The Multiflow Trace Scheduling Compiler

P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes,
W. D. Lichtenstein, Robert P. Nix, John S. O’Donnell, John C. Ruttenberg

(Formerly of) Multiflow Computer[1]

10/30/92

The Multiflow compiler uses the trace scheduling algorithm to find and exploit instruction-level parallelism beyond basic blocks. The compiler generates code for VLIW computers that issue up to 28 operations each cycle and maintain more than 50 operations in flight. At Multiflow the compiler generated code for eight different target machine architectures and compiled over 50 million lines of FORTRAN and C applications and systems code.

The requirement of finding large amounts of parallelism in ordinary programs, the trace scheduling algorithm, and the many unique features of the Multiflow hardware placed novel demands on the compiler. New techniques in instruction scheduling, register allocation, memory-bank management, and intermediate-code optimizations were developed, as were refinements to reduce the overhead of trace scheduling.

This paper describes the Multiflow compiler and reports on the Multiflow practice and experience with compiling for instruction-level parallelism beyond basic blocks.

Keywords: trace scheduling, compiler optimization, instruction scheduling, register allocation, memory-reference analysis, VLIW, performance analysis, instruction-level parallelism, speculative execution.

[1] Authors’ current addresses: P. Geoffrey Lowney, Digital Equipment Corporation, 85 Swanson Rd, Boxborough, MA, 01719; Stefan M. Freudenberger, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA, 94304; Thomas J. Karzes, D.E. Shaw and Co, 39th Floor, Tower 45, 120 West 45th Street, NY, NY, 10036; W. D. Lichtenstein, Thinking Machines Corp, 245 First St, Cambridge, MA, 02138; Robert P. Nix, Digital Equipment Corporation, 85 Swanson Rd, Boxborough, MA, 01719; John S. O’Donnell, Equator Technologies, 1738 26th Avenue East, Seattle, WA, 98112; John C. Ruttenberg, Silicon Graphics, 31 Cherry St, Milford, CT, 06460.

1 Introduction

For the last 30 years, declining hardware costs have encouraged computer scientists and engineers to seek increased performance through parallelism. In the area of single-CPU performance, this search yielded high-performance computers as early as 1964 [67, 70] that performed on-the-fly data precedence analysis to keep multiple functional units busy. Great interest has been focused on the upper limits of parallelism within existing software. Studies now 20 years old [69, 29], confirmed by later work [38], show that only small benefits are available when parallelism is sought within basic blocks.

This limitation is troublesome for scientific programs, where regularity and operation independence is intuitively obvious, and where CPU performance is critical. Because no practical technique for scheduling individual operations from beyond basic blocks was known, data-parallel operations (vector instructions) were added to scientific computers. Compiler techniques were developed for recognizing vector opportunities in loop-based patterns of operations [7, 8, 43, 5]. These techniques, now known as *vectorization*, suffered from limitations in their applicability [49].

In 1979 Fisher [26] described an algorithm called trace scheduling, which proved to be the basis for a practical, generally applicable technique for extracting and scheduling parallelism from beyond basic blocks [28, 53]. The work of Fisher’s group at Yale, particularly that of Ellis [23], showed that large potential speedups from parallelism were available on a wider range of applications than were amenable to vectorization. Multiflow Computer, Inc. was founded in 1984 to build on this line of research, and to develop processors similar to those envisioned in the Yale research: VLIW machines that could execute many operations at once. An overview of the machines is provided here; for detailed discussion see [19, 20, 39]. Multiflow closed its doors in March, 1990. This paper reports on the compiler developed at Multiflow from 1984 to 1990.

The Multiflow compilers are the key component of computer systems that utilize instruction-level parallelism on a larger scale than ever before attempted. Parallelism is achieved on a wider range of applications than vectorization can handle. Furthermore, this parallelism is achieved with relatively simple hardware; most of the complexities attendant upon identifying and scheduling operation ordering is handled in software, while hardware simply carries out pre-determined schedules.

The techniques developed in the Multiflow compiler will be very applicable to future generations of RISC processors, which will integrate many functional units on a single chip.

1.1 Trace Scheduling Overview

The trace scheduling algorithm permits instruction scheduling beyond basic blocks (Figure 1-1). It provides a framework for a unified approach to the scheduling of simple loops, loops with conditionals, and loop-free stretches of code. Multiflow demonstrated that it was possible to have a single instruction-scheduling strategy that yielded many of the benefits of more complex approaches to loop scheduling [60, 44, 45, 2]. The algorithm allows a natural separation between global and local correctness issues. This leads to a compiler structure which closely resembles that of a traditional, basic-block scheduling compiler, with the addition of a trace scheduling module.

A summary description of the basic algorithm follows; much more detail is presented later in the paper.

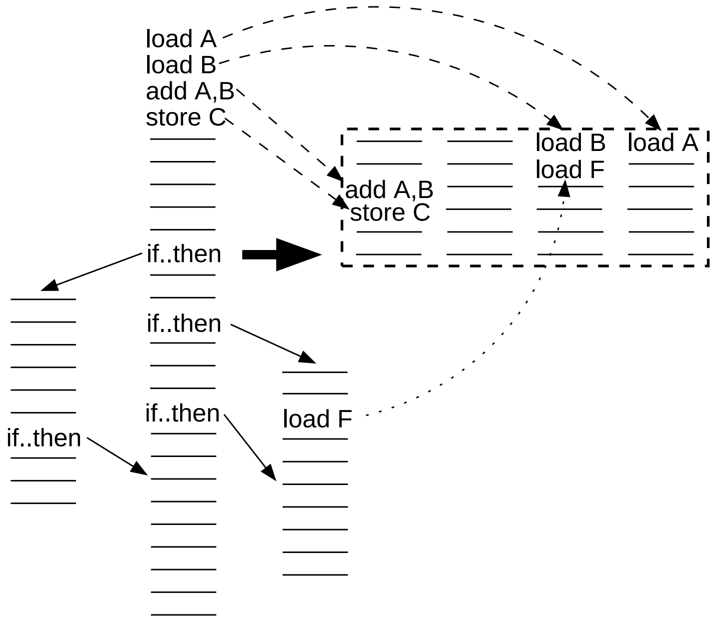


Figure 1-1: Code scheduling across basic block boundaries

After all intermediate optimization has been done, and all operations have been expanded to machine-level opcode sequences, the flow graph of operations is passed to the trace scheduler (TS). It first annotates the graph with expected execution frequencies. These are generated by linear combination of branch probabilities and loop trip counts, obtained either from heuristics or from measurements of prior runs of the application. The TS then enters the following loop:

A. Select a sequence of operations to be scheduled together (Figure 1-2). This sequence is called a trace. Traces are limited in length by several kinds of boundaries; the most significant ones are module boundaries (entry/return), loop boundaries (no trace includes operations within and without a given loop), and previously scheduled code.

B. Remove the trace from the flow graph, passing it to the instruction scheduler (called the code generator in Bulldog [23]).

C. When the instruction scheduler returns the finished schedule for the trace, place the schedule in the flow graph, replacing the operations originally in the trace (Figure 1-3). At schedule boundaries other than the main entry and exit, correct logical inconsistencies which arise from operations moving above and below splits and above joins. This can require generating copies of some of the operations scheduled on the trace (see Figure 1-4).

D. Loop, until all operations have been included in some trace and all traces have been replaced by schedules (Figure 1-5).

E. Select the best linear order for the object code and emit it.

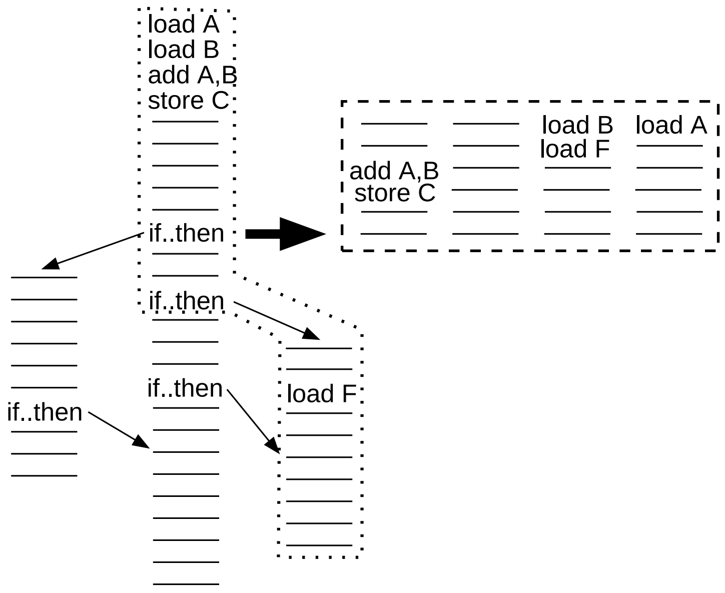


Figure 1-2: Select a trace and schedule code within the trace

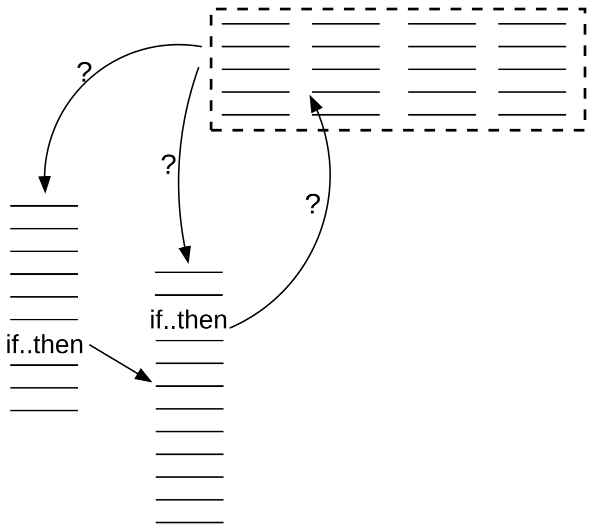


Figure 1-3: Replace the trace with the scheduled code and analyze state at split and join points

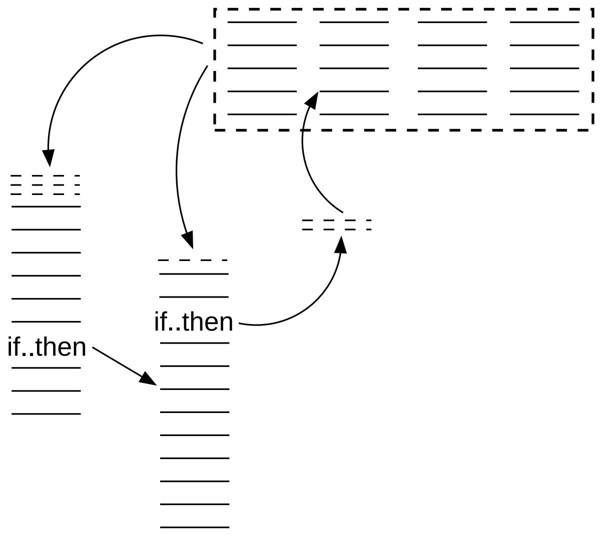


Figure 1-4: Generate compensation code to resolve split and join state differences

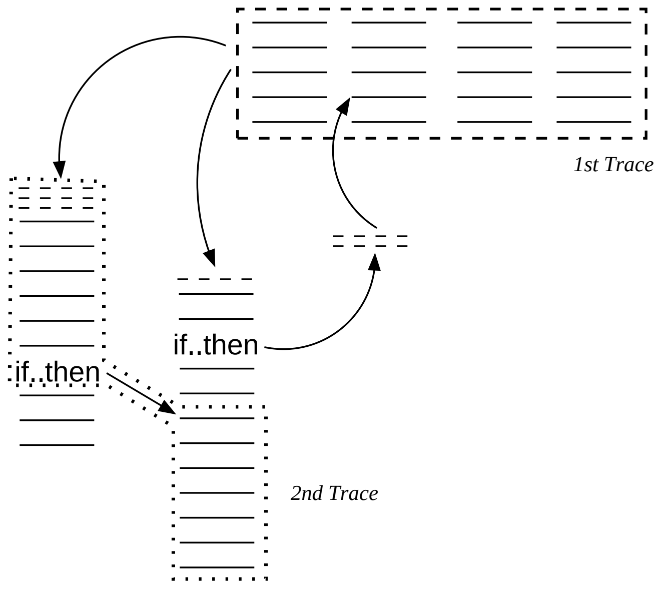


Figure 1-5: Iterate, selecting each trace based on successive priority

The task of the Instruction Scheduler (IS) within this framework is as follows. The IS receives one trace at each invocation. It builds a data precedence graph (DPG) to represent data precedence constraints on execution order and adds various heuristic edges to the DPG to further constrain the schedule. The entire scheduling problem which the IS must solve is incorporated in the DPG, which can represent a single basic block, a sequence of basic blocks with complicated conditionals, or the (possibly unrolled) body of a loop. Special handling for a particular kind of code is effected by adding heuristic edges to the DPG.

The scheduler attempts to build the fastest possible sequence of instructions for the DPG. Operations are frequently moved past one or more splits or joins to issue them "greedily" or "speculatively", in order to shorten the length of the expected execution path.

This simple picture is complicated somewhat by register allocation and the management of machine resources in use across multiple execution paths. These issues are discussed in sections 9 and 10.

2 Outline

In the following section we present an overview of the Multiflow Trace machines. Section 4 gives some history of the compiler, and section 5 describes its structure. We then describe the phases of the compiler: the front end in section 6, the optimizer in section 7 and the back end in section 8. To present the back end in more detail, we describe the trace scheduler in section 9, the instruction scheduler in section 10, the machine model in section 11, the calling sequence in section 12, and the disambiguator in section 13. We evaluate the performance of the compiler in sections 14 and 15 and close with some retrospective conclusions in section 16.

3 The Trace machines

3.1 Basics

All Multiflow computers share a set of common architectural features. They are all VLIWs; they encode many operations in a single long instruction. Operations are RISC-like: fixed 32-bit length, fixed-format, three register operations with memory accessed only through explicit loads and stores. Operations are either completed in a single cycle or explicitly pipelined; pipelines are self-draining. The machines are not scoreboarded and machine resources can be oversubscribed. The memory system is interleaved. The compiler must avoid register conflicts, schedule the machine resources, and manage the memory system.

There are three series of Multiflow machines, eight models in all.

- The 200 series, which first shipped in January, 1987. It is implemented in CMOS Gate arrays and TTL logic with Weitek CMOS floating point chips. It has a 65ns cycle time.
- The 300 series, which first shipped in July, 1988. This is a 10%-redesign of the 200 with Bipolar Integrated Technologies (BIT) ECL floating point parts. The cycle time remained at 65ns.
- The 500 series. This is an ECL semi-custom implementation that was fully designed but not completely fabricated or tested when the company closed in March, 1990. It targeted a 15ns cycle time.

