SUDS: Automatic Parallelization for Raw Processors

by

Matthew Ian Frank

B.S., University of Wisconsin (1994)
S.M., Massachusetts Institute of Technology (1997)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2003

Certified by . . . .
Anant Agarwal
Professor
Thesis Supervisor

Certified by . . . .
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

SUDS: Automatic Parallelization for Raw Processors

by

Matthew Ian Frank

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2003, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A computer can never be too fast or too cheap. Computer systems pervade nearly every aspect of science, engineering, communications and commerce because they perform certain tasks at rates unachievable by any other kind of system built by humans. A computer system's throughput, however, is constrained by that system's ability to find concurrency. Given a particular target work load the computer architect's role is to design mechanisms to find and exploit the available concurrency in that work load.

This thesis describes SUDS (Software Un-Do System), a compiler and runtime system that can automatically find and exploit the available concurrency of scalar operations in imperative programs with arbitrary unstructured and unpredictable control flow. The core compiler transformation that enables this is *scalar queue conversion*. Scalar queue conversion makes scalar renaming an explicit operation through a process similar to closure conversion, a technique traditionally used to compile functional languages.

The scalar queue conversion compiler transformation is speculative, in the sense that it may introduce dynamic memory allocation operations into code that would not otherwise dynamically allocate memory. Thus, SUDS also includes a transactional runtime system that periodically checkpoints machine state, executes code speculatively, checks if the speculative execution produced results consistent with the original sequential program semantics, and then either commits or rolls back the speculative execution path. In addition to safely running scalar queue converted code, the SUDS runtime system safely permits threads to speculatively run in parallel and concurrently issue memory operations, even when the compiler is unable to prove that the reordered memory operations will always produce correct results.

Using this combination of compile time and runtime techniques, SUDS can find concurrency in programs where previous compiler based renaming techniques fail because the programs contain unstructured loops, and where Tomasulo's algorithm fails because it sequentializes mispredicted branches. Indeed, we describe three application programs, with unstructured control flow, where the prototype SUDS system, running in software on a Raw microprocessor, achieves speedups equivalent to, or better than, an idealized, and unrealizable, model of a hardware implementation of Tomasulo's algorithm.

Thesis Supervisor: Anant Agarwal
Title: Professor

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I believe that engineering is a distinctly social activity. The ideas in this thesis were not so much "invented" as they were "organically accreted" through my interactions with a large group of people. For the most part those interactions took place within the Computer Architecture Group at the Laboratory for Computer Science at MIT. That the Computer Architecture Group is such a productive research environment is a testament, in large part, to the efforts of Anant Agarwal. Anant somehow manages to, simultaneously, find and attract brilliant and creative people, keep them focused on big visions, and acquire the resources to turn those visions into realities.

Anant also has the incredible ability to judge the difference between an engineering advance and a "tweak," between the long term and the short. He's occasionally wrong, but I've lost count of the number of times that I stubbornly went my own way only to discover, sometimes years later, that he had been right in the first place. If I've learned anything about how to do relevant computer engineering research, then I learned it from Anant.

A student who is as bad at taking advice as I am actually requires two advisors, so that he can be given a full amount of advice, even when he is ignoring half of what he is told. Saman Amarasinghe took on the thankless task of trying to keep me directed and focused. He listened to my constant griping and complaining about our research infrastructure, and then patiently taught me how to use it correctly. Saman, somehow, always knows when to push me, when to back off, and when to give me a kick in the pants. The fact that I am graduating at all is as much a testament to Saman's will power as it is to my own.

Saman was also the main sounding board for most of the ideas described in this thesis. Saman was the first to realize the importance and novelty of the program transformations that I had been doing "by hand" for several years, and convinced me, at some point in 2000 or 2001, that I needed to automate the process. The result of that suggestion is Chapters 3 through 6.

When a large research group works together for many years people's ideas "rub off" on each other, and it becomes difficult (for me) to attribute specific ideas to their originators. The computer architecture group at MIT is huge, and thus I've had the opportunity to interact with a large number of people, most of whom have had an influence on my work.

Ken Mackenzie acted as my de facto advisor when I first came to MIT, (before Saman arrived, and while Anant was on leave). Ken taught me how to do collaborative research, and, more importantly, also honest research.

Jonathan Babb, in many ways, initiated the Raw project, both in terms of the influence of his Virtual Wires logic emulator on Raw's communication networks, and with his interest in reconfigurable computing. Towards the beginning of the project we had daily conversations (arguments) that would often last six or eight hours. Almost everything I know about computer aided design I learned from Jon.

Michael Taylor lead the Raw microprocessor implementation effort, without which this work would have been impossible. In addition, I always went to Mike for honest assessments of what I was doing, and for help in making my runtime implementations efficient. Moreover, Mike is largely responsible for pointing out the usefulness of Dataflow techniques on my work. In particular, the deferred execution queues, described in Chapter 3, are influenced by the communication channels that Mike created, and discussed with me at length, while he was doing some work on mapping Dataflow graphs to Raw during the early stages of the project.

Walter Lee and I learned the SUIF compiler infrastructure together. In addition to his massive contributions to that infrastructure (which is also used by my compiler), Walt has been my main sounding board for compiler implementation issues. I don't think my compiler work would have succeeded without his patient guidance, and excellent instincts, on what will work and what won't.

Andras Moritz and I had an extraordinarily productive collaborative relationship during the time we were at MIT together. In addition to actively contributing to early versions of the SUDS runtime system, the work Andras and I did together on software based cache controllers influenced all aspects of the SUDS runtime system.

I had the pleasure, during my last few years at MIT, of sharing an office with Nathan Shnidman. In addition to being a good friend, and contributing to a fun work environment, Nate was always willing to listen to me rant about what I was working on. Nate was also always willing to tell me about what he was working on, and even better, explain it so I could understand it. In the process he taught me just about everything I know about communication systems and signal processing.

Kevin Wilson and Jae-Wook Lee both contributed to early versions of the SUDS runtime system. My conversations with them informed many of the implementation choices in the final prototype. Sam Larsen and Radu Rugina have each contributed pieces of the SUIF compiler infrastructure that I use. Sam, like Walt, has been a constant source of good advice on compiler implementation issues. Radu contributed the near production quality pointer analysis that all of us in the Raw project have depended on. Bill Thies has also patiently let me rant about whatever was on my mind, and in return has educated me about array transformations. Numerous technical conversations with Krste Asanovic, Jason Miller, David Wentzlaf, Atul Adya, Emmett Witchel, Scott Ananian, Viktor Kuncak, Larry Rudolph and Frans Kaashoek have informed my work in general. Frans Kaashoek both served on my thesis committee and also provided numerous helpful comments that improved the presentation of the dissertation.

I don't believe any of my work would have been possible had I not been working with a group actually implementing a microprocessor. In addition to many of the people mentioned above, that implementation effort involved Rajeev Barua, Faye Ghodrat, Michael Gordon, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, Albert Ma, Mark Seneski, Devabhaktuni Srikrishna, Mark Stephenson, Volker Strumpen, Elliot Waingold and Michael Zhang.

During the last year several of my colleagues at the University of Illinois have been particularly helpful. Much of the presentation, in particular of the introductory material, and of the compiler transformations, was informed by numerous

conversations with Sanjay Patel, Steve Lumetta and Nick Carter.

I have also benefited from a substantial amount of technical and administrative support, which I simply could not have handled on my own. Among others, this support was provided by Rachel Allen, Scott Blomquist, Michael Chan, Cornelia Colyer, Mary Ann Ladd, Anne McCarthy, Marilyn Pierce, Lila Rhoades, Ty Sealy, Frank Tilley, Michael Vezza and Shireen Yadollahpour.

Financially, this work was supported in part by an NSF Graduate Research Fellowship and NSF and Darpa grants to the Fugu and Raw projects. While I was in graduate school, my wife, Kathleen Shannon, earned all of the money required for our living expenses. My tuition for the last year was funded through a fellowship generously arranged by Anant Agarwal through the Industrial Technology Research Institute/Raw Project Collaboration. In addition, I thank my colleagues in the Electrical and Computer Engineering Department at the University of Illinois for their patience while I finished this work.

Finally, the emotional support provided by a number of people has been more important to me than they, perhaps, realize. My parents unconditional love and support has been crucial. In Boston, Andras Moritz, Mike Taylor, Nate Shnidman, Tim Kelly and Jeffery Thomas were particularly supportive. In Illinois Nick Carter, Sanjay Patel, Steve Lumetta and Marty Traver provided a vital support network.

Most of all, I have relied on my wife, Kathleen Shannon, and my children, Karissa and Anya. Their love has carried me through this period. Without them none of this would have been possible, or worth doing.

I dedicate this work to the memories of my grandfathers, who taught me, by example, how to dream big dreams and then make them happen.

Contents

Chapter 1

Introduction

Computer programmers work under a difficult set of constraints. On the one hand, if the programs they produce are to be useful, they must be correct. A program that produces an incorrect result can be, literally, deadly. A medical radiation therapy machine that occasionally delivers the wrong dose can kill the patient it was intended to heal [76].

On the other hand, to be useful a program must also produce its results in a timely manner. Again, the difference can be critical. Aircraft collision avoidance systems would be useless if it took them longer to detect an impending collision than for the collision to occur. Similarly, today's vision and speech recognition systems work too slowly to be used as tools for interacting with human beings.

After correctness, then, the computer engineer's main area of focus is the "speed" or "performance" of the computer system. That this is the case, (and should remain so), is a consequence of the fact that performance can often be traded for other desirable kinds of functionality. For example, in the low-power circuits domain, improved system throughput enables reduced power consumption through voltage scaling [23]. In the software engineering domain, the widely used Java programming language (first released in 1995) includes garbage collection and runtime type checking features that were considered too expensive when the C++ programming language was designed (circa 1985) [112].

Unfortunately, the twin goals of correctness and speed conflict. To make it more likely that their programs are correct, programmers tend to write their programs to run sequentially, because sequential programs are easier to reason about and understand. On the other hand, the rate at which a computer can execute a program is constrained by the amount of concurrency in the program.

One solution to this conundrum is to allow the programmer to write a sequential program in a standard imperative programming language, and then automatically convert that program into an equivalent concurrent program by techniques that are known to be correct. There are two relatively standard approaches for converting sequential imperative programs into equivalent concurrent programs, Tomasulo's algorithm [117, 57, 104, 83, 105], and compiler based program restructuring based on a technique called scalar expansion [68].

Each of these techniques presents the architect with a set of tradeoffs. In particular, Tomasulo's algorithm guarantees the elimination of register storage dependences, and is relatively easily extended to speculate across predictable dependences, but does so at the cost of partially sequentializing instruction fetch. On the other hand, compiler based restructuring techniques can find all of the available fetch concurrency in a program, and have relatively recently been extended to speculate across predictable dependences, but have not, prior to this work, been capable of eliminating register storage dependences across arbitrary unstructured control flow. *The SUDS automatic parallelization system eliminates the tradeoffs between Tomasulo's algorithm and compiler based program restructuring techniques.*

Informally, renaming turns an imperative program into a functional program. Functional programs have the attribute that every variable is dynamically written at most once. Thus functional programs have no anti- or output- dependences. The cost of renaming is that storage must be allocated for all the dynamically renamed variables that are live simultaneously. The particular problem that any renaming scheme must solve, then, is how to manage the fixed, and finite, storage resources that are available in a real system.

Tomasulo's algorithm deals with the register storage allocation problem by taking advantage of its inherently sequential fetch mechanism. That is, if Tomasulo's algorithm runs out of register renaming resources, it can simply stall instruction fetch. Because instructions are fetched in-order, and sequentially, the previously fetched instructions that are currently using register renaming resources are guaranteed to make forward progress and, eventually, free up the resources required to restart the instruction fetch mechanism.

Traditional compiler based renaming techniques, like scalar expansion, take a different approach, renaming only those scalars that are modified in loops with structured control flow and loop bounds that are compile time constants. This enables the compiler to preallocate storage for scalar renaming, but limits the applicability of this technique to structured loops that can be analyzed at compile time.

The SUDS approach, in contrast, is to rename speculatively. The SUDS compile time scheduler uses a compile time technique called *scalar queue conversion* to explicitly rename scalar variables. Scalar queue conversion dynamically allocates storage for renamed scalars, and thus can rename across arbitrary control flow (even irreducible control flow). Unlike Tomasulo's algorithm, which depends on sequential fetch to avoid overflowing the finite renaming resources, SUDS fetches instructions from different parts of the program simultaneously and in parallel. As a result, scalar queue conversion's dynamically allocated renaming buffers may overflow.

SUDS deals with these overflow problems using a checkpointing and repair mechanism. SUDS periodically checkpoints machine state, and if any of the renaming buffer dynamic allocations should overflow, SUDS rolls back the machine state to the most recent checkpoint and reexecutes the offending portion of code without renaming. In the (hopefully) common case the renaming buffers do not overflow.

Because SUDS can fetch multiple flows of control simultaneously, and even when the control flow graph is unstructured or irreducible, SUDS exploits concurrency that neither Tomasulo's algorithm nor previous compiler based renaming techniques can exploit. Despite the fact that SUDS implements both scalar renaming and speculative checkpoint/repair in software, it is able to achieve speedups equal to, or better than, an idealized (unrealizable) hardware implementation of Tomasulo's algorithm.

The next section explains why finding concurrency is fundamental to computer system performance. Section 1.2 describes the SUDS approach to finding concurrency. Section 1.3 describes the specific technical contributions of this work.

1.1 Technology Constraints

Why is automatic parallelization important? There are two ways to make a computer system "faster." The first is to reduce the amount of time to execute each operation. This goal can only be achieved by improved circuit design and improved fabrication techniques.

The second technique is to increase the throughput of the system. This is the domain of the computer architect. In this section we will point out that the only way to increase system throughput is to increase the number of independent operations simultaneously in flight. And we will further demonstrate that technology constraints demand that system throughput can only increase sublinearly in the amount of available parallelism. Thus, *architectural performance improvements depend on our ability to find parallelism in real world workloads.*

One method for demonstrating this claim is to invoke Little's Law [78],

$$X = N/R. \tag{1.1}$$

Little's Law says that the system throughput, (number of operations completed per unit time), X, is equal to the quotient of the number of independent operations simultaneously active in the system, N, and the time required to complete each operation, R.

Assuming that we can increase parallelism without increasing operation latency, (*i.e.*, $R = O(1)$, which is not true, as we will see subsequently), then the achievable system throughput is limited to the number of independent operations that can run simultaneously. That is, *at best,* $X \propto N$.

Pipelining is one popular architectural technique for increasing system throughput. In a pipelined design each fundamental operation is divided into multiple stages of approximately equal latency, and latches are placed between the stages. Assume that the time to execute each fundamental operation is t_f (*i.e.*, the time for just the combinational logic) and the time to latch a result is t_l. Then if we divide the fundamental operation into N pipeline stages we increase the latency of each operation from t_f to $Nt_l + t_f$. Thus by Little's Law

$$X_{\text{pipeline}} = \frac{N}{Nt_l + t_f}.$$

We can conclude two things from this derivation. First, as N grows, pipelining improves throughput only to the limit of

$$\lim_{N \to \infty} X_{\text{pipeline}} = \frac{1}{t_l}.$$

That is, pipelining throughput is limited to the maximum rate at which we can cycle a latch in a particular technology.

Second, suppose we desire to pipeline until we achieve a desired fraction, f_x, where $0 < f_x < 1$, of the maximum throughput $1/t_l$. Then

$$\frac{f_x}{t_l} = \frac{N}{Nt_l + t_f}$$

and so

$$N = \frac{t_f}{t_l}\frac{f_x}{1 - f_x}.$$

The fraction $f_x/(1 - f_x)$ approximates a linear function when f_x is close to 0, but grows to infinity as f_x approaches 1. Thus only a small fraction (about half) of the maximum pipelining throughput is achievable, unless we can find a way to grow N, the available operation parallelism, *hyperbolically*. Recent microprocessor designs have come close to the limits of the linear regime [81, 3, 11], and thus future designs will need to find another approach if they are to achieve greater system throughput.

A second approach to increasing system throughput is to increase the number of functional units. If it were the case that we could fetch the operands for each operation in constant time, then we would be able to increase throughput linearly as we increased the number of independent operations available in the system. Unfortunately, this argument depends on the assumption that the functional units are executing work that is completely independent and that they never communicate. If even a small constant fraction of the results produced by each functional unit need to be communicated to another arbitrarily chosen functional unit, then we need to account for these communication costs in our calculation.

Recent analysis of technology scaling trends shows that communication costs will be the dominant concern in computer architecture design by the year 2013 [81, 3, 11]. For example, in 35nm technology, and assuming a clock cycle time equivalent to 8 fan-out-of-4 gate delays it is expected that it will cost more than two hundred cycles to propagate a signal all the way across a chip. We can accurately model these assumptions with the following simple abstract rules:

1. The propagation of information takes time linear in distance traveled.

2. The universe is finite dimensional.

3. Storing information consumes area linear in the quantity of information stored.

Thus, the area of the entire system is at least proportional to N, where N is the number of simultaneously active independent operations. An arbitrary communication operation in the system takes time proportional to the distance traveled, which, on a two-dimensional computer chip, will on average, be proportional to $\sqrt{N}$.[1] Plugging the result $R = \sqrt{N}$ into Little's Law we are led to the conclusion that at best[2]

$$X \propto \sqrt{N}.$$

Thus, to improve computer system throughput by a factor of two, one must find at least four times as much parallelism. Put another way, parallelism is the computer architect's constrained resource, and thus improving parallelism is the most critical component to future improvements in computer system throughput.

[1] Online locality management techniques, like caching, might be able to reduce this distance somewhat, but it is an open question whether the benefits would be substantial. Even *offline* techniques, like VLSI circuit placement algorithms, typically produce results in the range $R \propto N^{0.1}$ to $R \propto N^{0.3}$ [73, 36].

[2] I can find no previous publication of this argument, but the designers of the Tera computer system were clearly aware of it before 1990 [5].

```
do
  s = f(i)
  if s
    t = g(i)
    u = h(i)
    *t = u
  i = j(i)
  v = k(i)
while v
```

Figure 1-1: An example program.

1.2 Finding Parallelism

How, then, are we to find the parallelism required to improve throughput in the next generation of computer architectures? The execution of a program can be viewed as a process of unfolding the dynamic dependence graph of that program. The nodes of this graph correspond to arithmetic operations that need to be performed, while edges in the graph correspond to a partial ordering of operations that needs to be enforced if the program is to produce the correct results. When viewed in this way, then the process of finding parallelism becomes a process of finding operations in the dynamic dependence graph that don't depend on one another. Much of the difficulty in finding parallelism in imperative programs comes from the fact that existing compilers and architectures build dependence graphs that are too conservative. They insert false dependence arcs that impede parallelism without affecting the correctness of program execution.

The SUDS automatic parallelization system relies on three basic principles:

1. Every imperative program can be converted into a functional program by making renaming explicit. A functional (*i.e.*, explicitly renamed) program has the attribute that *every variable is (dynamically) written at most once* thus functional programs have no anti- or output- dependences.

2. The flow dependences produced by following the single flow of control in

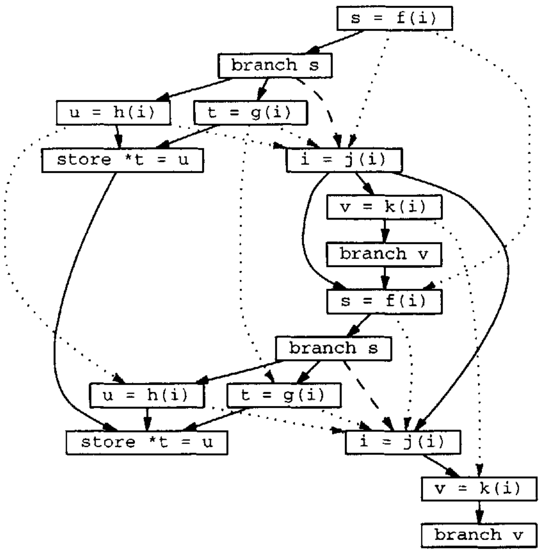
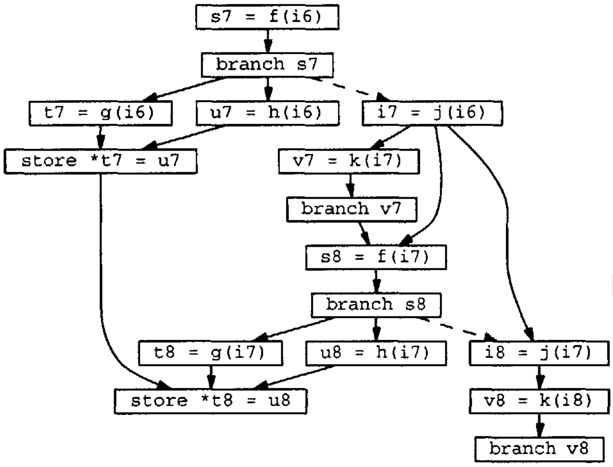


Figure 1-2: The conservative dynamic dependence graph generated from the code in Figure 1-1. Arcs represent dependences between instructions that must be honored for correct execution. Dotted arcs represent anti-dependences that can be removed through dynamic renaming. Dashed arcs represent flow dependences that can be removed through accurate control dependence analysis. The *height* of this conservative dynamic dependence graph is 12 nodes, because there is a path through the graph of length 12. The throughput of this program would be one iteration every six cycles.



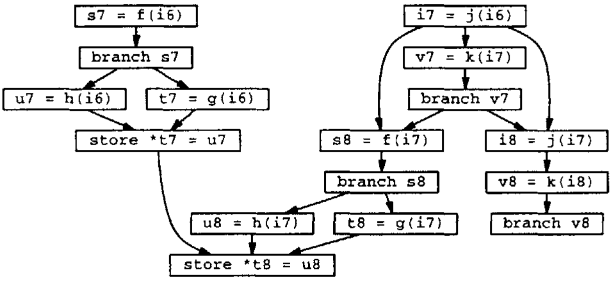


Figure 1-3: Dynamic renaming removes anti–dependences. The height of the graph has been reduced from the 12 nodes of the conservative dynamic dependence graph to 10 nodes. The throughput has been improved from one iteration every six cycles to one iteration every five cycles.

Figure 1-4: Control dependence analysis removes conservative branch-dependence arcs. The combination of dynamic renaming and control dependence analysis has reduced the height of the graph to 7 nodes. The throughput has been improved to one iteration every three cycles.

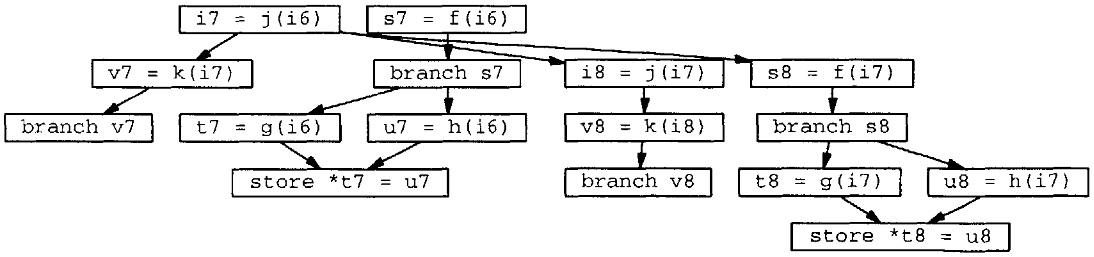


Figure 1-5: Speculation breaks predictable dependences. The graph height has been reduced to 5 nodes. The throughput has been improved to one iteration every cycle.

the standard control flow graph representation are more conservative than necessary. *Control dependence* analysis produces a more accurate, and sparser, representation of actual program structure that makes multiple flows of control explicit.

3. Many true-dependences (in particular those on data structures in memory) and control-dependences can be further eliminated by *speculation.*

Figure 1-1 shows an example of a simple loop with non-trivial dependences. Figure 1-2 shows the conservative dynamic dependence graph of two iterations of the loop. The figure is annotated with the dependences that limit parallelism. The variable i creates a *true-dependence,* because the value written to variable i in the first iteration is used in the second iteration. The reads of variables s, t, u and v in the first iteration create *anti-dependences* with the writes of the corresponding variables in the second iteration. In this conservative representation every operation is also *flow-dependent* on the branch that proceeds it. Finally, there is a *memory-dependence* between the potentially conflicting store operations in the two iterations. We can see by looking at the graph that, without any further improvement this loop can execute at a maximum rate of one iteration every six "cycles" (assuming that each instruction takes a cycle to execute).

Figure 1-3 shows the benefits of renaming to remove anti-dependences. Renaming creates a uniquely named location in which to hold each intermediate value produced by the program. Since each location is written exactly once the anti- and output-dependences are eliminated [57]. Renaming improves the throughput of the example loop from one loop iteration every six cycles to one loop iteration every five cycles.

Figure 1-4 shows the results of applying control dependence analysis [40, 30]. This eliminates the flow-dependence between the branch statement on variable `s` and later code (*e.g.*, the statement "`i = j(i)`") that execute irrespective of whether the branch is taken or not. The combination of renaming and control dependence analysis improves the throughput of the example loop from one loop iteration every six cycles to one loop iteration every three cycles.

Figure 1-5 illustrates what happens when two of the remaining dependences are eliminated using speculation techniques. While there is a true control dependence between the branch at the end of the first iteration and the execution of the code in the second iteration we can use traditional *branch speculation* techniques [103, 132] to parallelize across this dependence with high probability. The dependence between the stores in the two iterations is necessary in a conservative sense, in that the addresses in `t7` and `t8` could be the same under some program execution, but using *memory dependence speculation* [44] we can take advantage of the idea that probabilistically the addresses in `t7` and `t8` are different.

These speculative dependences are monitored at runtime. The system checkpoints the state occasionally and executes code in parallel, even though this may cause dependence violations that produce inconsistent states. The runtime system later checks for (dynamic) violations. If the runtime system finds any violations, execution is temporarily halted, the system state is restored to and restarted at the most recent checkpoint. If such violations are rare then the system achieves the parallelization benefits.

Figure 1-5 demonstrates that the combination of renaming, control dependence analysis and speculation have found a substantial amount of parallelism in the original code. While each iteration of the original loop includes eight operations, we can (conceptually) improve the throughput to one loop iteration every cycle, or eight instructions per cycle.

This thesis addresses the issues involved in the above example, in the context of SUDS (the Software Un-Do System), an "all-software" automatic parallelization system for the Raw microprocessor. SUDS performs explicit dynamic renaming by closure-converting C programs. SUDS exploits control independence by mapping control-independent code to independent branch units on Raw. Finally, the SUDS runtime system speculates past loop control dependence points, which tend to be highly predictable, and allows memory operations to speculatively execute out of order.

1.3 Contributions

The main contribution of this thesis is a compiler transformation called *scalar queue conversion*. Scalar queue conversion is an instruction reordering algorithm that simultaneously renames scalar variables. Scalar queue conversion has at least five unique features.

1. Scalar queue conversion works on flow graphs with *arbitrary* control flow. The flow graph can be unstructured, or even irreducible.

2. Scalar queue conversion can move instructions out of loops with bounds that can not be determined until runtime.

3. Scalar queue conversion *guarantees* the elimination of *all* scalar anti- and output- dependences that might otherwise restrict instruction reordering.

Thus scheduling algorithms based on scalar queue conversion can make instruction ordering decisions irrespective of register storage dependences.

4. Scalar queue conversion, unlike Tomasulo's algorithm, can rename and reorder instructions across mispredicted branches whenever the reordered instructions are not control dependent on that branch.

5. Scalar queue conversion is a speculative compiler transformation, in that it inserts dynamic memory allocation operations into code that might not otherwise dynamically allocate memory. We describe an efficient software based checkpoint repair mechanism that safely applies speculative compiler optimizations.

In addition to describing scalar queue conversion this thesis makes the following additional contributions.

1. It shows how to move the renaming operations introduced by scalar queue conversion to minimize the runtime overheads introduced by scalar renaming.

2. It shows how to use scalar queue conversion to implement a generalized form of loop distribution that can distribute loops that contain arbitrary inner loops.

3. It describes the pointer and array analysis issues that needed to be addressed when using scalar queue conversion in a practical context.

4. It describes the SUDS software runtime system, which performs memory dependence speculation while only increasing the latency of memory operations by about 20 machine cycles.

5. It provides a demonstration that the SUDS system effectively schedules and exploits parallelism in the context of a complete running system on the Raw microprocessor.

It is my hope that the work in this thesis will serve as a starting point for the research that I believe needs to be done to enable the next several generations of high performance microprocessors. Tomasulo's algorithm issues instructions out of order, but its ability to *fetch* out of order is limited by mispredicted branch points. To overcome this fetch limit the microprocessor must somehow transform a sequential thread into multiple, concurrent, threads of control. The research in this thesis demonstrates the kinds of problems that need to be overcome when the sequential thread is both imperative and has completely arbitrary control flow.

1.4 Road Map

The rest of this thesis is structured as follows. Chapter 2 defines the relatively standard graph-theoretic terms widely used in the compiler community. Readers with a strong background in compiler design can profitably skip Chapter 2.[3]

The next four chapters describe scalar queue conversion. Chapter 3 describes the transformation, and explains why scalar queue conversion is able to, provably, eliminate *all* the scalar anti- and output- dependences that might otherwise inhibit a particular schedule. Chapter 4 discusses an optimization that improves scalar queue conversion's placement of copy instructions. Chapter 5 describes several extensions and improvements that widen the applicability of scalar queue conversion. Chapter 6 describes the generalized loop distribution transformation that

[3] But please keep in mind the difference between the *value dependence graph* (the graph comprising the scalar def-use chains, control dependence arcs, and memory dependences) and the *conservative program dependence graph* (the graph comprising the value dependence graph with additional edges for the scalar use-def and def-def chains). Both of these graphs are sometimes called "program dependence graphs" in the literature, but the difference is important in the work described in subsequent chapters.

scalar queue conversion enables.

Several practical questions with regard to scalar queue conversion are addressed in Chapter 7. The first problem is that scalar queue conversion introduces dynamic memory allocation operations into loops that might not otherwise allocate memory dynamically. Thus, scalar queue conversion is unsafe in the sense that it does not provide strict guarantees on the memory footprint of the transformed program. Chapter 7 describes an efficient software based checkpoint repair mechanism that we use to eliminate this problem. The SUDS Software Un-Do System described in Chapter 7 allows scalar queue conversion to be applied *speculatively*. If scalar queue conversion introduces a dynamic memory allocation error then SUDS rolls back execution to a checkpointed state and runs the original version of the code. SUDS performs an additional important task in that it implements a memory dependence speculation system that breaks (speculatively and at runtime) memory dependences that would otherwise forbid the parallelization of many loops.

Chapter 8 describes the inter-relationship of the work described in Chapters 3 through 7 in the context of a working system. Several case studies describe, in some detail, how, and why, the transformations are applied to specific loops.

Chapter 9 describes the relationship of scalar queue conversion and generalized loop distribution to previous work in program slicing, scalar expansion, loop distribution, thread-level parallelization, critical path reduction and data speculation. Chapter 10 concludes.

Chapter 2

The Dependence Analysis Framework

As stated in Section 1.2 the SUDS approach to finding parallelism rests on three principles:

1. Dynamic renaming eliminates anti- and output-dependences.
2. Control dependence analysis eliminates conservative flow-dependences.
3. Speculation eliminates some dynamically predictable true- and control-dependences.

In this chapter we define basic terms and describe what we mean by a *dependence.*

2.1 The Flow Graph

To start with, let us define some basic terms. We will use the term *program* to refer to the finite set of instructions that specifies the set of operations that we wish to perform. For the purposes of the conceptual development in this chapter we choose a simple "control flow graph" representation of programs. An example of some code is shown if Figure 2-1. The resulting control flow graph is shown in Figure 2-2.

```
sum = 0
i = 0
do
  partial_sum = 0
  j = 0
  use(i, sum)
  do
    use2(sum, partial_sum, i, j)
    partial_sum = partial_sum + 1
    j = next(j)
    c1 = cond1(i, j)
  while c1
  i = i + 1
  sum = sum + partial_sum
  c2 = cond2(i)
while c2
use(sum)
```

Figure 2-1: An example program with a doubly nested loop.

The nodes in the control flow graph representation represent *instructions*. Each instruction specifies an operation that changes some part of the underlying machine state. The control flow graph has two additional nodes, labeled `begin` and `end` that correspond to the initial and final states of the program execution. The edges in the control flow graph represent (programmer specified) temporal constraints on the order of operations. More specifically, if there is a directed path from instruction `A` to instruction `B` in the control flow graph, then there may be a correct sequence of (dynamic) state transitions where transition `A` occurs before transition `B`. Note that Figure 2-2 includes an edge that flows directly from the `begin` node to the `end` node. This edge represents the possibility that the program will not executed at all. We will call the control flow graph edges *flow dependences*.

The kinds of instructions permitted in our representation include

1. 3-Address operations (*e.g.*, `a = b + c`, where "`a`" is a register name, "`b`" and "`c`" are register names or constants, and "`+`" is a binary operation with no side effects. The semantics are that the contents of register `a` are replaced with

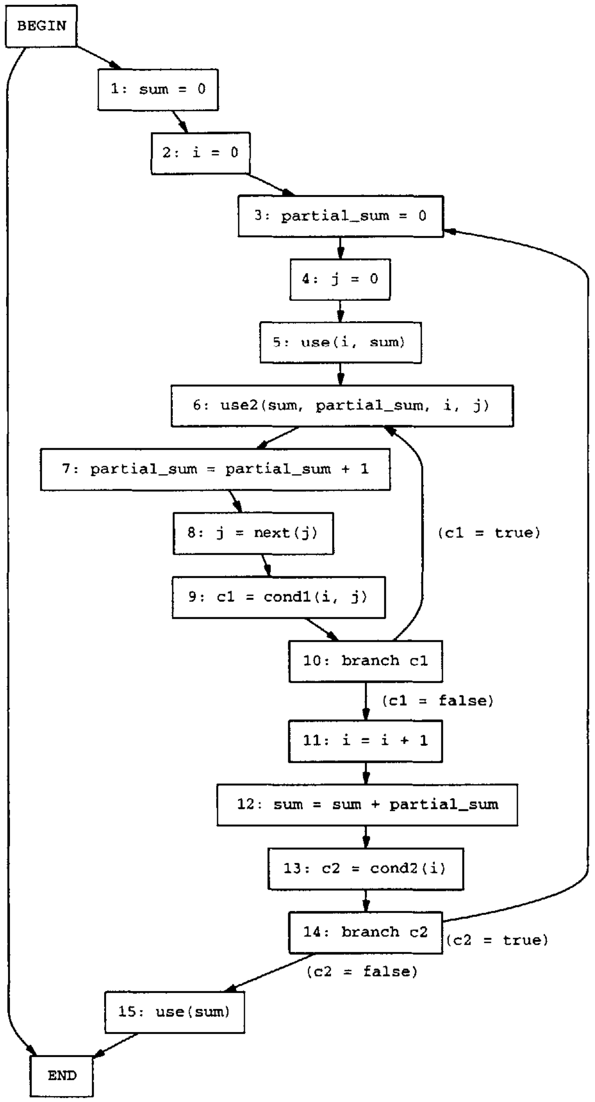


Figure 2-2: The control flow graph corresponding to the program in Figure 2-1.

the value produced by performing the specified operation on the contents of registers b and c. We call a the *destination operand* and b and c *source operands*.

2. Load instructions, `x = *y`, where "x" and "y" are register names. The semantics are that the current contents of the memory location with address y are loaded into the x register.

