM Press, 1989.

[83] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *19th ISCA*, pages 256–266, Gold Coast, Australia, May 1992.

[84] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of PLDI*, pages 1–12, June 1995.

[85] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, 1992.

[86] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.

[87] Donald Yeung and Anant Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, pages 187–197, May 1993.