The 200 and 300 series come in three widths: a 7-wide, which has a 256-bit instruction issuing 7 operations; a 14-wide with a 512-bit instruction; and a 28-wide with a 1024-bit instruction. The 500 was designed in only 14-wide and 28-wide versions. The wider processors are organized as multiple copies of the 7-wide functional units; we call the 7-wide group of functional units a cluster. For most of the paper, we will focus on the 300 series. Figure 3-1 shows a 7/300, and Figure 3-2 shows a 28/300.

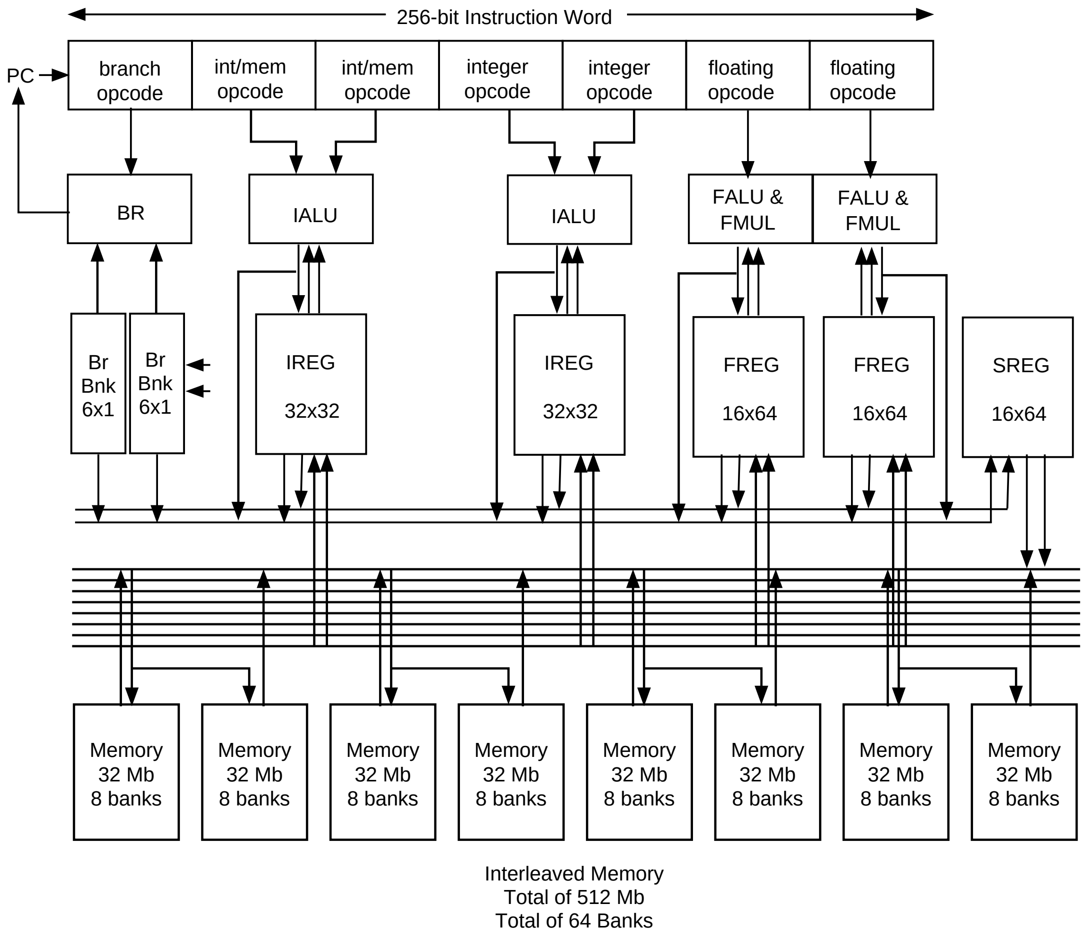


Figure 3-1: The Multiflow TRACE 7/300

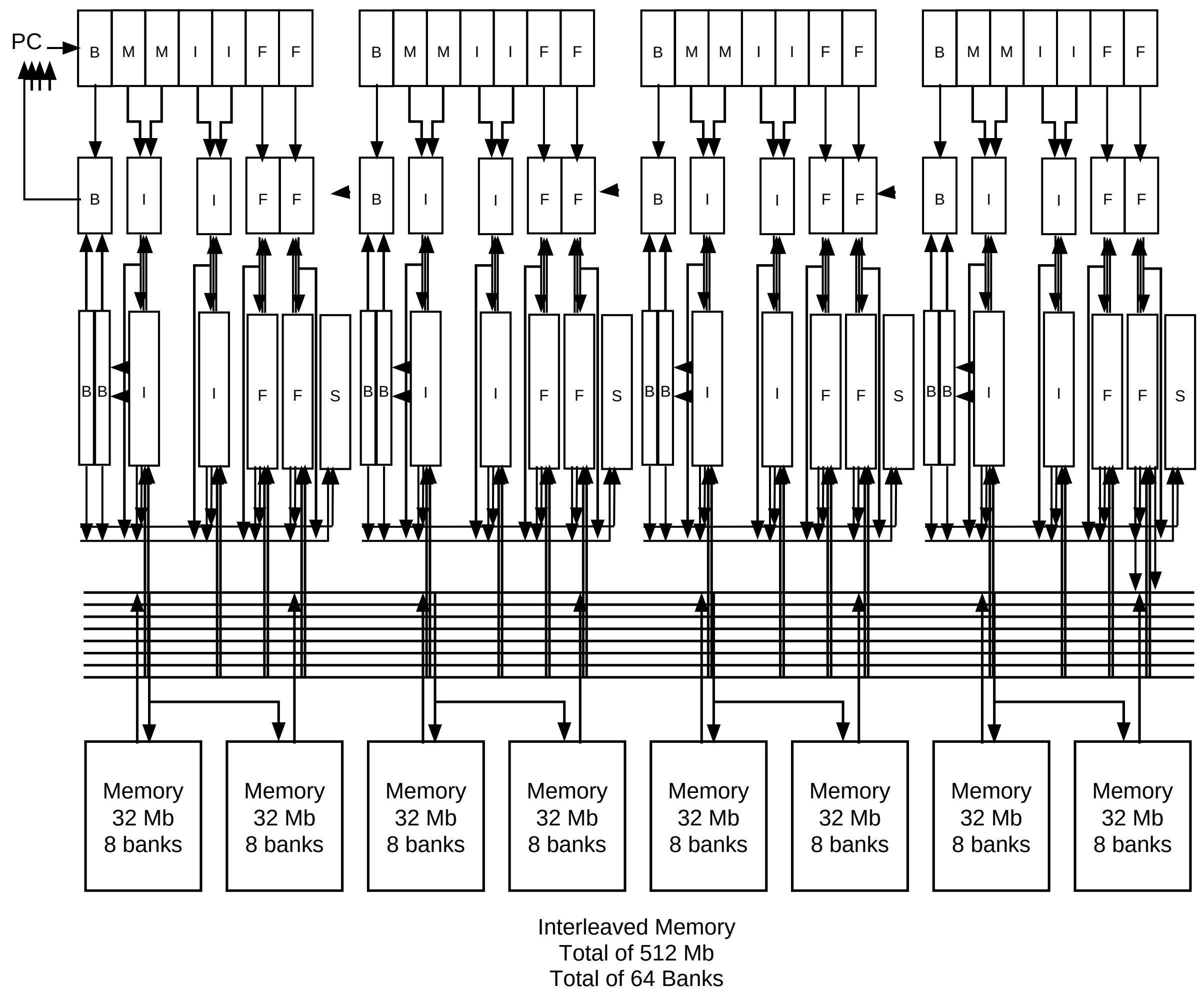


Figure 3-2: The Multiflow TRACE 28/300

In the 300 series, instructions are issued every 130ns; there are two 65ns beats per instruction. Integer operations can issue in the early and late beats of an instruction; floating point operations issue only in the early beat. Most integer ALU operations complete in a single beat. The load pipeline is 7 beats. The floating point pipelines are 4 beats. Branches issue in the early beat and the branch target is reached on the following instruction, effectively a two beat pipeline. An instruction can issue multiple branch operations (4 on the 28/300); the particular branch taken is determined by the precedence encoded in the long instruction word.

- There are 4 functional units per cluster: two integer units and two floating units. In addition, each cluster can contribute a branch target. Since the integer units issue in both the early and the late beat, a cluster has the resources to issue 7 operations for each instruction.

- There are 9 register files per cluster (36 register files in the 28/300). See Figure 3-3. Data going to memory must first be moved to a store file. Branch banks are used to control conditional branches and the select operation.

Register-File Type	*Number*	*Elements*	*Size*
Integer	2	32	32
Floating	2	16	64
Store	1	16	64
Integer branch	2	6	1
Floating branch	2	1	1

Figure 3-3: Register files per cluster

- The instruction cache holds 8K instructions (1Mb for the 28/300). There is no data cache.
- The memory system supports 512 Mb of physical memory with up to 64 way interleaving. There is a 4Gb virtual address space.
- There are two IO processors. Each supports a 246 Mb/sec DMA channel to main memory and two 20 Mb/s VME busses.

Figure 3-4 presents the basic performance figures for the 300 series.

	7/300	*14/300*	*28/300*
MOPs	53	107	215
Mflops	30	60	120
Main memory Mb/s	123	246	492
Linpack 1000x1000	23	42	70
Linpack 100x100	11	17	22
SPECmark	NA	23	25
Sustainable ops-in-flight	10-13	20-26	40-52

Figure 3-4: Hardware performance of the TRACE 300 family

Figure 3-5 shows two instructions of 14/300 code, extracted from the inner loop of the 100x100 Linpack benchmark. Each operation is listed on a separate line. The first two fields identify the cluster and the functional unit to perform the operation, the remainder of the line describes the operation. Note the destination address is qualified with a register-bank name (e.g., sb1.r0); the ALUs could target any register bank in the machine (with some restrictions). There is extra latency in reaching a remote bank.

```
instr   cl0     ialu0e    st.64          sb1.r0,r2,17#144
        cl0     ialu1e    cgt.s32        li1bb.r4,r34,6#31
        cl0     falu0e    add.f64        lsb.r4,r8,r0
        cl0     falu1e    add.f64        lsb.r6,r40,r32
        cl0     ialu0l    dld.64         fb1.r4,r2,17#208
        cl1     ialu0e    dld.64         fb1.r34,r1,17#216
        cl1     ialu1e    cgt.s32        li1bb.r3,r32,zero
        cl1     falu0e    add.f64        lsb.r4,r8,r6
        cl1     falu1e    add.f64        lsb.r6,r40,r38
        cl1     ialu0l    st.64          sb1.r2,r1,17#152
        cl1     ialu1l    add.u32        lib.r32,r36,6#32
        cl1     br        true and r3    L23?3
        cl0     br        false or r4    L24?3;

instr   cl0     ialu0e    dld.64         fb0.r0,r2,17#224
```

```
cl0      ialu1e    cgt.s32        li1bb.r3,r34,6#30
cl0      falu0e    mpy.f64        lfb.r10,r2,r10
cl0      falu1e    mpy.f64        lfb.r42,r34,r42
cl0      ialu0l    st.64          sb0.r4,r2,17#160
cl1      ialu0e    dld.64         fb0.r32,r1,17#232
cl1      ialu1e    cgt.s32        li1bb.r4,r35,6#29
cl1      falu0e    mpy.f64        lfb.r10,r0,r10
cl1      falu1e    mpy.f64        lfb.r42,r32,r42
cl1      ialu0l    st.64          sb0.r6,r1,17#168
cl1      ialu1l    bor.32         ib0.r32,zero,r32
cl1      br        false or r4    L25?3
cl0      br        true and r3    L26?3;
```

Figure 3-5: TRACE 14/300 Code Fragment

3.2 Data types

The natural data types of the machine are 32-bit signed and unsigned integers, 32-bit pointers, 32-bit IEEE-format single precision, and 64-bit IEEE-format double precision. 16-bit integers and characters are supported with extract and merge operations; bit strings with shifts and bitwise logicals; long integers by an add with carry; and booleans with normalized logicals. There is no support for extended IEEE precision, denormalized numbers, or gradual underflow.

Accesses to memory return 32 or 64 bits. Natural alignment is required for high performance: *0 mod 4* for 32 bit references; *0 mod 8* for 64 bit. Misaligned references are supported through trap code, with a substantial performance penalty.

Memory is byte addressed. Using byte addresses eases the porting of C programs from byte addressed processors such as the VAX and 68000. The low bits of the address are ignored in a load or a store, but are read by extract and merge operations. Accessing small integers is expensive. Each load of a character requires a ld.8/ext.s8 sequence, and each store requires a ld.8/mrg.s8/st.8 sequence. Figure 3-6 shows the schedule generated for two character copies.

```
void copy2(a,b)
char *a, *b;
{
        a[0] = b[0];
        a[1] = b[1];
}

mark_trace 1;
instr   cl0    ialu0e    ld.8       ib0.r32,r4,zero
        cl0    ialu0l    sub.u32    lib.r0,r0,32#?2.1?2auto_size;
instr   cl0    ialu0e    bor.32     lib.r33,zero,r4
        cl0    ialu0l    ld.8       ib0.r35,r3,zero
        cl0    ialu1l    add.u32    lib.r34,r33,6#1;
instr   cl0    ialu0e    bor.32     lib.r36,zero,r3
        cl0    ialu1l    add.u32    lib.r1,r36,6#1;
instr   cl0    ialu1l    ext.s8     lib.r32,r32,r33
```

```
        cl0    gc        mnop      2;
instr   cl0    ialu1e    mrg.s8    lsb.r0,r35,r32,r3
        cl0    ialu0l    st.8      sb0.r0,r3,zero;
instr   cl0    ialu0e    ld.8      ib0.r32,r4,6#1
        cl0    ialu0l    ld.8      ib0.r33,r3,6#1
        cl0    gc        mnop      4;
instr   cl0    ialu1l    ext.s8    lib.r32,r32,r34;
instr   cl0    ialu1e    mrg.s8    lsb.r0,r33,r32,r1
        cl0    ialu0l    st.8      sb0.r0,r3,6#1;
instr   cl0    ialu0l    add.u32   lib.r0,r0,32#?2.1?2auto_size
        return;
```

Figure 3-6: Sequence for copying two characters

3.3 Memory system and data paths

The Multiflow Trace has a two level interleaved memory hierarchy exposed to the compiler. All memory references go directly to main memory; no data cache is present. There are 8 memory cards, each of which contains 8 banks. Each bank can hold 8Mb, for a total capacity of 512Mb. Memory is interleaved across the cards and then the banks. The low byte of an address determines its bank: bits 0-1 are ignored, bits 2-4 select a card and bits 5-7 select a bank.