3. Store instructions, `*y = x`, where "x" and "y" are register names. The semantics are that the current contents of register x overwrite the value in the memory location with address given by register y.

4. Branch instructions, `branch c`, where "c" is a register name. The semantics are that of a dynamic decision point with respect to which of two output edges we take out of the node.[1]

5. Call instructions, `call p`, where "p" is a register or constant containing the identifier of some node in some flow graph. The call instruction *implicitly* places the identifier of its own node on an implicit stack, so that it can be used by the `return` instruction.

6. Return instructions, `return`, that pop the identifier of a node off the top of the implicit stack, and return flow of control to the successor of that node.

7. Jump instructions, `jump c`, where "c" is a register name. It is assumed the register contains the identifier of some flow graph node, and control flow is rerouted to that node. This permits "multiway" branches, such are required to efficiently implement C `switch` statements.

[1] We could have made state transitions on a program counter an explicit part of the representation, but have chosen not to because control flow graphs are standard. Control flow graphs represent transitions on the program counter implicitly through the flow dependences, with branches representing the only points at which runtime information effects transitions on the program counter state.

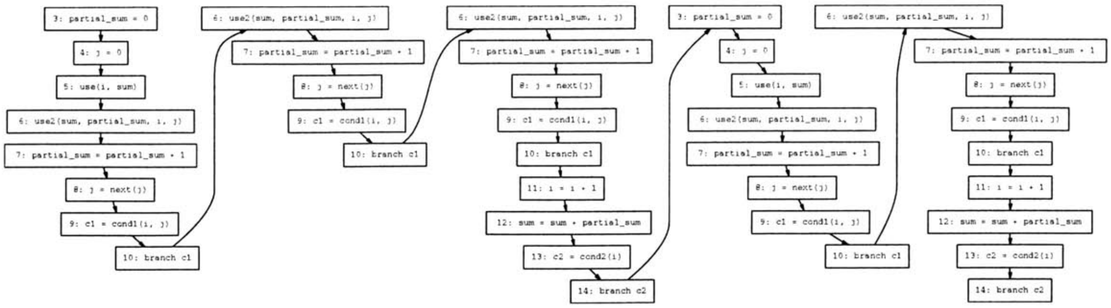


Figure 2-3: The *sequential ordering* of the state transitions produced by two iterations of the outer loop of the flow graph in Figure 2-2. The inner loop executes three times during the first outer loop iteration and twice during the second.

The semantics of a particular program can be determined (operationally) by starting with a predetermined machine state with one register for each named register in the program, and a memory, and then stepping through the control flow graph, performing the state transitions specified by each instruction one at a time. We call the sequence of state transitions produced by this process the *sequential order*. A sequential order for two iterations of the outer loop of the flow graph in Figure 2-2 is shown in Figure 2-3. In this example, the inner loop executes three times during the first outer loop iteration and twice during the second. There are 39 total instructions shown in this total order.

The question we are trying to address is whether there are sequences of state transitions, other than the sequential order, in which we can execute the state transitions and get the same final state. That is, the sequential order is a *total* order on the set of state transitions. We would like to find less restrictive *partial* orders that produce the same final state.

2.2 The Conservative Program Dependence Graph

The first observation we make is that the flow dependences on individual instructions are overly conservative with respect to register operands. A combination of standard dataflow analyses can produce less restrictive orderings.

We say that given nodes d and n in a control flow graph d *dominates* n if every directed path from `begin` to n passes through d [75]. Every node dominates itself. For example, in Figure 2-2 node 14 dominates nodes 14 and 15, but not end. This is because every path from `begin` to node 15 goes through node 14, but there is a path (`begin` → end) that does not go through node 14. The *postdominance* relation is defined similarly, with the flow graph reversed. Node d postdominates n if d is on every path from n to end. In Figure 2-2 node 15 postdominates every node in the flow graph except nodes `begin` and end.

We can also define the set of dominators of a node n, Dom[n], recursively as the least fixed point of the set of simultaneous equations:

$$\mathrm{Dom}[n] = \{n\} \cup \left(\bigcap\nolimits_{p \in \mathrm{pred}[n]} \mathrm{Dom}[p]\right) \quad \forall n,$$

where we work downwards in the lattice of sets from full sets towards empty sets.

The dominance relation introduces a well defined partial order on the nodes in a flow graph. Thus, we can define a *backward dependence edge* as any edge from a node n to a node d that dominates n. We will informally refer to any edge that is not a backward edge as a *forward edge.* (Note that this overloads the word "forward" somewhat since it includes edges $x \rightarrow y$ where neither x nor y dominate the other). For example, in Figure 2-2 node 6 dominates node 10 so the edge $10 \rightarrow 6$ is a backedge in the flow graph.

The intuitive reason that the dominance relation is central to our analysis (as it is in most modern compiler optimizations) is that it summarizes information about all possible sequential orderings of state transitions, no matter the initial state at the beginning of execution. That is, if node d dominates node n in the flow graph, then *every* sequential ordering generated from the flow graph will have the

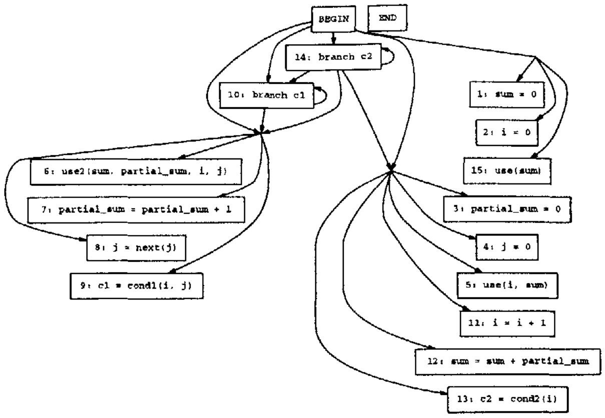


Figure 2-4: The *control dependences* corresponding to the flow graph in Figure 2-2.

property that the first appearance of d will come before the first appearance of n. If node d does not appear in the sequential ordering, then node n can not appear either.

Given two nodes, we say that x *strictly dominates* w iff x dominates w and x $\neq$ w. The *dominance frontier* of a node x is the set of all *edges* v $\rightarrow$ w such that x dominates v, but does not strictly dominate w [75]. (The original work on dominance frontiers used the set of *nodes* w, but the edge formulation is more accurate and more useful. See, for example, [92].) In Figure 2-2 node 6 dominate nodes 10, 14 and 15, but does not strictly dominate any of nodes 6, 3 or end, so the dominance frontier of node 6 is the edges 10 $\rightarrow$ 6, 14 $\rightarrow$ 3 and 15 $\rightarrow$ end. The *postdominance frontier* of node x is the set of all edges v $\rightarrow$ w such that x postdominates w, but does not strictly postdominate v (note that we have, essentially, reversed the edge).

The postdominance frontier gives us information about control dependence [40, 30]. In particular we say that a node n is *control dependent* on edge x $\rightarrow$ y iff the edge is in the postdominance frontier of n. The intuitive reason for this is that the postdominance frontier represents the set of edges that cross from regions of the program where n is not guaranteed to execute to regions of the program where n is guaranteed to execute. The nodes x in the control dependence edges are thus the branch points that decide whether or not node n should execute. For example, in Figure 2-2 the postdominance frontier of node 7 is the set of edges `begin` $\rightarrow$ 1, 10 $\rightarrow$ 6 and 14 $\rightarrow$ 3, and indeed, it is *exactly* the `begin` node and the branches at nodes 10 and 14 that determine how many times node 7 will execute. (Recall that one should think of the `begin` node as a branch that decides whether or not the program will execute at all.) The complete set of control dependences for the flow graph from Figure 2-2 is shown in Figure 2-4.

For each node x that contains an instruction that has register r as a destination operand we call x a *definition* of r. For each node y that contains an instruction that uses register r as a source operand we call y a *use* of r. For example, in Figure 2-2 nodes 4 and 8 define the variable `j`, while nodes 6, 8 and 9 use the variable `j`.

We say that a definition (use) x of register r *reaches* a node y if there exists a path from x to y in the flow graph such that there is no other definition of register r on that path. For example, the definition of variable `j` at node 4 reaches node 8, because there is a path from 4 to 8 with no other definition of `j`, but the definition at node 4 does not reach node 9 because every path from 4 to 9 goes through the definition at node 8.

More generally, given any directed graph (N, E) and subsets $Gen \subset N$ and $Pass \subset N$, we define the *Reaching* relation on the graph with respect to *Gen* and *Pass* as the set of nodes $y \in N$ such that there is a path from a node $x \in Gen$ to y such that all the intermediate nodes on the path are in *Pass*. Techniques for efficiently generating the reaching relation can be found in any standard undergraduate compiler textbook [4]. Typically it is found as the least fixed point of the equation

$$Reaching = Succs(Gen \cup (Reaching \cap Pass)).$$

Where $Succs(X) = \{n \in N | x \in X \wedge (x \rightarrow n) \in E\}$.

Then we can more specifically define the *reaching definitions* relation for a node x that defines a register r as the solution to the *Reaching* relation where $Gen = \{x\}$ and

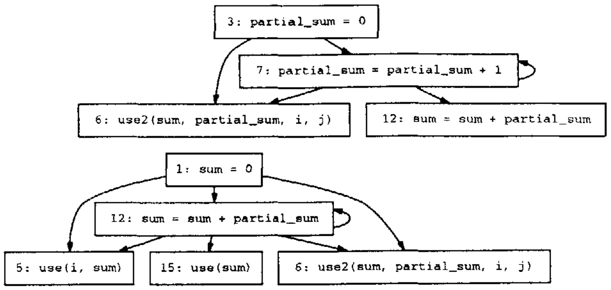

3: partial_sum = 0
7: partial_sum = partial_sum + 1
6: use2(sum, partial_sum, i, j)
12: sum = sum + partial_sum
1: sum = 0
12: sum = sum + partial_sum
5: use(i, sum)
15: use(sum)
6: use2(sum, partial_sum, i, j)

Figure 2-5: The du-webs corresponding to variables `partial_sum` and `sum` for the flow graph from Figure 2-2.

Pass is the set of nodes that do not define `r`. Likewise the *reaching uses* relation for a node `x` that uses a register `r` is the solution to the *Reaching* relation where $\mathit{Gen} = \{\mathtt{x}\}$ and *Pass* is the set of nodes that do not define `r`. For example, in Figure 2-2, the definition of variable `j` in node 8 reaches node 6 (through the path, 8, 9, 10, 6). But the definition at node 8 does not reach node 5, because node 4 is not in the *Pass* set.

Of particular interest to us is the subset of the reaching definitions relation that relates the definitions to the uses of a particular register `r`. This subset of the reaching definitions relation is typically called the *def-use-chains* or *du-chains* for the variable `r`. A maximally connected subset of the du-chains for a particular register `r` is called a *du-web*. The du-chains for variable `j` in Figure 2-2 are $4 \rightarrow 6, 4 \rightarrow 8, 8 \rightarrow 9$, $8 \rightarrow 6$ and $8 \rightarrow 8$. This set of du-chains is also a du-web, since it is a connected set. The du-webs for variables `partial_sum` and `sum` are shown in Figure 2-5. Given the du-chains for a register `r`, the du-webs can be efficiently calculated by computing the connected components (*e.g.*, using depth first search) on the graph of du-chains [68].

Similarly, the *def-def-chains* relation for the register `r` is the subset of the reaching defs relation that relates the definitions of `r` to other definitions of `r`. For example, $8 \rightarrow 4$ is a def-def chain for variable `j` in Figure 2-2. The *use-def-chains* for a variable `r` are the subset of the reaching uses of `r` that are also definitions. Note that the use-def chains are *not* simply the def-use chains turned around backwards. For example, in Figure 2-2 $7 \rightarrow 12$ is a def-use chain for variable `partial_sum`, but $12 \rightarrow 7$ is *not* a use-def chain, because every path from node 12 to node 7 must go through node 3, which redefines `partial_sum`.

We have defined the def and use chains with respect to registers only. We will also define a particularly conservative set of dependences with respect to memory operations (load and store instructions). We say that any memory operation, x, *reaches* memory operation, y, if there is a path from x to y in the control flow graph. (*Pass* is the set of all nodes). We say there is a *memory dependence* from x to y if at least one of x and y is a store instruction. (That is, we don't care about load-load dependences).

Now we are ready to define the conservative program dependence graph, and relate the conservative program dependence graph (which is a *static* representation of the program) to the allowable *dynamic* orderings of instructions.

We define the *conservative program dependence graph* as the graph constructed by the following procedure. Take the nodes from the control flow graph. For every pair of nodes, x, y, insert an edge, $x \rightarrow y$, if there is either a def-use-chain from x to y, a use-def-chain from x to y, a def-def-chain from x to y, a memory dependence from x to y or a control dependence from x to y.[2]

Suppose the sequential execution of a control flow graph on a particular initial state produces a particular sequential (total) ordering of state transitions (as described above for the semantics for control flow graphs). Now for every pair of dynamic instruction nodes x, y, such that x comes before y in the sequential ordering, we insert an edge from x to y if there is an edge in the conservative program dependence graph between the corresponding (static) instruction nodes. We call the resulting graph the *conservative dynamic dependence graph*. The conservative dynamic dependence graph corresponding to the sequential order shown in

[2] We defined control dependence from *edges* to nodes, (*i.e.*, $(b \rightarrow d) \rightarrow n$). Here we are using the standard *node* definition of control dependence, $b \rightarrow n$ for simplicity.

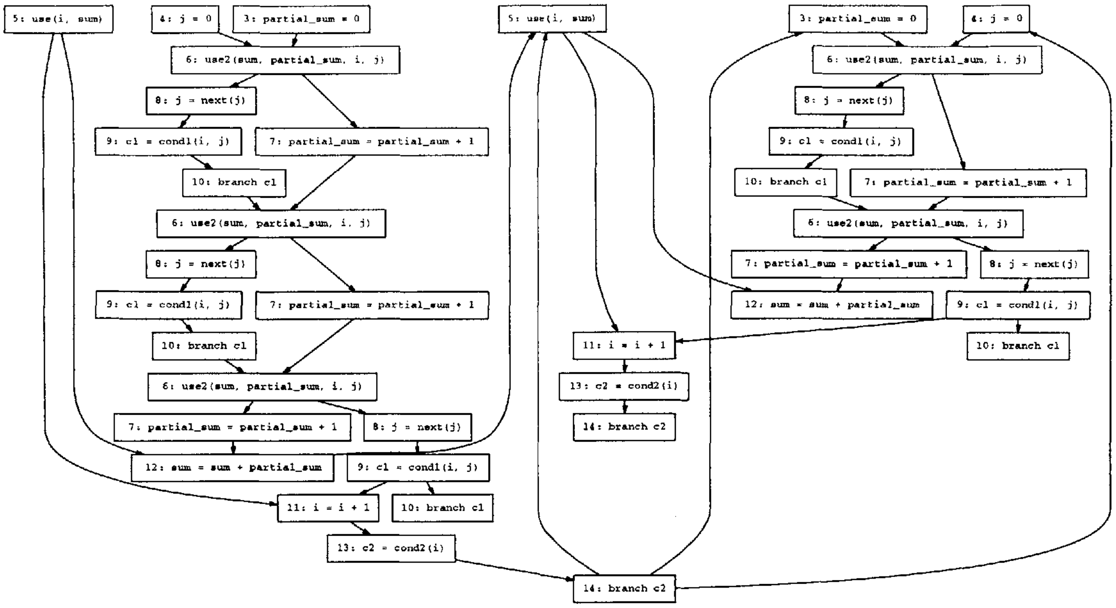


Figure 2-6: The *conservative dynamic dependence graph* for two iterations of the outer loop of the flow graph in Figure 2-2. The inner loop executes three times during the first outer loop iteration and twice during the second. The depth of the graph has been reduced to 26 instructions, from the 39 instructions in the sequential order shown in Figure 2-3.

Figure 2-3 is shown in Figure 2-6.

The edges in the conservative dynamic dependence graph have standard names [57], which we will also use. If the edge on the dynamic ordering was created because there was def-def-chain or use-def-chain in the conservative program dependence graph we call the edge in the dynamic ordering a *register storage dependence*. We will sometimes distinguish between these two types by calling them output-dependences and anti-dependences, respectively. If the edge on the dynamic ordering was created because there was a def-use-chain in the conservative program dependence graph we call the edge in the dynamic ordering a *value-dependence,* or less formally, a "true-dependence". If the edge on the dynamic ordering was created because there was a memory dependence from a store to a load we will call it a *memory value dependence*. If the edge on the dynamic ordering was created because there was a memory dependence from a load to a store we will call it a *memory anti-dependence*. If the edge on the dynamic ordering was created because there was a memory dependence from a store to a store we will call it a *memory output-dependence*. Finally, if the edge on the dynamic ordering was created because there was a control dependence in the conservative program dependence graph we will call it a *dynamic control dependence*.

Note that the conservative dynamic dependence graph is a directed acyclic graph, and thus defines a partial order on the state transitions during the execution of the program. The value of the conservative program dependence graph comes from the fact that *any sequence of these state transitions that obeys the partial ordering demanded by the conservative dynamic dependence graph will produce the same final state as the sequential ordering*. This can be argued informally by noticing that we have

1. Placed a total order on changes to the memory state (through memory-dependences).

2. Guaranteed that every instruction executes after the branches in the sequential order that control whether or not that instruction executes (through control-dependences).

3. Placed a total order on changes to each *individual* register state (through def-def-chains).

4. Guaranteed that source operands always receive the value they would have received in the sequential order by placing each use of register r in the conservative dynamic dependence graph between the same two defs of register r that it was between in the sequential order (through def-use and use-def chains).

We have gained some parallelization flexibility by moving from the control flow graph to the conservative program dependence graph, because we have moved from the total order on state transitions imposed by the sequential order, to the somewhat less restrictive partial order given by the conservative dynamic dependence graph. For example, in Figure 2-6 we have reduced the dependence distance to 26 nodes from the 36 nodes shown in the sequential order from Figure 2-3.

2.3 The Value Dependence Graph

One of the main constraints to further parallelization of the conservative program dependence graph is the existence of a large number of storage dependences. In Chapter 3 we will describe scalar queue conversion, a compiler transformation that can *always* add copies to the flow graph that eliminate *all* register storage dependences. Thus, instruction scheduling algorithms can make instruction ordering decisions irrespective of register storage dependences. In particular, instruction scheduling algorithms can work on a less restrictive graph than the conservative program dependence graph.

To differentiate this graph from the conservative program dependence graph we will call it the *value dependence graph*. We define the value dependence graph as the graph constructed by the following procedure. Take the nodes from the control flow graph. For every pair of nodes, x, y, insert an edge, $x \dashrightarrow y$, if there is either a def-use-chain from x to y, a memory dependence from x to y or a control dependence from x to y. Thus the value dependence graph is the subgraph of the conservative program dependence graph created by removing the use-def and def-def chains from the conservative program dependence graph.

Suppose the sequential execution of a control flow graph on a particular initial state produces a particular sequential (total) ordering of state transitions (as described above for the semantics for control flow graphs). Now for every pair of

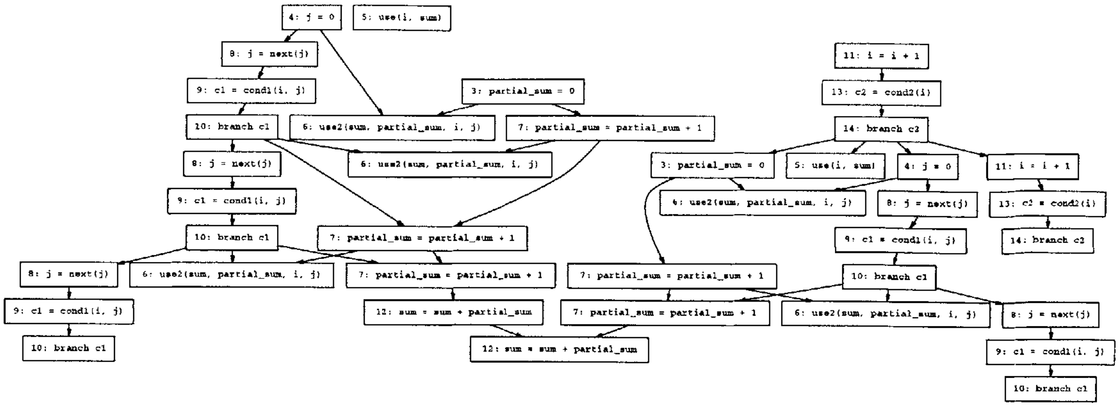


Figure 2-7: The *dynamic value graph* for two iterations of the outer loop of the flow graph in Figure 2-2. The inner loop executes three times during the first outer loop iteration and twice during the second. The depth of the graph has been reduced to 10 instructions, from the 39 instructions in the sequential order shown in Figure 2-3.

dynamic instruction nodes x, y, such that x comes before y in the sequential ordering, we insert an edge from x to y if there is an edge in the value dependence graph between the corresponding (static) instruction nodes. We call the resulting graph the *dynamic value graph.* The dynamic value graph corresponding to the sequential order shown in Figure 2-3 is shown in Figure 2-7.

Renaming scalars to avoid register storage dependences produces substantial concurrency gains. This concurrency comes at the cost of increasing the number of simultaneously live values, and thus the required storage space. For example, in Figure 2-7 we have reduced the dependence distance to 10 nodes from the 26 nodes in the conservative dynamic dependence graph from from Figure 2-6. As a result the graph is, informally, both "shorter" and "fatter." In the following chapters we will describe scalar queue conversion, a compiler transformation that effects this renaming.

Chapter 3

Scalar Queue Conversion

As described in the last chapter, scalar renaming is one of the most effective techniques known for exposing instruction concurrency in a program. In this section we will show that the compiler can restructure the code to eliminate *all* register storage dependences. The ability to eliminate any register storage dependence means that *instruction scheduling algorithms can make instruction ordering decisions irrespective of register storage dependences.* The increased flexibility results in schedules that would otherwise be impossible to construct.

We call this transformation to eliminate register storage dependences *scalar queue conversion,* because it completely generalizes the traditional technique of scalar expansion [68] to arbitrary unstructured (even irreducible) control flow, and provably eliminates all register anti- and output-dependences that would violate a particular static schedule. In Chapter 6 we show how to use scalar queue conversion as the key subroutine to enable a generalized form of loop distribution. Loop distribution is best viewed as a scheduling algorithm that exposes the available parallelism in a loop [68]. The loop distribution algorithm in Chapter 6 generalizes previous scheduling techniques by scheduling across code with completely arbitrary control flow, in particular, code with inner loops. This generalization is possible only, and exactly, because scalar queue conversion guarantees the elimination of all register anti- and output-dependences.

3.1 Motivation

Consider node 6 in the flow graph in Figure 2-2. Suppose we want to run this instruction out of order. For example, execution of the operation "`use2(sum, partial_sum, i, j)`" might consume many cycles, and we might wish to start execution of node 7 before node 6 completed its work. Unfortunately there is a use of variable `partial_sum` in node 6 and a definition of `partial_sum` in node 7, so dynamically executing an instance of node 6 out of order with the immediately following instance of node 7 could produce incorrect results. If, however, we were to make a copy of the variable `partial_sum` into a new variable, called, for example `partial_sum_tmp`, then we could execute nodes 6 and 7 in either order. This transformation is demonstrated in Figure 3-1.

Suppose, however, that we want to defer execution of *all* dynamic instances of node 6 until after execution of all the dynamic instances of node 7. In this case we need to generalize the transformation so that rather than saving the values required by node 6 in a (statically allocated) register, we save the values in dynamically allocated storage. By this process we can simultaneously save the machine states required to execute an arbitrary number of dynamic instances of node 6.

More concretely, we turn node 6 into a closure. A *closure* can be thought of as a suspended computation [72, 107]. It is typically implemented as a data structure that contains a copy of each part of the state required to resume the computation, plus a pointer to the code that will perform the computation. There are then a set of operations that we can perform on a closure:

1. We can *allocate* a closure by requesting a portion of memory from the dy-

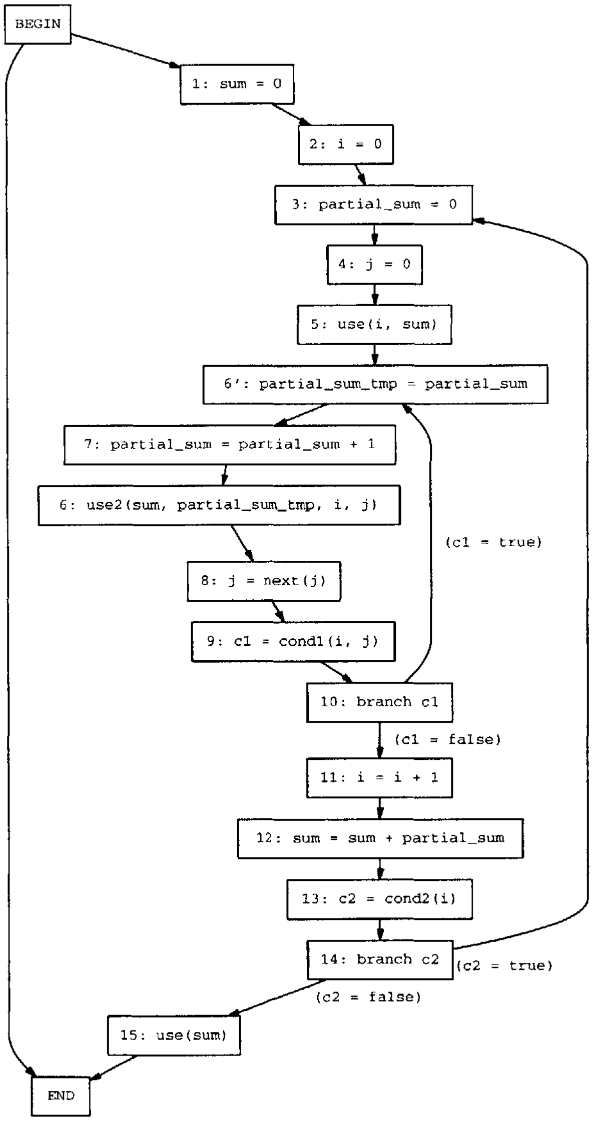


Figure 3-1: Copying the value of the variable `partial_sum` allows reordering of nodes 6 and 7.

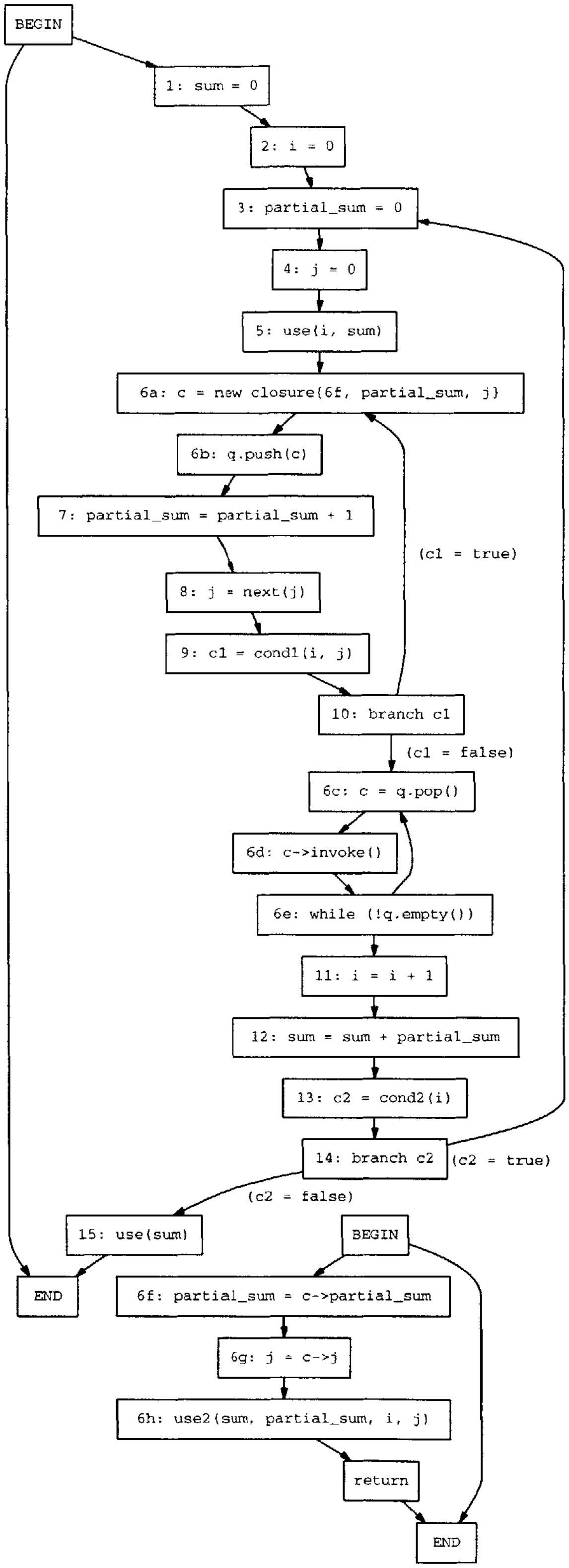


Figure 3-2: Copying the value of variables `partial_sum` and `j` to the dynamic storage represented by `closure_queue` allows us to defer executions of instantiations of node 6 past an arbitrary number of instantiations of nodes 7 and 8.

namic memory allocator that is sufficient to hold the required state plus code pointer.

2. We can *fill* a closure by copying relevant portions of the machine state into the allocated memory structure.

3. We can *invoke* a closure by jumping to (calling) the closures code pointer and passing a pointer to the associated data structure that is holding the relevant machine state.

Closures will be familiar to those who have used lexically scoped programming languages. For example, in `C++` and `Java` closures are called *objects*. In these languages closures are *allocated* by calling operator `new`, *filled* by the constructor for the object's class, and *invoked* by calling one of the methods associated with the object's class.

In the general case we can *defer* execution of some subset of the code by creating a closure for each deferred piece of code, and saving that closure on a queue. Later we can *resume* execution of the deferred code by invoking each member of the queue in FIFO order. For example, Figure 3-2 demonstrates how we use queues of closures to defer execution of every dynamic instance of node 6 until after the execution of every dynamic instance of node 7.

The intuition behind this result is that *every imperative program is semantically equivalent to some functional program* [72, 58, 7]. Since a functional program never overwrites any part of an object (but rather creates an entirely new object) there are no storage dependences. Another way to view the result is in terms of the dynamic register renaming performed by Tomasulo's algorithm [117, 57, 104, 83, 105]. Tomasulo's algorithm performs a dynamic mapping of "virtual" register names to "physical" registers, each of which is written only once. After this renaming all register storage dependences are eliminated, because (conceptually) no physical register ever changes its value. Thus, the instruction scheduling algorithm is less constrained by register storage dependences.

Tomasulo's algorithm, however, fetches branches in the order they are given by the *flow* dependences from the control flow graph. Similarly, existing techniques for proving the equivalence of imperative to functional programs [58, 7] rely on continuation passing style. Conversion to continuation passing style requires that continuations nest in an order corresponding to flow dependences [6]. Scalar queue conversion, in contrast, places closure allocation and fill operations only where they are required in the *value dependence graph*. As a result, scheduling algorithms based on scalar queue conversion (such as the generalized loop distribution algorithm described in Chapter 6), are not restricted to fetching a single sequential flow of control.

3.2 Road Map

The remainder of this chapter addresses the questions of when it is legal to defer execution of a region of code, and where closures need to be created to perform the renaming required by the requested code deferment. In Sections 3.3, 3.4, 3.5 and 3.6 we demonstrate that scalar queue conversion can defer *any* set of instructions that does not violate the dependences in the value dependence graph. The additional register storage dependences of the conservative program dependence graph can be *completely* ignored.

Subsequent chapters deal with a number of practical issues surrounding scalar queue conversion. In Chapter 4 we give an *eager* dead-copy elimination algorithm, motivated by algorithms that convert to SSA form, that optimizes (in a minimax sense) the number of dynamic copy operations introduced by scalar queue conversion.

Section 5.1 demonstrates how to extend the results from this chapter from regions with single exits to regions with multiple exits. Section 5.2 shows how to use the closures created by scalar queue conversion as a basic unit of concurrency. Scalar queue conversion eliminates scalar anti- and output- dependences, but does not eliminate memory dependences. Chapter 5 also describes a set of program transformations that reduce or eliminate memory dependences, thus extending the applicability of scalar queue conversion.

Chapter 6 additionally shows how to use scalar queue conversion as the key enabling technology for a generalized form of loop distribution. In particular, the generalized loop distribution transformation described in Chapter 6 relies on the ability of scalar queue conversion to place closure allocation and fill operations only at points where they are required by the value dependence graph, rather than the more restrictive control flow graph.

A key practical question with regard to scalar queue conversion is addressed in Chapter 7. The problem is that scalar queue conversion introduces dynamic memory allocation operations (*i.e.*, closure allocations) into loops that might not otherwise allocate memory dynamically. Thus, scalar queue conversion is *unsafe* in the sense that it does not provide strict guarantees on the memory footprint of the transformed program. In particular, a scalar queue converted program could, potentially, try to allocate more memory than is available in the system, and thus create an error condition that would not have occured in the untransformed program.

Chapter 7 describes an efficient software based checkpoint repair mechanism that we use to eliminate this problem. The SUDS Software Un-Do System described in Chapter 7 allows scalar queue conversion to be applied *speculatively*. If scalar queue conversion introduces a dynamic memory allocation error condition then SUDS rolls back execution to a checkpointed state and runs the original version of the code.

The relationship of scalar queue conversion to program slicing, scalar expansion, loop distribution, Tomasulo's algorithm and thread level speculation is described in Chapter 9.

Running Example

The concepts, definitions and proofs in the rest of this chapter are all illustrated with respect to an example based on the program shown in Figure 2-2. I have done my best to choose the example such that it illustrates the relationships between the relevant ideas, but so that it is not so complicated as to overwhelm the reader.

The example problem is as follows. Suppose we wish to reschedule the loop in Figure 2-2 into two loops, one that does the work corresponding to nodes 2, 3, 4, 7, 8, 9, 10, 11, 13 and 14, and one corresponding to nodes 1, 5, 6, 12 and 15. Is there a legal way to restructure the code to effect this rescheduling? In this chapter we will demonstrate that this transformation is legal exactly because the flow of value and control dependences across the partitioning of nodes in the region is *unidirectional*.