Data is returned from memory on a set of global busses which are approximately shown in Figure 3-2.[1] These busses are shared with moves of data between clusters; to maintain full memory bandwidth on a 28/300 the number and placement of data moves must be carefully planned.

Each level of interleaving has a potential conflict.

- Card/bus conflict. Within a single beat, all references must be to distinct cards, and they must use distinct busses. If two references conflict on a card or a bus, the result is an undefined program error.
- Bank conflicts. A memory bank is busy for 4 beats from the time it is accessed. If another reference touches the same bank within the 4-beat window, the entire machine stalls. To achieve maximum performance, the compiler must schedule successive references to distinct banks.

A 28/300 can generate 4 references per beat, and, if properly scheduled, the full memory bandwidth of the machine can be sustained without stalling. The 28/300 is asymmetric in that it can perform only 2 stores per beat; 2 compatible loads need to be paired with the 2 stores to achieve maximum bandwidth.

3.4 Global resources

In addition to functional units and register banks, the Trace machines have a number of global shared resources that need to be managed by the compiler.

- Register file write ports. Each integer and floating register file can accept at most 2 writes per beat, one of which can come from a local ALU. Each branch bank can accept 1 write per beat. Each store file can accept 2 writes per beat.
- Global busses. There are 10 global busses; each can hold a distinct 32-bit value each beat. The hardware contains routing logic; the compiler need only guarantee that the number of busses is not oversubscribed in a beat. (Actually the busses come in 4 types, and each type must be separately scheduled.)
- Global control. There is one set of global controller resources, which control access to link registers used for subroutine calls and indirect branches.

[1] For a more detailed presentation of the Trace memory system, see [19].

3.5 Integer units

The integer units execute a set of traditional RISC operations. There are a number of features added to support trace scheduling:

- There are a set of dismissable load opcodes. These opcodes set a flag to the exception handler that indicates that the load operation is being performed speculatively (see 3.8 below).
- All operations that compute booleans are invertible. The trace scheduler prefers invertible branches, so that it can layout a trace as straight-line code by inverting branch conditions as necessary.
- A 3-input, one output select operation (a = b ? c : d) is provided. This permits many short forward branches to be mapped into straight line code.

A conditional branch is a two operation sequence. An operation targets a branch bank register, and then the branch reads the register. A conditional branch requires 3 beats, and, since the branch can only issue on an instruction boundary, in sparse code it can require 4. Having separate register files for the branch units relieves pressure on the integer register files and provides additional operand bandwidth to support the simultaneous branch operations.

The two integer ALUs per cluster are asymmetric; only one can issue memory references. This presented problems for the instruction scheduler, as we discuss in section 10.

Due to limits on the size of our gate arrays, no integer ALUs shared a register file. This fact, coupled with the low latency of integer operations, makes it difficult for the instruction scheduler to exploit parallelism in integer code. The cost of moving data between register files often offsets the gains of parallelism. Figure 3-7 shows how a simple parallel integer sequence can be slower on two ALUs with separate register files than one ALU. With a single register file the parallelism can easily be exploited.

1 ALU		*2 ALU*		*2 ALU*	
1 RF		*1 RF*		*2 RF*	
0:	cmp i>n	0:	cmp i>n; i=i + 1	0:	mov i to other bank
1:	i = i + 1			1:	cmp i > n ; i = i + 1
				2:	mov i back

Figure 3-7: The multiple register file dilemma

3.6 Floating units

The floating units in the 300 series are the BIT ECL floating point parts. There are two units per cluster, and each can execute the same repertoire of floating operations. Each also implements a full complement of integer operations, but only the move and select operations are used by the compiler.

The floating units have relatively small register files; 15 64-bit registers per file are available to the compiler. All pipelines on the Trace are self-draining; if an interrupt occurs the pipelines drain before the interrupt is serviced. This means that operations may complete earlier than determined by the compile-time schedule, so a subsequent operation cannot target the destination register of an operation until the first operation is completed. The load latency is 7 beats, and the floating point latency is 4. A load can issue every beat; a flop every other beat. Thus 9 distinct destination registers are required in each floating bank to keep the pipelines full. This leaves only 6 registers per bank to hold variables, common subexpressions, and the results of operations that are not immediately consumed.

There is no pipelined floating move. A move between floating registers takes 1 beat and consumes a register write-port resource. This can prevent another floating point operation issued 3 beats earlier from using the same write port; thus a floating move can in some situations lock out two floating point operations. To address this, a pipelined move operation was added to the 500 series.

The floating units can be used in two special modes: multiply-accumulate mode and pair mode. In multiply-accumulate mode, each operation can perform both a multiply and an add. This mode was added late in the machine design, and because of some unusual constraints, it cannot be used by the compiler. Pair mode, which is supported by the compiler, allows each 64-bit register to be treated as a 2-element vector of single precision operands.

3.7 Instruction encoding

The instruction encodings are large, especially for code that does not use all of the functional units. To save space in memory and on disk, object code is stored with NOP operations eliminated. Instructions are grouped in blocks of 4, and the non-NOP operations are stored with a preceding mask word that indicates which NOPs have been eliminated. When an instruction is loaded into the icache, it is expanded into its full width.

To save space in the icache, an instruction can encode a multi-beat NOP, which instructs the processor to stall for the specified number of beats before executing the following instruction.

The large instruction format permits a generous number of immediate operands. There is a full 32-bit word worth of immediates for each cluster in each beat. The word can be divided into 16-bit pieces to construct shorter offsets. In addition one of the two source register specifiers for each integer operation can be interpreted as a 6-bit immediate. The compiler uses immediates heavily. Constants are never loaded from memory; they are constructed from the instruction word. Double precision constants are pieced together out of two immediate fields. The global pointer scheme used by most RISC machines[15] is not required. However, due to the large number of branches and memory references that can be packed into a single instruction, the immediate resource can be oversubscribed. Our unrollings are tuned to keep the offsets created from induction variable simplification small, and, on the wide machine, care is taken to place the targets of loop exits so that they can be reached with a short branch offset.

3.8 Trap code and timings

Trap hardware and trap code supports virtual memory, trapping on references to unmapped pages and stores to write-protected pages. To prevent unwarranted memory faults on speculative loads, the compiler uses the dismissable load operation. If a dismissable load traps, the trap code does not signal an exception, but returns a NAN or integer zero, and computation continues; if necessary, a translation buffer miss or a page fault is serviced. NANs are propagated by the floating units, and checked only when they are written to memory or converted to integers or booleans. Correct programs exhibit correct behavior with speculative execution on the Trace, but an incorrect program may not signal an exception that it would have signaled if compiled without speculative execution.

The hardware also supports precise floating exceptions, but the compiler cannot move floating operations above conditional branches if this mode is in use. This mode was only used when compiling for debugging.

The trap code supports access to misaligned data, piecing together the referenced datum from multiple 32-bit words. In doing so, it places 8 and 16-bit quantities in the correct place in a 32-bit word, so that extracts and merges work correctly.

The 300 series supports three performance counters: a beat counter, a cache miss counter, and a bank-stall counter. These are accessible to software and provide very accurate measurements of program execution.

3.9 Compiler issues

In summary, the main compiler issues for the Trace machines are as follows.

- There are a large number of pipelined functional units, requiring from 10-50 data independent operations in flight to fill the machine. This large amount of instruction level parallelism requires scheduling beyond basic blocks and a strategy for finding parallelism across loop iterations. In addition, there is functional-unit bandwidth to support speculative execution.
- Machine resources can be oversubscribed, and the pipelines use resources in every beat. The compiler must precisely model the cycle-by-cycle state of the machine even in situations where such a model is not critical for performance.

- There is an interleaved memory system that must be managed by the compiler. Card conflicts cause program error; bank conflicts affect performance.
- Each functional unit has its own register file. It costs an operation to move a value between registers, although remote register files may be targeted directly at the cost of an extra beat of latency.

4 Compiler history

The roots of Multiflow compiler technology are Fisher's thesis on trace scheduling [26, 27], and the Bulldog compiler developed by Fisher, Ellis, Ruttenberg, Nicolau and others at Yale [28, 23, 54]. Bulldog implements Fisher's algorithm in a prototype compiler. It presents a complete design of optimization, memory-reference analysis, register allocation, and instruction scheduling in a trace scheduling compiler for a hypothetical VLIW. At Multiflow, Bulldog was used for experiments to guide the design of the Trace architecture. It also served as the high level design for the production compiler.

We deviated from the design of Bulldog only when profitable or necessary, but numerous changes were made. These changes were made for two reasons. Most changes were due to the different goals and scope of the two projects. The Bulldog compiler is a 30,000 line Lisp program, compiling a small FORTRAN subset to a hypothetical machine; its goal is to explore the issues in compiling for a VLIW. The Multiflow compiler is a 500,000 line C program, compiling FORTRAN and C to a series of production VLIWs. The goal of the compiler is to generate high performance code for a VLIW and to present the traditional C and FORTRAN environment found on a workstation or a mini-computer. The different goals and scope of the two compilers led to major changes. The internal representations used by Bulldog were not adequate to represent the full programming language semantics of FORTRAN and C; this led to a rethinking of the front end and the optimizer. The memory-reference analyzer was redesigned to exploit the complex view of memory relationships presented by FORTRAN and C. The machine model was recreated to represent the Multiflow Trace series of machines; the instruction scheduler became much more complex. The heuristics used throughout the compiler are more developed and tuned, and several new optimizations were introduced. Yet at the high level, the compiler has the same structure as Bulldog, particularly the implementation of the trace scheduling algorithm.

The second source of changes were two fundamental issues not addressed by Bulldog: relative memory-bank disambiguation and spilling registers.

- Memory-bank disambiguation. Bulldog performs static bank disambiguation; the compiler determines which memory bank would be addressed by each memory reference. This requires that the compiler be able to compute *address mod b*, where *b* is the number of banks, at compile-time; when the bank cannot be determined, a central sequential memory controller is used.

  Static bank disambiguation is impractical for languages with pointers and by-reference arguments. The Multiflow Trace permits relative bank disambiguation.[1] The compiler must ensure that all of the references issued simultaneously are to distinct banks, but it does not need to know which banks. Relative conflicts are more frequently resolvable at compile-time. For example, if A is an aligned double precision array, A(I) and A(I+1) are known to refer to different banks, though we typically do not know which bank A(I) references.

  In Bulldog, bank disambiguation is performed as a single pass over the memory references in the program. In the Multiflow compiler, memory-bank management needs to be integrated with the scheduling of functional units; this is a major complication to the instruction scheduler.

- Spilling registers. The Bulldog compiler does not spill registers to memory; it assumes the machine provides enough registers for the routine being compiled. Our experience at Multiflow is that registers are a critical resource of the machine. In routines, other than simple kernels, that present a large amount of parallelism to the compiler, the decision of when to spill and restore values is very important to achieving high performance.

[1] In the context of the compiler, we frequently refer to both bus/card and bank conflicts as bank conflicts.

5 Compiler structure

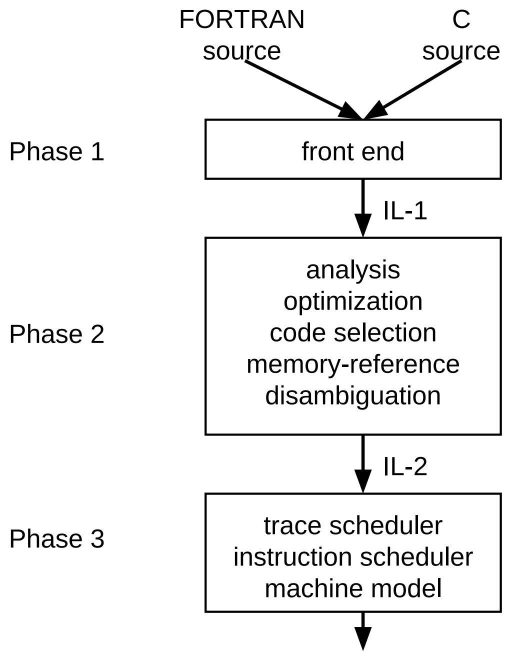


Figure 5-1: Structure of the Multiflow compiler

The Multiflow compiler has the three phase structure presented in Figure 5-1. Phase 1 is a FORTRAN or C front end, which produces a high level intermediate representation called IL-1. Phase 2 analyzes and optimizes the program, and lowers the representation into IL-2. Phase 3 performs the trace scheduling algorithm and produces machine code.

Phase 2 and Phase 3 operate on IL semantics, independent of the source language. Operations in IL are n-tuples: an op-code, followed by a list of written operands and a list of read operands. Operands are either constants or virtual registers called temporaries. *N-ary* operations provide opportunity for flexible expansions into binary operations; the optimizer can find more parallelism and more potential loop invariants and common subexpressions. They also provide a simple solution to the requirements of FORTRAN parentheses.

IL-1 is the interface between Phase 1 and Phase 2; it defines a high-level virtual machine. At this level we attempt to capture the memory access model defined by the programming language and defer lowering memory references until Phase 2; this is very useful in performing memory-reference analysis. Memory is referenced through explicit load and store operations. The stack pointer and the argument pointer are not introduced by Phase 1. Array index lists are preserved. Addressing is not expanded unless required by language semantics. Accessing the address of a data object is marked with an explicit operation.

Data objects are grouped into packets, where a packet represents a group of variables with a language defined storage relationship.[1] Packets are the unit of storage allocation. Two direct memory references can reference the same storage only if they are in the same packet. For indirect references, we define a template packet, which describes the template the pointer wants to impose on memory. This is similar to the Pascal record type, or the C structure or union type. Restrictions on alias relationships (such as between FORTRAN by-reference arguments) can be associated with a template. Each packet has a known *0 mod B* alignment, i.e., *mod(address(packet),B) = 0*;

[1] Our definition of a packet is derived from the definition presented in [3].

address(packet) is of the form *s*B* for some integer *s*. We refer to *s* as a packet seed. Packet seeds are useful when determining whether two references can refer to the same bank.

IL-1 operations are successively lowered during optimization. The output of the optimizer is a flow graph of IL-2 operations, which correspond to machine operations, except that we have not yet assigned the functional units to perform the operations or the registers to hold their operands.

6 Front ends

The Multiflow compiler includes front ends for C [41] and ANSI FORTRAN 77 [6] with VAX/VMS extensions. Other languages (Pascal, Ada, Lisp) are supported by translators that generate C. The front ends were derived from the ATT pcc compiler suite [37, 25] by mapping the pcc intermediate representation to IL-1. IL-1 is higher-level than the pcc intermediate. We implemented tree-synthesis algorithms to recapture array semantics from the pointer arithmetic of pcc. In retrospect, it may have been easier to generate our IL-1 directly from semantic actions.

We implemented user level directives in both FORTRAN and C as structured comments. Loop unrolling directives allow the user to specify how a loop should be unrolled. Inline directives allow functions to be inlined. Memory-reference directives allow the user to assert facts about addresses. Trace-picking directives allow the user to specify branch probabilities and loop trip counts. In addition, the front end can instrument a program to count basic block executions. The instrumentation is saved in a data base which can be read back on subsequent compilations. This information is used to guide the trace picker.

We support the Berkeley Unix run-time environment. Care was given to structure layout rules to ease porting from VAX and 68000 base systems. Despite the unusual architecture of the Trace, it was easier to port from BSD VAX or 68000 systems to the Trace than to many contemporary RISC-based systems.

The FORTRAN IO library distributed with the ATT compilers is not adequate for high performance computing. A new high performance FORTRAN IO library was written and integrated with the FORTRAN front end.

7 The optimizer

The goal of the optimizer is to reduce the amount of computation the program will perform at run time and to increase the amount of parallelism for the trace scheduler to exploit. Computation is reduced by removing redundant operations or rewriting expensive ones; this is the goal of most optimizers. Parallelism is increased by removing unnecessary control and data dependencies in the program and by unrolling loops to expose parallelism across loop iterations. The Multiflow compiler accomplished these goals with standard Dragon-book style optimization technology [1] enhanced with a powerful memory-reference disambiguator.

7.1 Organization

The optimizer is designed as a set of independent, cooperating optimizations that share a common set of data structures and analysis routines. The analysis routines compute control flow (dominators, loops) and data flow (reaching defs and uses, live variables, reaching copies). In addition, the disambiguator computes symbolic derivations of address expressions. Each optimization records what analysis information it needs, and what information it destroys. The order of optimizations is quite flexible (in fact it is controlled by a small interpreter). The order used for full optimization is given in Figure 7-1.

Basic optimizations:
Expand entries and returns to IL2
Find register variables
Expand memory ops to IL2
Eliminate common subexpressions
Propagate copies
Remove dead code
Rename temps
Transform ifs into selects

Prepare for first loop unrolling:
Generate automatic assertions
Move loop invariants
Find register memory references
Eliminate common subexpressions
Transform ifs into selects
Find register expressions
Remove dead code

First unroll:
Unroll and optimize loops
Rename temps
Propagate copies
Simplify induction variables
Eliminate common subexpressions
Propagate copies
Remove dead code

Second unroll:
Unroll and optimize loops
Rename temps

Prepare for Phase3:
Expand calls into IL2
Walk graph and allocate storage
Analyze for dead code removal
Remove assertions
Remove dead code
Expand remaining IL1 ops to IL2
Propagate copies
Remove dead code
Rename temporaries

Figure 7-1: Optimizations invoked by Multiflow compiler

7.2 Control dependence

By control dependence, we mean barriers to instruction-level parallelism that are caused by the control flow of the program. Control dependence is introduced by conditional branches, function calls and loops.

Control dependence introduced by conditional branches is directly addressed by trace scheduling, which performs speculative execution above a branch; this works best for highly predictable branches. In addition, the 300 series supports predicated execution with a three input select operation, and for *if-then* branches, the compiler generates a select if the *then* clause is not too expensive. For example, the compiler maps

```
                                t = Y + Z
IF (cond) X = Y + Z   ──────►   X = cond ? t : X
```

This removes many short forward branches and makes trace scheduling much more effective. The 500 series includes predicated stores and predicated floating point operations.

Function calls are addressed by inlining. The compiler will inline small leaf procedures. In addition to removing the overhead of the procedure call, this increases the size of traces and provides more opportunities for parallelism. More aggressive inlining can be performed by the user with command line arguments or directives.

Loops are addressed by unrolling. Loops are unrolled by copying the loop body including the exit test (see Figure 7-2). Most compilers remove exit tests when unrolling a loop by *preconditioning*. To unroll by *n*, a pre-loop is added to handle *trip-count mod n* iterations, and the loop processes *n* loop bodies at a time. Unlike most machines, the Trace has a large amount of branch resource, and there is no advantage to removing exit branches. By leaving the branches in, we eliminate the short-trip count penalty caused by preconditioning. In addition, loops with data dependent loop exits, which cannot be preconditioned (e.g., *while* loops), can also be unrolled and optimized across iterations. For loops with constant trip-count, all but one exit test can be removed, and small loops can be unrolled completely.

Preconditioning is a separate optimization; it is only used to support pair mode (a hardware feature that allowed vectors of length 2) or the math intrinsics (as explained below). In practice, when preconditioning, we actually postcondition the loop, for this makes it possible to make assertions about alignment on loop entry. The 500 series has less branch bandwidth, and postconditioning is required to achieve peak performance.

```
loop               unrolled by 4        pre-cond by 4        post-cond by 4

L:  if-- goto E    L:  if-- goto E          if-- goto L      L:  if-- goto X
    body               body                 body                 body
    goto L             if-- goto E          if-- goto L          body
E:                     body                 body                 body
                       if-- goto E          if-- goto L          body
                       body                 body                 goto L
                       if-- goto E      L:  if-- goto E      X:  if-- goto E
                       body                 body                 body
                       goto L               body                 if--goto E
                   E:                       body                 body
                                            body                 if-- goto E
                                            goto L               body
                                        E:                   E:
```

Figure 7-2: Styles of loop unrolling

Loops are unrolled heavily. A loop must be unrolled enough to expose sufficient parallelism within the loop body to enable the instruction scheduler to fully utilize the machine. Unrolling is controlled by a set of heuristics which measure the number of operations in the loop, the number of internal branches, and the number of function calls. A loop is unrolled until either the desired unroll amount is reached or one of the heuristic limits is exceeded. For example, if the target unroll amount is 8 and the operation limit is 64, we will unroll a loop body with 8 or less operations 8 times. If the body contains 9 operations, it will be unrolled 7 times. If it contained 30 operations, it would only be unrolled twice, and if it contained more the an 32 operations, it would not be unrolled at all.

The default unrolling in FORTRAN for a Trace 14-wide is 16; at the highest level of optimization, we unroll by 96. The corresponding limits are given below.

optimization level	*1*	*2*	*3*	*4*
total unrolling	16	32	64	96
first unrolling	16	32	32	32
second unrolling	—	—	2	3
max operations	128	256	512	768
max branches	0	2	4	8
max calls	0	0	0	0

Figure 7-3: FORTRAN unrolling for Trace 14/300

The limits were determined experimentally on a small set of benchmarks, and worked well in practice. The compiler includes another set of heuristics that controlled the unrolling based on the utilization of machine resources by the loop body, but in practice it did not outperform the simpler scheme.

Loop unrolling is done in two steps: a first unrolling where the loop body is copied, and a second unrolling where the first-unrolled bodies are copied as a unit. The two step approach allows us to keep the first unrollings relatively small and still unroll heavily. The ordering of optimizations is heavily influenced by our two phase loop unrolling design (see figure 7-1). Most standard optimizations are performed before the first unrolling; this permits us to have an accurate estimate of loops size for our heuristics. After the first unrolling, the induction variables are rewritten, and optimization is performed across the unrolled bodies (achieving the effect of predictive commoning [55]). Both induction variable simplification and commoning across loop bodies may increase register pressure. By keeping the first unrolling small, we prevent the register pressure in the loop from exceeding the available registers. We also keep the constant displacements introduced by the induction variable simplification small, so that the immediate resource in each instruction is not oversubscribed.

7.3 Data dependence

The compiler's strategy for eliminating unnecessary data dependence is to map as many variables as possible to temporaries (virtual registers) where they can be more easily analyzed and optimized. The major optimizations for removing data dependence are copy propagation and temporary renaming. The compiler also rewrites reduction loops to minimize recurrences between loop iterations.

7.3.1 Allocating variables to temporaries

The optimizer attempts to place each variable and value in a temporary, unless the semantics of the program requires it to be in memory; this includes aggregates (i.e., structures) except for arrays. The instruction scheduler will then spill values to memory as needed when allocating registers. This eliminates as many references as possible from the program and permits the instruction scheduler to place the necessary ones (spills and restores) at optimal points in the program. Minimizing memory references is important on Trace systems, for they have no data cache.

The attempt to place values in temporaries is done over the entire program and then for each loop. This is first done by a straightforward analysis of the IL-1 program presented to Phase 2, which places unaliased scalars and aggregates in registers on a routine-wide and loop-by-loop basis. Later optimizations use the disambiguator to place loop invariant array references and indirect references in registers.

To enhance the capabilities of the disambiguator, the compiler will assert the condition tested by a conditional branch after the branch. In particular, the compiler will assert that the induction variable of the loop is within the loop bounds. For example, in the following, the compiler asserts that `i >= k+1` and `i <= n` in the inner loop. This permits the disambiguator to know that `a(k,j)` cannot reference the same location as `a(i,j)`, and thus can safely be moved out of the loop.

```
do j = k+1, n
    do i = k+1,n
            a(i,j) = a(i,j) + a(k,j)* a(i,k)
    end do
end do
```

Two classes of loop-varying array and indirect references are maintained in registers, achieving an effect similar to scalar replacement [9]. The first, called *register memory detection*, detected address invariant references that were unaliased in a loop (such as c(i,j) below).

```
do j = 1,n
    do i = 1,n
        do k = 1,n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
    end do
end do
```

The second, called *register expression detection*, detects recurrences of the form below and maintains the corresponding array element in a temporary. (Actually, f(v) can be any expression tree.)

```
original loop body          transformed body
...                         ....
a := f(x)                   a := f(x)
...                         ...
b := f(v)                   b : = t
...                         ...
v := c ? v : x              v := c ? v : x
...                         t := c ? t : a

                            where t is initialized in the
                            loop pre-header as t:= f(v)
```

Both of these transformations enable some loops to be rewritten as reductions, as discussed below.

7.3.2 Removing data dependence

After memory references are removed, the compiler attempts to remove data dependence with temporary renaming and copy propagation. Temporary renaming splits temporaries into disjoint use-def webs. Copy propagation propagates a copied value to its use. These optimizations are invoked multiple times. Together, they are used to prepare for induction variable simplification on an unrolled loop.

Initially, an unrolled loop has very little parallelism between unrollings, because of the dependencies on induction variables; see the dependencies on i in Figure 7-4. By renaming i ,and then copy propagating, we can remove those dependencies. By propagating integer additions of a constant, we can express the i0, i1, i2 as secondary induction variables of the primary induction variable i , and we have set the loop up for successful induction variable simplification.

```
unrolled                  renamed                   copy propagated

 i = 1                     i = 1                     i = 1
L1:                       L1:                       L1:
 if (i > n) goto exit      if (i > n) goto exit      if (i > n) goto exit
 ld a(i)                   ld a(i)                   ld a(i)
 st a(i)                   st a(i)                   st a(i)
 i = i + 1                 i1 = i + 1                i1 = i + 1

 if (i > n) goto exit      if (i1 > n) goto exit     if (i1 > n) goto exit
 ld a(i)                   ld a(i1)                  ld a(i1)
 st a(i)                   st a(i1)                  st a(i1)
 i = i + 1                 i2 = i1 + 1               i2 = i + 2

 if (i > n) goto exit      if (i2 > n) goto exit     if (i2 > n) goto exit
 ld a(i)                   ld a(i2)                  ld a(i2)
 st a(i)                   st a(i2)                  st a(i2)
 i = i + 1                 i3 = i2 + 1               i3 = i + 3

 if (i > n) goto exit      if (i3 > n) goto exit     if (i3 > n) goto exit
 ld a(i)                   ld a(i3)                  ld a(i3)
 st a(i)                   st a(i3)                  st a(i3)
 i = i + 1                 i = i3 + 1                i = i + 4
 goto L1                   goto L1                   goto L1
```

Figure 7-4: Removing dependencies on an induction variable

When a variable is live on loop exit, unrolling a loop can create a situation where multiple definitions of the same variable reach the same use. The various definitions cannot be independently renamed, and the definitions will not be able to be issued speculatively above the preceding loop exit. To remove this problem, we insert a self assignment at each loop exit for every variable that is both defined in the loop and live on the exit. This permits the variables to be renamed. We pay an extra move on exit from the loop to enable parallelism between loop iterations. See Figure 7-5.

```
unrolled                  self assignments          renamed

  i = 1                    i = 1                     i = 1
 L1:                      L1:                       L1:
  if (i > n)               if (i > n)                if (i > n)
    { goto exit }            { x = x; goto exit }      { x = x4; goto exit }
  x = ld a(i)              x = ld a(i)               x1 = ld a(i)
  i = i + 1                i = i + 1                 i1 = i + 1

  if (i > n)               if (i > n)                if (i1 > n)
    { goto exit }            { x = x; goto exit }      { x = x1 ; goto exit }
  x = ld a(i)              x = ld a(i)               x2 = ld a(i1)
  i = i + 1                i = i + 1                 i2 = i + 2
```

```
if (i > n)                  if (i > n)                  if (i2 > n)
  { goto exit }               { x = x; goto exit }        { x = x2 ; goto exit }
x = ld a(i)                 x = ld a(i)                 x3 = ld a(i2)
i = i + 1                   i = i + 1                   i3 = i + 3

if (i > n)                  if (i > n)                  if (i3 > n)
  { goto exit }               { x = x; goto exit }        { x = x3 ; goto exit }
x = ld a(i)                 x = ld a(i)                 x4 = ld a(i3)
i = i + 1                   i = i + 1                   i = i + 4
goto L1                     goto L1                     goto L1
```

Figure 7-5: Removing dependencies on a variable live on loop exit

7.3.3 Reductions

The compiler will rewrite a loop containing a reduction, where a reduction is a recurrence of the form *a = a op fn(i)*, where *op* is commutative and associative. The reduction will be rewritten into *n* interleaved reductions, and the *n* results will be combined on loop exit. For example, a dot product would be transformed:

```
                                        t1 = 0.0
                                        t2 = 0.0
                                        t3 = 0.0
                                        t4 = 0.0
    x = 0.0                             x = 0.0
    i = 1                               i = 1
l1:                                 l1:
    if (i>n) goto exit                  if (i > n) goto exit
    x = x + y(i)*z(i)                   t1 = t1 + y(i)*z(i)
    goto l1                             i = i + 1
    exit:                               if (i > n) goto exit
                                        t2 = t2 + y(i)*z(i)
                                        i = i + 1
                                        if (i > n) goto exit
                                        t3 = t3 + y(i)*z(i)
                                        i = i + 1
                                        if (i > n) goto exit
                                        t4 = t4 + y(i)*z(i)
                                        i = i + 1
                                        goto l1
                                    exit:
                                        x = t1 + t2 + t3 + t4
```

The interleave amount is determined by the number of the reduced operations that can be simultaneously active in the machine (pipeline latency times number of functional units). The dot product above would be interleaved by 4 on the Trace 14/300. To give the instruction scheduler more freedom, the optimizer will unroll the loop at least twice the interleave amount, and insert self-assignments on the loop exits and rename as described above.