Consider a connected, single-entry, single-exit region R of the flow graph. We induce the *region flow graph* by taking the set of nodes in the region and all the edges $x \rightarrow y$ such that both x and y are in the region. With the begin node we associate a set of definitions for variables that correspond to the def-use chains that reach from nodes $\bar{r} \notin R$ to nodes $r \in R$. With the end node we associate a set of uses for variables that correspond to the def-use chains that reach from nodes $r \in R$ to nodes $\bar{r} \notin R$. On the resulting region flow graph we calculate the value dependence graph. Recall that this is the def-use chains, the memory dependence chains, and the control dependences calculated on the flow graph. Note that we have explicitly used the value dependence graph, rather than the conservative program dependence graph, which also includes use-def and def-def

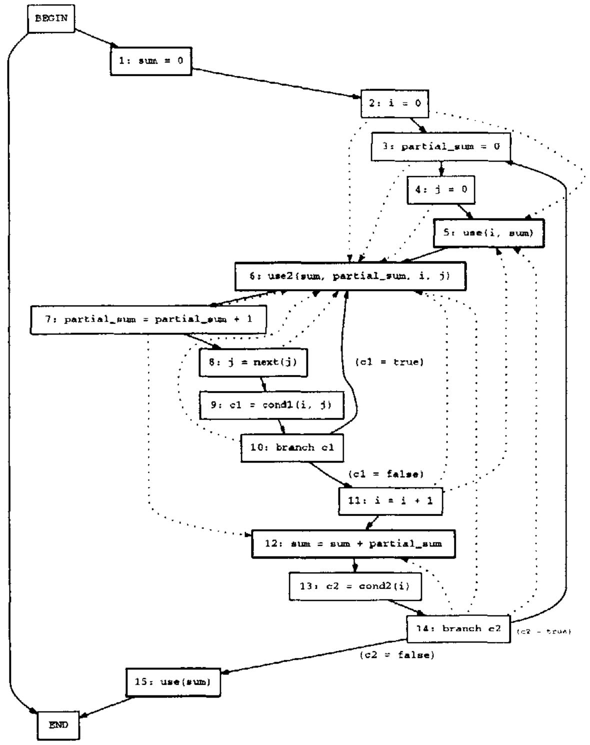


Figure 3-3: Partitioning the outer loop into the two subsets, 2, 3, 4, 7, 8, 9, 10, 11, 13, 14 and 1, 5, 6, 12, 15 produces a *unidirectional cut* because no dependence edges flow from the second subset into the first. Cut dependence edges are shown in dotted lines. They all flow from the first subset into the second.

chains, because these are exactly the dependences that will be eliminated using scalar queue conversion.

3.3 Unidirectional Cuts

Now we define a *cut* of the set of nodes in a region, R, as a partitioning of the set of nodes into two subsets, A, B such that $A \cap B = \emptyset$ and $A \cup B = R$. We say that a cut is *unidirectional* iff there are no edges $x \dashrightarrow y$ such that $x \in B$ and $y \in A$. That is, all the edges either stay inside A, stay inside B or flow from A to B, and *no* edges flow from B to A. For example, given the region corresponding to the outer loop in Figure 3-3, the partition $\{2, 3, 4, 7, 8, 9, 10, 11, 13, 14\}$ and $\{1, 5, 6, 12, 15\}$ is a unidirectional cut because there are no def-use chains, memory or control dependences flowing from the second set to the first.

In the following sections we will demonstrate that by the process of *queue conversion* we can always transform a unidirectional cut A-B of a single-entry single-exit region into a pair of single-entry single-exit regions, that produce the same final machine state as the original code, but have the feature that all of the instructions from partition A execute (dynamically) before all the instructions from partition B.

Any particular value dependence graph might have many different unidirectional cuts. The criteria for choosing a specific cut will depend on the reasons for performing the transformation. In Section 5.1 and Chapter 6 we will discuss two different applications in which unidirectional cuts appear naturally. In particular, we will present two different methods for finding a unidirectional cut efficiently, each depending on a different set of goals.

3.4 Maximally Connected Groups

First we will show that we can create a "reasonable" flow graph that consists only of the nodes from subset A of a unidirectional A-B cut. The property that makes this possible is that every *maximally connected* group of the nodes from subset B will have only a single exit. Thus we can remove a maximally connected subset of nodes from subset B from the region flow graph and "fix-up" the breaks in the flow graph by connecting the nodes that precede the removed set to the (unique) node that succeeds the removed set.

Given a unidirectional cut A-B of a flow graph then we will call a subset of nodes $\beta \subset B$ in the graph a *maximally connected group* iff every node in β is connected in the flow graph only to other nodes of β or to nodes of A. That is, given $\bar{\beta} = B - \beta$ and nodes $b \in \beta$, $\bar{b} \in \bar{\beta}$ there are no edges $b \rightarrow \bar{b}$ or

$\bar{b} \to b$. For example, given the unidirectional cut shown in Figure 3-3 where $A = \{2, 3, 4, 7, 8, 9, 10, 11, 13, 14\}$ and $B = \{1, 5, 6, 12, 15\}$, the maximally connected groups are the subsets $\{1\}$, $\{5, 6\}$, $\{12\}$ and $\{15\}$ of B.

But now suppose that we are given a unidirectional cut A-B. This means that there can be no control dependences from B to A. Informally, there are no branches in B that can in any way determine when or if a node in A is executed. Now suppose that we are given a maximally connected group $\beta \subset B$. If β has an exit edge $b \to a$ (an edge where $b \in \beta$, $a \notin \beta$), then, because β is maximally connected it must be the case that $a \in A$. The node a can not be in B because then β would not be maximally connected.

If there are two (or more) such exit edges, $b_0 \to a_0$ and $b_1 \to a_1$, where $b_0 \neq b_1$ then it must be the case that there is a branch or set of branches in β that causes the flow graph to fork. In particular, b_0 and b_1 must have different control dependences, and at least one of those control dependences must be on a node inside β. But a_1 and a_0 can not be control dependent on any node inside β, because they are on the wrong side of the A-B cut.

Consider node a_0. There is an edge from b_0 to a_0, thus there is at least one path from b_0 to `exit` that passes through a_0. But a_0 is not control dependent on b_0, so *every* path from b_0 to `exit` must pass through a_0. Thus a_0 postdominates b_0. Similarly, for every node $b_i \in \beta$ such that there is any path from b_i to b_0, it must be the case that a_0 postdominates b_i.

Consider this set of $b_i \in \beta$ that are on a path to b_0. Now, β is connected, thus there must either be a path from b_i to b_1 or there must be a path from b_1 to b_i. If there is a path from b_1 to b_i then there is a path from b_1 to b_0 and thus a_0 also postdominates b_1. Suppose there is no path from b_1 to b_0, then there must be a path from one of the b_i to b_1. But we already know that *every* path from b_i to `exit` goes through a_0, so every path from b_1 to `exit` must go through a_0. Thus

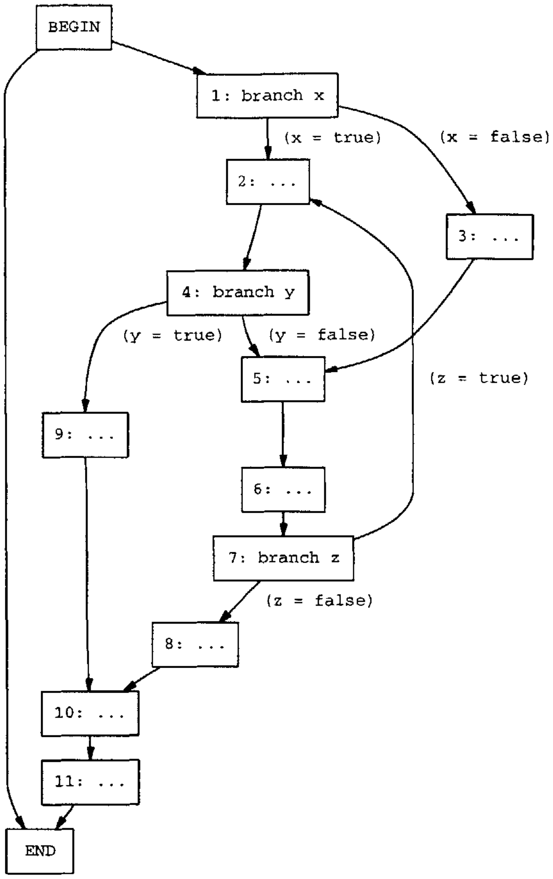


Figure 3-4: Any maximally connected subset of nodes from the bottom of a unidirectional cut *always* exits to a single point. In this case (an irreducible loop) if either node 4 or 7 is in the bottom of a unidirectional cut then so must all the nodes 2, 4, 5, 6, 7, 8 and 9. Thus a maximally connected subset containing node 4 or node 7 will exit to node 10.

a_0 postdominates both b_0 and b_1.

By a similar argument a_1 postdominates both b_1 and b_0. More specifically, a_1 *immediately* postdominates b_1, because there is a flow graph edge $b_1 \rightarrow a_1$. Thus a_0 must postdominate a_1 if it is to also postdominate b_1. A similar argument shows that a_1 must postdominate a_0. Postdominance is a partial order, thus $a_0 = a_1$. So the maximally connected group β exits to a unique node in A.

As an example, consider Figure 3-4. This figure shows a flow graph containing an irreducible loop. Suppose that we would like to include node 4 (a branch instruction) in set B of a unidirectional A-B cut. We will demonstrate that any

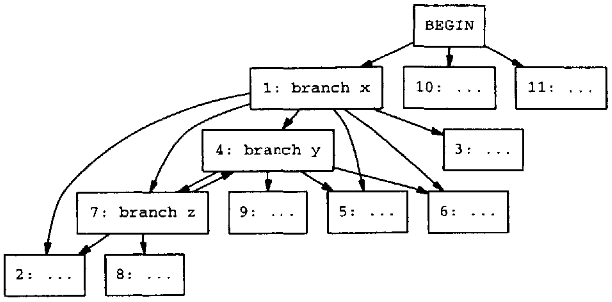


Figure 3-5: The control dependence graph for the flow graph in Figure 3-4 has a cycle between nodes 4 and 7. Thus both nodes must be on the same side of a unidirectional cut of the flow graph.

maximally connected group $\beta \subset B$ that contains node 4 must also contain nodes 8 and 9, and will, therefore, exit through node 10. We can see this by examining Figure 3-5, which shows control dependence graph corresponding to the flow graph in Figure 3-4. There is a cycle in the control dependence graph between the two exit branches in nodes 4 and 7. Thus if either of the exit branches for the irreducible loop is included on one side of the unidirectional cut, then the other must as well, because we require that no control dependences in a unidirectional cut flow from B to A.

Given a unidirectional cut A-B of a flow graph we can efficiently find all the maximally connected groups $\beta \subset B$ as follows. First we scan the edges of the flow graph to find all the edges $b_j \rightarrow a_i$ where $b_j \in B$ and $a_i \in A$. By the argument above the set of nodes a_i found in this manner represent the set of unique exits of maximal groups $\beta_i \subset B$. Then for each a_i we can find the associated maximally connected group β_i by performing a depth first search (backwards in the flow graph by following predecessor edges) starting at a_i, and where we follow only edges that lead to nodes in B.

For example, recall that in Figure 3-3 the maximally connected subgroup $\{5, 6\}$ exits to node 7. A backwards search from node 7 finds nodes 5 and 6 from set B but does not find node 12, because that would require traversing intermediate nodes

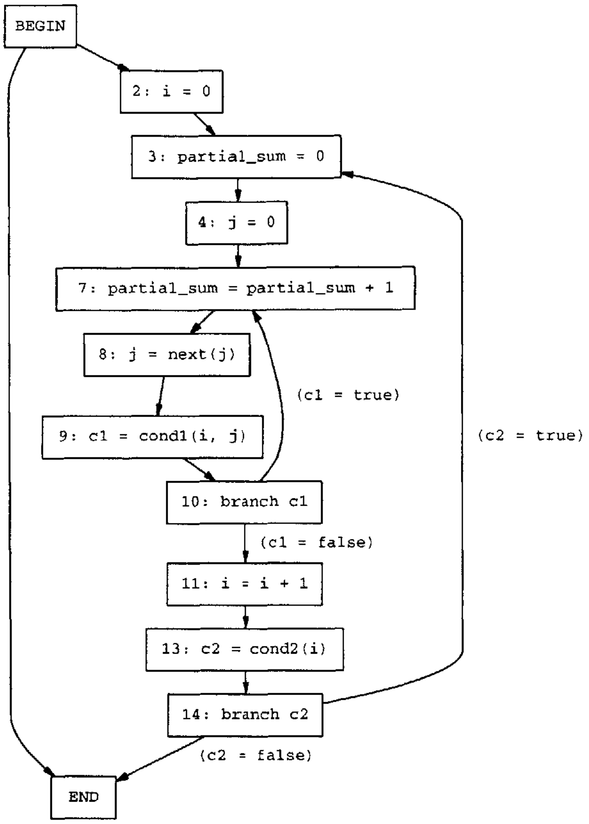


Figure 3-6: The sliced flow graph for nodes 2, 3, 4, 7, 8, 9, 10, 11, 13 and 14. For example, nodes 4 and 10 (the entries to the maximal group consisting of nodes 5 and 6) are connected to node 7, (the single exit node for group 5, 6).

(*e.g.*, node 4) that are in set A.

Now we can create a flow graph that performs exactly the work corresponding to part A of the unidirectional A-B cut by removing each of the maximally connected groups of B one by one. Given a maximally connected group $\beta_i \subset B$ with entry edges $a^y_{i_0} \rightarrow b^y_{i_0}, \ldots, a^y_{i_n} \rightarrow b^y_{i_n}$ and exits $b^x_{i_0} \rightarrow a^x_i, \ldots, b^x_{i_n} \rightarrow a^x_i$ to the unique node a^x_i, then we can remove β_i from the flow graph by removing all the nodes of β_i from the flow graph, and inserting the edges $a^y_{i_0} \rightarrow a^x_i, \ldots, a^y_{i_n} \rightarrow a^x_i$. We call the resulting flow graph the *sliced flow graph for partition* A.

Figure 3-6 shows the sliced flow graph for the partition $\{2,3,4,7,8,9,10,11,13,14\}$. The maximal groups in the original flow graph (Figure 3-3) were the sets $\{5,6\}$, and $\{12\}$. The entry edges to $\{5,6\}$ were $\{4 \rightarrow 5\}$

and $\{10 \rightarrow 6\}$, while the exit edge was $\{6 \rightarrow 7\}$. Thus in the sliced flow graph we remove nodes 5 and 6 and insert edges $\{4 \rightarrow 7\}$ and $\{10 \rightarrow 7\}$. Node 12 is removed and the edge $\{11 \rightarrow 13\}$ is inserted. Similarly, nodes 1 and 15 have been removed and edges connecting their entries to their exits have been inserted.

3.5 The Deferred Execution Queue

In addition to creating a flow graph that performs exactly the work corresponding to part A of a unidirectional A-B cut, we can also annotate the flow graph so that it keeps track of exactly the order in which the maximal groups $\beta_i \subset B$ will be executed. We do this by creating a queue data structure at the entry point of the region flow graph. We call this queue the *deferred execution queue*.

Every edge $a^y_{i_j} \rightarrow b^y_{i_j}$, $a^y_{i_j} \in A$, $b^y_{i_j} \in \beta_i$ in the flow graph represents a point at which control would have entered the maximal group β_i. Likewise, every edge $b^x_{i_k} \rightarrow a^x_i$, $b^x_{i_k} \in \beta_i$, $a^x_i \in A$, represents exactly the points at which control would have returned to region A.

Thus, after creating the sliced flow graph for partition A, by removing the regions β_i from the flow graph (as described in the previous section), we can place an instruction along each edge $a^y_{i_j} \rightarrow a^x_i$ that *pushes* the corresponding code pointer for the node $b^y_{i_j}$ on to the deferred execution queue. The edges $a^y_{i_j} \rightarrow a^x_i$ execute in exactly the order in which the β_is would have executed in the original flow graph. Thus after execution of the sliced flow graph for partition A, the deferred execution queue will contain all of the information we need to execute the code from partition B in exactly the correct order and exactly the correct number of times.

We can accomplish this by converting each β_i into a procedure that contains a flow graph identical to the flow graph that corresponds to the original β_i, but

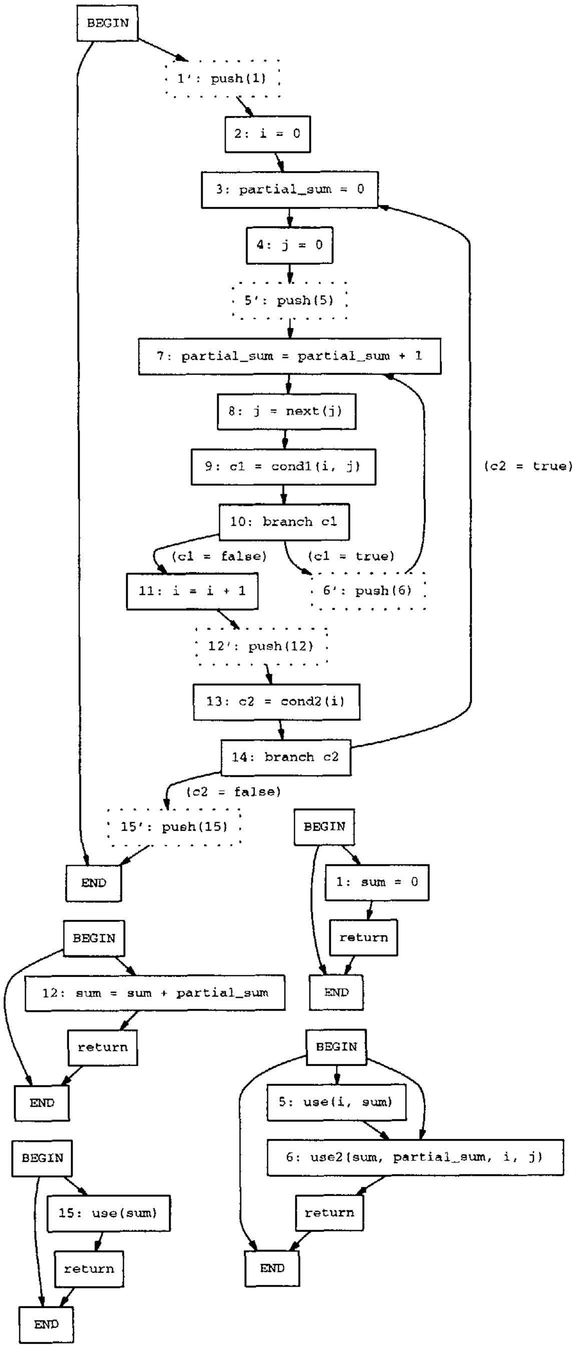


Figure 3-7: Queue conversion annotates the sliced flow graph for A with instructions that record which maximal groups of B would have executed, and in what order. Each maximal group of B is converted into its own procedure.

returns at each exit point of β_i.[1] Then we can recreate the original execution sequence of partition B by *popping* each code pointer $b_{i_j}^y$ off the front of the deferred execution queue and calling the corresponding procedure.

The queue conversion of our example program is shown in Figure 3-7. Push instructions for the appropriate maximal group entry points have been inserted along the edges begin → 2, 4 → 7, 10 → 7, 11 → 13 and 14 → end. The maximal groups $\{1\}, \{5, 6\}, \{12\}$ and $\{15\}$ are each converted into a procedure.

Closure Conversion

If it were the case that there were no register storage dependences flowing from B to A then the deferred execution queue would be sufficient. Our definition of a unidirectional A-B cut did not, however, exclude the existence of use-def or def-def chains flowing from region B to region A. Thus, we must solve the problem that partition A might produce a value in register x that is used in region B but then might overwrite the register with a new value before we have a chance to execute the corresponding code from partition B off the deferred execution queue.

The problem is that the objects we are pushing and popping on to the deferred execution queue are merely code pointers. Instead, we should be pushing and popping *closures*. A closure is an object that consists of the code pointer together with an *environment* (set of name-value pairs) that represents the saved machine state in which we want to run the corresponding code. Thus a closure represents a suspended computation.[2]

Consider the registers (variables) associated with the set of def-use chains that

[1] If the underlying infrastructure does not support multiple-entry procedures, then each maximal group β_i can be further partitioned into a set of subprocedures, each corresponding to a maximal basic block of β_i. Each subprocedure that does not exit β_i tail calls [107] its successor(s) from β_i.

[2] Closures that take no arguments, as is the case here, are sometimes called *thunks*, but typically only in the context of compiling call-by-name languages, which is not the case here.

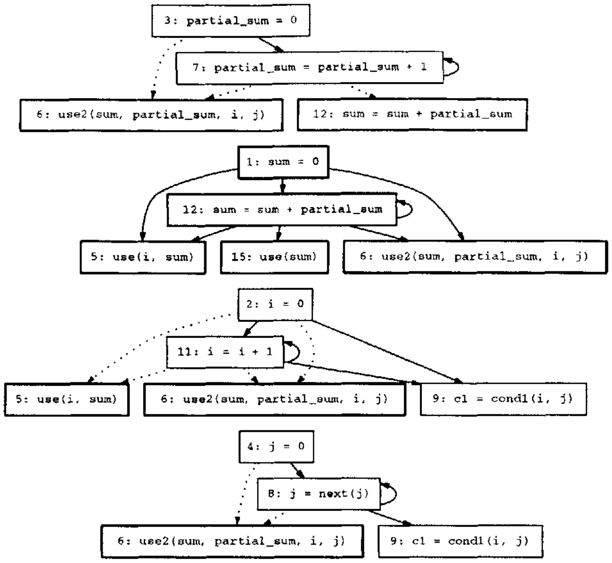

3: partial_sum = 0
7: partial_sum = partial_sum + 1
6: use2(sum, partial_sum, i, j)
12: sum = sum + partial_sum
1: sum = 0
12: sum = sum + partial_sum
5: use(i, sum)
15: use(sum)
6: use2(sum, partial_sum, i, j)
2: i = 0
11: i = i + 1
5: use(i, sum)
6: use2(sum, partial_sum, i, j)
9: c1 = cond1(i, j)
4: j = 0
8: j = next(j)
6: use2(sum, partial_sum, i, j)
9: c1 = cond1(i, j)

Figure 3-8: Cuts in the du-webs for variables `i`, `j`, `sum` and `partial_sum` given the cut from nodes 2, 3, 4, 7, 8, 9, 10, 11, 13, 14 to nodes 1, 5, 6, 12, 15 (shown in bold). Def-use chains that cross the cut are shown as dotted edges.

reach into a maximal group $\beta_i \subset B$. If we save a copy of the values associated with each of these registers along with the code pointer, then we can eliminate all the use-def chains that flow from B to A, and replace them, instead, with use-def chains that flow only within partition A.

To convert each maximal group $\beta_i \subset B$ into a closure we transform the code as follows.

1. Consider the graph of nodes corresponding to β_i. For each of the entry nodes $b^y_{i_j}$ of this graph find the set of nodes $\beta_{i_j} \subset \beta_i$ reachable from $b^y_{i_j}$. For each set β_{i_j} find the set of variables, $V_{ij} = \{v_{ijk}\}$ such that there is a def-use chain flowing from partition A into β_{i_j}. (That is, there is a definition of v_{ijk} somewhere in A and a use of v_{ijk} somewhere in β_{i_j}). Figure 3-8 shows that this set can be easily derived from the du-webs corresponding to the flow graph. For example, $V_{\{12\}} = \{\texttt{partial\_sum}\}$ and $V_{\{15\}} = \emptyset$. The maximal group $\beta_{\{5,6\}}$ has two entry points, (at 5 and 6). In this case it happens that $V_{\{5,6\},5} = V_{\{5,6\},6} = \{\texttt{i},\texttt{j},\texttt{partial\_sum}\}$.

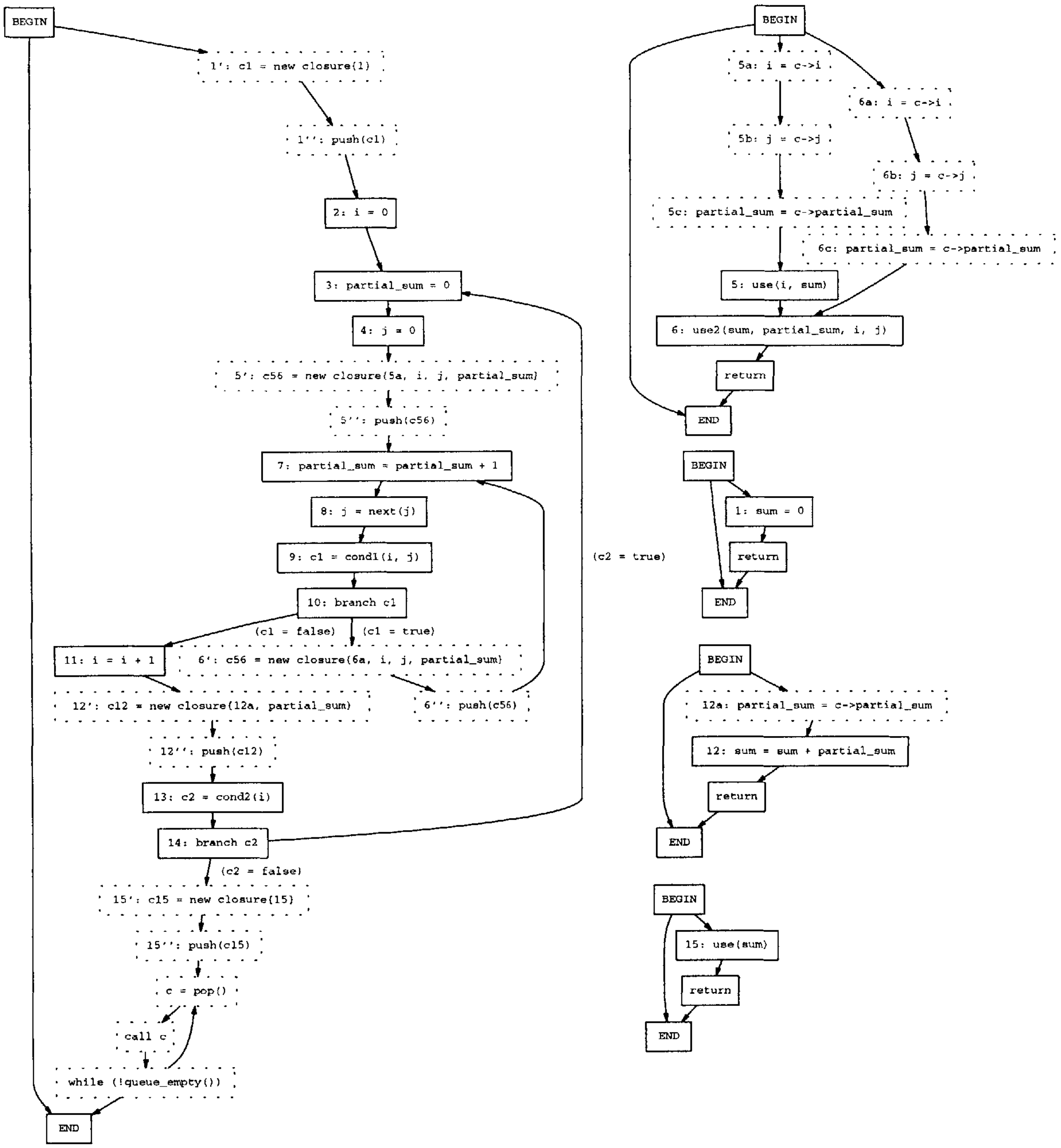


Figure 3-9: Closure conversion ensures that each value crossing the cut gets copied into a dynamically allocated structure before the corresponding register gets overwritten.

2. Consider each edge $a^y_{i_j} \rightarrow a^x_i$ in the sliced flow graph for partition A that corresponds to entry point $b^y_{i_j}$ of maximal group β_i. Along this edge we place an instruction that dynamically allocates a structure with $|V_{ij}| + 1$ slots, then copies the *values* $\langle b^y_{i_j}, v_{ij1}, \ldots, v_{ij|V_{ij}|}\rangle$ into the structure, and then pushes a pointer to this structure onto the deferred execution queue. Figure 3-9 demonstrates this process. For example, along the edge $4 \rightarrow 7$ we have placed instructions that allocate a structure containing the values of the code pointer, "5", and the copies of the values contained in variables, `i`, `j` and `partial_sum`.

3. For each β_i we create a procedure that takes a single argument, `c`, which is a pointer to the structure representing the closure. The procedure has the same control flow as the original subgraph for β_i except that along each entry we place a sequence of instructions that copies each entry from each slot of the closure into the corresponding variable v_{i_k}. Figure 3-9 shows that the two entries to the procedure corresponding to the maximal group $\{5, 6\}$ have been augmented with instructions that copy the values of variables `i`, `j` and `partial_sum` out of the corresponding closure structure.

4. To invoke a closure from the deferred execution queue we pop the pointer to the closure off the front of the queue. The first slot of the corresponding structure is a pointer to the code for the procedure corresponding to β_i. Thus we call this procedure, passing as an argument the pointer to the closure itself. In Figure 3-9 this process is shown towards the bottom of the original procedure, where we have inserted a loop that pops closures off the deferred execution queue, and invokes them.

This completes the basic scalar queue conversion transformation. Because a copy of each value reaching a maximal group β_i is made just before the point in the program when it would have been used, the correct set of values reaches each maximal group, even when execution of the group is deferred. Additionally, since the copy is created in partition A, rather than partition B, we have eliminated any use-def chains that flowed from partition B to partition A. In the next section we will demonstrate how to generalize the result to eliminate def-def chains flowing from B to A. In Chapter 4 we will show how to move the closure creation points so that they least restrict further transformations to partition A.

3.6 Unidirectional Renaming

In the previous section we demonstrated that we could transform a unidirectional A-B cut on a single-entry single-exit region into an equivalent piece of code such that all the instructions in partition A run, dynamically, before all the instructions in partition B. Further we demonstrated that we could do this even in the presence of use-def chains flowing from partition B to partition A. In this section we will show that the result can be generalized, in a straightforward way, to A-B cuts where there are additionally def-def chains flowing from partition B to partition A.

The result depends on the fact that given a unidirectional A-B cut, we can insert a new instruction anywhere in the flow graph, and that if we give that instruction a labeling that includes it in partition B, then we will not introduce any new control dependences that flow from partition B to partition A. (The opposite is not true. That is, if we place a new instruction in partition A at a point that is control dependent on an instruction in partition B, then we will introduce a control dependence edge that will violate the unidirectionality of the cut.)

For the remainder of the thesis we will assume that each du-web in the program has been given a unique name. This transformation is already done by most

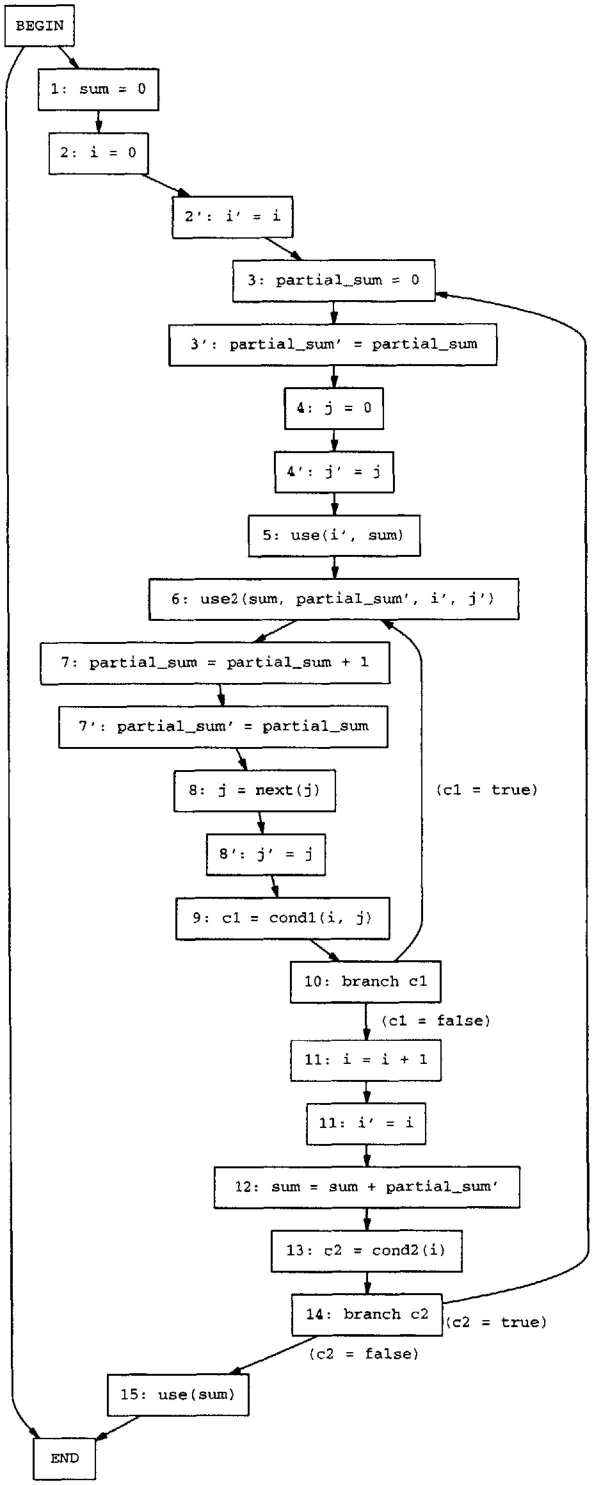


Figure 3-10: An example of statically renaming the variables `i`, `j` and `partial_sum`.

optimizing compilers because it is so common for programmers to reuse variable names, even when the variables are completely independent. For example, many programmers reuse the variable name i for the index of most loops. Once the du-webs are calculated, as described in Section 2.2, we iterate through the set of du-webs for each variable x, renaming all the uses and definitions in each node in the ith web to x_i. Thus we can, without loss of generality, talk about *the* du-web for a particular variable.

Now consider the du-web for variable x on a unidirectional cut A-B where some of the definitions of x are in A and some of the uses of x are in B. Thus, there is a value dependence flowing from A to B. It may be the case that there are definitions of x in B and uses of x in A, but, because A-B is a unidirectional cut, it cannot be the case that there are any def-use chains reaching from B to A. Thus the du-web has a unidirectional structure, just as the value dependence graph did. (In fact, another way of seeing this is to observe that each du-web is an induced subgraph of the value dependence graph). For example, in the du-webs shown in Figure 3-8 one can observe that the def-use chains crossing the cut (shown with dotted edges) all flow in one direction.

The du-web for variable x thus has a structure that is *almost* renameable, except for those edges in the web that cross the cut. Suppose, however that we were to place a copy instruction "x' = x" directly after each of the definitions of x from A that reach a use in B. Then we could rename all the definitions and uses of x in B to x'. The program will have exactly the same semantics, but we will have eliminated all of the def-def chains flowing from B to A. We will call such a renaming of of a du-web that crosses a unidirectional cut a *unidirectional renaming*.

An example of a unidirectional renaming is shown in Figure 3-10. Each time one of the variables i, j and partial_sum is modified it is copied to a corresponding variable i', j' or partial_sum'. The uses of i, j and partial_sum

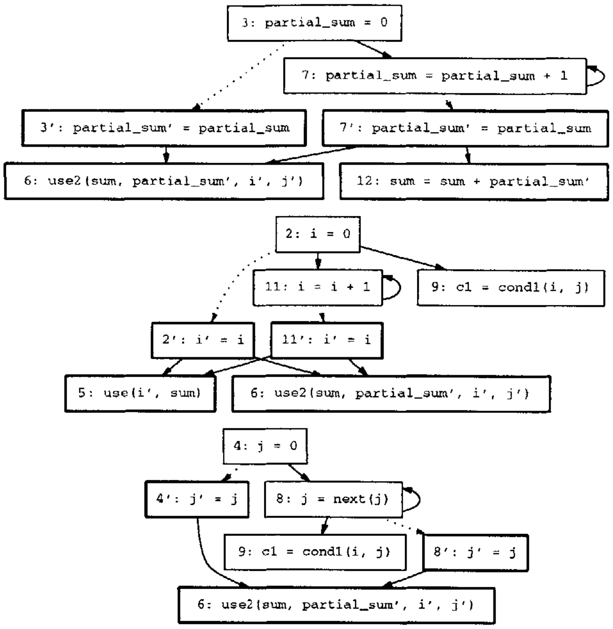

3: partial_sum = 0
7: partial_sum = partial_sum + 1
3': partial_sum' = partial_sum
7': partial_sum' = partial_sum
6: use2(sum, partial_sum', i', j')
12: sum = sum + partial_sum'
2: i = 0
11: i = i + 1
9: c1 = cond1(i, j)
2': i' = i
11': i' = i
5: use(i', sum)
6: use2(sum, partial_sum', i', j')
4: j = 0
4': j' = j
8: j = next(j)
9: c1 = cond1(i, j)
8': j' = j
6: use2(sum, partial_sum', i', j')

Figure 3-11: The unidirectionally renamed du-webs for variables `i`, `j` and `partial_sum`.

in partition B are then renamed to `i'`, `j'` and `partial_sum'`. The du-webs for this unidirectional renaming are shown in Figure 3-11.