For some floating point operations, the reassociation performed by reductions is (strictly speaking) illegal according to FORTRAN semantics, but in almost all cases when performing such operations in loops, the order is unimportant. Also, we provide a switch to prevent this optimization in cases where the result could differ (e.g., floating point addition/subtraction), without inhibiting other cases (e.g., min/max, integer operations, etc.)

The compiler will also detect *parasite* reductions, where the desired value is maintained in parallel with the reduced value. For example, idamax, which returns the index of the maximum in a vector, is recognized as the parasite of a max reduction and interleaved. See Figure 7-6.

```
idamax = 1
dmax = dabs(dx(1))
do 30 i = 2,n
    if(dabs(dx(i)).le.dmax) goto 30
    idamax = i
    dmax = dabs(dx(i))
30 continue
```

```
     idamax = 1
    dmax = dabs(dx(1))
     i = 2
L1:
    if (i > n) goto exit
    a = dabs(dx(i))
    b = a .gt.  dmax
    idamax = b ? i : idamax
    dmax = b ? a : dmax
    i = i + 1
    goto L1
exit:
```

```
    idamax0 = 1
    idamax1 = idamax0
    dmax0 = abs(dx(1))
    dmax1 = dmax0
    i = 2
L1:
    if (i > n) goto exit
    a0 = dabs(dx(i))
    b0 = a0 .gt.  dmax0
    idamax0 = b0 ? i : idamax0
    dmax0 = b0 ? a0 : dmax0
    i1 = i + 1

    if (i1 > n) goto exit
    a1 = dabs(dx(i1))
    b1 = a1 .gt.  dmax1
    idamax1 = b1 ? i1 : idamax1
    dmax1 = b1 ? a1 : dmax1
    i = i + 2
    goto L1

exit:
    t0 = dmax0 .gt.  dmax1
    t1 = dmax0 .eq. dmax1
    t2 = idamax0 .lt. idamaxa1
    t3 = t1 and t2
    b = t0 or t3
    dmax = b ? dmax0 : dmax1
    idamax = b ? idamax0 : idamax1
```

Figure 7-6: idamax reduction

7.4 Reducing computation

The basic optimizations used to reduce computation are standard, though they have some interesting features. Both loop invariant motion and common subexpression elimination (CSE) use the disambiguator to detect location conflicts between memory references; this allows these optimizations to deal effectively with array references and indirect references. As discussed above, induction variable simplification is performed on heavily unrolled loops. In addition to strength reducing the address expressions, it must minimize the number of live registers required across the loop bodies (either induction variables or loop invariants) and make the best use of the constants in each instruction. Common subexpression elimination is performed on extended basic blocks. When performed after loop unrolling, it can detect redundant computations across loop bodies. For example, in Livermore kernel 7, only 3 new loads are required each iteration; the compiler detects this and the corresponding redundant flops as well.

```
        do 7 l= 1,loop
        do 7 k= 1,n
                x(k)= u(k ) + r*( z(k ) + r*y(k )) +
     .          t*( u(k+3) + r*( u(k+2) + r*u(k+1)) +
     .          t*( u(k+6) + r*( u(k+5) + r*u(k+4))))
    7   continue
```

Our common subexpression algorithm performs local cleanups in addition to commoning. Within an extended basic block it performs constant folding, copy propagation, operation simplification, and dead code removal. These are optimizations that are easy to perform once a data dependency graph for the extended basic block is built. Copy propagation and dead code removal are also performed globally in separate optimizations.

CSE also detects and optimizes calls to the math intrinsics. The core math intrinsics (atan, atan2, cos, cos_sin, exp, log, pow, sin) are implemented in *n-at-a-time* versions, where n is one of 1, 2, 4, 8, or 16. N arguments are passed to the intrinsics and n values returned. CSE will look for multiple calls to an intrinsic, and substitute a call to an *n-at-a-time* function. The loop unroller postconditions loops that contain an intrinsic (we do not allow an intrinsic to be called speculatively) producing the equivalent of vector intrinsics.

```
before                          after

x1 = sin(y1)                    (x1,x2) = sin_2(y1,y2)
x2 = sin(y2)
z1 = cos(y)                     (z1,z2) = cos_sin(y)
z2 = sin(y)
do i = 1,n                      do i = 1,8*(n/8),8
        x(i) = sin(y(i))                (y1,...,y8) = y(i:i+7)
end do                                  (x1,...,x8) = sin(y1,...,y8)
                                        x(i:i+7) = (x1,...,x8)
                                end do

                                <postloop not shown>
```

Our approach can also be applied to programs that do not have a structured use of intrinsics. For example, in Figure 7-7 below, 13 calls to sin and cos are replaced with 5 calls (1 to cos, 1 to cos_2, 1 to sin_2, 1 to sin_4, and 1 to cos_sin_2).

```
                    tht0 = thtp(me) - tht1zr*(ee - 0.75*rr)
     1              -tdel3*(bet(me)+wep(me,1))
     2              +talf1*(del(me)+vep(me,1))
     3              +gimbl(me)/dsin(psigim(me))*
     4              (tht2(me)*dcos(psii+psigim(me))
     5              tht1(me)*dsin(psii+psigim(me)))
                    dtht0 = dthtp(me) -tdel3*(dbet(me)+dwep(me,1))
     1              +talf1*(ddel(me)+dvep(me,1))
     2              +gimbl(me)/dsin(psigim(me))*omeg(me)*
     3              (-tht2(me)*dsin(psii+psigim(me))
     4              tht1(me)*dcos(psii+psigim(me)))
     5              +gimbl(me)/dsin(psigim(me))*
     6              (dtht2(me)*dcos(psii+psigim(me))
     7              dtht1(me)*dsin(psii+psigim(me)))
                    stht0 = dsin(tht0)
                    ctht0 = dcos(tht0)
                    s2tht0 = dsin(2.0*tht0)
                    c2tht0 = dcos(2.0*tht0)
                    kbett = kbet(me)
                    kdell = kdel(me)
                    clds = cldstr(me)
                    gj0 = gj(me,1)
```

Figure 7-7: Non-vector candidate for n-at-a-time intrinsics.

Dead code removal is an iterative mark-and-sweep algorithm that removes operations that do not contribute to the final result of the program; it also deletes dead control flow (e.g., conditional branches which always go one way due to a constant condition and branches whose true and false successors are the same). Dead code removal is most profitable in conjunction with constant propagation and procedure inlining. We use it additionally to clean up information maintained for our disambiguator. To enhance memory-reference analysis, we maintain subscript information until the end of phase 2. Array references are lowered twice, first to a form that contains both base-displacement addressing and a subscript list, and finally to a form that contains only the base-displacement. Induction variable simplification inserts special deriv_assign operations that relate the new induction variables to the original subscript expressions. After the second lowering, the code that maintained the subscript expressions is dead, and can be removed. Similarly, assertions about addresses are maintained in the flow graph until the end of phase 2. In the final code expansions, the assertions are removed, and dead code removal eliminates the now-dead code that maintained the values asserted.

8 The back end

The output of Phase 2 is a flow graph of IL-2 operations lowered to machine level. Phase 3 performs functional-unit assignment, instruction scheduling, and register allocation. The work is divided into four modules: the trace scheduler, which manages the flow graph and assures inter-trace correctness; the instruction scheduler, which schedules each trace and assures intra-trace correctness; the machine model, which provides a detailed description of the machine resources; and a disambiguator, which performs memory-reference analysis.

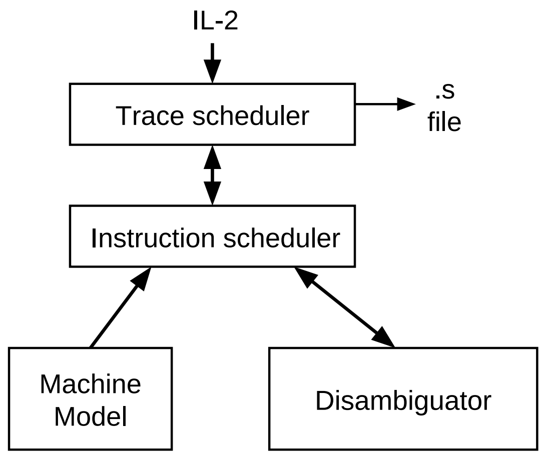


Figure 8-1: Structure of Phase 3

9 The Trace Scheduler

9.1 1. The algorithm

The trace scheduler performs the following steps:

A. Copy the flow graph produced by Phase 2 and estimate how many times each operation will be executed.

B. Perform the following loop until the entire flow graph has been scheduled.

   1. Using the execution estimates as a guide, pick a trace (a sequence of basic blocks) from the flow graph.
   2. Pass the trace to the instruction scheduler. The instruction scheduler schedules the trace and returns a machine language schedule.
   3. Replace the trace with the schedule in the flow graph and, if necessary, add copies of operations to compensate for code motions past basic block boundaries.

C. Emit the schedules in depth first order.

We describe these steps in detail below, except for the instruction scheduler, which is described in section 10.

9.1.1 Expect

Execution estimates (called *expect*) are calculated from loop trip count frequencies and the probabilities of conditional branches. We use the following rules:

```
If operation O is an entry to a routine
      expect(O) = 1.0/(number_of_entries)

If operation O is not the head of a loop
      expect(O) = Sum_P( prob_i * expect_i )
      where
                      Sum_P       is the sum over all preds of O, such that
                                  the pred is not a loop entrance.
                      prob_i      is the probability of traversing the edge
                                  from pred_i to this op.
                      expect_i    is the expect of pred_i.

If operation O is a loop head:
      expect(O) = iter_count * Sum_LE( prob_i * expect_i )
      where
                      iter_count  is the expected iteration count for the loop.
                      Sum_LE      is the sum over all loop entrances to the loop.
                      prob_i      is the probability of traversing the edge from
                                  loop_entrance_i to O
                      expect_i    is the expect of loop_entrance_i.
```

A loop entrance is an operation not in the loop which has a successor in the loop. We insert a pseudo op to ensure that each loop entrance has only one successor. When calculating *expect*, we handle irreducible loops by treating them as if they were reducible. In the formulas above, if an operation is not a loop head, we ignore all loop entrances that are predecessors, and if an operation is a loop head, we treat all loop entrances to that loop as predecessors.

The probabilities at conditional branches are obtained from either a database collected during previous executions of the program, a user directive, or the simple heuristic that a conditional branch is 50-50 unless it is a loop exit. The probability of an exit from a loop is set to 1/iteration_count, where iteration_count is the expected iteration count for that loop. The expected iteration count for a loop is assumed to be 100.

9.1.2 Trace Picking

Traces are picked by first selecting the yet-to-be-scheduled operation with the highest *expect*. This operation becomes the seed for the trace, and the trace is grown forward (in the direction of the flow graph) and then backward. We grow the trace by picking a successor (or predecessor if moving backward) that satisfies the current trace-picking heuristic. If no successor (or predecessor) satisfies the current heuristic, the trace ends. Traces always end when we hit an operation that is already scheduled, or an operation that is already on the trace. Also, traces never cross the backedge of a loop, as explained below. In addition, we end the trace when its length is equal to *max_trace_length*, which varies from 1024 to 2048 operations, depending on the width of the machine and the level of optimization.

The trace-picking heuristics are defined in terms of edges between operations in the flow graph. The same criteria are used to determine if an edge can be added to the trace, regardless of the direction we are growing the trace. We apply our heuristic to an edge from *pred* to *succ*. If we are growing the trace forward, *pred* is already on the trace; if backward, *succ* is already on the trace.

We implemented a large number of trace-picking heuristics, but used only the two listed below.

- *mutual most likely*. Both of the following conditions must be met:

  The edge from *pred* to *succ* has the highest probability of all exits from *pred* (i.e., if we are at *pred*, we are "most likely" to go to *succ*).

  The edge from *pred* to *succ* contributes the most *expect* to *succ* of all predecessors of *succ* (i.e., if we are at *succ*, we are "most likely" to have come from *pred*).

- *no compensation*. We want no compensation code to be required after instruction scheduling. If neither *pred* or *succ* is a rejoin or split, the edge is OK. Rejoins and splits require special attention:

  If *succ* is a rejoin (i.e., it has multiple predecessors) then end the trace; compensation would be required if *succ* is moved before *pred* in the schedule.

  If *pred* is a rejoin then the edge is OK.

  If *succ* is a split (i.e., it has multiple successors) then end the trace; compensation code would be required if *succ* is moved before *pred* in the schedule.

  If *pred* is a split then the edge is OK.

Mutual most likely is the heuristic used by default. The *no compensation* heuristic is used to avoid the creation of compensation code; it restricts traces to a variant of basic blocks. The compiler will switch into this heuristic if too much compensation code has been created; see section 9.2.5.

9.1.3 Compensation code

After picking a trace the trace scheduler passes it to the instruction scheduler. The instruction scheduler returns a schedule. The trace scheduler must examine the code motions the instruction scheduler performed to see if any operations must be copied. Splits (branches out of a trace) and joins (branches into a trace) determine the basic block boundaries in the flow graph. If a copy is necessary, it will be associated with a split or a join.

To discuss compensation code, we first need to introduce some notation.[1] *Trace_position(O)* is the position of operation *O* on the trace. *First_cycle(O)* is the position of the first cycle of operation *O* in the schedule. *Last_cycle(O)* is the the position of the last cycle of operation *O* in the schedule.

A split operation, or split, is an operation with more than one successor (for example, a conditional branch operation, or an indirect branch). When the instruction scheduler moves an operation below a split on the schedule, the trace scheduler must copy this operation on the off-trace edge. For example, A is copied in Figure 9-1. (In our examples, a machine instruction is denoted as [op; op;...])

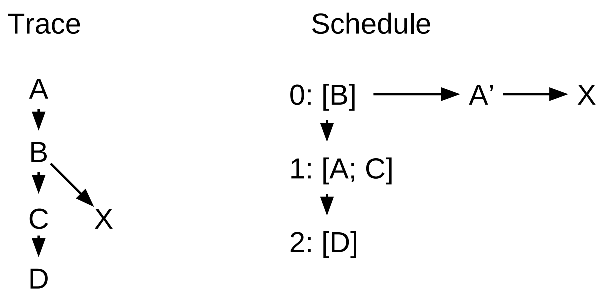


Figure 9-1: Split compensation code

Each split *S* will have a tuple of compensation copies $(s_1,...,s_m)$ where $trace\_position(s_i) < trace\_position(S)$ and $first\_cycle(s_i) > last\_cycle(S)$. The copies are placed on the split edge in source order (i.e., the order in which the operations appeared on the trace.)

[1] Our notation is adapted from Ellis [23].

A joined operation, or join, is an operation on the trace that is the target of a branch operation. Whenever the instruction scheduler moves an operation above a join, the trace scheduler must insert a copy of the operation on the off-trace joining edge. For example, C and D are copied in Figure 9-2.

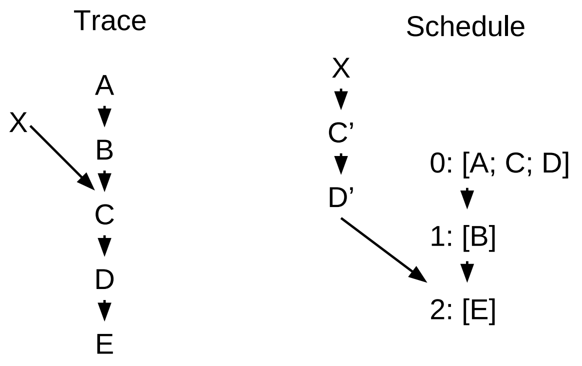


Figure 9-2: Join compensation code

Before join compensation code is generated the trace scheduler must determine where to rejoin the schedule. The rejoin cycle R of a join to trace position J must satisfy the constraint that all operations O that appeared prior to J in the trace (that is, *trace_position* $(O) < J$) must be complete before cycle R on the schedule. For example, the join to C on the trace above is moved to instruction 2 on the schedule.

Once the rejoin instruction is determined, the trace scheduler can determine the join compensation code. Each join to trace position J with rejoin cycle R will have a tuple of compensation copies $(j_1,...,j_m)$ where *trace_position*(j_i) $>= J$ and *last_cycle*$(j_i) < R$. They are placed on the join edge in source order.

If a split is copied onto a rejoin edge, additional copies are required. Consider the join to B in Figure 9-3; the rejoin instruction is 4. A copy of C is needed on the off-trace edge of D"; otherwise the path from X to Y will be incorrect. In general, all operations that are between the join and the split on the trace that are not above the rejoin instruction in the schedule must be copied on to the off-trace edge of the copied split. Each split S_J copied on the join to trace position J with rejoin cycle R will have a tuple of compensation code $(s_{j1},...,s_{jm})$, where *trace_position*$(s_{ji})>=J$, *trace_position*$(s_{ji})<$*trace_position*(S_J), and *last_cycle*$(s_{ji})>=R$.

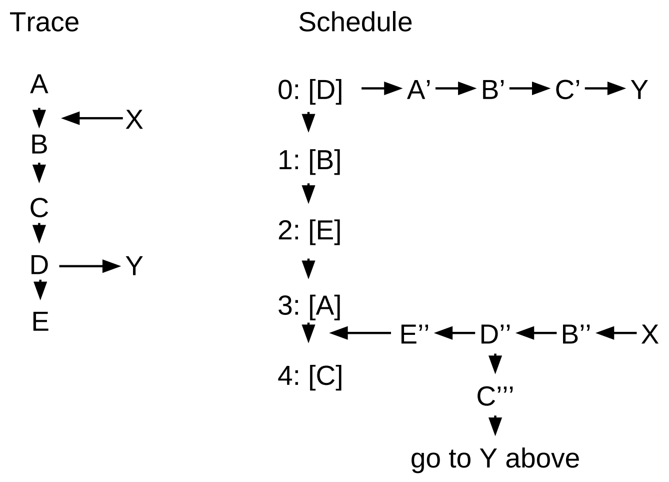


Figure 9-3: Split copied onto rejoin edge

9.1.4 Speculative code motion

Speculative execution, moving an operation from below a split on the trace to above a split on the schedule, does not produce compensation code. This is the most common code motion in the Multiflow compiler. High priority operations from late in the trace are moved above splits and scheduled early in the trace. The instruction scheduler will perform such a move only if it is safe: an operation cannot move above a split if it writes memory or if it sets a variable that is live on the off-trace path. The hardware provides support for suppressing or deferring the exceptions generated by speculative operations.

Although it has been suggested that the compiler could insert code into the off-trace path to undo any effects of a speculative operation, this is not done. For simple register operations, such as incrementing a counter, the operation is best "un-done" by targeting it to a register that is not live on the off-trace path. For operations that write memory or transfer control, the complexity of un-doing them outweighs the potential benefits.

9.1.5 Emitting the schedules

Since traces are selected along the most frequently traveled paths in the program, trace scheduling gives the effect of profile-guided code positioning [59]. When the entire flow graph has been scheduled, the graph has been transformed into a graph of schedules. The trace scheduler then does a depth first walk from the entries, emitting the schedules. To avoid unnecessary branches between schedules in the emitted code, we always visit the fall-through successor of a schedule first when performing the depth first walk.

On the Trace, the immediate resource in an instruction is shared across many operations, and obtaining peak performance in an unrolled loop requires using short branches for loop exits in order to free up immediate space the for memory offsets. However, VLIW instructions are very large (128 fully packed instructions are 16Kb on a 28/300); loop exits must be positioned shortly after the loop body, to keep them within reach of the short branch offset. When we encounter a schedule that begins a loop, we change from a depth-first to breadth-first walk of the graph, so that we can collect and position the loop exits. When we exit the loop, we resume our depth-first walk.

9.2 Restrictions to trace scheduling

The Multiflow compiler places a number of restrictions on trace scheduling in order to limit the amount of compensation code and to make the problem of engineering the compiler more tractable.

9.2.1 Loops

A trace does not cross a backedge of a loop. This restriction is partly historical; Fisher did not consider picking traces across a back edge in his first definition of trace scheduling [26]. But it has a number of advantages. It simplifies the instruction scheduler and trace scheduler, for they do not have to deal with the complexities of scheduling multiple iterations of a loop simultaneously; the trace for a loop body can be treated identically to a trace from a sequence of loop free code. It also simplifies the memory-reference analysis, as we discuss in section 13. In addition, Nicolau relies on this restriction in his proof that trace scheduling terminates [54].

In practice, this restriction does not impact performance very much. The most popular algorithm for scheduling across the back edge of a loop is software pipelining [60, 44, 45, 21]. A software-pipelined loop scheduler could be integrated into the Multiflow compiler in a straightforward manner. It may improve the performance of vector kernels, where we already perform excellently, but it would not address the weaker points of the compiler.

The attraction of software pipelining is that it is an algorithm for finding an optimal schedule for a loop kernel. An unrolling strategy that does not cross the backedge of the loop must always start up the loop bodies at the head of the loop and wind them down at the bottom; these portions of the schedule will not use the machine optimally. A software pipelined schedule can move the wind-up and wind-down portions of the schedule outside of the loop. See Figure 9-4.

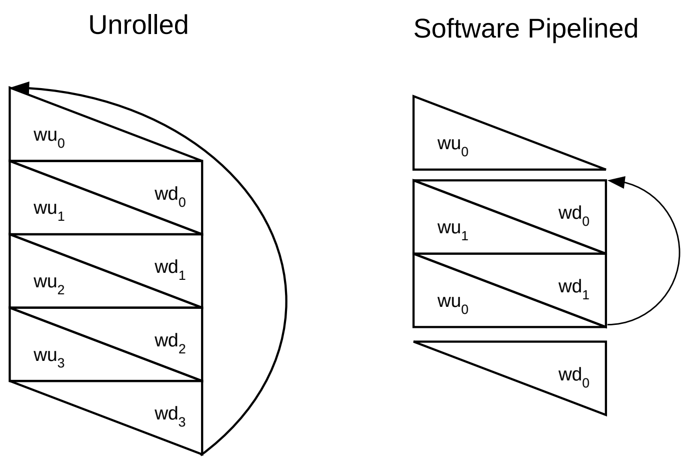


Figure 9-4: Loop unrolling and software pipelining

The Multiflow compiler will unroll loops heavily (up to 96 times on the 14/300) to amortize the loop wind-up and wind-down over many bodies, mitigating their performance effect. The unrolling does increase code size, but due to the large instruction cache on the Multiflow machines this does not affect performance significantly.

For low trip count loops, software pipelining has no advantage. The wind-up and wind-down dominate the execution time of the loop for both software pipelining and simple unrolling. Moreover, software pipeline algorithms require a pre-loop [44] unless special hardware support is supplied [61,21]; the overhead of the pre-loop must be amortized over the number of iterations spent in the software pipelined kernel. The Multiflow unrolling strategy does not require a pre-loop. However, as suggested in [68], software pipelining can be extended to use speculative execution and not require a pre-loop.

Very low trip count loops are best served by peeling off the first few iterations, so that a trace can be picked which bypasses the loop entirely. This permits the peeled iterations to be scheduled with the code preceding and following the loop. Testing for the zero trip case is always a performance advantage; peeling off additional iterations can be guided by feed-back from previous executions of the program. Unfortunately, the Multiflow compiler does not implement this optimization.

9.2.2 Splits

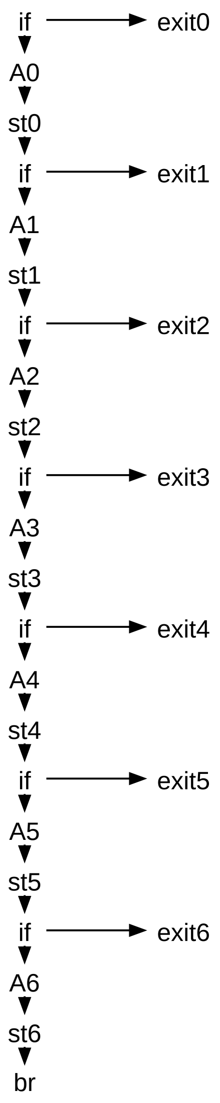
9.2.2.1 Controlling split compensation

To limit split compensation, the Multiflow trace scheduler requires all operations that precede a split on the trace to precede the split on the schedule, except for stores. Thus only store operations appear in split compensation code and the amount of split compensation code in a program is very small.

This restriction limits the parallelism available to the scheduler, but has a small effect on performance. An intuitive explanation is that executing a split early does not typically speed the execution of the on-trace path; it only speeds the off-trace path. The Multiflow compiler achieves its speed-ups when it predicts the on-trace path correctly. The scheduler attempts to schedule a split as soon as all of its predecessors are scheduled, so that control leaves the trace as early as it would with a conventional basic block compiler. The off-trace path is not penalized by trace scheduling, though it may not be sped up.

We permit stores to move below splits to avoid a serialization in the schedule; stores are never permitted to move above splits. Consider an alternating sequence of branches and stores, as in Figure 9-5. If we required stores to complete before any preceding branch, at most one store and one branch could be scheduled in one instruction. The Multiflow Trace 14/300 can issue 4 stores and 2 branches per instruction. By allowing the stores to move below splits, we gain parallelism on-trace in exchange for a small amount of split compensation code. See Figure 9-5 (b).[1] Note that an unrolled vector loop will contain a sequence of branches and stores intermixed with other operations.

Trace



(a) Schedule with store-branch constraint

```
0: [ A0; A1;          if ➤ exit0]
1: [ A2; A3; st0;     if ➤ exit1]
2: [ A3; A5; st1;     if ➤ exit2]
3: [ A6;     st2;     if ➤ exit3]
4: [         st3;     if ➤ exit4]
5: [         st4;     if ➤ exit5]
6: [         st5;     if ➤ exit6]
7: [         st6;   br]
```

(b) Schedule without store-branch constraint

```
0: [ A0; A1;          if ➤ exit0]
1: [ A2; A3; st0;     if ➤ exit1; if ]  ➤  st1’; goto exit2
2: [ A4; A5; st1; st2;   if ➤ exit3; if ]  ➤  st3’; goto exit4
3: [ A6;     st3; st4;   if ➤ exit5; if ]  ➤  st5’; goto exit6
4: [         st6; br]
```

Figure 9-5: Scheduling with and without a store-branch constraint

9.2.2.2 Source-order splits

By constraining a split by all of its trace predecessors (except for stores), splits are scheduled in source order. This has two important consequences. First, source-order splits restrict compensation code as required by Nicolau in his proof that trace scheduling terminates [54], though this is a stronger restriction than required. Second, source-order splits ensure that all paths created by the compensation code are subsets of paths (possibly

[1] A compiler for a machine that can issue at most one branch or store per cycle need not move stores below splits.

re-arranged) in the flow graph before trace scheduling. This fact is relied upon by our memory-reference analyzer. The proof of this fact requires a case-by-case analysis following Nicolau [54] and is beyond the scope of this paper. Figure 9-6 shows how rearranging splits can create a potential trace that does not correspond to a flow path in the original flow graph. Consider the compensation code for the rejoin w. It creates a path w-C'-D'-B''-x, which contains both x and C. In the original flow graph, C and x are never on the same path.

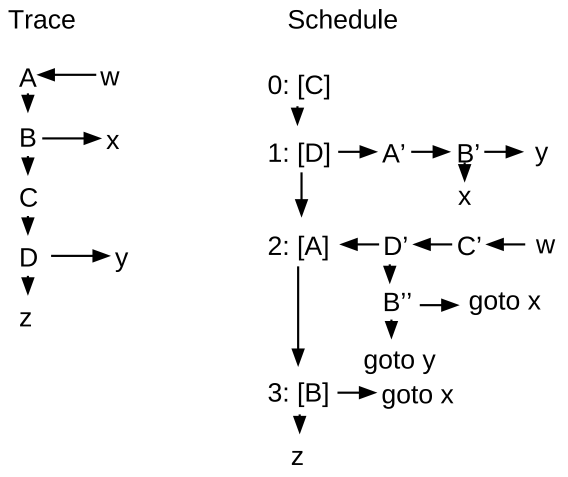


Figure 9-6: Splits scheduled out of source order

9.2.2.3 Indirect branches

Indirect branches (or igotos) are treated like other splits except when two indirect branches target the same label. In this case, no compensation code can be generated, and no motion below the igoto is permitted. An igoto branches to a location through an address previously computed and stored in a variable. If two igotos potentially branch to the same location, it is impossible to redirect one of the igotos to a new target location without redirecting the other. In the example in Figure 9-7, tag B has two predecessors, and there is no place to insert compensation code between igoto X and tag B that is not also executed along the path from igoto Y to Tag B. Therefore no compensation code can be allowed.