To see how unidirectional renaming eliminates backwards flowing def-def chains, consider Figure 3-12. We examine the cut from the set of nodes $\{1, 2, 3, 4, 6, 7\}$ to the set $\{5, 8\}$. This is a unidirectional cut because all of the value and control dependences flow from the first set to the second. Figure 3-13 shows the corresponding du-web for variable x. There is, however, a def-def chain flowing from node 5 to node 7 (against the cut direction).

Unidirectionally renaming the flow graph, as shown in Figures 3-14 and 3-15 solves this problem. After placing copy instructions "`x' = x`" after the definitions that reach across the cut, and renaming x to `x'` in nodes 5 and 7, *all* of the definitions of x are on one side of the cut while *all* of the definitions of `x'` are on the other side of the cut. Thus there are *no* def-def chains flowing across the cut. All the def-def chains are now contained within one partition or the other.

Placing the copy instructions for the unidirectional renaming directly after the

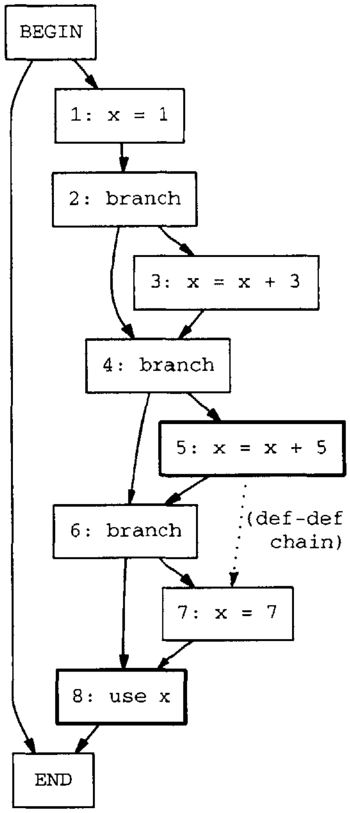


Figure 3-12: The cut separating nodes 1, 2, 3, 4, 6 and 7 from nodes 5 and 8 is unidirectional because all the value and control dependences flow unidirectionally. The def-def chain flowing from node 5 to node 7 does *not* violate the unidirectionality of the cut.

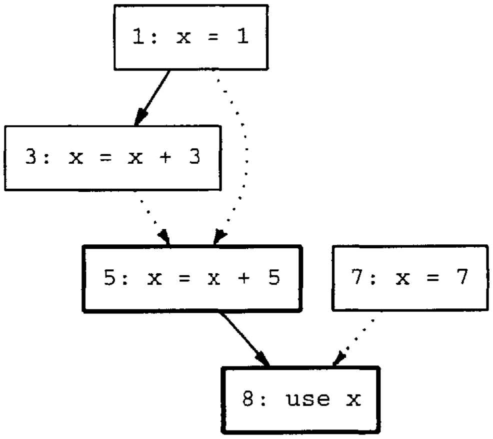


Figure 3-13: The du-web for variable x from the flow graph in Figure 3-12. The cut is unidirectional because all the def-use chains flow in one direction across the cut. Dotted edges show cut edges.

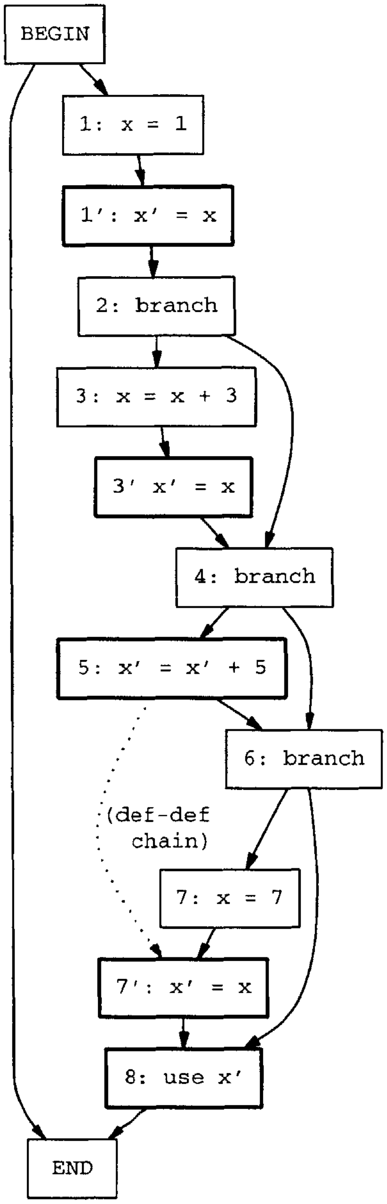


Figure 3-14: After unidirectionally renaming the variable x the def-def chain between nodes 5 and 7 is eliminated, and replaced instead with a def-def chain from node 5 to node 7′. The new def-def chain does not cross the cut because node 5 and 7′ are both in the same partition (indicated by nodes with a bold outline).

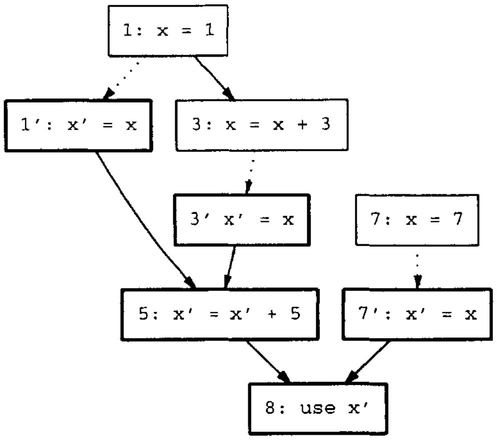


Figure 3-15: The du-web for variables x and x' from the flow graph in Figure 3-14. The cut is still unidirectional because all the def-use chains flow in one direction across the cut. Dotted edges show cut edges. Now, however, there is no def-def chain crossing the cut because definitions of variable x happen in one partition, while definitions of variable x' happen in the other.

corresponding definition of each variable produces a correct result, but, in fact, we can do better. We can maintain the program semantics and eliminate the output dependences if we place the copy instructions along *any* set of edges in the program that have the property that they cover all the paths leading from definitions of x in A that reach uses of x in B and are not reached by any of the definitions of x in B. In the next section we will show how to derive such a set of edges that is optimal, in the sense that they will execute only as often as the innermost loop that contains both the definitions and the uses.

Thus given any unidirectional cut A-B we can insert copy instructions into each du-web that has edges flowing from A to B and derive a semantically equivalent flow graph with the property that there are no def-def chains flowing from B to A.

There is a second, perhaps more important, benefit of performing unidirectional renaming on the du-webs that cross the cut. This is that after renaming, closure conversion and a single pass of local copy propagation, all the *uses* of a variable will be entirely contained on one side of the cut or the other. That is, all communication across the cut will occur through the deferred execution queue.

There will be no "shared" scalar variables. Because of this property we perform unidirectional renaming on *all* du-webs that cross the cut, even when there are no def-def chains that need to be broken. Specific examples are given in Chapter 4 and Section 5.2.

3.7 Wrapup

In this chapter we demonstrated that, through the process of scalar queue conversion, we can restructure *any* unidirectional cut of the *true* scalar dependences in *any* program, and reschedule the code so that all of the instructions in the top half of the cut run (dynamically) before all of the instructions in the bottom half. Scalar queue conversion *completely* eliminates scalar anti- and output-dependences that might otherwise make this rescheduling impossible.

In this chapter we described how to apply scalar queue conversion to a single-entry single-exit region of code. Chapter 5 demonstrates how to extend the result to regions of code with multiple exits, by a single application of scalar queue conversion to a somewhat larger region of code. Chapter 5 also describes a set of transformations that eliminate memory dependences from the program dependence graph, thus exposing unidirectional cuts in a wider variety of circumstances. Chapter 6 describes how to use scalar queue conversion as a subroutine of a generalized version of loop distribution that can reschedule regions of code with arbitrary control flow (including inner loops).

Chapter 7 describes the SUDS Software Un-Do System, which complements scalar queue conversion in two ways. First, as mentioned above, scalar queue conversion is unsafe in the sense that it does not strictly guarantee the amount of dynamic memory the transformed program will allocate. The SUDS system solves this problem by allowing scalar queue conversion to be applied *speculatively*.

SUDS checkpoints the system state, and then runs the transformed program. If the transformation causes a memory allocation error, then the execution can be rolled back to the checkpointed state, and resumed with the original (untransformed) code.

SUDS additionally complements scalar queue conversion by providing memory dependence speculation. Memory dependence speculation allows scalar queue conversion to work across memory dependences that can not be handled by the techniques in Chapter 5, and that would otherwise hide unidirectional cuts.

Chapter 4

Optimal Unidirectional Renaming

In Chapter 3 we demonstrated that, through the process of scalar queue conversion, we could transform a unidirectional cut A-B on a single-entry single-exit region into an equivalent piece of code such that all the instructions in partition A run, dynamically, before all the instructions in partition B. Further, in Section 3.6, we demonstrated that, through a process of static unidirectional renaming, we could do this even in the presence of use-def or def-def chains flowing from partition B to partition A. In this section we will demonstrate that we can move the unidirectional renaming points to a position in the flow graph that is optimal, in the sense that we place them at the legal points in the graph such that they are in the outermost possible loop.

We do this by implementing an eager form of partial dead code elimination [64]. The algorithm takes advantage of two additional facts. First, that the copy instructions we inserted for unidirectional renaming (Section 3.6) can be moved or replicated at any point in the graph that is not reached by any other definition that is not a copy instruction. Additionally, we take advantage of a useful property of the static single assignment (SSA) flow graph. After conversion to SSA every use of a variable in the program will be reached by only a single definition, and, further, that definition will dominate the use [30].

Informally, the algorithm moves copy instructions downward through join points in the flow graph until it reaches a join point that dominates a use. This node has the property that it is the earliest (static) point in the program where we can determine exactly the value that reaches the use. Then we use the partial dead code elimination algorithm to move the copy instruction through the intervening branches in the flow graph that might make the copy instruction less likely to execute at all.

4.1 "Least Looped" Copy Points

The objective of optimal unidirectional renaming is similar to the objective of conversion to static single assignment form [30]. That is, we desire to connect each use of a variable with a single copy statement. The only place where this condition might be violated is at join points in the flow graph. That is, places in the flow graph that two different definitions might reach. But recall the definition of the dominance frontier of a node x. This is the set of edges in the flow graph that flow between nodes y and z where all paths to y go through x, but where there are paths to z that do not go through x. In other words, z is a join node in the flow graph such that a definition at node x will no longer be unique. Consider, for example, the definitions of variable j in the flow graph in Figure 4-1. The definition at node 8 dominates nodes 9 and 10, but not node 6. So node 6 is on the dominance frontier of node 8, and indeed, two definitions of j can reach node 6. One from node 8 and the other from node 4.

The key to the construction of static single assignment form is that it places copy instructions on the *iterated* dominance frontier of each definition. A straightforward method of constructing the iterated dominance frontier for a set of defi-

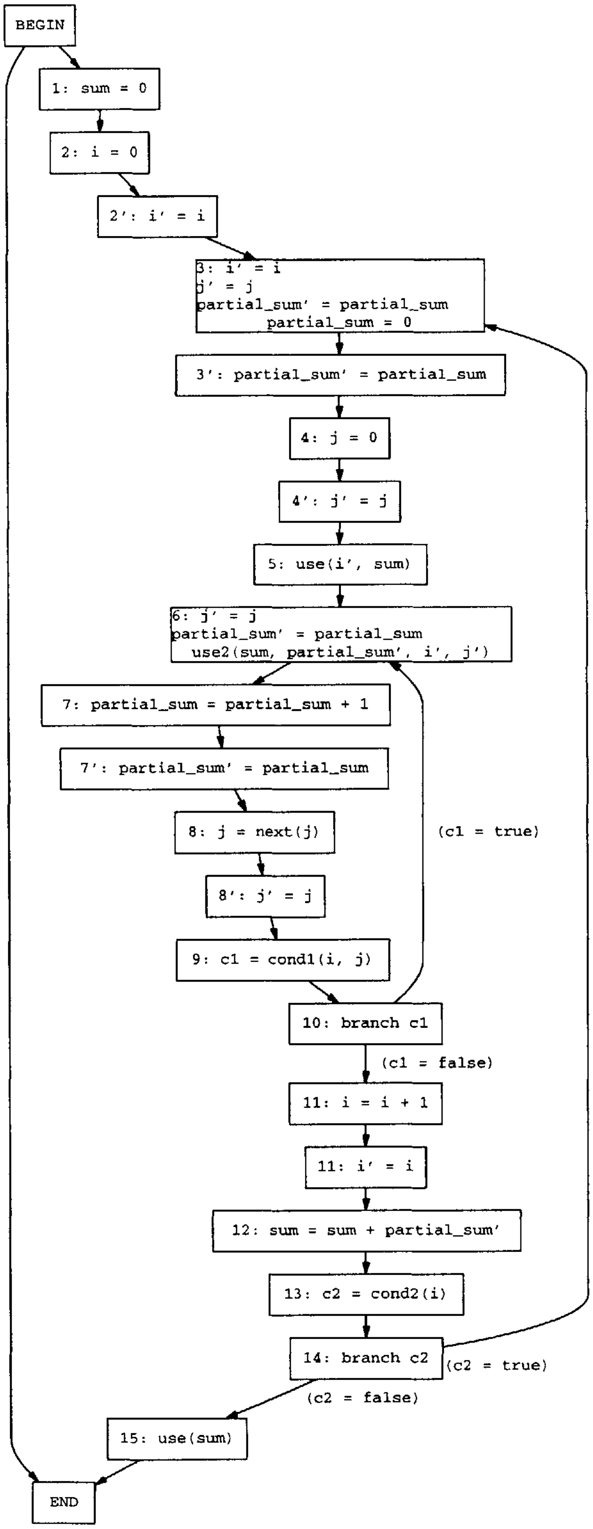


Figure 4-1: After unidirectional renaming we place replicas of each copy instructions at all the join points reachable by that copy instruction.

nitions that are already copy instructions is as follows. For each copy instruction "`x' = x`", replicate the instruction at each node in the dominance frontier of the definition that has not already been marked. Since the new instruction is also a new definition for the variable, the procedure must iterate until it reaches a fixed point [30]. The caveat, in this case, is that we must *not* place instruction replicas at any point in the graph that is also reached by a "real" definition of the variable `x'`.

In Figure 4-1 we show how replicas of the copy instructions "`i' = i`", "`j' = j`" and "`partial_sum' = partial_sum`" are placed at the iterated dominance frontier of the unidirectionally renamed flow graph from Figure 3-10.

More concretely, we proceed as follows. We are given a du-web for variable x over a unidirectional cut A-B. As described in Section 3.6 we give this web a unidirectional renaming by renaming x to x′ in all nodes belonging to B, and then inserting new copy instructions, $x' = x$ in the flow graph directly after the definitions of x in the original flow graph that both belong to A and reach one of the uses in B. The new copy instructions are included in partition B, rather than A, and thus we are left with a semantically equivalent flow graph that is still unidirectionally cut, but is guaranteed not to have any def-def chains flowing from B to A.

We now define five subsets of the nodes in the unidirectionally renamed du-webs with original variable x and renamed variable x′. The set $\mathrm{Copy}_{x' \leftarrow x}$ consists of the set of nodes that contain newly inserted copy instructions `x' = x`. The set Def_x is the set of nodes that define x. The set $\mathrm{Def}_{x'}$ is the set of nodes that define x′ *minus* the set $\mathrm{Copy}_{x' \leftarrow x}$. The set Use_x is the set of nodes that use x *minus* the set $\mathrm{Copy}_{x' \leftarrow x}$. The set $\mathrm{Use}_{x'}$ is the set of nodes that use x′.

Now recall the definition, from page 34 of the *Reaching* relation for subsets *Gen* and *Pass* of the set of nodes in a flow graph. This was the set of nodes for which there is a path in the flow graph from some node in *Gen*, passing only through nodes in *Pass*. We then let $\mathrm{Illegal}_{x' \leftarrow x}$ be the set of nodes reached by $\mathrm{Def}_{x'}$. That is, we generate the *Reaching* relation with *Gen* $= \text{Def}_{x'}$ and *Pass* $= \overline{\text{Def}_{x'} \cup \text{Def}_{x}}$. $\text{Illegal}_{x' \leftarrow x}$ is then the set of points in the program at which inserting an instruction `x' = x` might cause the program to produce incorrect results.

Now let $\text{ItDom}_{x' \leftarrow x}$ be the set of nodes that corresponds to the iterated dominance frontier of $\text{Copy}_{x' \leftarrow x}$. If we place copy instructions at all the nodes in $\text{AllCopies} = (\text{ItDom}_{x' \leftarrow x} \cup \text{Copy}_{x' \leftarrow x}) - \text{Illegal}_{x' \leftarrow x}$ we will have, by the properties of the iterated dominance frontier [30], found exactly the set of join nodes through which it would be legal to move the copy instructions.

4.2 Lazy Dead Copy Elimination

Now let $\text{Live}_{x'}$ be the set of nodes for which there is a path to a use of x', that does not pass through a definition of x' [4]. We can generate $\text{Live}_{x'}$ by generating *Reaching* on the *reverse* flow graph with *Gen* $= \text{Use}_{x'}$ and *Pass* $= \overline{\text{Def}_{x'} \cup \text{Copy}_{x' \leftarrow x}}$. We can eliminate any *dead* copies by keeping only those on nodes in $\text{AllCopies} \cap \text{Live}_{x'}$, and deleting the rest, since the value they produce will never be used by any instruction. For example, in Figure 4-1 the copies of `i` at nodes 2′, and 11′ are dead. Additionally, the copies of `j` at nodes 3, 4′ and 8′ are dead, and the copies of `partial_sum` are dead at nodes 3 and 3′. Removal of these dead copy instructions is shown in Figure 4-2.

Finally, following Knoop *et al* [64], we define $\text{ReachingUses}_{x'}$ as the set of nodes that can be reached by a use of x' without passing through a definition of x'. ("ReachingUses" corresponds to the complement of the set that Knoop *et al* call "Delayed"). Then if we let $\text{BadNodes}_{x'} = \text{ReachingUses}_{x'} \cup \text{Illegal}_{x' \leftarrow x}$ we can *sink* the copy instructions to the frontier between $\overline{\text{BadNodes}_{x'}}$ and $\text{BadNodes}_{x'}$. That is, to edges $m \rightarrow n$ in the flow graph where $m \in \overline{\text{BadNodes}_{x'}}$ and $n \in \text{BadNodes}_{x'}$. Iteration of dead copy elimination and copy sinking produces the optimal re-

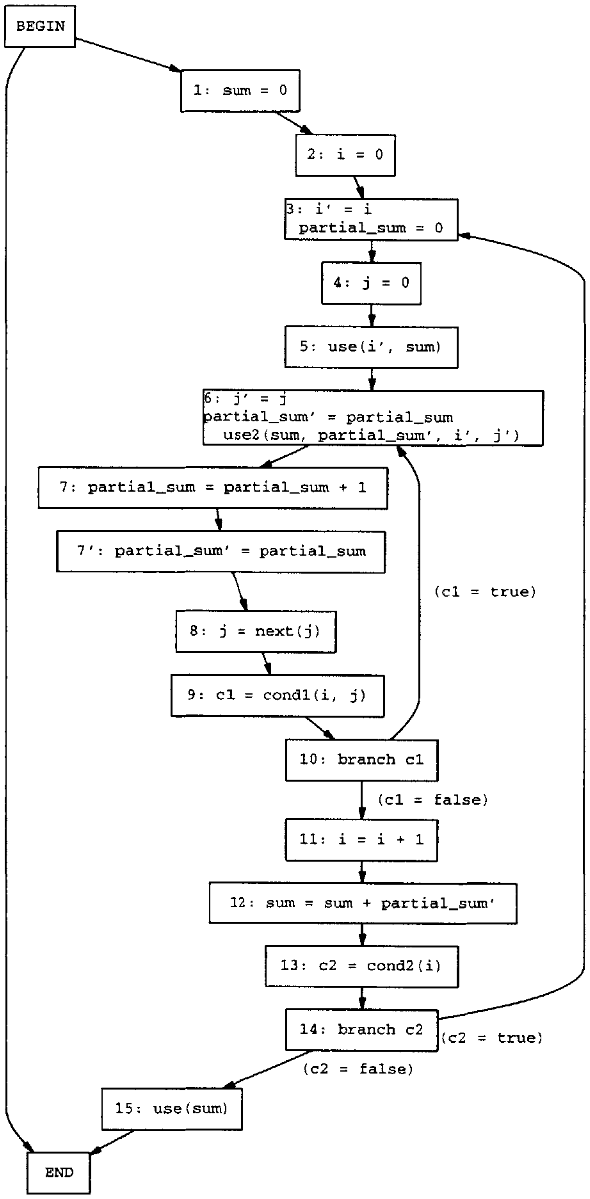


Figure 4-2: After placement of copies on the iterated dominance frontier at most one copy instruction will reach each use, and the remaining copy instructions can be dead-code eliminated.

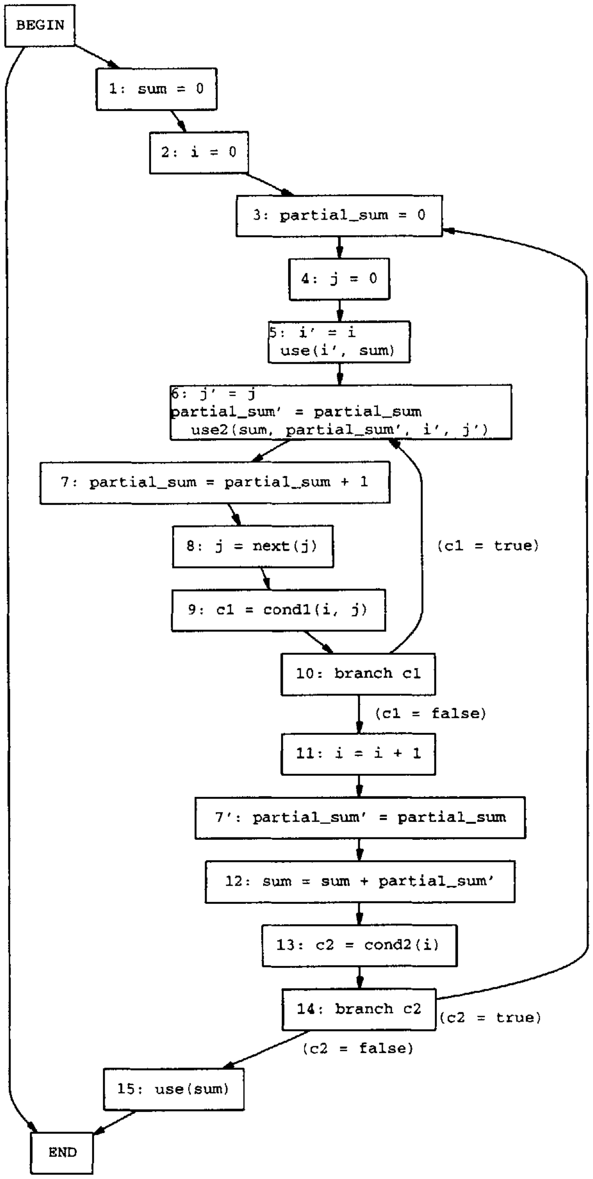


Figure 4-3: We can "sink" the copy instruction in node 7′ out of the inner loop.

sult. Figure 4-3 shows the sinking of the copy instruction partial_sum' = partial_sum at node 7′ from inside the inner loop to a position in the outer loop just before the corresponding use.

Figure 4-4 shows what happens when optimal unidirectional renaming precedes scalar queue conversion. We point out two things when comparing Figure 4-4 to Figure 3-9. First, the scalars used by the two halves of the partitioning are entirely distinct. The sliced flow graph corresponding to the top of the cut defines and uses only variables i, j and partial_sum. The flow graphs for the closures produced by scalar queue conversion define and use only variables i', j', partial_sum' and sum. Second, note that unidirectional renaming has made it possible to avoid the extra queueing and dequeuing of variable i that occurs in the inner loop in Figure 3-9.

Finally, we note one additional feature of optimal unidirectional renaming. It *tends* to be the case that optimal unidirectional renaming makes du-webs sparser. This is intuitively reasonable, given that the optimal unidirectional renaming process, like conversion to SSA form, puts copy instructions at the iterated dominance frontier of each definition. The result is that most (but not all) of the unidirectionally renamed uses will be reached by only a single definition. Compare, for example, the du-chains for partial_sum and partial_sum' after optimal unidirectional renaming (Figure 4-5) with the du-chains shown in Figure 3-11. After optimal unidirectional renaming the chains for partial_sum' actually form two independent webs, and can thus be given different static names.

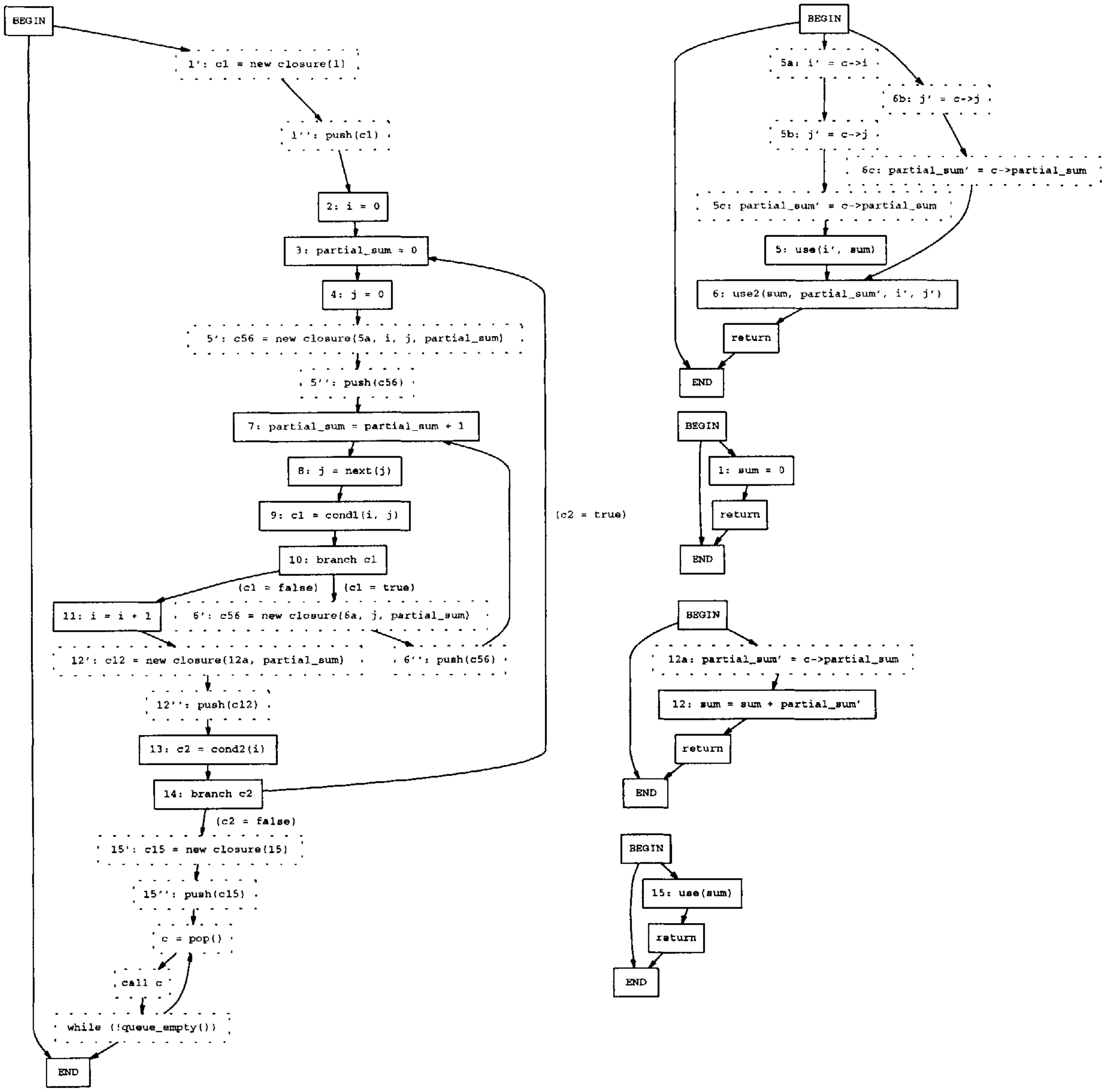


Figure 4-4: After scalar queue conversion of the optimally renamed flow graph from Figure 4-3 and a pass of local copy propagation there are neither any uses or definitions of variables i, j or partial_sum in any of the closures produced by scalar queue conversion.

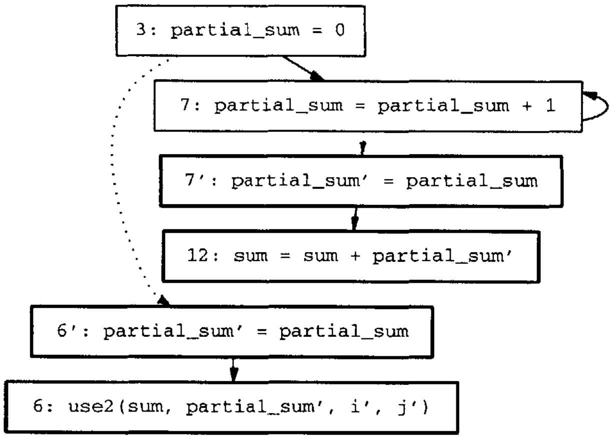

3: partial_sum = 0
7: partial_sum = partial_sum + 1
7': partial_sum' = partial_sum
12: sum = sum + partial_sum'
6': partial_sum' = partial_sum
6: use2(sum, partial_sum', i', j')

Figure 4-5: Optimal unidirectional renaming introduces additional opportunities for static renaming. The du-chains for the variables partial_sum and partial_sum' have been cut in such a way that the du-chains for partial_sum' actually form two independent webs.

Chapter 5

Extensions and Improvements to Scalar Queue Conversion

Scalar queue conversion provides the basic mechanism for renaming and rescheduling any unidirectional cut of a single-entry single-exit value dependence graph. In this chapter we discuss five practical extensions to scalar queue conversion. In Section 5.1 we demonstrate how to extend scalar queue conversion to single-entry multiple-exit regions of a flow graph. A particularly interesting feature of this extension is that it is also an *application* of scalar queue conversion, because we use scalar queue conversion itself to separate the multi-exit region from its successors in the flow graph. Section 5.2 further demonstrates how to *localize* scalars to the closures created by scalar queue conversion thereby enabling concurrent execution.

Scalar queue conversion guarantees that we can reschedule any *unidirectional* cut of the value dependence graph. In Chapters 2, 3 and 4 we took a conservative view of memory dependences by inserting edges in the value dependence graph for *all* load-after-store, store-after-load and store-after-store dependences. These, extra, conservative dependences may restrict the applicability of scalar queue conversion because they might create cycles in the value dependence graph across what would otherwise be unidirectional cuts. In Sections 5.3, 5.4 and 5.5 we discuss three methods of improving the quality of memory dependence information in the value dependence graph, widening the applicability of scalar queue conversion.

5.1 Restructuring Loops with Multiple Exits

The scalar queue conversion transformation given in Chapter 3 is described only in terms of single-entry single-exit regions of flow graph. It turns out, however, that a single application of scalar queue conversion to a single-exit region can be used to *extract* a multiple exit subloop of that region. The main intuition is that scalar queue conversion makes the *continuations* of each loop iteration explicit. That is, we can treat a region of code as a computation that, along with the rest of the work it does, also explicitly computes a "next program counter."

Given a flow graph we can identify a natural loop using standard techniques. Recall that a *back edge* in the flow graph is any edge $b \rightarrow h$ where h dominates b. Then h is called the *loop header*, b is called the *loop branch*, and every reducible loop can be uniquely identified by its back edge. The *natural loop* associated with a back edge is defined to be the set of nodes that can reach b without going through h [4]. Further the *loop exits* are exactly those edges $x \rightarrow y$ where node x is a node in the loop and node y is a node outside the loop.

Consider the flow graph in Figure 5-1. Here the back edge is the edge from node 8 to node 1. The natural loop associated with that back edge is the set of nodes $\{1, 2, 3, 5, 6, 7, 8\}$. The loop exits are the edge from node 3 to node 4 and the edge from node 8 to node 9.

Given a natural loop with more than one exit, we transform that loop into a

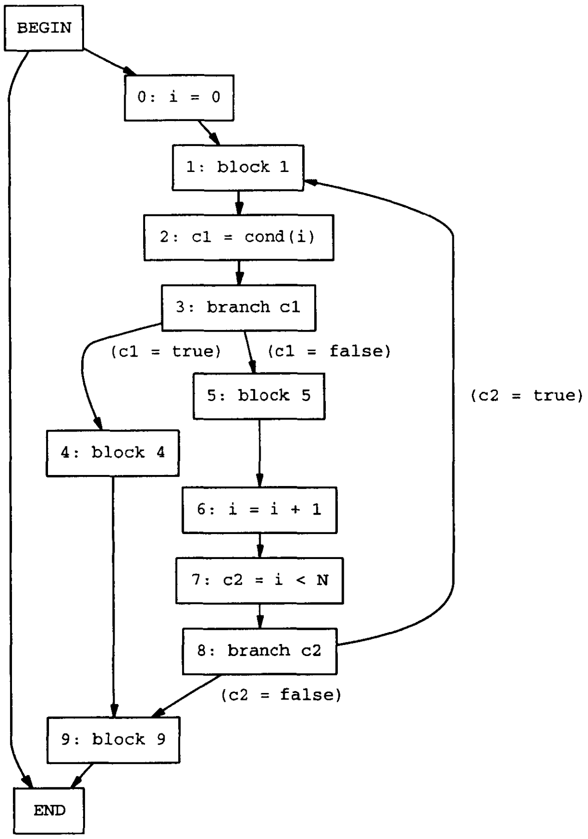


Figure 5-1: The flow graph for a loop with multiple exits.

single exit loop, using a stripped-down version of scalar queue conversion, as follows. We create a variable, k, and initialize it to 0 at the top of the loop. We create a new loop branch b' that branches to the top of the loop if $k = 0$ and exits the loop otherwise. Then for every loop exit $x \rightarrow y$ in the original loop we redirect the edge as $x \rightarrow y' \rightarrow b'$ where y' is a new node that sets k to the *label* of node y. Finally, we insert a new node b'' after the exit from b' where b'' is a multiway branch that jumps to the label stored in variable k.

This transforms the example flow graph as shown in Figure 5-2. Node $1'$ initializes the continuation variable k. To exit from the loop nodes 3 and 8 now conditionally set k to the correct non-zero value and then go to node $8'$. Node $8'$ is now the only loop exit, and exits only when k is non-zero. Finally, when the loop is exited, node $8''$ uses the value stored in the continuation variable k to jump to the correct code, either block 4 or block 9, depending on whether node 3 or node 8 caused the loop to exit.

In the expected common case, where the loop is not exited, control flows the same way it would have in the original code. The continuation variable k is initialized to zero at the top of the loop iteration. Neither loop exit is taken, so nodes $5, 6, 7$ and 8 execute while nodes $4'$ and $9'$ do not, and the value of k will be zero when node $8'$ is reached. Thus, node $8'$ branches back to node $1'$ at the top of the loop.

A similar transformation has been implemented previously in the loop distribution phase of the IBM PTRAN compiler [51]. The SUDS compiler implements the additional optimization that in the frequently observed case that all exit nodes x_i exit to the same node y along edges $x_i \rightarrow y$, then b' can simply exit to y and the multiway branch b'' can be omitted. This optimization is particularly desirable, because it allows the continuation variable, k to be treated as private to the loop body during subsequent compiler phases.

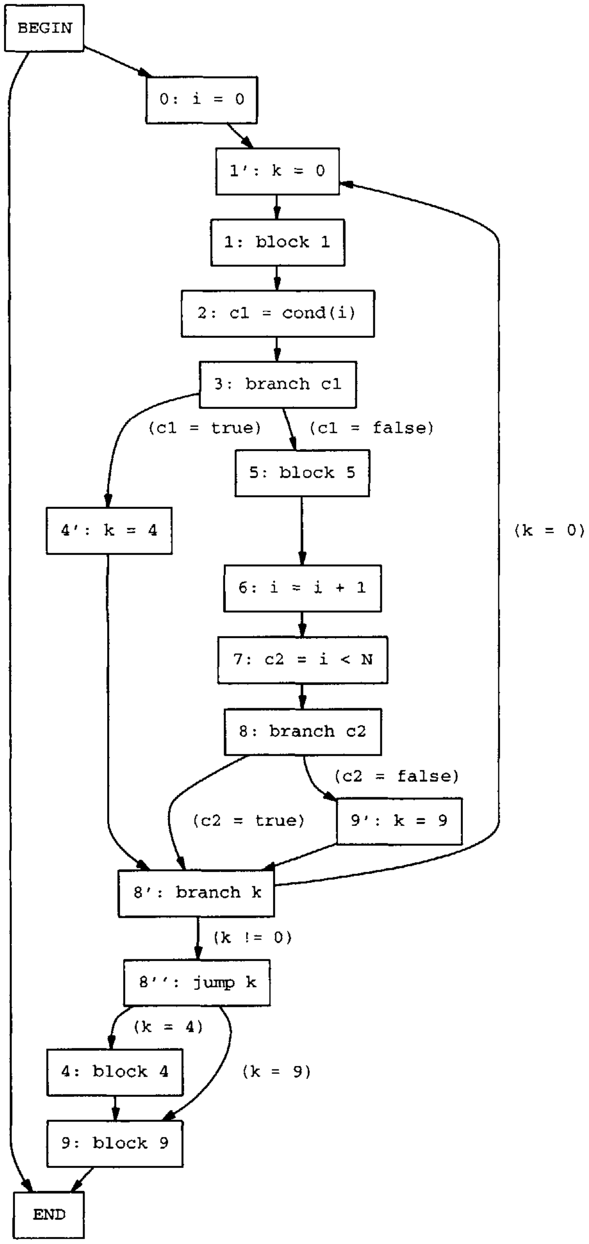


Figure 5-2: The loop of Figure 5-1 transformed so that it has only a single exit.

5.2 Localization

We assume that each closure is given a unique activation record when it is invoked. This requires heap allocation of activation records [71]. In practice this requires only a straightforward change to the code generator, and produces code that is competitive with stack allocated activation records [9]. In this section we describe how to localize scalars to a particular activation record. More specifically, we show that through this localization process we can eliminate register storage dependences between invocations of closures, enabling concurrent execution of the closures produced by scalar queue conversion.

In Chapter 4 we noted that optimal unidirectional renaming tends to produce more static renaming opportunities. As a result it tends to be the case that few du-chains flow between the closures produced by scalar queue conversion. We take advantage of this by introducing a notion of scope and, when possible, assign variables to a scope smaller than the entire program. By *scope* we simply mean the lifetime of an activation record, and thus we assign variables to scopes by associating variables with activation records.

We follow the straightforward rules that

1. If all the nodes of the du-web for a particular variable x fall into partition A of a unidirectional A-B cut. Then x is assigned the scope associated with the procedure containing the sliced flow graph for A.

2. If all the nodes of the du-web for a particular variable x fall into the same procedure, β_i, produced by scalar queue conversion, then x is assigned the scope associated with that procedure.

3. If the nodes of the du-web for a particular variable x fall into different procedures, then x is assigned the scope of B. (This is the scope containing the *set* of procedures $\beta_i \subset B$.)

Note that it is not necessary to have a global scope to cover the case that some of the nodes of a du-web are part of A and some part of B, because after unidirectional renaming and scalar queue conversion each du-web is guaranteed to be entirely contained on one side of the cut or the other.

For example, consider Figure 4-4. In this case the variables `i`, `j` and `partial_sum` can be localized to the procedure on the left. The variables `i'`, `j'` and `partial_sum'` can be localized to the procedure corresponding to nodes 5 and 6. An *independent* version of variable `partial_sum'` can be localized to the procedure corresponding to node 12. Finally, the variable `sum` can be localized to the scope containing the *set* of procedures on the right side of the figure.

The result of this localization process is the elimination of anti-dependences between different invocations of the same procedure. For example, each closure for the procedure corresponding to nodes 5 and 6 in Figure 4-4 will have its own, private, copies of variables `i'`, `j'` and `partial_sum'` in its own activation record, and thus these closures can be invoked concurrently.

5.3 Equivalence Class Unification

Our current compiler uses a context-sensitive inter-procedural pointer analysis [129, 97] to differentiate between memory accesses to different data structures. The result of the pointer analysis is a points-to set for each load, store and call site in the flow graph. The *points-to set* is a conservative list of all the possible allocation sites that could be responsible for allocating the memory touched by the operation in question. (Examples of "allocation sites" include points in the flow graph that call the `malloc()` routine, declarations of global aggregates, and declarations of any global scalars that might be aliased.)

The points-to sets resulting from the pointer analysis will be conservative in the sense that if the points-to sets for two instructions do not intersect, then the pointer analysis has *proved* that there is no situation under which the two instructions might access the same memory location. As a result, we can remove from the value dependence graph any memory dependence chain between instructions having non-intersecting points-to sets.

This technique is now widely used in parallelizing compilers whenever a decent pointer analysis is available [108, 14, 19].

5.4 Register Promotion

The renaming operations of scalar queue conversion work only for unaliased scalar variables. It is often the case, however, that in some region of code some invariant pointer will be repeatedly loaded and stored. When this is the case we can *register promote* [26, 79] the memory location to a scalar for the duration of the region. Register promotion is a generalization/combination of partial redundancy elimination and partial dead code elimination, targeted at load and store operations. When register promotion can be applied, especially when it can be applied to loops, it turns memory references into scalar references, which can then be renamed and rescheduled by scalar queue conversion.

Consider the following example of summing an array into a memory location (similar to an example given by Cooper and Lu [26]):

```
*p = 0
for (i = 0; i < N; i++)
  x = *p
  pA = &A[i]
  y = *pA
  z = x + y
  *p = z
```

If the pointer analysis can guarantee that p and pA always point to different memory locations then we know that (a) the pointer p is invariant during the execution of the loop and (b) memory references to the location pointed to by pA will never interfere with memory references to the location pointed to by p.

Thus we can transform the code by allocating a "virtual register" (scalar), rp, loading *p into rp before the start of the loop, storing rp back to *p after the end of the loop and replacing all references to *p inside the loop with references to rp. The resulting code is:

```
*p = 0
rp = *p
for (i = 0; i < N; i++)
  x = rp
  pA = &A[i]
  y = *pA
  z = x + y
  rp = z
*p = rp
```

This enables scalar queue conversion on the new scalar variable rp.

The idea of using register promotion to improve parallelism has been previously investigated in [19].

5.5 Scope Restriction

Scope restriction is an analysis performed at the front end of the compiler that uses scoping information on aggregates (arrays and structures) to restrict the live ranges of the aggregates to the scope they were originally declared in. The front end passes this information to the back end by changing the stack allocated data structure into a heap allocated data structure, with a call to a special version of malloc at the point where the object comes in to scope, and a call to a special version of free at the point(s) where the object goes out of scope.

The back end is augmented so that when it generates the reaching relation for memory dependence chains it recognizes that the special version of free *kills* (*i.e.*, does not pass) definitions and uses of the corresponding pointer. Thus memory anti- and output- dependence chains that otherwise would have reached backwards through loops can be eliminated before scalar queue conversion. At code generation time if the calls to matching versions of malloc and free are still in the same procedure, then they can be turned back into stack pointer increment/decrement operations.

This transformation relies on the programmer to declare each aggregate in the innermost scope in which it might be accessed. While this programmer behavior is desirable, from a software engineering standpoint, popular programming languages, like ANSI-C, have only (relatively) recently started supporting automatic allocation of aggregates. Thus, scope restriction is not applicable to "dusty deck" codes. If it is desired to support parallelization of such programs then one should consider incorporating an array privatization analysis in the compiler [77, 82, 118].

Chapter 6

Generalized Loop Distribution

In this chapter we describe how to apply scalar queue conversion to enable a generalized form of loop distribution that can reschedule *any* region of code with arbitrary control flow, including arbitrary looping control flow. The goal of loop distribution is to transform the chosen region so that *any externally visible changes to machine state will occur in minimum time*. Roughly speaking, then, we begin by finding externally visible state changes for the region in question, which we call critical definitions. We then find the smallest partition of the value dependence graph that includes the critical node, yet still forms a unidirectional cut with its complement. Finally we apply scalar queue conversion to create a provably minimal (and hopefully small) piece of code that performs *only* the work that cyclically depends on the critical definition. For simplicity we will present the transformation in terms of a single-entry single-exit region, R, of the value dependence graph. The transformation can be extended to multiple exit regions by applying the transformation from Section 5.1.

Section 6.4 discusses the relationship of generalized loop distribution to recurrences, (roughly speaking, recurrences are loop carried dependences that are updated with only a single associative operator (*e.g.*, addition)). In particular, we

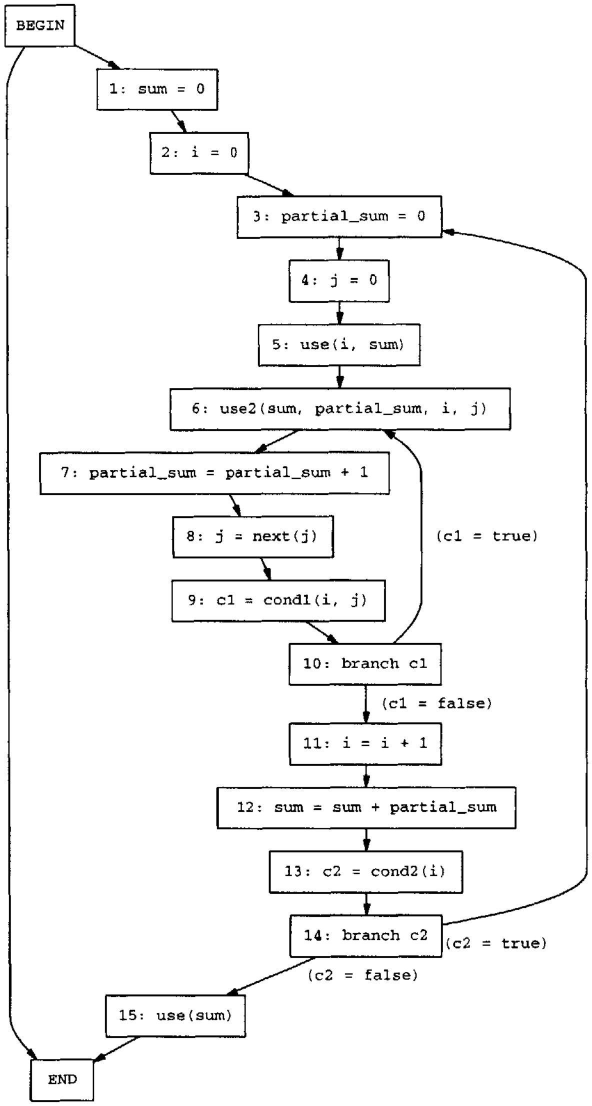


Figure 6-1: The control flow graph of the example loop. (This is the same as Figure 2-2, replicated here only for ease of reference.)

demonstrate that generalized loop distribution enables a broader class of recurrences to be reassociated than can be handled with less powerful scheduling techniques.

Loop distribution is closely related to a variety of recently proposed scheduling techniques called "critical path reductions." Section 9.2 describes this relationship, and how the generalized loop distribution technique also extends critical path reduction transformations.

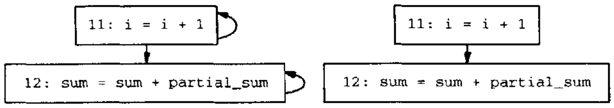


Figure 6-2: The critical node graph (left) and the critical node dag (right) for the outer loop of the flow graph in Figure 6-1.

6.1 Critical Paths

Consider again the example used throughout Chapters 2 and 3, which we replicate in Figure 6-1 for ease of reference. Roughly speaking, this loop has two loop carried dependences, on the variables `i` and `sum`. The other variables, (*e.g.*, `j`, `partial_sum`, `c1` and `c2`) are private to each loop iteration, and thus are not part of the state changes visible external to the loop.

Following this intuitive distinction, we more concretely identify the *critical definitions* of a region. We do this by finding all uses (anywhere in the program) such that at least one definition d_R within the region R reaches the use and at least one definition from outside the region $d_{\bar{R}}$ reaches the use. Then we call the definition d_R (the one inside region R) a critical definition. To reiterate, intuitively, the critical definitions represent changes to the part of the state that is visible from outside the region. Critical definitions represent points inside the region at which that visible state is changed. (As opposed to region (loop) invariant and externally invisible (private) state).

For the region corresponding to the outer loop in Figure 6-1 the critical definitions are the nodes 11 and 12. Nodes 5, 6, 9 and 11, for example, are reached both by node 11 (inside the loop) and node 2 (outside the loop), so node 11 is a critical definition for the loop. Likewise, nodes 5, 6 and 12 are reached both by node 12 (inside the loop) and node 1 (outside the loop), so node 12 is also a critical definition for the loop.

Next we construct the *critical node graph*. The nodes of the critical node graph are the critical definitions as defined above. There is an edge in the critical node graph between nodes d_0 and d_1 exactly when there is a *path* from d_0 to d_1 in the value dependence graph. The critical node graph for the outer loop of the flow graph from Figure 6-1 is shown on the left side of Figure 6-2. There is a critical node graph edge from node 11 to node 12 because there is a path in the dependence graph 11 $\rightarrow$ 13 $\rightarrow$ 14 $\rightarrow$ 12. (The dependence from node 14 to node 12 is a control dependence, while the other edges in the path are due to scalar value dependences.)

Finally, we construct the *critical node dag* by collapsing cycles in the critical node graph. This isn't strictly necessary, but a cycle in the critical node graph represents a sequence of state changes that is mutually dependent, and thus can't be reordered. Thus we gain no flexibility by not collapsing, and the collapsed result is easier to deal with. Note that a dag is just a pictorial representation of a partial ordering. That is we say that given two nodes a and b in the dag $a < b$ if there a path from a to b in the dag. This partial ordering is well defined since there are no cycles in the critical node dag. The critical node dag for the outer loop of the flow graph from Figure 6-1 is shown on the right side of Figure 6-2.

6.2 Unidirectional Cuts

We use the critical node dag to prioritize the instructions in the value dependence graph into a sequence of unidirectional cuts (see Chapter 3). There will be twice as many priorities as there are levels in the critical node dag.

We start by giving each critical node a priority corresponding to its level in the critical node dag. Next, for each critical node we find all nodes in the value dependence graph that have a *cyclic dependence* with the critical node. That is, given critical node d and node n, if there is a path from d to n in the value dependence

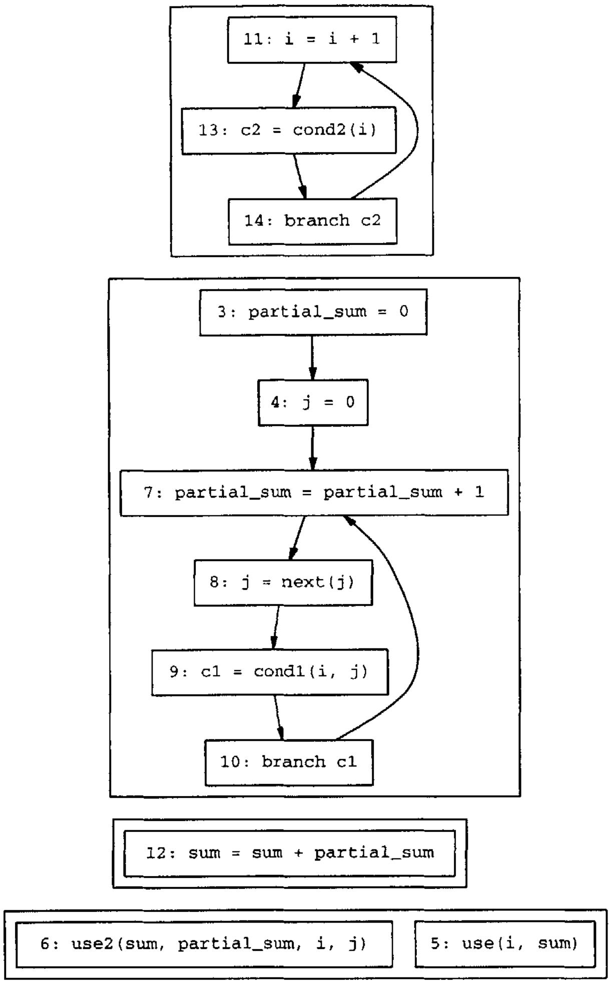


Figure 6-3: The Prioritization of the nodes in the outer loop of the flow graph in Figure 6-1.

graph and a path from n to d in the value dependence graph, then we give n the same priority as d. For example, in the loop in Figure 6-1 the cyclic path 11 → 13 → 14 → 11 in the value dependence graph indicates that nodes 13 and 14 form a cycle with the critical node 11.

All remaining nodes will receive priorities *between* the critical node priorities. That is, for each node n find the critical node d_{below} with the *highest* priority, such that there is a path from n to d_{below} in the value dependence graph. Then give n a priority higher than d_{below}'s priority, but just lower than the priority of d_{below}'s parent.

For example, in Figure 6-1 node 12 depends on node 7. Node 7, in turn, is dependent on nodes 3, 4, 7, 8, 9 and 10. (There exists, for example, the dependence path 4 $\longrightarrow$ 8 $\longrightarrow$ 9 $\longrightarrow$ 10 $\longrightarrow$ 7.) None of these nodes has a path in the value dependence graph leading to any of nodes 11, 13 or 14. Thus we give nodes 3, 4, 7, 8, 9 and 10 a priority between the priority of node 11 and the priority of node 12. The prioritization of the nodes from the outer loop of the flow graph of Figure 6-1 is shown in Figure 6-3.

More generally one can also solve the dual problem: find the critical node d_{above} with the lowest priority such that there is a path from d_{above} to n, and then give n any priority between that of d_{above} and d_{below}. Note that for any node n with cyclic dependences with a critical node d_{crit} it is the case that $d_{above} = d_{below} = d_{crit}$, and thus the priority of these nodes will be set consistently with the above criteria. In the transformation described below, it will turn out that cross-priority dependence edges are more expensive to handle than dependence edges within a priority, and the dual information could be used, in combination with a maxflow/mincut algorithm to minimize the number of cross-priority dependence edges. This will be investigated in future work. In any case both the primal and dual problems can be individually solved by a simple dataflow analysis based on depth-first search. The implemented algorithm uses only the primal information.

6.3 Transformation

For each priority we have a unidirectional cut from the higher priorities to this priority and those below. Thus we perform scalar queue conversion on each priority (from the bottom up) to complete our code transformation.

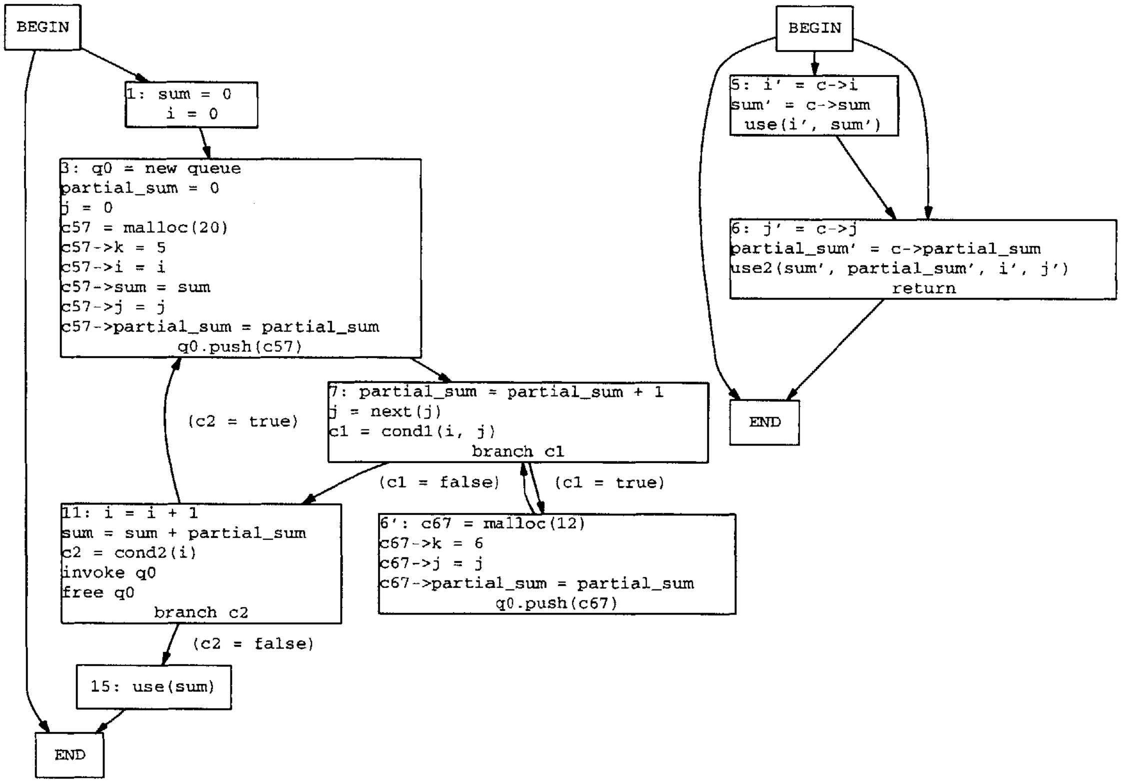
There are, however, two subtleties. The first is that as we perform scalar queue conversion on a unidirectional A-B cut we must place instructions to create, and fill, closures into the graph of partition A for each maximally connected group

$\beta_i \subset B$. The question then arises as to which priority the closure creation and fill instructions for each maximal group should belong to. We solve this problem by running the prioritization algorithm from Section 6.2 on the instructions introduced by each pass of scalar queue conversion. Note that because we are working with unidirectional cuts we never introduce nodes that can "undo" any of the priority decisions we have already made.

The second, practical, problem is that we are trying to use loop distribution to schedule concurrency. That concurrency exists in the non-critical priority groups produced by the prioritization scheme in Section 6.2. The problem is that the concurrency we have exposed is between iterations of the *outer* loop that we are distributing. Thus we would like to create a thread for each outer loop iteration, even if that thread invokes many closures. We solve this problem by running scalar queue conversion *twice* for the non-critical priority groups.

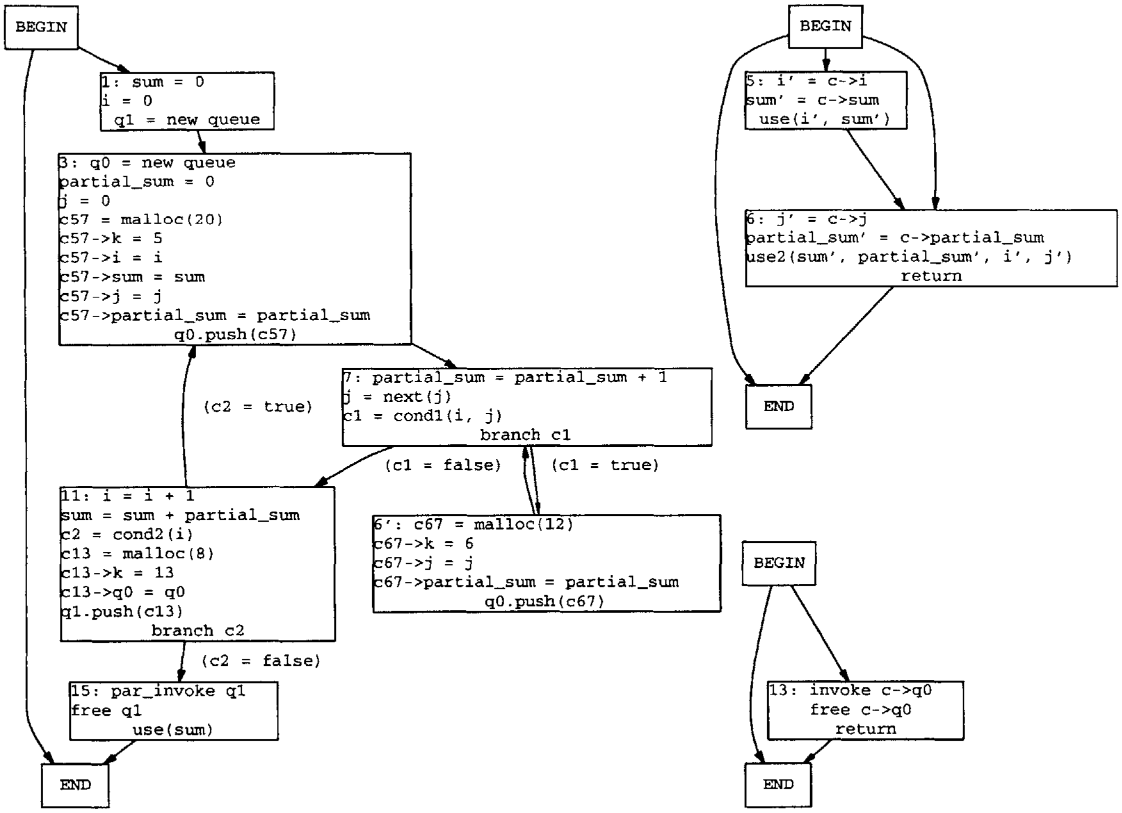
In the first pass we run scalar queue conversion with respect to the loop body (*i.e.*, not including the loop backedge). This packages the entire work done for that priority group in each iteration into a deferred execution queue (one deferred execution queue per iteration), which is then invoked. Figure 6-4 shows the results of performing this scalar queue conversion on the lowest priority group (nodes 5 and 6) of the example loop. A deferred execution queue (q0) is created, and the correct closures are pushed on to q0 to perform the low priority work from the entire inner loop of a *single* iteration of the outer loop.

In the second pass we run scalar queue conversion with respect to the entire loop (including the loop backedge). This creates a second deferred execution queue with one closure per loop iteration. The closures in this deferred execution queue can be invoked in parallel because the prioritization analysis from Section 6.2 has already determined that there are no dependences between these closures. This second transformation is shown in Figure 6-5. A deferred execution



```
1: sum = 0
     i = 0

3: q0 = new queue
partial_sum = 0
j = 0
c57 = malloc(20)
c57->k = 5
c57->i = i
c57->sum = sum
c57->j = j
c57->partial_sum = partial_sum
             q0.push(c57)

7: partial_sum = partial_sum + 1
j = next(j)
c1 = cond1(i, j)
             branch c1

11: i = i + 1
sum = sum + partial_sum
c2 = cond2(i)
invoke q0
free q0
         branch c2

6': c67 = malloc(12)
c67->k = 6
c67->j = j
c67->partial_sum = partial_sum
              q0.push(c67)

15: use(sum)

5: i' = c->i
sum' = c->sum
 use(i', sum')

6: j' = c->j
partial_sum' = c->partial_sum
use2(sum', partial_sum', i', j')
               return
```

Figure 6-4: The example loop after using scalar queue conversion to move the lowest priority group (nodes 5 and 6) to the bottom of the loop body.



```
1: sum = 0
i = 0
 q1 = new queue

3: q0 = new queue
partial_sum = 0
j = 0
c57 = malloc(20)
c57->k = 5
c57->i = i
c57->sum = sum
c57->j = j
c57->partial_sum = partial_sum
            q0.push(c57)

7: partial_sum = partial_sum + 1
j = next(j)
c1 = cond1(i, j)
              branch c1

11: i = i + 1
sum = sum + partial_sum
c2 = cond2(i)
c13 = malloc(8)
c13->k = 13
c13->q0 = q0
q1.push(c13)
         branch c2

6': c67 = malloc(12)
c67->k = 6
c67->j = j
c67->partial_sum = partial_sum
          q0.push(c67)

15: par_invoke q1
free q1
      use(sum)

5: i' = c->i
sum' = c->sum
 use(i', sum')

6: j' = c->j
partial_sum' = c->partial_sum
use2(sum', partial_sum', i', j')
             return

13: invoke c->q0
  free c->q0
   return
```

Figure 6-5: The example loop after a second use of scalar queue conversion to move the lowest priority group (nodes 5 and 6) *out* of the loop.

queue (q1) is created. The closures (c13) on this queue receive the deferred execution queue, q0, created in the first pass as a parameter. Then after the loop exits, the closures on deferred execution queue q1 can be invoked in concurrently.

Figure 6-6 shows the result of running scalar queue conversion on the lower priority critical path. Note that while the original critical path consisted only of the node "12: sum = sum + partial_sum," the prioritization algorithm has determined that the closure filling operation "c57->sum = sum" must be scheduled at the same priority. Thus a pointer to the c57 closure is passed as a parameter to the c12 closure so that c12 can fill in the current value of the sum variable before it is modified.

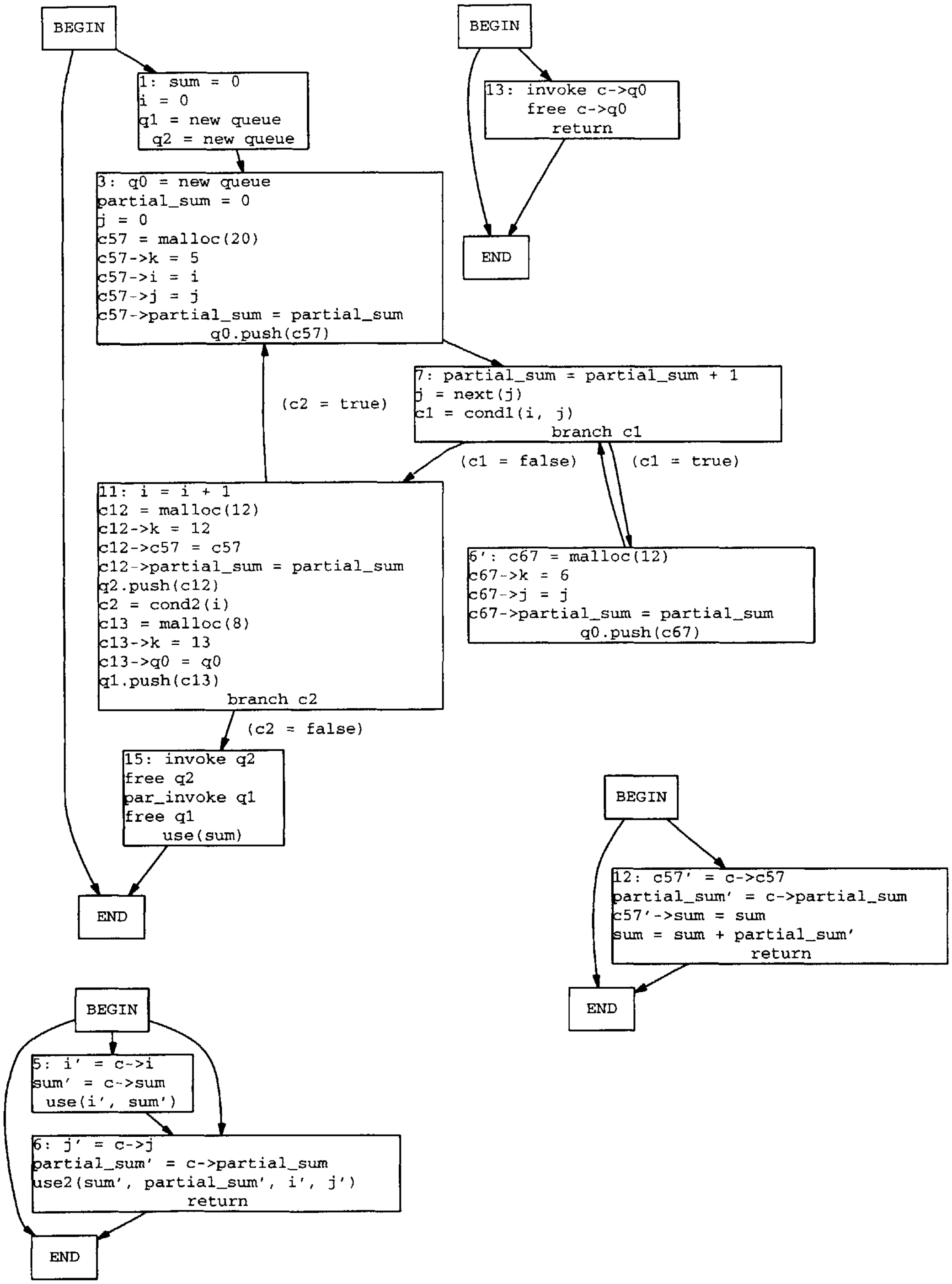
Figure 6-7 shows the end result of running generalized loop distribution. After another two passes of scalar queue conversion the work corresponding to the inner loop of the original code has been moved into a deferred execution queue, q3, the closures of which can be invoked concurrently.

6.4 Generalized Recurrence Reassociation

A common problem in the doubly nested loops that are handled by the generalized loop distribution algorithm described in Sections 6.1, 6.2 and 6.3 is that critical paths (loop carried dependences) of the outer loop will often also contain nodes in the inner loop. Since critical paths represent cycles in the code that must be run sequentially, we would like to reduce the length of these paths when ever possible.