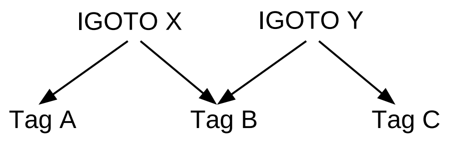


Figure 9-7: Indirect branches with a common target

By inserting multiple tag pseudo ops, the front end and the optimizer can avoid this flow graph construct for the common uses of indirect branching (C *switch* statements and FORTRAN computed goto). Only the translation of a FORTRAN assigned goto may require a flow graph where more than one igoto targets the same label.

The instruction scheduler restricts code motion to prevent an igoto from being copied onto a join edge, for this would cause the copy and the on-trace igoto to share the same targets.

9.2.3 Joins

9.2.3.1 Determining the rejoin point

As stated in Section 9.1.3, the rejoin cycle R of a join to trace position J must satisfy the constraint that all operations O that appeared prior to J in the trace must complete before cycle R on the schedule. On a machine with self-draining pipelines like the Trace, we could permit R to be the first cycle such that all operations which preceded the join on the trace have started. For example, in Figure 9-8, we could rejoin to cycle 1. This would avoid copying B, C, and D on the rejoin edge.

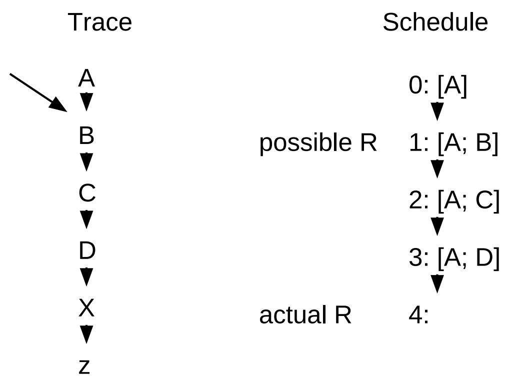


Figure 9-8: Early rejoin is beneficial to off-trace path

Unfortunately, there are schedules for which the early rejoin incurs an execution time penalty for the off-trace path. Consider the schedule in Figure 9-9. If we rejoin to cycle 1, we slow down the off-trace path, and do not avoid any rejoin compensation code.

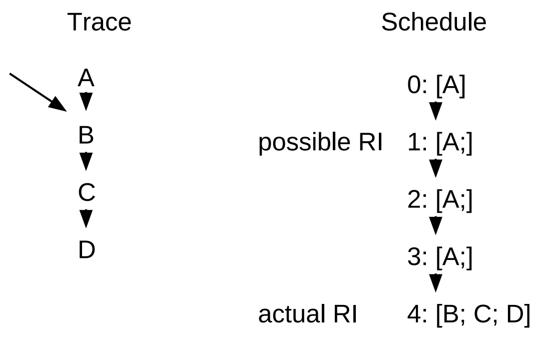


Figure 9-9: Early rejoin slows off-trace path

Note the instruction scheduler does not consider the placement of rejoins when creating a schedule; this analysis is performed after the schedule has been created. To avoid penalizing the off-trace code, we want to wait for the the pipelines of operations above the rejoin to drain to a point where they no longer constrain other operations in the schedule and then rejoin the schedule. To avoid a complicated heuristic, we always delay the rejoin until all operations that preceded the join on the trace are complete, creating the necessary rejoin copies.

9.2.3.2 Multiple joins

Sometimes a joined operation serves as the target of multiple branch operations. In this case we must decide whether there should be a separate compensation copy for each joining edge or if the joining edges should share a single instance of the compensation copies. If separate copies are inserted, it is possible that these copies can be merged into the off-trace code. Furthermore, only a single branch instruction is needed (to transfer control to cycle R). A single copy of the compensation code, on the other hand, reduces the amount of code growth. In the

Multiflow compiler, we opted to place a separate set of join copies on each joining edge. To control code growth, we do not allow code motion above an operation that has more than 4 predecessors. See Figure 9-10 .

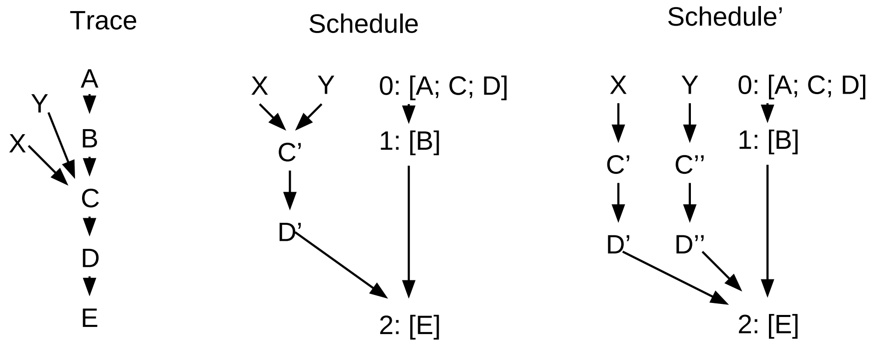


Figure 9-10: Compensation code alternatives for multiple joins

9.2.4 Copy suppression

Unrolling loops with internal branches can cause a very large amount of join compensation code. See Figure 9-11. In this loop all of the join compensation code is redundant; each potential rejoin copy has been scheduled in an instruction that dominates the rejoin. This problem, noted in [23], was the motivation for our copy suppression algorithm.

Copy suppression detects if an operation has been moved to a point in the schedule that dominates the rejoin. If it has, and the result of the operation is live at the rejoin, a copy is not necessary. The details of this algorithm and our implementation are described in [33]. With copy suppression, the compiler can profitably unroll loops with internal branches.

9.2.5 Fail safe trace scheduling shutdown

When the number of copies in a program is twice the number of original operations, the trace scheduler will no longer permit compensation code to be generated. This ensures that the program will finish compiling relatively rapidly. This is rarely activated in normal compilation, and is a fail-safe recovery from worst case copying (as might be generated for a heavily unrolled loop with internal branches, where the copy suppression algorithm is not successful.)

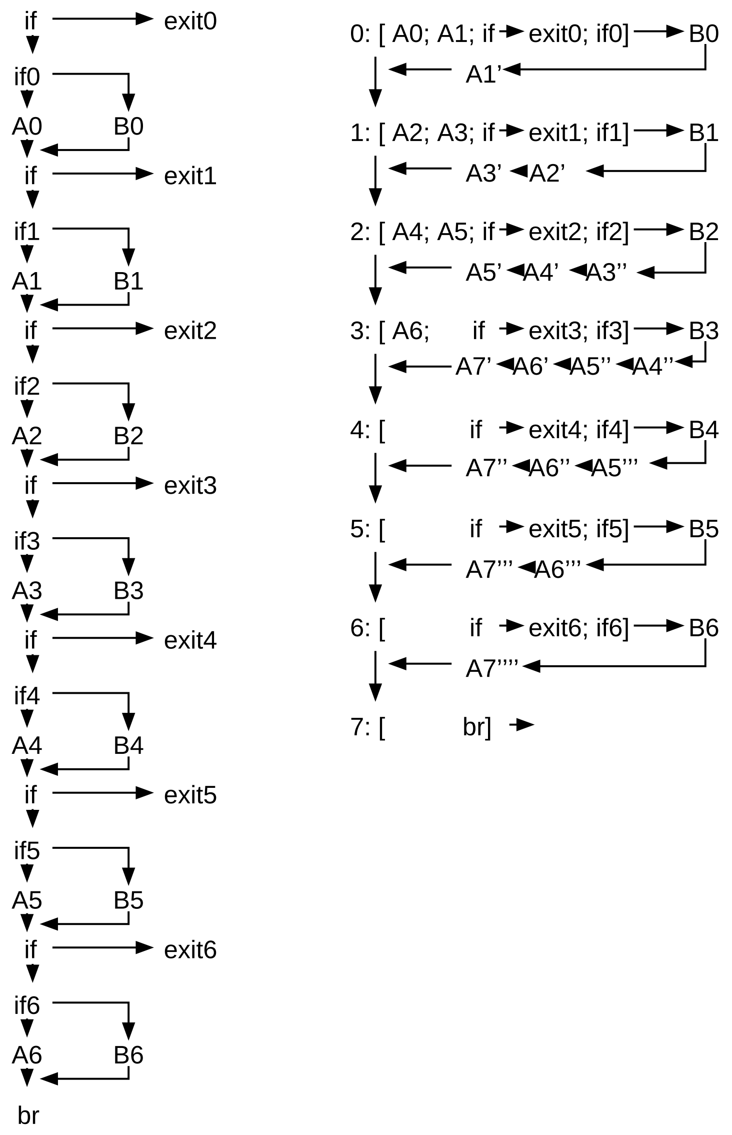


Figure 9-11: Join compensation code when scheduling a loop with internal branches.

9.3 Communication between schedules

The instruction scheduler schedules one trace at a time, but it requires information about neighboring traces for correctness and to optimize the performance on inter-trace transitions. In addition to a trace, the trace scheduler passes the following information to the instruction scheduler:

- Information about pipelines and memory references in the neighborhood of the split or join.
- Information about register bindings and live variables

9.3.1 Partial schedules

The functional-unit pipelines on the Multiflow machines use machine resources in every beat. For example, an integer ALU operation that writes a remote register file has a 3 beat pipeline.

- In beat 1, it uses an IALU, one or two register file read ports, and possibly an immediate field.
- In beat 2, it uses a bus from the IALU to the remote register file.
- In beat 3, it uses a write port in the remote register file. (The result is bypassed and can be read by the local functional unit in this beat.)

Because the hardware does not check for oversubscription of resources, the compiler must precisely model the resource utilization in every beat. The resources used by operations in a trace are modeled by the instruction scheduler as it creates a schedule. However, a split may leave a schedule with pipelines in flight. The compiler must track the resources used to wind down these pipelines. Similarly, a join to a schedule may require winding up some pipelines. Information about pipelines in flight must be associated with each split from and join to a schedule.

We call the the set of pipelines bisected by a split or join a *partial schedule*.

9.3.1.1 Creating partial schedules

A partial schedule is the upper or lower half of a set of pipelines bisected by a split or join [23]. Our notation for the *nth* beat of the pipeline for an operation O is O-n. In Figure 9-12, the join from X bisects C, and the split from B bisects A. The join partial schedule is [C-1]/[C-2]; the split partial schedule is [A-3]/[A-4].

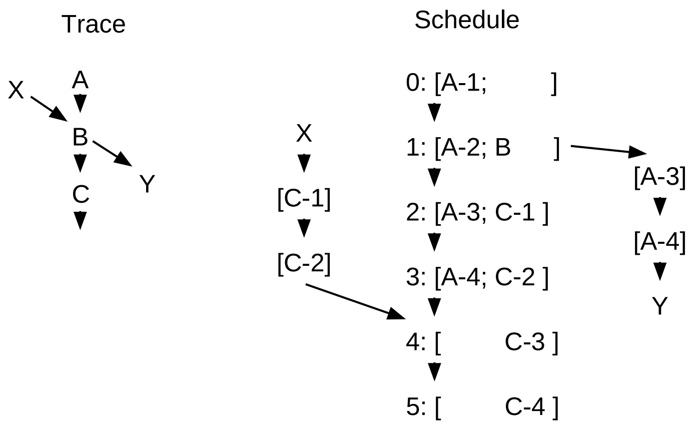


Figure 9-12: Partial schedules

On a join, the upper half of the bisected pipeline is copied onto the rejoining edge. An operation is in a join partial schedule for a join to trace position *J* with rejoin cycle *R* if *trace_position(O) >= J*, *first_cycle(O) < R*, and *last_cycle(O) >= R*. In targeting a rejoin, we have to make sure that a branch can be added to the last cycle of the join partial schedule. We cannot insert a new instruction with a branch between the partial schedule and the schedule, for that will shift the resources used by the pipelines and may disrupt the schedule we are joining. If a branch cannot be added to the last cycle of the partial schedule, we must try to join to subsequent cycles until we find a compatible cycle or we reach the end of the schedule (where we know we can rejoin). For example, if [C-2] locks out a branch in the example above, we cannot join to cycle 4, but must try to join at cycle 5; if [C-3] locks out a branch, we can join to the end of the schedule.

On a split, the bottom half of the bisected pipeline is copied onto the split edge. An operation is in a split partial schedule for split *s* if *first_cycle(O) <= last_cycle(s)*, and *last_cycle(O) > last_cycle(s)*. Note we must include the operations that are speculatively scheduled above the split and bisected by the split edge.

9.3.1.2 Merging partial schedules

A trace with a predecessor of previously scheduled code may have an associated wind-down partial schedule. This partial schedule will be passed to the instruction scheduler along with the trace, and it will be placed in the first instructions of the schedule. See Figure 9-13.

$$\begin{tikzcd}
\text{Trace} & \text{Schedule} \\
\text{[A-3;]} \arrow[d] & \text{0: [A-3; Y]} \arrow[d] \\
\text{[A-4;]} \arrow[d] & \text{1: [A-4; Z]} \\
\text{Y} \arrow[d] & \\
\text{Z} &
\end{tikzcd}$$

Figure 9-13: Wind-down partial schedule

Similarly, a trace with a successor of previously scheduled code may have an associated wind-up partial schedule. This partial schedule will be passed to the instruction scheduler along with the trace, and it will be placed in the last instructions of the schedule. See Figure 9-14.

$$\begin{tikzcd}
\text{Trace} & \text{Schedule} \\
\text{W} \arrow[d] & \text{0: [W-1;]} \arrow[d] \\
\text{X} \arrow[d] & \text{1: [W-2; X]} \arrow[d] \\
\text{[C-1]} \arrow[d] & \text{2: [W-3; C-1;]} \arrow[d] \\
\text{[C-2]} & \text{3: [W-4; C-2;]}
\end{tikzcd}$$

Figure 9-14: Wind-up partial schedule

Information about the machine resources used in the schedule joined by the wind-down partial schedule is also maintained. This permits the two traces to be more tightly merged. On a machine with self-draining pipelines, the tail of a pipelined operation only consumes resources; it does not require an operation to be initiated. The tail can be placed on a flow path where it will have no effect unless the first cycle of the operation is executed. In Figure 9-15, the last two cycles of W are placed in the schedule we join to. If control flows to instruction 4 with W in flight, W-3 continues the pipeline. If control falls through to instruction 4 from instruction 3, W-3 is a nop. The compiler performs this merge after creating the schedule for the trace.