This section describes how we leverage generalized loop distribution to shorten critical paths when the update operator in the critical path is *associative*.[1] When the update operator is associative we can often transform the code to make the

[1] The most common associative operator is addition. Associative operators are binary operators with the property that $(a + b) + c = a + (b + c)$. Other common programming operators having this property are multiplication, "max" and "min."



```
1: sum = 0
i = 0
q1 = new queue
 q2 = new queue
```

```
3: q0 = new queue
partial_sum = 0
j = 0
c57 = malloc(20)
c57->k = 5
c57->i = i
c57->j = j
c57->partial_sum = partial_sum
             q0.push(c57)
```

```
7: partial_sum = partial_sum + 1
j = next(j)
c1 = cond1(i, j)
              branch c1
```

```
6': c67 = malloc(12)
c67->k = 6
c67->j = j
c67->partial_sum = partial_sum
           q0.push(c67)
```

```
11: i = i + 1
c12 = malloc(12)
c12->k = 12
c12->c57 = c57
c12->partial_sum = partial_sum
q2.push(c12)
c2 = cond2(i)
c13 = malloc(8)
c13->k = 13
c13->q0 = q0
q1.push(c13)
             branch c2
```

```
15: invoke q2
free q2
par_invoke q1
free q1
    use(sum)
```

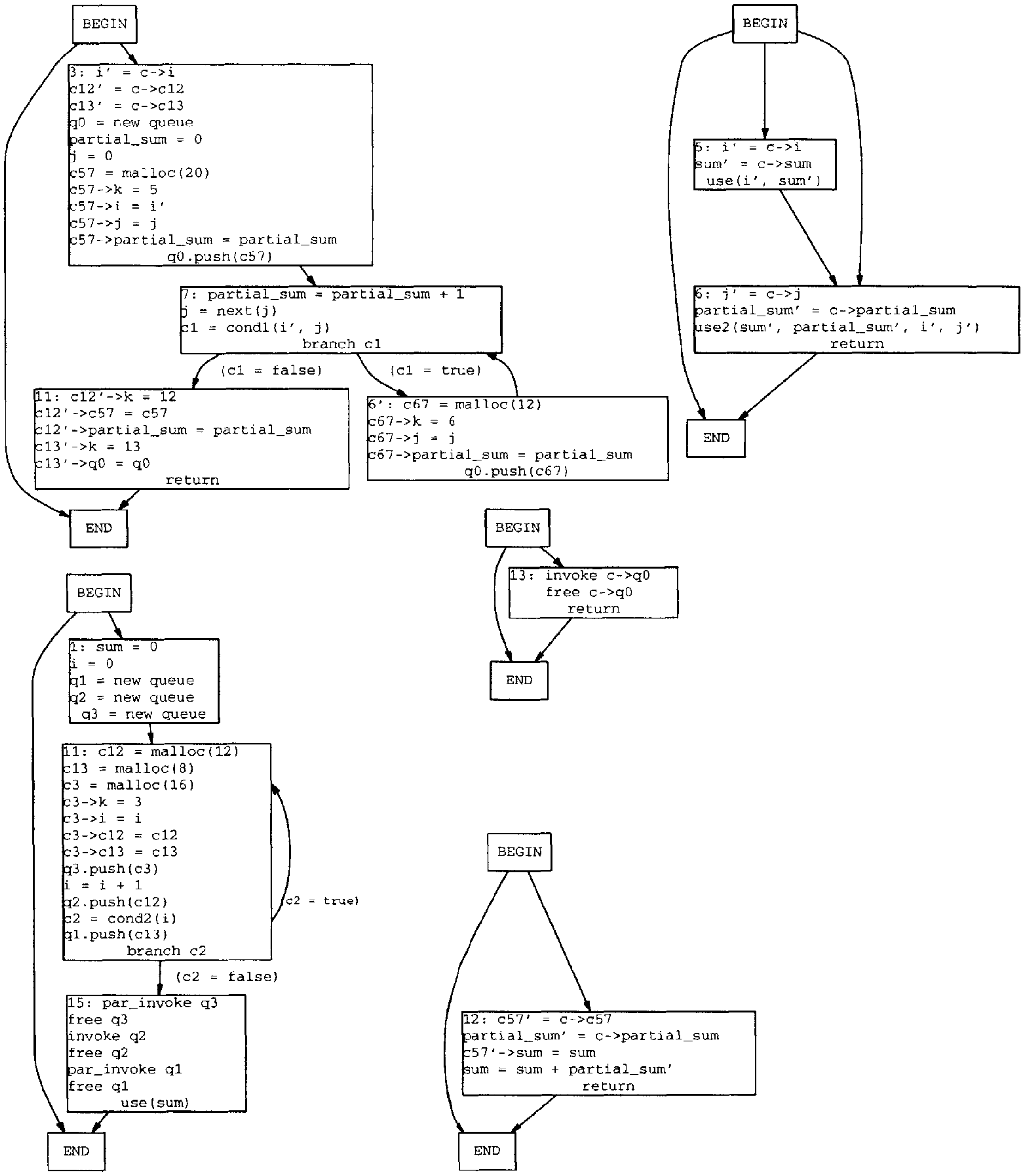
```
13: invoke c->q0
    free c->q0
      return
```

```
12: c57' = c->c57
partial_sum' = c->partial_sum
c57'->sum = sum
sum = sum + partial_sum'
               return
```

```
5: i' = c->i
sum' = c->sum
 use(i', sum')
```

```
6: j' = c->j
partial_sum' = c->partial_sum
use2(sum', partial_sum', i', j')
               return
```

Figure 6-6: The example loop after using scalar queue conversion to move the lower priority critical path (corresponding to node 12) out of the loop. Note that a pointer to the c57 closure (which initializes sum') is passed as a parameter to the closure c12 so that c12 can fill in the current value of the sum variable before it is modified.



```
3: i' = c->i
c12' = c->c12
c13' = c->c13
q0 = new queue
partial_sum = 0
j = 0
c57 = malloc(20)
c57->k = 5
c57->i = i'
c57->j = j
c57->partial_sum = partial_sum
          q0.push(c57)
```

```
7: partial_sum = partial_sum + 1
j = next(j)
c1 = cond1(i', j)
            branch c1
```

```
11: c12'->k = 12
c12'->c57 = c57
c12'->partial_sum = partial_sum
c13'->k = 13
c13'->q0 = q0
               return
```

```
6': c67 = malloc(12)
c67->k = 6
c67->j = j
c67->partial_sum = partial_sum
            q0.push(c67)
```

```
5: i' = c->i
sum' = c->sum
 use(i', sum')
```

```
6: j' = c->j
partial_sum' = c->partial_sum
use2(sum', partial_sum', i', j')
                 return
```

```
13: invoke c->q0
     free c->q0
       return
```

```
1: sum = 0
i = 0
q1 = new queue
q2 = new queue
 q3 = new queue
```

```
11: c12 = malloc(12)
c13 = malloc(8)
c3 = malloc(16)
c3->k = 3
c3->i = i
c3->c12 = c12
c3->c13 = c13
q3.push(c3)
i = i + 1
q2.push(c12)
c2 = cond2(i)
q1.push(c13)
     branch c2
```

```
15: par_invoke q3
free q3
invoke q2
free q2
par_invoke q1
free q1
      use(sum)
```

```
12: c57' = c->c57
partial_sum' = c->partial_sum
c57'->sum = sum
sum = sum + partial_sum'
             return
```

Figure 6-7: The example loop after using scalar queue conversion to reschedule and move the group 3, 4, 7, 8, 9 and 10 out of the loop. The outer loop of the original flow graph corresponds to the flow graph in the lower left corner of the figure. The inner loop of the original flow graph corresponds to the flow graph in the upper right of the figure.

dependence graph more "treelike."

Consider the following code:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    use(sum)
    sum = sum + f[i][j]
```

The loop carried dependent variable sum is traditionally called a *recurrence* [68]. The critical path for this recurrence contains the instruction "sum = sum + f[i][j]" in the inner loop. Using a combination of static renaming and forward substitution [68] we will demonstrate that because the update operator here is associative we can move this critical node out of the inner loop, into the outer loop.

Briefly, recurrence reassociation introduces a temporary variable that sums the values in the inner loop, and then adds the temporary to the original recurrence variable only in the outer loop. This transformation produces the following code:

```
for (int i = 0; i < N; i++)
  int partial_sum = 0
  for (int j = 0; j < M; j++)
    use(sum + partial_sum)
    partial_sum = partial_sum + f[i][j]
  sum = sum + partial_sum
```

Note that we have *simultaneously* introduced the temporary variable partial_sum and forward substituted the expression sum + partial_sum into the inner loop statement use(sum), creating the new inner loop statement use(sum + partial_sum).

The basic idea is that while any scheduling algorithm has to honor all the value dependences, generalized loop distribution, with scalar queue conversion, will eliminate all the anti-dependences. Thus recurrence reassociation takes ad-

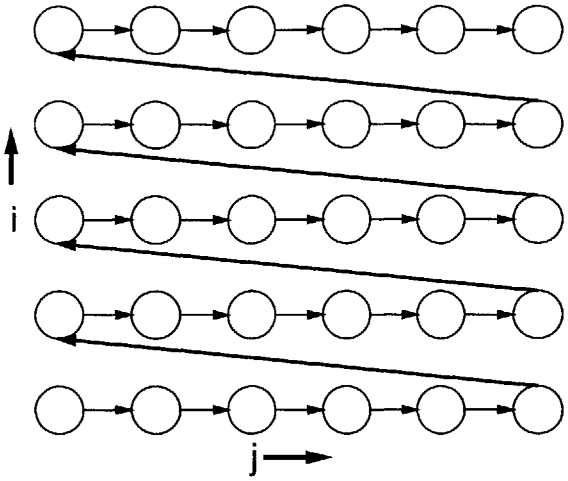


Figure 6-8: The dynamic dependence graph between updates to the sum recurrence variable in the original code.

vantage of operation associativity to turn loop-carried true-dependences into anti-dependences. Generalized loop distribution then eliminates the anti-dependence during scheduling.

In this context we define *reassociatable recurrence variables* to be loop-carried true-dependences that are modified only with a single associative operator. Note, in particular that in the example sum is both used and modified inside the inner loop, but is still considered to be a recurrence variable. Figure 6-8 shows sum's dependence pattern.

In the example, shown above, a temporary, partial_sum has been introduced above the inner loop. Each use of sum has been converted to a use of sum + partial_sum. The update of sum in the inner loop has been changed to an update of partial_sum and finally, partial_sum is added to sum after the inner loop is finished. At first glance it would appear that we have not improved the situation. But in fact, we are no longer modifying the variable sum in the inner loop. From the perspective of the outer loop, this *separates* the modification of sum from its use. Figure 6-9 shows how sum's dependence pattern has changed.

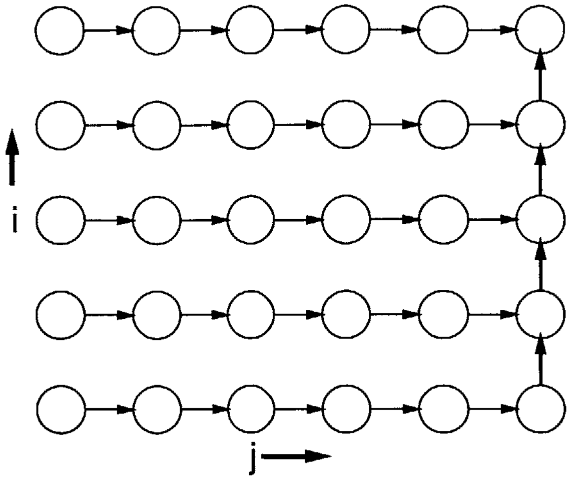


Figure 6-9: The dynamic dependence graph between updates to the sum variable after recurrence reassociation has been performed.

Traditionally, reassociatable recurrence variables are considered to be those that are

1. Loop-carried true-dependences.
2. Updated only with a single associative operator (*e.g.*, plus, times or max).
3. Unused except in the update operation(s) [68].

The *simultaneous* application of static renaming and forward substitution described above allows the third requirement to be circumvented in the case that we want to move a critical update out of an inner loop.

Chapter 7

SUDS: The Software Un-Do System

Since scalar queue conversion can only break *scalar* anti- and output- dependences, additional solutions are required to parallelize around memory dependences. The transformations described in Chapter 5 are a necessary component to solving the memory dependence problem, but they are not sufficient. More specifically, any transformation that relies only on information available at compile time can not legally remove edges from the memory dependence graph that are only *usually* irrelevant.

In this chapter we describe SUDS, the Software Un-Do System, which *speculatively* eliminates edges from the dependence graph. Informally, SUDS checkpoints the machine state and then runs a piece of code that has been parallelized assuming that certain dependences "don't matter." Once the code is done running SUDS checks that the parallel execution produced a result consistent with sequential semantics. If the result is found to be consistent, SUDS commits the changes and continues. If the result is found inconsistent, SUDS rolls back execution to the last checkpoint and re-runs the code sequentially.

As shown in Figure 7-1, SUDS partitions Raw's tiles into two groups. Some portion of the tiles are designated as *compute* nodes. The rest are designated as

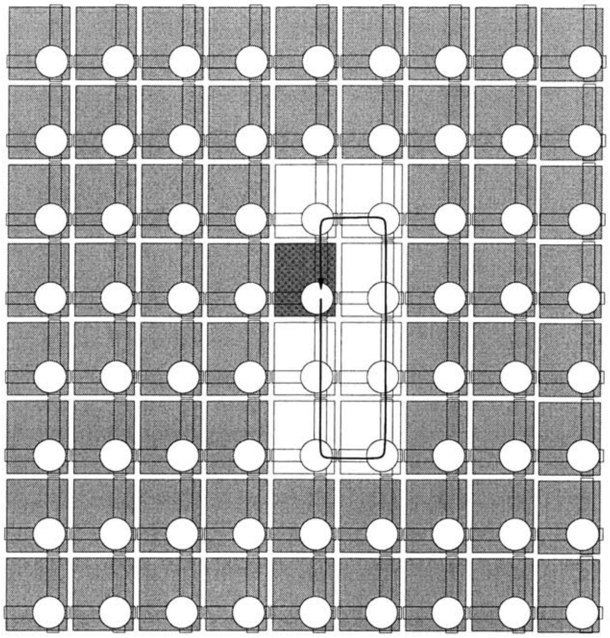


Figure 7-1: An example of how SUDS allocates resources on a 72 tile Raw machine. The 64 gray tiles are memory nodes. The 8 white tiles, approximately in the middle, are worker nodes, the gray hatched tile near the center is the master node. Loop carried dependences are forwarded between compute nodes in the pattern shown with the arrow.

memory nodes. One of the compute nodes is designated as the *master* node, the rest are designated as *workers* and sit in a dispatch loop waiting for commands from the master. The master node is responsible for running all the sequential code.

SUDS parallelizes loops by cyclically distributing the loop iterations across the compute nodes. We call the set of iterations running in parallel a *speculative strip*. Each compute node runs the loop iterations assigned to it, and then all the nodes synchronize through the master node.

In the next section we describe speculative strip mining, the technique SUDS uses to checkpoint and run a portion of a loop. In Section 7.2 we describe the SUDS runtime system component that efficiently checks the correctness of a particular parallel execution.

```
do
  (LOOP BODY)
while !c
```

Figure 7-2: An arbitrary loop.

```
do
  checkpoint machine state
  error = false
  for i = 0; i < 32; i++
    (LOOP BODY)
     error |= c
  if (error)
    roll back to checkpointed state
    for i = 0; (i < 32) && !c; i++
      (LOOP BODY)
while !c
```

Figure 7-3: The same loop after speculative strip mining. Machine state is checkpointed. A strip of 32 iterations is run. After the strip completes the `error` variable is checked. It running the strip caused any kind of misspeculation, (early exit from the loop, out-of-order memory access or a deferred execution queue dynamic memory allocation error), then machine state is rolled back to the checkpoint, and the original code is run non-speculatively for up to 32 iterations to get past the misspeculation point.

7.1 Speculative Strip Mining

Speculative strip mining is the technique SUDS uses to checkpoint and run a portion of a loop. Like traditional strip mining techniques [1], speculative strip mining turns a loop into a doubly nested loop, where each invocation of the newly created inner loop iterates a fixed number of times.

Speculative strip mining differs from traditional strip mining in that it generates the control structure shown in Figure 7-3. After the transformation, the outer loop body starts by checkpointing machine state. Then a *speculative strip* of 32 iterations are run. This inner loop is the loop that generalized loop distribution will be applied to, and that the SUDS system will try to run speculatively and in parallel.

A new variable, `error`, is introduced that is used to keep track of any misspeculation that might happen during the speculative strip. This variable can get set in any of three ways. First, the speculative strip runs for exactly 32 iterations. If during any one of those 32 iterations the loop condition variable, c, becomes set, then, semantically, the inner loop should have exited in fewer than 32 iterations, and thus the error variable gets set. Second, the error variable is implicitly set if any of the memory operations sent to the SUDS memory dependence speculation system (described below) are found to have executed out of order. Third the error variable will be set if any of the dynamic memory allocation operations (those introduced by generalized loop distribution) fail because of an out of memory condition.

After the speculative strip runs, the `error` condition is checked. If it is not set (hopefully the common case), then the outer loop iteration is finished, and a new outer loop iteration will start. The process of checkpointing, running a speculative strip, and checking the `error` condition will be repeated. If, on the other hand, the `error` condition *is* set, then the code rolls back to the checkpointed state, and a *different* copy of the inner loop is run.

In this case, the inner loop runs the original (unoptimized) loop body code. This "nonspeculative strip" runs for at most 32 iterations, but unlike the speculative strip, this strip runs sequentially and the original loop conditional, c, is checked on every iteration for early exit. Since generalized loop distribution is not applied, the nonspeculative strip can not take an out-of-memory exception (unless the semantics of the original code would have done so). Since the loop is run sequentially, the memory operations can not execute out-of-order.

Speculative strip mining, as described here, works only on loops with a single exit. If we wish to apply speculative strip mining to a loop with multiple exits then the transformation from Section 5.1 is applied first. Note that speculative strip mining assumes that the loop conditional, c will be false if the loop is to continue, and non-false if the loop is to exit. The transformation from Section 5.1, which turns multiple exit loops into single exit loops, produces condition variables that have this property. If the loop was single exit to begin with, and has a loop conditional with the opposite boolean sense, then a new loop conditional must be introduced, before applying speculative strip mining.

The "checkpoints" that speculative strip mining introduces need to be handled carefully. There are two parts to this. The first has to do with "checkpointing" the memory state of the machine. The memory state is typically enormous, and checkpointing the entire memory would be too costly. What the SUDS memory dependence speculation system (described below) does instead, is to *log* all of the modifications to memory requested during the speculative strip. Then, if rollback is required, the log is "run backwards" to restore the original memory state. If, after running the speculative strip, rollback is not required, then the log is erased and reused.

The second part of checkpointing has to do with the register (scalar) state of the machine. Speculative strip mining makes a copy of every scalar whose state might visibly change during the running of the speculative strip. But, these variables are exactly the "loop carried dependences" that generalized loop distribution recognizes in its critical path analysis. Thus speculative strip mining performs the critical definition analysis described in Section 6.1. Any scalars identified as critical during this analysis are copied into temporaries before the speculative strip. If rollback is required the values in the temporaries are copied back to the original scalars. If the speculative strip runs with no errors the temporaries are discarded.

Speculative strip mining allows generalized loop distribution to *legally* introduce dynamic memory allocations into the program. Because all memory operations are logged during a speculative strip, and speculative strip mining also makes copies of the (visible) register state, any dynamic memory allocation error introduced by generalized loop distribution can be fixed. This checkpoint/repair mechanism allows a second, important, performance optimization. Because all

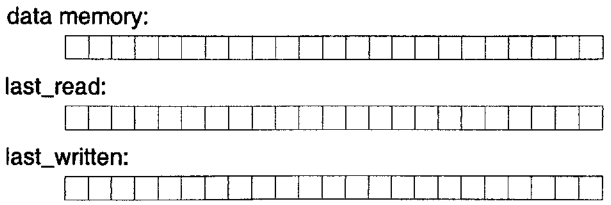


Figure 7-4: A conceptual view of Basic Timestamp Ordering. Associated with every memory location is a pair of timestamps that indicate the logical time at which the location was last read and written.

memory operations are logged, we can speculatively execute memory operations *out-of-order*. Thus, after speculative strip mining, and before generalized loop distribution, we remove from the value dependence graph all of the memory dependences that are carried on the outer loop. These memory dependences can be removed from the value dependence graph by generating reaching information for memory operations on the region flow graph for the loop body, with the loop back edge removed.

In addition to logging memory operations, if memory operations are issued out-of-order, then the memory access pattern must also be checked. The SUDS runtime memory dependence speculation system does this logging and checking. The memory dependence speculation system is described in the next section.

7.2 Memory Dependence Speculation

The memory dependence speculation system is in some ways the core of the system. It is the fallback dependence mechanism that works in all cases, even if the compiler cannot analyze a particular variable. Since only a portion of the dependences in a program can be proved by the compiler to be privatizable or loop carried dependences, a substantial fraction of the total memory traffic will be directed through the memory dependence speculation system. As such it is necessary to minimize the latency of this subsystem.

7.2.1 A Conceptual View

The method we use to validate memory dependence correctness is based on Basic Timestamp Ordering [15], a traditional transaction processing concurrency control mechanism. A conceptual view of the protocol is given in Figure 7-4. Each memory location has two timestamps associated with it, one indicating the last time a location was read (`last_read`) and one indicating the last time a location was written (`last_written`). In addition, the memory is checkpointed at the beginning of each speculative strip so that modifications can be rolled back in the case of an abort.

The validation protocol works as follows. As each load request arrives, its timestamp (`read_time`) is compared to the `last_written` stamp for its memory location. If `read_time` $\geq$ `last_written` then the load is in-order and `last_read` is updated to `read_time`, otherwise the system flags a miss-speculation and aborts the current speculative strip.

On a store request, the timestamp (`write_time`) is compared first to the `last_read` stamp for its memory location. If `write_time` $\geq$ `last_read` then the store is in-order, otherwise the system flags a miss-speculation and aborts the current speculative strip.

We have implemented an optimization on store requests that is known as the Thomas Write Rule [15]. This is basically the observation that if `write_time` $<$ `last_written` then the value being stored by the current request has been logically over-written without ever having been consumed, so the request can be ignored. If `write_time` $\geq$ `last_written` then the store is in-order and `last_written` is updated as `write_time`.

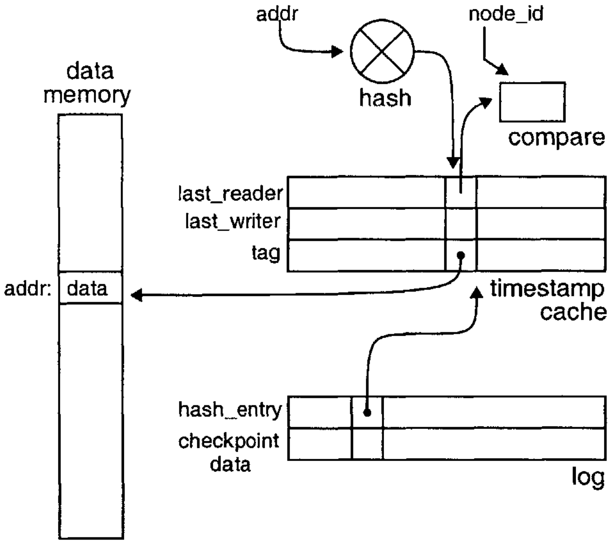


Figure 7-5: Data structures used by the memory dependence speculation subsystem.

7.2.2 A Realizable View

We can't dedicate such a substantial amount of memory to the speculation system, so the system is actually implemented using a hash table. As shown in Figure 7-5, each processing element that is dedicated as a memory dependence node contains three data structures in its local memory. The first is an array that is dedicated to storing actual program values. The next is a small hash table that is used as a *timestamp cache* to validate the absence of memory conflicts. Finally, the *log* contains a list of the hash entries that are in use and the original data value from each memory location that has been modified. At the end of each speculative strip the log is used to either commit the most recent changes permanently to memory, or to roll back to the memory state from the beginning of the speculative strip.

The fact that SUDS synchronizes the processing elements between each speculative strip permits us to simplify the implementation of the validation protocol. In particular, the synchronization point can be used to commit or roll back the logs and reset the timestamp to 0. Because the timestamp is reset we can use the requester's physical node-id as the timestamp for each incoming memory request.

Operation	Cost
Send from compute node	1
Network latency	4 + distance
Memory node	8
Network latency	4 + distance
Receive on compute node	2
Total	19 + 2 × distance

Figure 7-6: The round trip cost for a load operation is 19 cycles + 2 times the manhattan distance between the compute and memory node. The load operation also incurs additional occupancy of up to 40 cycles on the memory node after the data value is sent back to the compute node.

In addition, the relatively frequent log cleaning means that at any point in time there are only a small number of memory locations that have a non-zero timestamp. To avoid wasting enormous amounts of memory space storing 0 timestamps, we cache the active timestamps in a relatively small hash table. Each hash table entry contains a pair of `last_read` and `last_written` timestamps and a cache-tag to indicate which memory location owns the hash entry.

As each memory request arrives, its address is hashed. If there is a hash conflict with a different address, the validation mechanism conservatively flags a miss-speculation and aborts the current speculative strip. If there is no hash conflict the timestamp ordering mechanism is invoked as described above.

Log entries only need to be created the first time one of the threads in a speculative strip touches a memory location, at the same time an empty hash entry is allocated. Future references to the same memory location do not need to be logged, as the original memory value has already been copied to the log. Because we are storing the most current value in the memory itself, commits are cheaper, and we are able to implement a fast path for load operations. Before going through the validation process, a load request fetches the required data and returns it to the requester. The resulting latency at the memory node is only 8 cycles as shown in

Figure 7-6. The validation process happens after the data has been returned, and occupies the memory node for an additional 14 to 40 cycles, depending on whether a log entry needs to be created.

In the common case the speculative strip completes without suffering a miss-speculation. At the synchronization point at the end of the speculative strip, each memory node is responsible for cleaning its logs and hash tables. It does this by walking through the entire log and deallocating the associated hash entry. The deallocation is done by resetting the timestamps in the associated hash entry to 0. This costs 5 cycles per memory location that was touched during the speculative strip.

If a miss-speculation is discovered during the execution of a speculative strip, then the speculative strip is aborted and a consistent state must be restored. Each memory node is responsible for rolling back its log to the consistent memory state at the end of the previous strip. This is accomplished by walking through the entire log, copying the checkpointed memory value back to its original memory location. The hash tables are cleaned at the same time. Rollback costs 11 cycles per memory location that was touched during the speculative strip.

The synchronization between speculative strips helps in a second way. Hash table entries are only deleted in bulk, during the commit or rollback phases. Thus, we are guaranteed that between synchronization points the hash table will only receive insertion and lookup requests. As a result, the hash table can be implemented using open addressing with double hashing [65]. (That is, if a hash of a key produces a conflict, then we deterministically rehash the key until we find an open entry). The SUDS implementation does up to sixteen rehashes. Open addressing with double hashing has the properties that it avoids the costs of linked list traversal but still keeps the average number of hashes low.[1]

[1] For example, when the hash table is half full, the average number of rehashes will be 1 and the probability of not finding an open entry within sixteen rehashes will be $\frac{1}{65536}$.

7.2.3 Implementation

The SUDS memory dependence speculation system is designed to run on Raw microprocessors [123]. A Raw microprocessor can roughly be described as a single-chip, distributed-memory multiprocessor. Unlike traditional distributed-memory multiprocessors, however, the Raw design is singularly focused around providing low-latency, register-level communication between the processing units (which we call *tiles*). In particular, the semantics of the network and network interface are carefully designed to remove message dispatch overheads [74, 115] and deadlock avoidance/recovery overheads [67] from the critical path. Because of these considerations, a single-word data message can be sent from one tile to a neighboring tile, *and dispatched*, in under *six* 4.44ns machine cycles.

Each Raw tile contains an eight-stage single-issue RISC microprocessor, about 96 Kbyte of SRAM caches, an interface to the on-chip interconnect, and one of the interconnect routers. The tiles on each chip are arranged in a two-dimensional mesh, or grid, similar to the structure shown in Figure 7-1. While each tile contains a general-purpose RISC microprocessor pipeline, it is sometimes more appropriate to view this microprocessor as a deeply pipelined *programmable microcontroller* for a set of hardware resources that include an ALU and some SRAM memory. This, in any case, is the view I adopted for the implementation of the SUDS memory dependence speculation system.

As shown in Figure 7-1, SUDS partitions Raw's tiles into two groups. Some portion of the tiles are designated as *compute* nodes. The rest are designated as dedicated *memory* nodes. The memory nodes work together to implement a logically shared memory on top of Raw's physically distributed memory. Each time a compute node wishes to make a memory request from the logically shared memory it injects a message into the on-chip interconnect directed at the memory node that *owns* the corresponding memory address. The owner is determined by a simple xor-based hash of the address, similar to that used in some L1 caches [46]. Thus, if there are 64 tiles dedicated as memory nodes, the logically shared memory can be viewed as being banked 64 ways.

After the request is injected, it travels through Raw's on-chip interconnect at one machine cycle per hop (except when the message turns, which takes two machine cycles). Messages are handled, at their destination, in the order they are received, and *atomically*. The Raw network interface provides support so that if, when a request arrives, the tile processor is still busy processing a previously received request, the new request is queued in a small buffer local to the destination tile.[2] Protocol replies are sent on a network logically distinct from that used to send protocol requests, and storage to sink reply messages is preallocated before requests are made, so the communication protocol is guaranteed not to deadlock [67].

The hand optimized code at the memory node uses the header of each arriving request to dispatch to the appropriate request handler in just two cycles in the case of a load request, and seven cycles for store requests or control messages. The dispatch loop and load request handler are optimized to minimize load reply latency, at the expense of slightly poorer overall bandwidth. The load handler thus accesses the requested memory location and injects the data reply message to the requesting compute node before accessing the timestamp cache or log. As a result, the total end-to-end latency observed by a compute node making a load request is 19 machine cycles + 2x the manhattan distance between the compute node and the memory node. (Unless there is contention at the memory node or the memory node takes a cache miss while accessing the requested memory location).

Consider, for example, the 72 node Raw system shown in Figure 7-1, and as-

[2] Raw's network provides flow-control support so that if a destination node becomes heavily contended the sending nodes can be stalled without either dropping packets or deadlocking the network [33, 32].

sume the 225MHz clock speed of the existing Raw prototype. The end-to-end memory latency would be between 21 and 39 4.44ns machine cycles, or between 93ns and 174ns. If we assume that each Raw tile has a 64Kbyte data cache, then the effective size of the logically shared memory accessible with this latency is about 4Mbytes. (Actually, slightly less, since the data cache on each memory tile is used to store the timestamp cache and log in addition to any memory locations accessed.)

Given that half to three-fourths of this latency (between 44ns and 125ns) is in Raw's highly tuned interconnect, it is difficult to imagine that a dedicated, custom designed, cache controller could deliver significantly lower latency in this technology.

A dedicated, custom designed, cache controller might, however, deliver higher bandwidth. Each transaction handled by the SUDS memory dependence speculation protocol requires access to, at least, one 64-bit timestamp cache entry, one 64-bit log entry, and a 32-bit data memory access. One might improve the transaction rate by accessing these data structures simultaneously. In addition, each transaction must make at least four decisions in the timestamp cache, based on the requested address and timestamp. (Two of these decisions are to check that the correct hash entry has been found, the other two are for timestamp comparisons). One might additionally improve the transaction rate by simultaneously generating, and dispatching on, these conditions.

Even without these optimizations, the SUDS memory dependence speculation system delivers sufficient (although in no way superb) bandwidth. In the current system each of the eight worker nodes is allowed at most four outstanding store operations or one outstanding load operation. Thus there can be at most thirty-two requests simultaneously active in the sixty-four memory banks. The maximum probability of observing contention latency at a memory bank is thus less than

50%. Each transaction generates a total of between 22 and 53 machine cycles of work at the memory node (including the cost of commit), depending on whether or not a timestamp cache entry needs to be allocated during the request. Thus, the SUDS memory dependence speculation system can deliver an average throughput of better than one transaction per machine cycle.

7.2.4 The Birthday Paradox

This section explains a fundamental limit of parallelism on essentially randomly generated dependence graphs (such as one sees in many sparse matrix algorithms). The limitation basically boils down to the "birthday paradox" argument that with only 23 people in a room, the probability that some pair of them have the same birthday is greater than 50%.[3] As demonstrated here, the same argument shows that a memory dependence speculation system can expect to achieve a maximum speedup proportional to $\sqrt[3]{n}$ when randomly updating a data structure of size n.

Suppose we have b different processors, each of which is updating a randomly chosen array element, $B_i \in 1 \ldots n$. What is the probability that every processor updates a different array element?

We have n ways of choosing the first array element, $n-1$ ways of choosing the second array element, so that it is different from the first, $n-2$ ways of choosing the third array element so that it is different than the first two, and so on. Thus there are $\frac{n!}{(n-b)!}$ ways of assigning n array elements to b processors, so that the updates do not interfere. Yet there are a total of n^b ways of assigning n array elements to b processors randomly. Thus the probability, p, that all b accesses are non-interfering is

$$p = \frac{n!}{n^b(n-b)!}. \tag{7.1}$$

[3] The origin of the birthday paradox is obscure. Feller [39] cites a paper by R. von Mises, circa 1938, but Knuth [65], believes that it was probably known well before this.

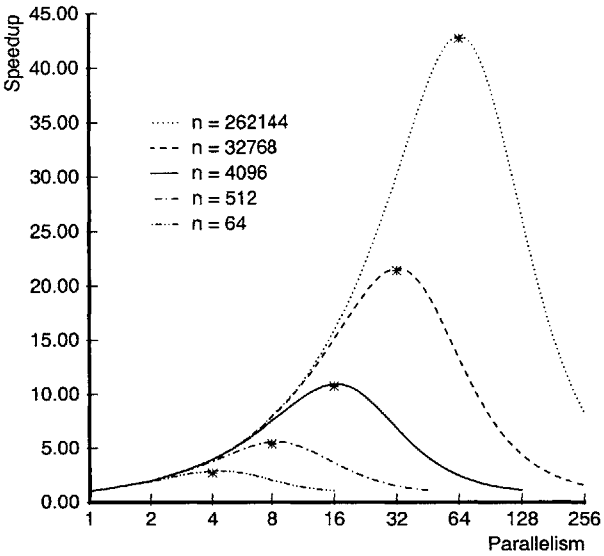


Figure 7-7: Speedup curves for speculatively parallelizing a loop that randomly updates elements of an array of length n as we change the number of processors that are running in parallel. The stars show the points at which parallelism equals $\sqrt[3]{n}$, as described in Equation 7.5.

Let us optimistically assume that, if a sequential processor can run b iterations in time b, that running in parallel on b processors, we can run b iterations in time 1 if none of the accesses conflict (which occurs with probability p), and time 1 + b if there is an access conflict (which occurs with probability 1 − p). The average speedup, S, will be

$$S = \frac{b}{p + (1-p)(1+b)} = \frac{b}{1 + b(1-p)}. \tag{7.2}$$

Note that the assumption that each speculative strip of parallel work is rerun from the beginning on misspeculation, rather than from the point of failure, affects the result only by a small constant factor, since the point of failure will, on average, be about halfway into the speculative strip.

Speedup curves for a variety of n are shown in Figure 7-7. As b varies on the x axis, the speedup increases nearly linearly to some optimal point, but then falls off dramatically as the probability of conflicting iterations starts to increase.

Now let us find the point at which speedup is maximized as a function of b. This will occur when $dS/db = 0$. We work this out as follows. Let $\nu = 1 + b(1-p)$. Then $S = b/\nu$,

$$\frac{d\nu}{db} = -b\frac{dp}{db} + 1 - p,$$

$$\begin{aligned}\frac{dS}{db} &= \frac{\nu - b\frac{d\nu}{db}}{\nu^2} \\ &= \frac{1 - b\frac{d\nu}{db} + b(1-p)}{(1+b(1-p))^2} \\ &= \frac{1 + b^2\frac{dp}{db}}{(1+b(1-p))^2}.\end{aligned}$$

Setting $dS/db = 0$ yields

$$\frac{dp}{db} = \frac{-1}{b^2}. \tag{7.3}$$

Equation 7.1 defines p using factorials, an integer function for which the derivative is not well defined. But we can approximate dp/db by recalling the definition of the derivative.[4] We examine the function $(p(b+h) - p(b))/h$. Letting $h = 1$ we get:

$$\begin{aligned}\frac{dp}{db} &\approx p(b+1) - p(b) \\ &= \frac{n!}{n^{b+1}(n-(b+1))!} - \frac{n!}{n^b(n-b)!} \\ &= \frac{-bn!}{nn^b(n-b)!} \\ &= \frac{-b}{n}p.\end{aligned}$$

[4] I am indebted to my father, David L. Frank, for suggesting this approach.

Solving this differential equation yields

$$p \approx e^{-b^2/2n}. \tag{7.4}$$

Combining the condition on dp/db given by Equation 7.3 with this approximation we get

$$\frac{-1}{b^2} = \frac{-be^{-b^2/2n}}{n},$$

or

$$b^3 = ne^{b^2/2n}.$$

Let us approximate the solution to this equation as $b^* = c\sqrt[3]{n}$. The error from this approximation is $n(c^3 - e^{c^2/(2\sqrt[3]{n})})$. If $c = 1$ then the error is negative for all $n > 0$. For $c > 1$ note that the error is positive whenever $c^3 > e^{c^2/(2\sqrt[3]{n})}$, or taking logarithms, when $n > c^6/8(\ln c^3)^3$. If $c = e^{1/6} \approx 1.18136$ then the error is positive for all $n > e$. Thus for all $n > e$,

$$\sqrt[3]{n} \leq b^* \leq 1.19\sqrt[3]{n}. \tag{7.5}$$

This approximation is demonstrated in Figure 7-7, with stars placed at the optimal points, as calculated by Equation 7.5. Every time the array size is multiplied by a factor of 8, the maximum parallelism increases by a factor of only 2. The intuition behind this cubic result is that as b increases, the probability of success decreases approximately proportional to b^2 while the cost of failure increases approximately as b.

7.3 Discussion

Another way to think about a speculative concurrency control system is to break it into two subsystems. The first subsystem is the checkpoint repair mechanism. The second subsystem checks that a particular concurrent execution produced a result consistent with a sequential ordering of the program. In the SUDS system, the log provides checkpoint repair functionality, while the timestamp cache performs concurrency checking.

Section 7.2.4 discussed the fundamental limits inherent to any system that uses speculation to discover concurrency in essentially randomly generated dependence graphs. This section describes two implementation choices made with respect to the design of the SUDS log and the qualitative impact those implementation choices had on system performance. First, the log implements a bulk commit mechanism instead of a rolling commit mechanism. Second, the log design permits only a single version of each memory location to exist at any one time, rather than a more sophisticated approach where multiple values may be stored simultaneously at a particular memory location.

The impact of several other design and implementation choices is discussed in Chapters 8 and 10. One of the main themes of Chapter 8 involves an implementation mistake with regard to the caching structure implemented *above* the SUDS system. In fact, the SUDS concurrency control subsystem is designed in such a way that implementing a better cache above SUDS would have been particularly easy, and Chapter 8 explains why I failed to do so. Chapter 10 discusses a longer term issue having to do with flat versus nested transaction models. In particular SUDS, like all existing memory dependence speculation and thread level speculation systems implements an inherently flat transaction model. Chapter 10 explains why I believe that future concurrent computer architectures will require *nested* transaction models.

Bulk Commit

The SUDS log is designed in such a way that commit only happens, in bulk, at the end of a speculative strip. Many other memory dependence speculation systems, especially those based directly on Franklin and Sohi's Multiscalar Address Resolution Buffer [43, 44], permit commits to occur on a rolling basis. That is, Multiscalar systems contain an implicit "commit token" that is passed from thread to thread as each completes. When a thread receives the token, the log entries corresponding to that thread are committed and flushed. Thus, in Multiscalar systems, the log commit operations occur concurrently with program execution, as long as no misspeculations occur.

SUDS, in contrast, runs a set of threads corresponding to a speculative strip, and then barrier synchronizes the entire system before committing the logs. The cost of this barrier synchronization step is not overlapped with program execution, and one might worry that the synchronization cost could overwhelm speedup gains. A simple implementation trick, however, amortizes the synchronization cost across several thread invocations, making the effective cost nearly irrelevant. The trick is that, in the SUDS implementation, a speculative strip contains four times as many threads as there are execution units in the system (thirty-two versus eight). As a result, the runtime system only needs to synchronize one-fourth as often, and the synchronization costs are significantly amortized.

While the cost of bulk synchronization is easily amortized, the benefit is substantial. In particular, the work required for log entry allocation and garbage collection becomes nearly trivial. In SUDS, log entries are allocated from a memory buffer in-order (with respect to the arrival of write requests). This can be accomplished simply by incrementing a pointer into this buffer. Deallocation of buffer entries is even more trivial. The pointer is just reset to point to the beginning of the buffer.

With a rolling commit scheme, on the other hand, log entries would be committed in a different order than they were received. Thus the log manager either needs to keep log entries sorted in timestamp order, or else deallocation creates "holes" in the log buffer, forcing the log manager to keep and manage an explicit free list.

Single Version Concurrency Control

The second design choice with respect to the SUDS log is that the SUDS concurrency control system is based on basic timestamp ordering [15], and thus makes only a single version of each memory location available at any time. Memory dependence speculation systems based on the Multiscalar Address Resolution Buffer, in contrast, essentially implement multiversion timestamp ordering [94].

This choice involves a tradeoff. On the one hand, multiversion timestamp ordering is capable of breaking the memory anti-dependence between a load and the following store to the same memory location. On the other hand, since there may be multiple versions associated with each memory location, each load operation must now perform an associative lookup to find the appropriate value.

The empirical question, then, becomes the relative importance of load latency to the cost of flagging some memory anti-dependences as misspeculations. Load latency is almost always on the critical path, and is particularly important in the SUDS runtime, since every load operation goes through the software implemented concurrency control system. How frequent, then, are memory anti-dependences between threads in the same speculative strip?

The key empirical observation is that *most short-term memory anti-dependences are caused by the stack allocation of activation frames,* (rather than heap allocation). That is, if two "threads" are using the same stack pointer, then register spills by the two threads will target the same memory locations. Most contemporary computer systems allocate activation frames on a stack, rather than the heap, because stacks provide slightly lower cost deallocation than does a garbage collected heap [9]. The SUDS compiler allocates activation frames on the heap, rather than a stack, simply because it was the most natural thing to do in a compiler that was already closure converting.[5] Thus, in the SUDS system every thread in the speculative strip gets its own, distinct, activation frame, and register spills between threads never conflict.

This separation of concerns between concurrency control, on the one hand, and memory renaming, on the other, enables the SUDS memory system to implement a particularly low latency path for loads. The SUDS log is specifically, and only, a mechanism for undoing store operations. That is, for each store operation, the store writes directly to memory, and the previous value at that memory location is stored in the log so that the store can be "backed out," if necessary. Thus load operations can read values directly from the memory without touching the log at all.

Caching

Finally, we note the relationship of the SUDS concurrency control mechanism to caching. Unlike other proposals for memory dependence speculation systems, SUDS does not integrate the concurrency control mechanism with the cache coherence mechanism. More specifically, the SUDS concurrency control system sits below the level of the cache coherence protocol in the sense that it assumes requests for each particular memory location arrive in a globally consistent order. Thus decisions about caching can be made almost independently of the concurrency control mechanism. The caveat is that most caching mechanisms are implemented at the level of multi-word cache lines, while the SUDS concurrency control

[5] "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil," Knuth, Computing Surveys, 6(4), 1974.

mechanism is implemented at the level of individual memory words.

Chapter 8

Putting It All Together

In this chapter we describe how all of the parts described in Chapters 3 through 7 fit together in the context of a working prototype SUDS system. The prototype system is described in Section 8.1. Section 8.2 describes, in some detail, several case studies of the use of generalized loop distribution to find concurrency.

SUDS is designed to run on Raw microprocessors. A Raw microprocessor can roughly be described as a single-chip, distributed-memory multiprocessor. Unlike traditional distributed-memory multiprocessors, however, the Raw design is singularly focused around providing low-latency, register-level communication between the processing units (which we call *tiles*). In particular, the semantics of the network and network interface are carefully designed to remove message dispatch overheads [74, 115] and deadlock avoidance/recovery overheads [67] from the critical path. Because of these considerations, a single-word data message can be sent from one tile to a neighboring tile, *and dispatched*, in under *six* 4.44ns machine cycles.

As reported elsewhere [113, 114], each Raw chip contains a 4 by 4 array of tiles; multiple chips can be composed to create systems as large as 32 by 32 tiles. A complete prototype single chip Raw system, running at 225 MHz, has been operational since February 2003. The processor was designed and implemented at MIT over a period of six years by a team that included several dozen students and staff members (although there were probably never more than a dozen people on the project at any one time). The processor was fabricated by IBM in their 0.15 micron SA-27E ASIC process.

8.1 Simulation System

We wanted to understand the properties of SUDS in the context of systems with sizes of 72 tiles, rather than the 16 available in the hardware prototype. Thus, the results in this thesis were generated on a system level simulator of the Raw microprocessor, called `usstdl`.[1] The simulator is both relatively fast, allowing us to run big programs with large data sets, and accurate, providing cycle counts that are within about 10% of the cycle counts provided by the hardware prototype. (`usstdl` is more than 100x faster than the completely cycle accurate behavioral model used by the hardware designers).

There are a few minor functional differences between the simulator and the hardware prototype. First, the simulator does not model interconnect network contention. This is of little consequence to the results reported here, since the total message traffic in the system is sufficiently low compared to the available network bandwidth on the prototype. Although the simulator does not simulate contention inside the interconnect, it does simulate contention at the network interfaces to the tile processors.

The second functional difference between `usstdl` and the hardware prototype is the addition of a second set of load/store instructions. These instructions make

[1] The name `usstdl` is an acronym for "Unified SUDS Simulator and Transactional Data Library," because (for no particularly good reason) both the simulator and library are checked in to the same subtree of the local version control system.

it possible to compile, and use, the C library routines (*e.g.*, strcpy()) so that they will work with either arrays stored in the local memory of a tile, or in the SUDS logically-shared, speculative memory described in Section 7.2.2 and 7.2.3.

These load/store instructions work as follows. They examine the high bit of the requested address. If that bit is a 0, then the the request is destined for one of the software-based memory nodes described in Section 7.2.3. For these requests the machine constructs, and sends, an appropriate message to the memory node, *with the same instruction latencies that would be experienced if the message were constructed in software on the hardware prototype.* This variety of load instruction (called "glw") does *not* have a destination register. Instead, the requested data is returned by a message arriving in the register-mapped network interface.

The code generator is thus designed so that, whenever the semantics of a load instruction are required, two instructions are generated. The first instruction is a glw instruction, which has one register operand specifying the address to be loaded. The second instruction copies the result out of the network interface register to one of the general purpose registers. (Raw's network interface registers are designed so that accesses to a register stall the processor until a message arrives in that register).

If the high bit of the address in a glw instruction is a 1, on the other hand, then the address is accessed from the tile's *local* data cache and the data is fed back to the network input register *as if* a data message had arrived from the interconnect. Because of this functionality, loads and stores in the C library can be compiled using glw and gsw instructions, instead of the normal load and store instructions. As a result, library routines can access data from both the local data cache or from the logically shared memory without recompilation.

Since the latency of local loads is somewhat higher when a library is compiled with this scheme, the C library routines are slightly slower when running under

SUDS than they are when running on a conventional microprocessor. On the other hand, the convenience of not having to compile multiple versions of the library, and then determine which version should be used in each circumstance amply makes up for the small loss in performance. (Consider, for example, the `strcpy` routine from the C library. Without the `glw` and `gsw` instructions we would have to compile four different versions, one for each possible combination of the source and destination strings being in remote memory or on the local stack. Worse, we would then need to determine, for each call, where the two parameters were located, which would require whole program analysis.)

The final, and most important, functional difference between `usstdl` and the Raw hardware prototype is the addition of a set of eight additional dedicated registers for receiving messages from Raw's dynamic network. Each message header includes an index into this register file, and when a message arrives it is directed to the register corresponding to that index. This simply extends the "zero-cycle" message dispatch concept from Raw's other networks so that it works with the particular network that is used by SUDS [115]. (The Raw hardware prototype implements zero-cycle message dispatch on its "static" network, but not on the "dynamic" network that SUDS uses).

Microarchitecturally, adding zero-cycle dispatch to Raw's dynamic network would be a straightforward change, in that it involves changes only in the register fetch stage of the local tile processor pipelines. From a performance standpoint, on the other hand, this change was critical. For example, Section 7.2 gave a breakdown of the 21 cycle round trip cost of performing a load in the SUDS speculative transactional memory system. Without `usstdl`'s zero-cycle message dispatch support, the critical path cost of performing a load increases by more than 12 cycles. This greater than 50% cost increase for each message received at the compute nodes is due entirely to the cost of message dispatching in software. Without zero-cycle dispatch the 2 to 3x SUDS speedup numbers reported below would be impossible to achieve. Instead SUDS would get slowdowns.

Programs running with the SUDS system are parallelized by a prototype SUIF based compiler that outputs SPMD style C code. The transformations performed by this compiler are described in Chapters 3, 4, 5, 6 and 7. The resulting code is compiled for the individual Raw tiles using gcc version 2.8.1 with the `-O3` flag. Raw single-tile assembly code is similar to MIPS assembly code, so our version of the gcc code generator is a modified version of the standard gcc MIPS code generator.

Comparison Systems

For comparison purposes I implemented simulators for two additional systems. The first is a baseline, single-issue 8-stage pipelined RISC processor with a MIPS ISA (similar to a single Raw tile). Programs are compiled directly to this system using the MIPS version of gcc 2.8.1 with the `-O3` flag. The second comparison system is an eight-way issue superscalar running an idealized version of Tomasulo's algorithm. This processor also has a MIPS ISA and programs are compiled directly to the system using the MIPS version of gcc 2.8.1 with the `-O3` flag.

The superscalar simulation is "idealized" in the sense that (a) the trace-fetch mechanism permits traces to be contained in multiple arbitrary cache lines (*i.e.*, the instruction cache is arbitrarily multi-ported), (b) the processor has an effectively infinite set of physical registers, (c) the processor has an effectively infinite set of functional units and (d) the processor has "perfect" zero-latency and infinite bandwidth, bypass networks, scheduling windows and register-file write back paths. The four ways in which the comparison superscalar is *not* idealized are (a) it is limited to fetching a trace of at most eight instructions per cycle, (b) a branch misprediction causes fetch to stall for two cycles, (c) the instruction scheduler obeys register value dependences and (d) only a single data store operation can occur in each cycle. The store buffer implements load bypassing of stores with forwarding. The cache to memory interface permits an effectively infinite number of simultaneous overlapping cache misses. Both the baseline, in-order, and comparison, out-of-order, models use a 32 Kbit gshare branch predictor.

Memory Systems

The memory systems for the baseline (in-order) and comparison (out-of-order) processors include a 4-way associative 64KByte combined I&D L1, 4 MByte L2 with 12 cycle latency and 50 cycle cost for L2 misses to DRAM. The memory system for the SUDS simulations is less idealized.

For the SUDS simulations we use a 72-tile Raw microprocessor. Eight of these tiles are dedicated as "workers" and the other sixty-four are dedicated as "memory nodes." Each of the worker tiles has a 4-way associative 64Kbyte combined I&D L1 cache that is used *only* for caching instructions and thread-local stacks.

In the SUDS system the sixty-four memory tiles work together, as described in Section 7.2.2, to provide a logically shared, speculative, L2 cache accessible to the eight worker nodes. Since this L2 cache is implemented in software on the sixty-four memory nodes, it has an effective size of slightly less than 64×64Kbyte = 4 MBytes. This is because the instructions for the memory dependence speculation software, the hash table data structures, and the log data structures, all compete for use of the 64Kbyte SRAM cache local to each memory tile. In the SUDS simulations L1 cache misses are assumed to take 50 cycles (this is equivalent to the *L2* cache miss penalty for the baseline and superscalar systems).

As described, above and in Section 7.2.2, the worker nodes do *not* cache potentially shared data in their local L1 caches. Rather every access to potentially shared data is forced to undergo the relatively expensive process of remotely accessing the software based memory dependence mechanism on one of the memory nodes. usstdl simulates every aspect of this process in full detail.

I chose all of these parameters simply because I was trying to see whether, in the context of a large and complex system, generalized loop distribution was making a difference. In all cases I have tried to bias the results slightly toward the superscalar. The superscalar's 4 Mbyte L2 cache is of similar size to the 4 Mbytes of cache collectively available on the SUDS memory system, thus any particular program has about the same off-chip miss rates on both systems. The L2 cache latency on the superscalar is lower (by a factor of almost two) than the minimum latency of a SUDS access to the software based memory dependence system. The superscalar L2 cache bandwidth is effectively unlimited, while the SUDS logically shared L2 cache has 64 banks, each of which is limited (by the software based protocol) to servicing approximately one request every 53 cycles (see Section 7.2.3).

Both systems can fetch at most eight useful, user-program, instructions per cycle. The superscalar model is permitted to issue, dispatch, and execute an effectively unlimited number of operations each cycle. usstdl accurately simulates the eight in-order pipelines that SUDS has at its disposal. The scalar operand matching/bypass network on the superscalar has no latency. usstdl accurately models the interconnect latencies of the implemented Raw hardware prototype.

The superscalar model automatically, and in zero cycles, renames every scalar in to an effectively infinite and zero-latency physical register file. The SUDS system renames, in software, into the deferred execution queues created by the loop distribution compiler pass. These queues are stored in the L1 caches of the worker nodes, must be accessed by load and store instructions, and can even suffer cache misses.

The same back end code generator is used for both systems (gcc 2.8.1) and is, at least, decent. Even this, however, slightly favors the superscalar since the glw and gsw instructions are inserted by the parallelizing compiler as volatile gcc inline assembly directives. The semantics of these directives are unknown to the gcc back end, and thus somewhat restrict the compiler's ability to optimize or reorder code.

I have tried, for every architectural parameter that I could think of, to either model that parameter the same way (*e.g.*, off chip memory access latency), or to bias the comparison towards the idealism of Tomasulo's algorithm and against the realistically implementable version of Raw and SUDS. The Raw group at MIT has demonstrated that the Raw hardware prototype, in IBM's SA-27E ASIC process, can be clocked at 225 MHz. It is doubtful whether the idealized superscalar could be clocked at a similar rate, especially given the (zero-cycle) latency chosen for its scalar bypass network.

Thus I feel justified in making the qualitative claim that, when running the same program under the idealized superscalar model and under SUDS on the usstdl simulator, then if the two runs have similar cycle counts, *generalized loop distribution is finding at least as much concurrency, if not more, than does Tomasulo's algorithm*. In fact, for two out of the three programs discussed below, the result is unequivocal, because the cycle counts for SUDS are *better* than the cycle counts for the idealized version of Tomasulo's algorithm.

8.2 Case Studies

This section describes how generalized loop distribution, the SUDS speculation system, and the other transformations described in this thesis interact in the context of three applications. We describe the application of generalized loop distribution to a molecular dynamics simulation program, a decompression program and a program that makes heavy use of recursion.

```
ComputeForces(vector<particle> molecules,
              real cutoffRadius)
  epot = 0.0
  foreach m in molecules
    foreach m’ in m.neighbors()
      if (distance(m, m’) < cutoffRadiusSquare)
        force_t force = calc_force(m, m’)
        m.force += force
        m’.force -= force
        epot += calc_epot(m, m’)
  return epot
```

Figure 8-1: Pseudocode for `ComputeForces`, the Moldyn routine for computing intermolecular forces. The neighbor sets are calculated every 20th iteration by calling the `BuildNeigh` routine (Figure 8-2).

8.2.1 Moldyn

Moldyn is a molecular dynamics simulation, originally written by Shamik Sharma [102], that is difficult to parallelize without speculation support. Rather than calculate all $O(N^2)$ pairwise force calculations every iteration, Moldyn only performs force calculations between particles that are within some cutoff distance of one another (Figure 8-1). The result is that only $O(N)$ force calculations need to be performed every iteration.

The original version of Moldyn recalculated all $O(N^2)$ intermolecular distances every 20 iterations. This made it impossible to run the program on any reasonably large data set. We rewrote the distance calculation routine so that it would also run in $O(N)$ time. This is accomplished by chopping the space up into boxes that are slightly larger than the cutoff distance, and only calculating distances between particles in adjacent boxes (Figure 8-2). This improved the speed of the application on a standard workstation by three orders of magnitude.

Generalized loop distribution and SUDS can parallelize each of the outer loops (those labeled "`foreach m in molecules`" in Figures 8-1 and 8-2). Although the `ComputeForces` routine accounts for more than 90% of program runtime on a standard workstation, each loop has different characteristics when run in parallel,

```
BuildNeigh(vector<list<int>> adjLists,
           vector<particle> molecules,
           real cutoffRadius)
  vector<list<particle>> boxes

  foreach m in molecules
    int mBox = box_of(m.position())
    boxes[mBox].push_back(m)

  foreach m in molecules
    int mBox = box_of(m.position())
    foreach box in adjLists[mBox]
      foreach m' in box
        if (distance(m, m') <
             (cutoffRadius * TOLERANCE))
          m.neighbors().push_back(m');
```

Figure 8-2: Pseudocode for BuildNeigh, the Moldyn routine for recalculating the set of interacting particles. adjLists is a pre-calculated list of the boxes adjacent to each box.

and it is thus instructive to observe the behavior of the other two loops as well.

The first loop in the BuildNeigh routine moves through the array of molecules quickly. For each molecule it simply calculates which box the molecule belongs in, and then updates one element of the (relatively small) boxes array. This loop does not parallelize well on the SUDS system because updates to the boxes array have a relatively high probability of conflicting when run in parallel.

The second loop in the BuildNeigh routine is actually embarrassingly parallel, although potential pointer aliasing makes it difficult for the compiler to prove that this loop is parallel. (The list data structures, "m.neighbors()," are dynamically allocated, individually, at the same program point, and thus the pointer analysis package we are using puts them in the same equivalence class). SUDS, on the other hand, handles the pointer problem by speculatively sending the pointer references to the memory nodes for resolution. Since none of the pointer references actually conflict, the system never needs to roll back, and this loop achieves scalable speedups.

The ComputeForces routine consumes the majority of the runtime in the program. For large problem sizes, the molecules array will be very large, while the number of updates per molecule stays constant, so the probability of two parallel iterations of the outer loop updating the same element of the molecules array is relatively small. Unfortunately, while this loop parallelizes well up to about a dozen compute nodes, speedup falls off for larger numbers of compute nodes because of the birthday paradox problem with memory dependence speculation described in Section 7.2.4. (Recall that this is a fundamental limitation of data speculation systems, not one unique to the SUDS system.)

Despite its small size and seemingly straight-forward structure, parallelization of the ComputeForces routine required nearly every compiler transformation and analysis described in Chapters 5, 6 and 7. The recurrence on the epot variable is reassociated as described in Section 6.4. The memory accesses for the updates specified by the statement "m.force += force" are register promoted to the outer loop, as described in Section 5.4. Equivalence class unification (Section 5.3) is used to discover that there is no memory dependence between the distance and force calculations (which require the position of each molecule), and the updates to the force vector associated with each molecule.

Speculative strip mining (Section 7.1) speculatively breaks the (true) memory dependences between outer loop iterations caused by the force updates. Finally, generalized loop distribution (Section 6) finds two critical nodes in the outer loop. One critical node corresponds to the "index variable" m and the other corresponds to the reassociated updates of the epot variable. The memory dependences between iterations of the outer loop are (speculatively) removed by speculative strip mining, so generalized loop distribution identifies the rest of the work in the outer loop (the distance and force calculations and force updates) as parallelizable.

Figure 8-3 shows the speedups of running Moldyn with an input dataset of

SUDS	3.38
idealized superscalar	3.16

Figure 8-3: Comparison of speedups over an inorder pipeline for Moldyn running on SUDS versus a superscalar.

256,000 particles on SUDS and the idealized superscalar. The baseline in order MIPS R4000 design achieves an average of only 0.223 instructions per cycle (IPC). This is largely due to poor L1 cache behavior. As is common with many numerical/scientific workloads, the working set of this program is considerably larger than the caches. Thus the program gets cache miss rates of about 5% during `BuildNeigh` and 3% during `ComputeForces`.

The idealized superscalar design achieves an IPC of about 0.705, or about 3.16x speedup over the single issue in-order processor. This improvement is achieved largely because the superscalar is able to overlap useful work with some of the cache miss latency.

The SUDS system achieves a speedup of about 3.38x over the single issue in-order processor. This despite the fact that on the superscalar more than 95% of memory accesses are to the L1 cache, while in the SUDS system only about one third of the memory accesses are to stack-allocatable values that can be stored in the L1. The other two thirds of the SUDS memory accesses are routed directly to the logically shared L2 cache, and the SUDS L2 cache is roughly 2x slower than the superscalar L2 cache, because it is implemented in software. Thus, we can conclude that the SUDS system is, somehow, managing to overlap a great deal more work with the long latency memory operations than is the idealized superscalar.

The key to understanding the difference lies in a closer examination of the doubly nested loops that consume most of the program running time. In both cases the number of times that the innermost loop will execute is almost completely unpredictable. Thus, even though the prediction rate for this loop branch is very high (greater than 99%) the superscalar will take a branch misprediction during almost every iteration of the outer loop. The superscalar is thus restricted to finding parallelism in the inner loop, while the SUDS system exploits parallelism in the outer loop. In fact, the conditional inside the `ComputeForces` loop makes things even worse for the superscalar. This branch is not particularly predictable, and thus the superscalar is restricted to looking for parallelism over only a relatively small number of iterations of the inner loop. The SUDS system is finding more concurrency largely because it is able to exploit the control independence of the outer loop upon the inner loop branches.

8.2.2 LZW Decompress

Compression is a technique for reducing the cost of transmitting and storing data. An example is the LZW compression/decompression algorithm [126], widely used in modems, graphics file formats and file compression utilities. Example pseudocode for the version of LZW decompress used in the Unix `compress` utility is shown in Figure 8-4. Each iteration of the outer loop reads a symbol from the input data stream, and traverses an adaptive tree data structure to output a (variable length) output string corresponding to the input symbol.

Note that the input data stream has been *engineered* to remove redundant (easy to predict) patterns. Thus, while the branch prediction rate for any particular static branch is relatively good, the probability of performing an entire iteration of the outer loop without *any* branch mispredictions is close to 0. As a result, Tomasulo's algorithm is limited to searching for concurrency in a relatively small window of instructions.

Generalized loop distribution and memory dependence speculation, on the other hand, can be used to search for concurrency between iterations of the outer loop. The structure of this loop is much more complex than is that of the program

```
outptr = 0
incode = getcode(&input_buf)
while ( incode > EOF )
  int stackp = 0
  code_int code = incode
  char_type the_stack[1<<BITS]

  if (incode == CLEAR)
    free_ent = FIRST
    if (check_error(&input_buf))
      break; /* untimely death! */
    incode = getcode(&input_buf);

  /* Special case for KwKwK string. */
  if ( code >= free_ent )
    the_stack[stackp] = finchar
    stackp = stackp + 1
    code = oldcode

  /* Generate output characters
     in reverse order */
  while ( code >= 256 )
    the_stack[stackp] = tab_suffix[code]
    stackp = stackp + 1
    code = tab_prefix[code]
  the_stack[stackp] = tab_suffix[code]
  stackp = stackp + 1

  /* And put them out in forward order */
  do
    stackp = stackp - 1
    out_stream[outptr] = the_stack[stackp]
    outptr = outptr + 1
  while ( stackp > 0 )

  /* Generate the new tree entry. */
  if ( free_ent < maxcode )
    tab_prefix[free_ent] = oldcode
    tab_suffix[free_ent] = tab_suffix[code]
    free_ent = free_ent + 1
    if (check_error(&input_buf)
      break; /* untimely death! */

  finchar = tab_suffix[code]
  oldcode = incode
  incode = getcode(&input_buf)
```

Figure 8-4: Pseudocode for lzw decompress.

described in the previous section, and thus a considerable amount of analysis and transformation needed to be done to expose this outer-loop concurrency.

First, several of the scalar variables (*e.g.*, `outptr`) are globals and one is a call-by-reference parameter (`input_buf`). Since these scalars would normally be referenced through loop carried dependent loads and stores, they must be register promoted before generalized loop distribution runs. The `input_buf` variable is referenced inside the `getcode` subroutine, and thus to enable register promotion, this subroutine needed to be inlined.

The local array variable, `the_stack`, must be privatized. In this case the array privatization is performed using the scope restriction technique described in Section 5.5. The equivalence class unification (Section 5.3) analysis proves that memory accesses to the `tab_prefix`, `tab_suffix` and `out_stream` data structures are mutually independent.

Both the `tab_prefix` and `tab_suffix` data structures are read and written in a data-dependent fashion during every iteration of the outer loop. This creates true memory dependences between iterations of the outer loop. Speculative strip mining and memory dependence speculation are used to dynamically break these dependences when the data-dependent access pattern allows it.

Next the `outptr` recurrence variable is reassociated, because it is updated with an associative operator in the second inner loop. Recall that this variable was originally a global, and thus register promotion has already been run to turn the references from memory operations into register accesses.

Finally, generalized loop distribution finds six critical nodes in the loop. These correspond to updates to the variables `incode`, `finchar`, `outptr`, `oldcode`, `free_ent` and `input_buf`. When collapsing to the critical node dag we discover that the updates to `incode`, `input_buf`, and `free_ent` form a cyclic critical path. The updates to these variables are "intertwined" in the sense that they depend

SUDS	1.8
idealized superscalar	2.0

Figure 8-5: Comparison of speedups over an inorder pipeline for lzw running on SUDS versus a superscalar.

upon one another. incode is data dependent on input_buf (through the getcode routine). The updates to input_buf, in turn, are control dependent on the outer loop branch, which is data dependent upon incode. Control dependences form a cycle between free_ent and incode.

Generalized loop distribution thus creates four sequential loops corresponding to the four cyclic critical paths. The first sequential loop updates incode, input_buf and free_ent, and creates deferred execution queues for all subsequent loops. The second sequential loop corresponds to updating oldcode.

The next loop can be (speculatively) parallelized, and corresponds to evaluating the parts of the first inner loop and the conditional for the "special case" that updates the, private, code and stackp variables, and to the conditional updates to tab_prefix and tab_suffix.

The third sequential loop corresponds to updating finchar. This is followed by another parallelizable loop that corresponds to all the updates to the private array, the_stack. The iterations of this loop are provably independent of one another, but the code in this loop has *also* been reordered with respect to the updates to tab_suffix performed in the previous speculatively parallel loop. This is what makes speculative strip mining necessary. Both parallel loops are part of the same strip, so the reordering of the code between the two loop bodies can be checked and, if necessary, corrected.

The fourth sequential loop corresponds to updating outptr. This enables the final parallelizable loop, which corresponds to the writes to out_stream.

Figure 8-5 shows the speedups for the idealized superscalar and for SUDS compared to running lzw decompress on the in-order single-issue pipeline. Neither system does particularly well. The superscalar only achieves a speedup of 2x, while SUDS achieves a speedup of 1.8x. Again the superscalar is limited by its inability to predict past the exits of the inner loops.

Generalized loop distribution actually finds a great deal more concurrency in this case than does the idealized version of Tomasulo's algorithm. On the other hand, this code is particularly memory intensive, and, on the in-order microprocessor, almost all of these memory operations hit in the L1 cache. The SUDS memory dependence speculation system, on the other hand, dramatically increases the latency of accessing the `tab_prefix` and `tab_suffix` data structures. While generalized loop distribution is finding enough concurrency to cover a substantial portion of this additional latency, it is not finding quite enough to completely make up for the lack of L1 caching in this case.

8.2.3 A Recursive Procedure

Recursive procedural calls are another way of organizing the control flow in a program. Procedural calls, however, are semantically equivalent to jumps, and thus can be automatically transformed to jumps using generalizations of tail recursion elimination [107]. In particular, this generalization creates explicit trees of activation records, and saves continuations to and restores continuations from this tree [111, 107, 8].

If it is known that sibling procedure calls in the activation tree do not depend upon one another, then it is legal to traverse the tree in either depth first or breadth first order [18, 96, 98]. The breadth first traversal tends to execute sibling nodes in the tree concurrently, but does so at the cost of pushing continuations on to the front, and popping continuations from the back, of a FIFO queue. Each of the pointers (to the front and back of the queue) thus forms an implicit critical node in

```
Traverse(node)
  read(node->parent)
  modify(node)
  foreach c in node->children
    spawn Traverse(c)
```

Figure 8-6: Pseudocode for the tree traversal routine from the `health` program. The `spawn` keyword is simply an annotation that indicates that sibling calls to Traverse are guaranteed to correctly run concurrently.

the resulting program. An additional issue with this breadth first implementation is that a child node may (but will not usually) execute concurrently with its parent, thus violating a true memory dependence.

I wanted to make sure that generalized loop distribution and the SUDS memory dependence speculation system were capable of handling these implicit dependences. For this purpose I chose a program called `health`, written by Martin Carlisle, that was already annotated with information about the legal concurrency between sibling calls in its recursive tree traversal routine [20, 135].

Highly simplified pseudocode for the tree traversal routine in `health` is shown in Figure 8-6. The `spawn` keyword annotation was inserted by the author to indicate that it is legal to run sibling calls to the `Traverse` subroutine concurrently. Note that while it is legal to run sibling calls concurrently, it is not legal to run a child concurrently with its parent, because the child call will read a memory location that may also be modified by the parent. Also, note that, again, Tomasulo's algorithm will be limited to searching for concurrency within a single call to the `Traverse` routine, because the probability of executing an entire call to `Traverse` without any branch mispredictions is near 0.

Figure 8-7 demonstrates how the code in Figure 8-6 is transformed to continuation passing style, and then converted to breadth first traversal, by introducing the explicit `fifo` array, and `head` and `tail` pointers into that array. Note that this transformation has not improved (or particularly degraded) the performance

```
while head < tail
  node = fifo[head++]
  read(node->parent)
  modify(node)
  foreach c in node->children
    fifo[++tail] = c
```

Figure 8-7: After conversion of the recursive `Traversal` routine to "continuation passing style," and introduction of a fifo to make traversal breadth-first rather than depth-first. The routine now has the structure of a loop that can be handled by generalized loop distribution.

SUDS	2.22
idealized superscalar	1.92

Figure 8-8: Comparison of speedups over an inorder pipeline for `health` running on SUDS versus a superscalar.

of Tomasulo's algorithm, because it is still the case that, due to branch mispredictions, it can only search for concurrency within a single iteration of the outer loop.

After speculative strip mining on this loop the two critical nodes found by generalized loop distribution correspond to the updates to `head` and `tail`. In addition, the `tail` variable is updated in the inner loop, and thus must be reassociated by generalized recurrence reassociation.

When the loop distributed code is run concurrently under the SUDS memory dependence speculation system it tends to be the case that there are very few memory misspeculations. There do tend to be a few memory misspeculations when the loop first starts to execute, because the root of the tree attempts to execute concurrently with its immediate children. After getting past this initial misspeculation phase however, the tree branches out widely enough that no more memory system conflicts occur.

As a result of these factors, the SUDS system runs this code about 15% faster than does the idealized model of Tomasulo's algorithm. Again, this is despite the fact that only about 50% of the memory accesses in this program are spills to activation frames that the SUDS system is capable of L1 caching. This program is particularly memory intensive because the operations performed on each node involve linked list traversals. The working set for this program, moreover, is relatively small, and fits completely in the superscalar's L1 cache, so the superscalar is paying almost no latency for L2 cache accesses, while SUDS is paying L2 cache latencies for approximately 50% of the loads operations it performs.

8.3 Discussion

In this section I discuss three limitations of the SUDS prototype system, and ways in which they might be addressed in future work. The first limitation is that SUDS lacks an effective L1 cache. I will discuss subsequently why I believe this to be an implementation error, rather than a more fundamental design flaw. The second limitation has to do with the scalability of the required compiler support. In particular, both the `lzw` and `health` applications required inter-procedural analysis and inlining to perform the required register promotion transformation described in Section 5.4. It is not clear that the inlining transformation, in particular, will effectively scale to applications that are significantly larger than the ones described here. Finally, I will discuss limitations on the parallel scalability of the SUDS system.

An additional limitation of the SUDS system, as with all existing memory dependence speculation systems, is that it implements a flat, rather than nested, transaction protocol. As a result, only one granularity of parallelism can be exploited at a time. This limitation, and some issues that need to be solved before it can be addressed, are discussed in Chapter 10.

L1 Caching

In all three of the applications discussed above, SUDS achieved speedups approximately equal to, or better than, those achieved by the idealized model of Tomasulo's algorithm. In all cases the SUDS system was particularly handicapped by its lack of L1 caching. The lack of L1 caching in the SUDS system, however, is *not* fundamental. As was mentioned briefly at the end of Section 7.3, it is relatively straight forward to add a caching system on top of SUDS, and in fact a software based L1 cache was implemented on top of an earlier version of SUDS [128].

The basic idea behind adding caching on top of SUDS would be to implement a standard directory based cache coherence scheme [21, 10, 2]. The key to a directory based cache coherence scheme is that the directory is guaranteed to see all the traffic to a particular memory location, and in the same global order that is observed in all other parts of the system. Thus, the directory controller can simply forward the list of requests to the concurrency control system, which can then process the information out-of-band.

This would all work fine, but our initial studies showed that the temporal locality exploitable by this scheme is extremely low. This is true for several reasons. First, because the L1 caches are distributed among eight execution units, an L1 fetch by one execution unit does not improve the cache hit rate of any of the seven other execution units. There is another issue, which is not a problem, but that severely limits the temporal locality exploitable by an L1 cache implemented over the SUDS runtime. This is that the SUDS system is *already* directing between 30% and 50% of the memory traffic to the L1 cache in the form of (non-shared) accesses to activation frames. (Primarily for register spills). While the register spill traffic does have a high temporal locality of reference, the rest of the memory traffic, which would be directed to the cache coherence system, does not [128].

The reason that the idealized superscalar is getting L1 cache miss rates significantly better than is the SUDS system lies in the superscalar's 8-word wide L1 cache lines. Essentially, the superscalar is able to *prefetch* useful data before it is required. The SUDS system, without L1 caching, is not able to leverage this advantage.

There are three reasons I did not implement coherent L1 caches for SUDS. The first had to do with my misunderstanding the importance of the spatial locality exploited by wide cache lines. After the initial studies showing the low temporal locality available in the memory system (both in Wilson's thesis [128] and in several informal studies we never published), I became mistakenly convinced that caching wouldn't really buy much.

The second reason I did not implement coherent L1 caches for SUDS was that I wasn't sure how to reconcile word-level concurrency control with multi-word cache lines. I now believe that information about the specific words in a line that have been accessed can probably be piggy-backed on the standard coherence messages, but more work will need to be done to make this efficient.

The third reason I did not implement coherent L1 caches for SUDS had to do with an, arguably, unreasonable fixation that I had on implementing cache controllers in software. It turned out that while this can be spectacularly successful in specific cases [84], it works rather less well for random access data memories. Thus, as described in Wilson's thesis [128], we were unable to implement an L1 data cache with latencies that were significantly lower than the observed latencies in the transactional L2 cache implemented in the final SUDS prototype. In the future I plan to address this deficiency in the context of a hardware implemented L1 cache coherence scheme.

Compiler Scalability

A second question with regard to the SUDS prototype has to do with the scalability of the compiler analyses and transformations. The scalar queue conversion transformation, unidirectional renaming transformation and generalized loop distribution transformations are all intra-procedural. Although I have not done any complexity analysis on these algorithms, several of the control flow graphs in the programs I looked at are relatively large (hundreds of nodes), and on the occasions when I made the mistake of implementing $O(N^3)$ algorithms, I noticed immediately, and was forced to reimplement.

The equivalence class unification and register promotion algorithms described in Sections 5.3 and 5.4, on the other hand, require inter-procedural pointer analysis. For this analysis I relied on Radu Rugina's `span` tool, which is believed to scale, in practice, to programs that are relatively large [97].

A potentially more severe problem was that several of these programs (and in particular `lzw decompress`) were written using global (scalar) variables that were modified inside subprocedures. In order to perform scalar queue conversion on these variables it was necessary to promote them to registers used and modified within the loop being transformed. To perform this register promotion, however, required inlining the corresponding subprocedures.

It is unlikely that such inlining will scale to programs much larger than several tens of thousands of lines of code. It is an open question (and as far as I know, an unexamined question), whether there is a way of performing efficient inter-procedural register promotion. One approach might be to, on an procedure-by-procedure basis, promote globals to call-by-reference parameters, and then promote call-by-reference to copy-in-copy-out. Effecting such a scheme would be an interesting direction for future research.

Parallel Scalability

A third question with regard to the SUDS prototype has to do with the ability of the system to scale to larger degrees of parallelism. The answer to this question actually depends on the application one is looking at. In the case of the three applications discussed in this chapter, the answer is that they do not scale beyond about eight compute nodes.

The reasons are threefold. First, parallel speedups are limited by Amdahl's law, and all of the applications considered here are "do-across" loops, rather than "do-all" loops. That is, these loops contain scalar loop carried dependences (the "critical nodes" identified by generalized loop distribution), and these loop carried dependences limit the available parallelism. For example, the `lzw` program has six critical nodes, and the fraction of execution time spent in the sequential code corresponding to these critical nodes grows as parallelism is increased. Informal experimentation showed that `lzw` sped up by only an additional 2% when run on a system with 16 compute nodes instead of the 8 node system described above.

The second impediment to speedup involves the "birthday paradox" problem described in Section 7.2.4. Recall that this is a problem fundamental to *all* memory dependence speculation systems, not one specific to the SUDS system. For example, the `moldyn` program modifies a sparse-matrix data structure in an effectively random pattern. Informal experimentation showed that `moldyn` exhibits speedup curves qualitatively similar to those shown in Figure 7-7. In fact, the speedup curves for `moldyn` are worse than those shown in the figure, because the figure models only a single update per thread, while in `moldyn` each thread makes, on average, several hundred updates to the shared data structure. For the problem size of 256,000 particles described in Section 8.2.1 maximum speedup occured in a system with eight compute nodes. A sixteen compute node system exhibited less speedup due to an increased number of concurrency violations versus the eight node system.

The final impediment to parallel speedup involves the fundamentally distributed nature of the memory system, as described in Section 1.1. That is, as the size of the memory system grows, the average latency to access a random element in the memory system grows as the square root of the memory size. This problem does not place any maximum limit on the speedup achievable by any application, but it does mean that one can not expect performance to scale linearly as problem size grows.

Chapter 9

Related Work

This chapter describes the relationship of the work in this thesis to previous work in scalar expansion, loop distribution, program slicing, thread-level speculation, critical path reduction and data speculation.

9.1 Scalar Queue Conversion

The idea of renaming to reduce the number of storage dependences in the dependence graph has long been a goal of parallelizing and vectorizing compilers for Fortran [68]. The dynamic closure creation done by the queue conversion algorithm in Section 3 can be viewed as a generalization of earlier work in *scalar expansion* [68, 29]. Given a loop with an index variable and a well defined upper limit on trip count, scalar expansion turns each scalar referenced in the loop into an array indexed by the loop index variable. The queue conversion algorithm works in any code, even when there is no well defined index variable, and no way to statically determine an upper bound on the number of times the loops will iterate. Moreover, earlier methods of scalar expansion are heuristic. Queue conversion is the first compiler transformation that *guarantees* the elimination of all register storage dependences that create cycles across what would otherwise be a unidirectional cut.

Given a loop containing arbitrary forward control flow, *loop distribution* [68] can reschedule that graph across a unidirectional cut [59, 51], but since loop distribution does no renaming, the unidirectional cut must be across the *conservative* program dependence graph (*i.e.*, including the register storage dependences). Queue conversion works across any unidirectional cut of the *value* dependence graph. Because scalar queue conversion always renames the scalars that would create register storage dependences, those dependences need not be considered during analysis or transformation. It is sometimes possible to perform scalar expansion before loop distribution, but loop distribution must honor any register storage dependences that are remaining.

Moreover, existing loop distribution techniques only handle arbitrary *forward* control flow inside the loop, and do so by creating arrays of predicates [59, 51]. The typical method is to create an array of three valued predicates for each branch contained in the loop. Then on each iteration of the top half of the loop a predicate is stored for each branch (*i.e.*, "branch went left", "branch went right" or "branch was not reached during this iteration"). Any code distributed across the cut tests the predicate for its closest containing branch. This can introduce enormous numbers of useless tests, at runtime, for predicates that are almost never true.

Queue conversion, on the other hand, creates and queues closures if and *only if* the dependent code is guaranteed to run. Thus, the resulting queues are (dynamically) often much smaller than the corresponding set of predicate arrays would be. More importantly, queue conversion works across inner loops. Further, because queue conversion allocates closures dynamically, rather than creating static arrays, it can handle arbitrary *looping* control flow, in either the outer or inner loops, even when there is no way to statically determine an upper bound on the number of times the loops will iterate.

Feautrier has generalized the notion of scalar expansion to the notion of array expansion [38]. As with scalar expansion, Feautrier's array expansion works only on structured loops with compile time constant bounds, and then only when the array indices are affine (linear) functions of the loop index variables. Feautrier's technique has been extended to the non-affine case [62], but only when the transformed array is not read within the loop (only written). The equivalence class unification and register promotion techniques described in Chapter 5 extend scalar queue conversion to work with structured aggregates (*e.g.*, C `structs`), but not with arrays. Instead, scalar queue conversion relies on the memory dependence speculation system described in Chapter 7 to parallelize across array references (and even arbitrary pointer references).

The notion of a unidirectional cut defined in Section 3.3 is similar to the notion, from software engineering, of a static program slice. A *static program slice* is typically defined to be the set of textual statements in a program upon which a particular statement in the program text depends [125]. Program slices are often constructed by performing a backward depth first search in the value dependence graph from the nodes corresponding to the statements of interest[90]. This produces a unidirectional cut.

In Section 3.4 we proved that we could produce an executable control flow graph that includes exactly the nodes from the top of a unidirectional cut of the value dependence graph. Yang has proved the similar property, in the context of *structured* code, that an executable slice can be produced by eliding all the statements from the program text that are not in the slice [131]. Apparently it is unknown, given a program text with unstructured control flow, how to produce a control flow graph from the text, elide some nodes from the graph and then *accurately* back propagate the elisions to the program text [13].[1] Generalizations of

[1] A potential solution, of which I am unable to find any mention in the literature, would be to associate information about goto statements with *edges* in the control flow graph, rather than nodes.

Yang's result to unstructured control flow work only by inserting additional dependences into the value dependence graph [13, 24], making the resulting slices larger and less accurate. The proof in Section 3.4 demonstrates that when working directly with control flow graphs (rather than program texts) this extra work is unnecessary, even when the control flow is irreducible.

Further, program slicing only produces the portion of the program corresponding to partition A of a unidirectional cut A-B. In Sections 3.5 and 3.6 we demonstrated how to queue and then resume a set of closures that reproduce the execution of partition B as well.

The reason queue conversion generalizes both loop distribution and program slicing is that *queue conversion makes continuations [111, 107, 8] explicit.* That is, any time we want to defer the execution of a piece of code, we simply create, and save, a closure that represents that code, plus the suspended state in which to run that code. It is standard to compile functional languages by making closures and continuations explicit [107, 8], but this set of techniques is relatively uncommon in compilers for imperative languages.

In fact, the SSA based static renaming optimization in Chapter 4 was anticipated by work from formal programming language semantics that demonstrates that continuation passing style representations and SSA form flow graphs of imperative programs are semantically equivalent [58]. Based on this work, Appel has suggested that a useful way of viewing the ϕ nodes at the join points in SSA flow graphs is as the point in the program at which the actual parameters should be copied into the formal parameters of the closure representing the code dominated by the ϕ node [7]. This roughly describes what the algorithm given in Chapter 4 does.

That is, given a maximal group β containing a use of variable x for which we

Hopefully, this will be investigated in the future.

are going to create a closure, we rename x to x' (which can be viewed as the formal parameter). Then we introduce a new closure, containing the instruction $x' = x$, at the ϕ point which *shares* an environment containing x' with β. It is useful to view the new closure as simply copying the actual parameter, x, to the formal parameter x'.

Traditional superscalar micro-architectures do renaming *only at the top of the stack* by having the compiler register allocate automatic variables and then renaming the registers at runtime [117, 57, 104, 83, 105]. This technique is used ubiquitously in modern architectures because it performs at least enough renaming to reach the parallelism limits imposed by flow dependences [124]. Unfortunately, the renamed registers are an extraordinarily constrained resource, making it impossible for superscalars to exploit flow dependences that can be eliminated through control dependence analysis [40].

Dataflow architectures [35, 34, 91, 28] do as much (or more) renaming as does queue conversion, but at the cost of insisting that all programs be represented purely functionally. This makes converting to dynamically allocated closures easy (because loops are represented as recursive procedures), but substantially restricts the domain of applicability. Queue conversion works on imperative programs, and, at least for scalar variables, performs renaming in a similar way.

Even more troubling than the inability of dataflow architectures to execute imperative programs, was that they contained no provision for handling overflow of renaming buffers [27]. Recently, work on efficient task queue implementations for explicitly parallel functional programming languages [85] has been extended to provide theoretical bounds on the renaming resources required by such systems [18, 17]. SUDS provides constant bounded resource guarantees through its checkpoint repair mechanism. This mechanism allows SUDS to rollback and sequentially reexecute any program fragment that exhausts renaming resources when run in parallel. Further, the SUDS memory dependence speculation mechanism allows SUDS to automatically parallelize sequential programs, written in conventional imperative programming languages, rather than relying on programmers to explicitly parallelize their programs.

The original motivation for queue conversion comes from previous work in *micro-optimization*. Micro-optimization has two components. The first, *interface decomposition* involves breaking up a monolithic interface into constituent primitives. Examples of this from computer architecture include Active Messages as a primitive for building more complex message passing protocols [121], and interfaces that allow user level programs to build their own customized shared memory cache coherence protocols [22, 70, 95]. Examples of the benefits of carefully chosen primitive interfaces are also common in operating systems research for purposes as diverse as communication protocols for distributed file systems [99], virtual memory management [50], and other kernel services [16, 55].

The second component of micro-optimization involves using automatic compiler optimizations (*e.g.*, partial redundancy elimination) to leverage the decomposed interface, rather than forcing the application programmer to do the work. This technique has been used to improve the efficiency of floating-point operations [31], fault isolation [122], shared memory coherence checks [100], and memory access serialization [37, 14]. On Raw, micro-optimization across decomposed interfaces has been used to improve the efficiency of both branching and message demultiplexing [74], instruction cache tag checks [84, 80], and data cache tag checks [86, 130].

Queue conversion micro-optimizes by making the renaming of scalar variables an explicit operation. Because queue conversion renames into dynamic memory, rather than a small register file, instructions can be scheduled over much longer time frames than they can with Tomasulo's algorithm. On the other hand, queue conversion can limit the costs of renaming to exactly those points in a program where an anti- dependence or output- dependence might be violated by a specific schedule. Further, we will show in Chapter 4 that, because scalar queue conversion makes renaming an explicit operation, the compiler can move the renaming point to a point in the program *between* the production of a value and its consumption, thus minimizing the number of times the renaming must occur.

9.2 Loop Distribution and Critical Path Reduction

As described above, generalized loop distribution generalizes loop distribution [68, 59, 51], by using scalar queue conversion to guarantee the elimination of all scalar anti- and output- dependences. Thus, generalized loop distribution simultaneously does the work of scalar expansion and loop distribution. In addition, generalized loop distribution distributes loops that contain arbitrary control flow, including inner loops.

A transformation similar to loop distribution, called *critical-path reduction* has been applied in the context of thread-level speculative systems [120, 109, 133]. Rather than distribute a loop into multiple loops, critical-path reduction attempts to reschedule the body of the loop so as to minimize the amount of code executed during an update to a critical node. While the transformation is somewhat different than that performed by loop distribution, loop distribution and critical-path reduction share the goal of trying to minimize the time observed to update state visible outside the loop body.

Schlansker and Kathail [101] have a critical-path reduction algorithm that optimizes critical paths in the context of superblock scheduling [53], a form of trace scheduling [41]. Vijaykumar implemented a critical-path reduction algorithm for the multiscalar processor that moves updates in the control flow graph [120]. Steffan *et al* have implemented a critical-path reduction algorithm based on Lazy Code Motion [63] that moves update instructions to their optimal point [109, 133]. As with previous loop distribution algorithms, none of these critical-path reduction algorithms can reschedule loops that contain inner loops.

9.3 Memory Dependence Speculation

Timestamp based algorithms have long been used for concurrency control in transaction processing systems. The memory dependence validation algorithm used in SUDS is most similar to the "basic timestamp ordering" technique proposed by Bernstein and Goodman [15]. More sophisticated multiversion timestamp ordering techniques [94] provide some memory renaming, reducing the number of false dependences detected by the system at the cost of a more complex implementation. Optimistic concurrency control techniques [69], in contrast, attempt to reduce the cost of validation, by performing the validations in bulk at the end of each transaction.

Memory dependence speculation is even more similar to virtual time systems, such as the Time Warp mechanism [54] used extensively for distributed event driven simulation. This technique is very much like multiversion timestamp ordering, but in virtual time systems, as in data speculation systems, the assignment of timestamps to tasks is dictated by the sequential program order. In a transaction processing system, each transaction can be assigned a timestamp whenever it enters the system.

Knight's Liquid system [61, 60] used a method more like optimistic concurrency control [69] except that timestamps must be pessimistically assigned *a priori*, rather than optimistically when the task commits, and writes are pessimistically buffered in private memories and then written out in serial order so that different processing elements may concurrently write to the same address. The idea of using hash tables rather than full maps to perform independence validation was originally proposed for the Liquid system.

Knight also pointed out the similarity between cache coherence schemes and coherence control in transaction processing. The Liquid system used a bus based protocol similar to a snooping cache coherence protocol [47]. SUDS uses a scalable protocol that is more similar to a directory based cache coherence protocol [21, 10, 2] with only a single pointer per entry, sometimes referred to as a Dir1B protocol.

The ParaTran system for parallelizing mostly functional code [116] was another early proposal that relied on speculation. ParaTran was implemented in software on a shared memory multiprocessor. The protocols were based on those used in Time Warp [54], with checkpointing performed at every speculative operation. A similar system, applied to an imperative, C like, language (but lacking pointers) was developed by Wen and Yelick [127]. While their compiler could identify some opportunities for privatizing temporary scalars, their memory dependence speculation system was still forced to do renaming and forward true-dependences at runtime, and was thus less efficient than SUDS.

SUDS is most directly influenced by the Multiscalar architecture [43, 106]. The Multiscalar architecture was the first to include a low-latency mechanism for explicitly forwarding dependences from one task to the next. This allows the compiler to both avoid the expense of completely serializing do-across loops and also permits register allocation across task boundaries. The Multiscalar validates memory dependence speculations using a mechanism called an address resolution buffer (ARB) [43, 44] that is similar to a hardware implementation of multiversion timestamp ordering. From the perspective of a cache coherence mechanism the ARB is most similar to a full-map directory based protocol.

More recent efforts have focused on modifying shared memory cache coherence schemes to support memory dependence speculation [42, 48, 110, 66, 56, 49]. SUDS implements its protocols in software rather than relying on hardware mechanisms. In the future SUDS might permit long-term caching of read-mostly values by allowing the software system to "permanently" mark an address in the timestamp cache.

Another recent trend has been to examine the prediction mechanism used by dependence speculation systems. Some early systems [61, 116, 49] transmit all dependences through the speculative memory system. SUDS, like the Multiscalar, allows the compiler to statically identify true-dependences, which are then forwarded using a separate, fast, communication path. SUDS and other systems in this class essentially statically predict that all memory references that the compiler can not analyze are in fact *independent*. Several recent systems [87, 119, 25] have proposed hardware prediction mechanisms, for finding, and explicitly forwarding, additional dependences that the compiler can not analyze.

Memory dependence speculation has also been examined in the context of fine-grain instruction level parallel processing on VLIW processors. The point of these systems is to allow trace-scheduling compilers more flexibility to statically reorder memory instructions. Nicolau [89] proposed inserting explicit address comparisons followed by branches to off-trace fix up code. Huang *et al* [52] extended this idea to use predicated instructions to help parallelize the comparison code. The problem with this approach is that it requires $m \times n$ comparisons if there are m loads being speculatively moved above n stores. This problem can be alleviated using a small hardware set-associative table, called a memory conflict buffer (MCB), that holds recently speculated load addresses and provides single cycle checks on each subsequent store instruction [45]. An MCB is included in the Hewlett Packard/Intel IA-64 EPIC architecture [12].

The LRPD test [93] is a software speculation system that takes a more coarse grained approach than SUDS. In contrast to most of the systems described here, the LRPD test speculatively block parallelizes a loop as if it were completely data parallel and then tests to ensure that the memory accesses of the different processing elements do not overlap. It is able to identify privatizable arrays and reductions at runtime. A directory based cache coherence protocol extended to perform the LRPD test is described in [134]. SUDS takes a finer grain approach that can cyclically parallelize loops with true-dependences and can parallelize most of a loop that has only a few dynamic dependences.

Chapter 10

Conclusion

I believe that the time is right for a revival of something similar to the "dataflow" architectures of the last decade. The dataflow machines of the past, however, had two problems. Fortunately, a system like SUDS can help to address these problems.

The first problem was that dataflow machines did not run imperative programs, but only programs written in functional programming languages [35, 34, 91, 85, 28, 18, 17].[1] Scalar queue conversion can help address this problem because it converts scalar updates into function (closure) calls.

The second problem with dataflow machines was that their renaming mechanisms were not fundamentally deadlock free [27]. Checkpoint repair mechanisms, like that provided by the SUDS transactional memory speculation system, can help address this problem by rolling back to a checkpointed state whenever the renaming mechanism overflows the available buffers.

On the other hand, dataflow machines have a desirable property that the SUDS system does not. This is that dataflow machines allow the expression of concurrency at the finest granularity, while the runtime system can be made

[1] More recent dataflow languages, like Cilk [18], permit imperative state updates. Unfortunately, the programmer is still forced to write their parallel code in terms of recursive calls to stateless functions. This actually makes the problem worse, since unlike the pure functional languages used in early dataflow machines, Cilk provides no way for the compiler to automatically check that programmers have not unwittingly inserted data races into their programs.

responsible for choosing the granularity most appropriate for the available resources [85, 18, 17].

SUDS, like all existing memory dependence speculation and thread speculation systems, implements a flat transaction model, and thus allows only *one* level of parallelism to be expressed in any particular loop. Consider the example program used throughout Chapters 2, 3 and 4 (shown in Figure 2-2). In this example we decided to use generalized loop distribution to parallelize the *outer* loop, but, depending on the relative trip counts of the inner and outer loops this choice could have been disastrous. If the outer loop iterates many times while the inner loop iterates only a few times then generalized loop distribution on the outer loop will work quite well. Most of the concurrency would be discovered and exploited. The deferred execution queues, created by scalar queue conversion, would have stayed relatively small because the size of these queues is proportional to the trip count of the inner loop.

On the other hand, if the outer loop iterates only a few times and the inner loop iterates many times, then applying generalized loop distribution to the outer loop would produce poor results. The system would try to exploit concurrency only in the outer loop. Meanwhile, each iteration of the outer loop would correspond to a thread, and that thread would create a deferred execution queue corresponding to the work in the inner loop. Since the inner loop executes many times, the deferred execution queues grow large, and could potentially overflow the available memory resources. This overflow would invoke the (relatively expensive) checkpoint recovery mechanism. Thus, the loop would end up executing completely sequentially with the added cost of attempting and then aborting each speculative strip.

One solution to this problem has been to develop heuristic compiler analyses to try to guess which loops will be most profitable to parallelize [133]. Another (not very attractive) solution would be to apply speculative strip mining and generalized loop distribution to *every* loop, and then use a runtime predictor to decide which loop in each loop nest should be speculatively parallelized.

Combining dataflow with scalar queue conversion and transactional concurrency control might provide an attractive alternative. In this case scalar queue conversion could be applied to every unidirectional cut in an imperative program that might expose concurrency. The result would be that (except for memory dependences) the program would be, essentially, transformed into a fine grain, functional, dataflow program. The runtime system could then, as in lazy task systems [85, 18, 17] dynamically choose to invoke each closure either as a conventional procedure call or as a concurrent thread as parallel resource become available.

To actually build such a system one would have to solve a number of problems. There are at least two problems that seem particularly difficult (and therefore, fun) to me. The first is how to reconcile the nested concurrency exposed by the system with the speculative transactional model. SUDS, like all other existing memory dependence and thread speculation systems, implements a flat transaction model. Theoretical nested transaction processing protocols have been proposed [88, 54], but actual, efficient, implementations of such systems seem to be in short supply. In nested transaction systems even seemingly simple problems, like efficient timestamp implementation, seem to require baroque solutions (see, for example, [116]). A second problem has to do with how one could extend the existing work on dynamic memory dependence prediction [87, 119, 25] to nested transaction systems.

Perhaps then, this dissertation, in the end, raises more questions than it answers. In the introduction I stated that the SUDS system was built on three techniques. They were dynamic scalar renaming, control dependence analysis, and speculation. I believe that these three techniques are *necessary* for finding and exploiting concurrency. On the other hand, I have *not* shown, (and do not believe), that these three techniques are *sufficient* for finding and exploiting concurrency. I like to think that this dissertation brings us a step closer to the goal of building a microprocessor that effectively finds and exploits concurrency in the kinds of programs that programmers really write. Reaching that goal will, I think, require a journey that is both long and enjoyable.

Bibliography

[1] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *15th International Symposium on Computer Architecture*, pages 280–289, Honolulu, HI, May 1988.

[3] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000.

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

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the International Conference on Supercomputing*, June 1990.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[7] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4), 1998.

[8] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the Symposium on Principles of Programming Languages*, 1989.

[9] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1), 1996.

[10] James Archibald and Jean-Loup Baer. An economical solution to the cache coherence problem. In *11th International Symposium on Computer Architecture*, pages 355–362, Ann Arbor, MI, June 1984.

[11] Semiconductor Industry Association. International technology roadmap for semiconductors, 2001.

[12] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen mei W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 227–237, Barcelona, Spain, June 1998.

[13] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, 1993.

[14] Rajeev Barua, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Maps: A compiler-managed memory system for Raw machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 4–15, Atlanta, GA, May 2–4 1999.

[15] Philip A. Bernstein and Nathan Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 285–300, Montreal, Canada, October 1980.

[16] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, December 3-6 1995.

[17] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[18] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM*, pages 356–368, November 1994.

[19] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization*, March 2003.

[20] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[21] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[22] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, Chicago, Illinois, April 18–21, 1994.

[23] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[24] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of GOTO statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, 1994.

[25] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 142–153, Barcelona, Spain, June 1998.

[26] Keith D. Cooper and John Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, NV, June 1997.

[27] David E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, May 1988.

[28] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM: A compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, June 1993.

[29] Ron Cytron and Jeanne Ferrante. What's in a name? The value of renaming for parallelism detection and storage allocation. In *Proceedings of the 16th Annual International Conference on Parallel Processing*, pages 19–27, St. Charles, IL, August 1987.

[30] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[31] William J. Dally. Micro-optimization of floating-point operations. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–289, Boston, MA, April 3–6, 1989.

[32] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, 1992.

[33] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, 1987.

[34] J. B. Dennis. Dataflow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[35] Jack B. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 126–132, 1974.

[36] Wilm E. Donath. Placement and average interconnnection lengths of computer logic. *IEEE Transactions on Circuits and Systems*, 26(4):272–277, April 1979.

[37] John R. Ellis. *Bulldog: A Compiler for VLIW Architecture*. PhD thesis, Department of Computer Science, Yale University, February 1985. Technical Report YALEU/DCS/RR-364.

[38] Paul Feautrier. Array expansion. In *Proceedings of the International Conference on Supercomputing*, pages 429–441, July 1988.

[39] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, New York, NY, 3rd edition, 1968.

[40] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[41] Joseph A. Fisher. Trace Scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[42] Manoj Franklin. Multi-version caches for Multiscalar processors. In *Proceedings of the First International Conference on High Performance Computing (HiPC)*, 1995.

[43] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *19th International Symposium on Computer Architecture (ISCA-19)*, pages 58–67, Gold Coast, Australia, May 1992.

[44] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[45] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–193, San Jose, California, October 1994.

[46] A. González, M. Valero, N. Topham, and J.M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Eleventh International Conference on Supercomputing*, 1997.

[47] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *10th International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, June 1983.

[48] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High Performance Computer Architecture (HPCA-4)*, pages 195–205, Las Vegas, NV, February 1998.

[49] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, San Jose, CA, October 1998.

[50] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, MA, October 12–15, 1992.

[51] Bor-Ming Hsieh, Michael Hind, and Ron Cryton. Loop distribution with multiple exits. In *Proceedings Supercomputing '92*, pages 204–213, Minneapolis, MN, November 1992.

[52] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 200–210, Chicago, Illinois, April 1994.

[53] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1625–1640, December 1995.

[54] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[55] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, Saint-Malo, France, October 5-8 1997.

[56] Iffat H. Kazi and David J. Lilja. Coarse-grained speculative execution in shared-memory multiprocessors. In *International Conference on Supercomputing (ICS)*, pages 93–100, Melbourne, Australia, July 1998.

[57] Robert M. Keller. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, December 1975.

[58] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, January 1995.

[59] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings Supercomputing '90*, pages 407–416, New York, NY, November 1990.

[60] Thomas F. Knight, Jr. System and method for parallel processing with mostly functional languages, 1989. U.S. Patent 4,825,360, issued Apr. 25, 1989 (expired).

[61] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 88–93, August 1986.

[62] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*, January 1998.

[63] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[64] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[65] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[66] Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *International Conference on Supercomputing (ICS)*, Melbourne, Australia, July 1998.

[67] John D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1998.

[68] David J. Kuck, R. H. Kuhn, David A. Padua, B. Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 1981.

[69] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[70] 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 *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 18–21, 1994.

[71] Butler W. Lampson. Fast procedure calls. In *Proceedings of the First International Conference on Architectural Support for Programming Languages and Operating Systems*, 1982.

[72] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[73] B.S. Landman and R.L .Russo. On pin versus block relationship for partitions of logic circuits. *IEEE Transactions on Computers*, 20(12):1469–1479, December 1971.

[74] Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.

[75] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[76] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, pages 18–41, July 1993.

[77] Zhiyuan Li. Array privatization for parallel execution of loops. In *Conference Proceedings, 1992 International Conference on Supercomputing*, Washington, DC, July 1992.

[78] J.D.C. Little. A proof of the queueing formula $L = \lambda W$. *Operations Research*, 9(3):383–387, May 1961.

[79] Raymond Lo, Fred C. Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, Montreal, Quebec, June 1998.

[80] Albert Ma, Michael Zhang, and Krste Asanovìc. Way memoization to reduce fetch energy in instruction caches. In *Workshop on Complexity-Effective Design, ISCA-28*, Göteborg, Sweden, June 2001.

[81] Doug Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[82] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of the Symposium on Principles of Programming Languages*, volume POPL-20, pages 2–15, Charleston, SC, January 1993.

[83] Wen mei W. Hwu and Yale N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Transactions on Computers*, 36(12):1496–1514, 1987.

[84] Jason Eric Miller. Software based instruction caching for the RAW architecture. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1999.

[85] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[86] Csaba Andras Moritz, Matthew Frank, and Saman Amarasinghe. FlexCache: A framework for flexible compiler generated data caching. In *Proceedings of the 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 12 2000. to appear Springer LNCS.

[87] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th Annual International Symposium on Microarchitecture (MICRO)*, Research Triangle Park, NC, December 1997.

[88] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1981.

[89] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[90] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In Peter B. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, pages 177–184, Pittsburgh, PA, April 1984.

[91] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 82–91, 1990.

[92] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence and the roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3), May 1997.

[93] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, CA, June 1995.

[94] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[95] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, Illinois, April 18–21, 1994.

[96] Radu Rugina and Martin C. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 72–83, Atlanta, GA, May 1999.

[97] Radu Rugina and Martin C. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 77–90, Atlanta, GA, May 1999.

[98] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195, Vancouver, BC, June 2000.

[99] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[100] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1–5, 1996.

[101] Michael Schlansker and Vinod Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[102] Shamik D. Sharma, Ravi Ponnusamy, Bogki Moon, Yuan shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing*, pages 97–106, Washington, DC, November 1994.

[103] James E. Smith. A study of branch prediction strategies. *Proceedings of the International Symposium on Computer Architecture*, ISCA-8:135–148, May 1981.

[104] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[105] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 29(3):349–359, March 1990.

[106] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[107] Guy Lewis Steele. RABBIT: A compiler for Scheme. Technical Report AITR-474, MIT Artificial Intelligence Laboratory, May 1978.

[108] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, January 1995.

[109] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2002.

[110] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 2–13, Las Vegas, NV, February 1998.

[111] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1):135–152, April 2000. (Republication of Oxford University Computing Laboratory Technical Monograph PRG-11, 1974).

[112] Bjarne Stroustrup. *The Design and Evolution of* C++. Addison-Wesley, 1994.

[113] Michael Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, March 2002.

[114] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[115] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2003.

[116] Pete Tinker and Morry Katz. Parallel execution of sequential scheme with ParaTran. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 40–51, July 1988.

[117] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[118] Peng Tu and David Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, Portland, OR, August 1993.

[119] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *30th Annual International Symposium on Microarchitecture (MICRO)*, Research Triangle Park, NC, December 1997.

[120] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, January 1998.

[121] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992.

[122] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, North Carolina, December 5-8 1993.

[123] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, September 1997.

[124] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume IV, pages 176–188, April 1991.

[125] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[126] Terry Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[127] Chih-Po Wen and Katherine Yelick. Compiling sequential programs for speculative parallelism. In *Proceedings of the International Conference on Parallel and Distributed Systems*, Taiwan, December 1993.

[128] Kevin W. Wilson. Integrating data caching into the SUDS runtime system. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2000.

[129] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[130] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanovìc. Direct addressed caches for reduced power consumption. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, volume 34, Austin, TX, December 2001.

[131] Wuu Yang. *A New Algorlthm for Semantics-Based Program Integration*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, August 1990.

[132] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. *Proceedings of the International Symposium on Microarchitecture*, MICRO-24:51–61, November 1991.

[133] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, October 2002.

[134] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 162–173, Las Vegas, NV, February 1998.

[135] Craig B. Zilles. Benchmark Health considered harmful. *Computer Architecture News*, pages 4–5, January 2001.