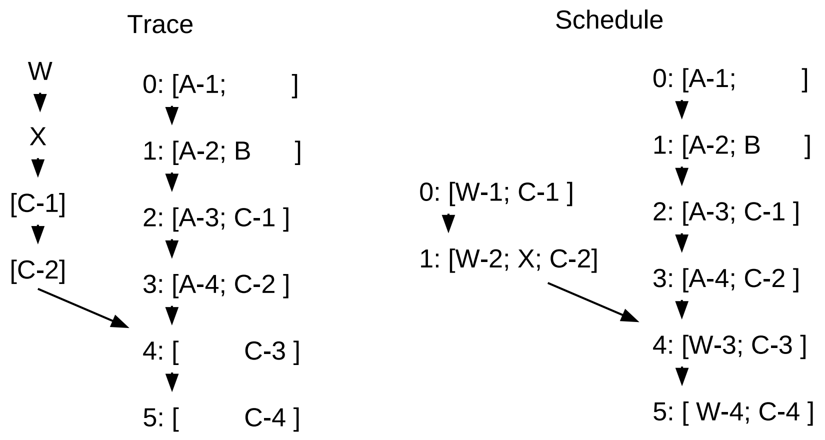


Figure 9-15: Merging partial schedules

9.3.1.3 Memory-reference information

When the instruction scheduler creates a partial schedule it also saves information about the memory references in the partial schedule and about the memory references in the 4-beat bank-stall window before the split or after the join. This permits the instruction scheduler to perform card and bank analysis when merging partial schedules.

9.3.2 Register bindings

When scheduling a trace, the instruction scheduler must decide where to read the variables that are live on entry to the trace and where to write the variables that are live on exit from the trace. For the first trace, no binding decisions have been made and the instruction scheduler is free to reference these variables in the locations that result in the best schedule. Most subsequent traces will be entered from or branch to machine code schedules where binding decisions have already been made. When a trace is entered from a machine code schedule, it must read its upward-exposed variables from the locations where the machine code last wrote them. When a trace branches to a machine code schedule it must write downward exposed variables into the locations read in the machine code below. This analysis is applied to all register candidates, which are IL temporaries and constants; we use the term value to describe both.

9.3.2.1 Value-location bindings

The information about value-location bindings within scheduled code is recorded and communicated via a data structure called a *Value-Location Mapping* (VLM) [30].[1] A VLM maps a set of values into a set of locations; one value may have more than one location, but each location will have at most one value. VLMs are created by the instruction scheduler after it has generated machine code for a trace. A distinct VLM is required for each split from and for each join to the schedule. The VLM at a split describes where the schedule has placed its downward

[1] Ellis called these *defs* and *uses* [23].

exposed values. The VLM at a join describes where the schedule reads its upward-exposed values. The following example shows the VLMs created by the instruction scheduler for a simple trace.

```
     Trace                Schedule

   z = A[i]             (< A, ireg1 >
                         < i,  ireg2 >)
                         0: freg1 = ireg1[ireg2]
                        (< A, ireg1 >
                         < i,  ireg2 >
                         < z, freg1 >)
```

After the VLMs are created, they are added to the flow graph to guard the borders of the schedule from the surrounding intermediate code. The trace picker then treats them like other operations. A VLM will always be either the first or last element of a trace. When a VLM is the first operation on a trace, the instruction scheduler must read any upward-exposed values from one of the locations given by the VLM. When a VLM is the final operation on a trace, the instruction scheduler must leave a copy of each value in the VLM in each of the associated locations.

9.3.2.2 Delayed Bindings

Some values are live at the top and bottom of a trace but are not referenced on the trace itself. We want to avoid making binding decisions for these values at the time the trace is scheduled. Eventually, we will have to decide on at least one location for each value through all the schedules that make up its live range. It is not necessary, though it is desirable, that a value reside in the same location in all the schedules that make up its live range.

Delaying the binding decision for a value until we actually schedule an operation that references it has several advantages:

- We implicitly prioritize the values by the expected frequency of the access to them. This allows us to weigh the benefits of keeping different values in registers; given a conflict, we would most like to keep those values in registers that are accessed in the highest frequency code, relegating spills and restores to the low frequency “suburbs”. (This achieves an effect similar to hierarchical coloring [10]).
- We can wait to choose locations until the scheduler actually sees how each particular value is accessed. This is especially important in the presence of function calls, which will require that their arguments and return values be in certain prespecified locations.
- The Trace machines have many separate register banks with each functional unit connected to a distinct register bank. A value must be in the proper register bank to be a functional-unit operand. We can make a much better register assignment if we defer the assignment until we know the functional unit of a value’s reader.

In order to delay the binding decisions for unreferenced live values, we devised a mechanism that we call *delayed bindings*. A delayed binding is a pseudo-location that may be assigned to an unreferenced value by the instruction scheduler. A delayed binding represents a connected subset of the value’s live range —a set of schedules that have already been generated and through which the value must pass without data motion. For each given value, the same delayed binding is used in the VLMs at all the boundaries of the connected set of schedules. As the connected subgraph of schedules grows, the delayed bindings for the same value are merged. When a binding decision is finally made, the delayed binding is updated to reflect the assigned location.

A delayed binding accumulates information that will be needed to choose a physical location for a value. For each schedule, we maintain a set of registers not yet allocated in the schedule. To resolve a delayed binding, we pick a register in the intersection of the unallocated register sets of the delayed binding’s set of schedules. If no such register exists, a memory location is chosen.

Figure 9-16 is an example of a delayed binding and its resolution. The binding decision for Z has been delayed as each of the schedules S6, S7, S8, and S9 were created and the delayed bindings for Z have been merged. At this point VLMs exist at the entry and exit to the subgraph which map Z into a delayed binding spanning the schedules in the subgraph. When the trace containing the assignment to Z and the entry VLM is scheduled, register freg3 is chosen and the binding is resolved. The binding is propagated to the exit VLM.

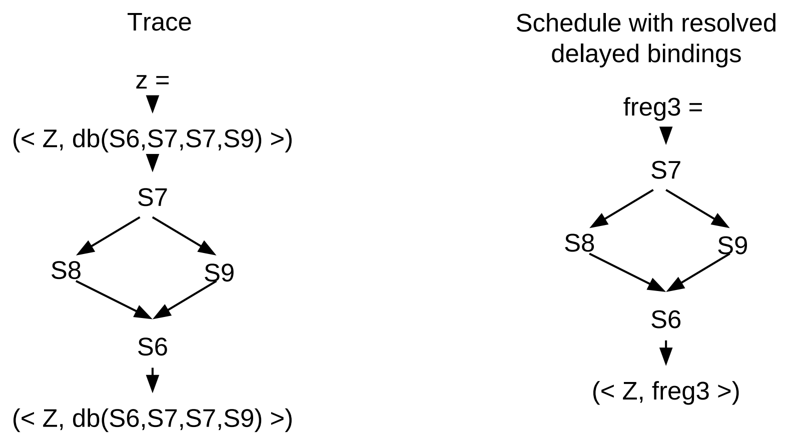


Figure 9-16: Resolution of a delayed binding

10 Instruction scheduler

Our instruction scheduler transforms a trace of IL2 operations into a schedule of wide instructions; it encompasses both the scheduling and register allocation phases of other compilers [16, 32, 34, 17, 18, 56, 31, 55, 72, 11, 12]. The operations have been lowered to machine level by Phase2. For each operation, the instruction scheduler must assign registers for the operands, assign a functional unit for the operation, and place the operation in a wide instruction. It performs a three step algorithm.

A. Build a data precedence graph (DPG) from the trace.

B. Walk the DPG and assign operations to functional units and values to register banks.

C. Perform list scheduling, creating the schedule and allocating registers. While scheduling, group memory references to minimize memory-bank conflicts.

10.1 The Data Precedence Graph

The instruction scheduler's basic data structure is a Data Precedence Graph. Each node in this graph represents a value with its associated defining operation, and each edge represents a scheduling constraint. There are three types of edges in the DPG:

- Operand edges, which describe the creation and the use of a value. These edges will require a register assignment.
- Memory edges, which describe definite and possible conflicts between load/store and store/store pairs.
- Constraining edges, which define scheduling constraints inserted for both correctness and performance.

Scheduling a trace rather than a basic block or an extended basic block introduces very little complexity in the instruction scheduler. When building the DPG, edges are added to constrain some operations from moving above splits. After scheduling, the trace scheduler compensates for any global code motions performed by the instruction scheduler.

10.2 Clustering

In assigning functional units and register banks, we want to cluster the assignments so that neighbors in the DPG are assigned to neighbors in the machine. However, only by spreading a computation around the machine and using all functional units can peak performance be achieved. The clustering algorithm trades off the benefit of parallelism with the cost of global data motion. The width of the Trace 14/300 and 28/300 make this a difficult problem.

It can be expensive to spread a computation across the machine. Each functional unit has its own register bank where it accesses its inputs. If an operand is not in the local register bank, a register transfer is required. A register transfer adds latency to the calculation and consumes a functional-unit resource, preventing another operation from being scheduled. The functional units can write to either local or remote register banks, but writing to a remote bank increases the latency and consumes a global bus. These busses are also used to load values from memory, and they are a critical resource on the Trace machines.

Some functional units have an overlapping but non-identical repertoire. In particular, each cluster has two integer ALUs of which only one can perform memory operations. This presents a conundrum for induction variables and memory references. If the induction variable is kept locally to the memory-reference ALU, incrementing it consumes the same resource required for a memory operation. On the other hand, keeping the induction variable local to the simple integer ALU requires two operations per induction variable update — one to increment the induction variable and one to move it into the memory-reference ALU. Fortunately, this latter strategy worked well for vector loops, which are unrolled and strength reduced so that frequently an induction variable update can be shared among a group of memory references.

10.2.1 BUG - The Bottom Up Greedy Algorithm

Our original approach to the clustering problem was taken directly from the Bulldog compiler — the *Bottom Up Greedy* (BUG) algorithm. BUG is a complex algorithm and is well described in [23]. The idea is to make a trial schedule focusing only on the issue of minimizing the amount of data transfer latency. The algorithm is "greedy" because it chooses the best functional unit available to it at the time an operation is considered. The resulting schedule is discarded; only the functional-unit and register-bank assignments are retained.

The BUG algorithm is a depth first traversal of the DPG using the recursive function bug_Assign, starting at the outputs (typically stores) working towards the inputs (typically loads.) The search is guided at each level by the latency-weighted depth of the nodes, so that a critical path of the computation is always searched first.

```
bug_Assign(op,destination_fus)
{
        if op is upwards-exposed VLM return
        if op.fu is assigned return

        foreach predecessor P in priority order {
                if (P is operand of op) {
                        possible_fus = best_fus(op,destination_fus)
                        bug_Assign(P,possible_fus)
                        }
                else
                        bug_Assign(P, nil)
        }
        op.fu = choose_fu(op,destination_fus)
}
```

Figure 10-1: The Bottom Up Greed (BUG) Assignment Algorithm

Here is an outline of how bug_Assign works:

A. When invoked on an operation, destination_fus, a list of possible destinations, is passed. If the list is non-null, at least one of the functional units will be a reader of the operation's result when the final assignments are made.

B. The predecessors P are visited in priority order, determined by the latency-weighted depth P in the DPG. Depth is measured from the entries to the DPG; it is a lower bound on the earliest time P could be scheduled. Items of greater depth are visited first; we visit the predecessors in order of maximum delay imposed on scheduling op. bug_Assign is recursively invoked on each P.

C. If P is an operand of op, a list of the best functional-unit assignments for op is computed and passed along with P. Each functional unit on the list is an equally good assignment for op; it can compute the result and deliver it to one of the destinations at as early as any other. The estimate of completion time is based on:

   1. Scheduled times of already assigned operands,
   2. Latency-weighted depths of the unassigned operands,
   3. Functional-unit assignments of the already assigned operands and the time required to move the operand from the functional unit that produced it,
   4. Earliest availability of the functional unit after its operands could optimistically be accessed (based on 1-3),
   5. The latency of the functional unit, and
   6. Earliest time to deliver the result to one of the destination functional units.

D. After the predecessors have been scheduled, register banks are assigned to the operands of op, and op is assigned a functional unit and placed in the schedule.

10.2.2 Problems with a greedy algorithm

A greedy algorithm is often inappropriate for the types of computations that attain peak performance on the wider Multiflow machines. Such computations are very parallel, for example, vector loops unrolled by the compiler. These codes have no single critical path; rather, all the computations are peers. The goal is to layout the computations on the machine in a manner that maximizes throughput rather than minimizing the latency of any particular thread. This general problem has a number of concrete manifestations. Among them:

- A greedy algorithm might schedule non-critical path operations in one component that result in delaying critical path operations in later components. For example, consider an unrolled daxpy

$$\begin{tikzcd}[column sep=small]
 & & T \arrow[dr] & & LD\ A,0 \arrow[dl] & & & T \arrow[dr] & & LD\ A,1 \arrow[dl] \\
LD\ B,0 \arrow[drr] & & & * \arrow[dl] & & LD\ B,1 \arrow[drr] & & & * \arrow[dl] & \\
 & & + \arrow[d] & & & & & + \arrow[d] & & \\
 & & ST\ B,0 & & & & & ST\ B,1 & &
\end{tikzcd}$$

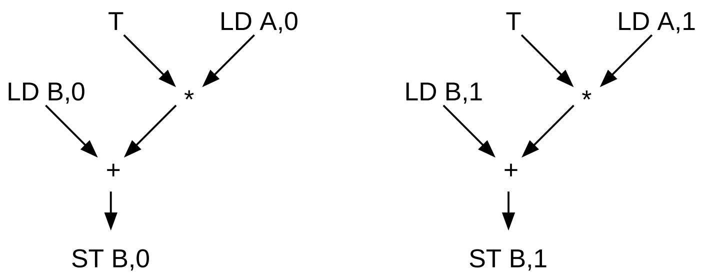


Figure 10-2: DAXPY, unrolled 2 times

A greedy depth first traversal will visit the LD B,0 before the LD A,1. It will assign and schedule LD B,0 first, potentially delaying the start of LD A,1, and thus the completion of the second tree. However, the LD B,0 is not on the critical path of the first tree, and scheduling it first has no advantage.

- Optimizing the completion time of the first component visited may result in spreading this component out across the machine. The resulting data motion will require the use of the global busses, reducing the number of memory references that can be issued. For example, in the daxpy4 shown in Figure 10-3, if all of the loads and multiplies are performed in parallel on separate functional units, the first component can be completed early. However, the results of the multiplies must be moved to a common functional unit to perform the adds. This data motion competes with loads from later components for the global memory busses, and effectively cuts the memory bandwidth in half.

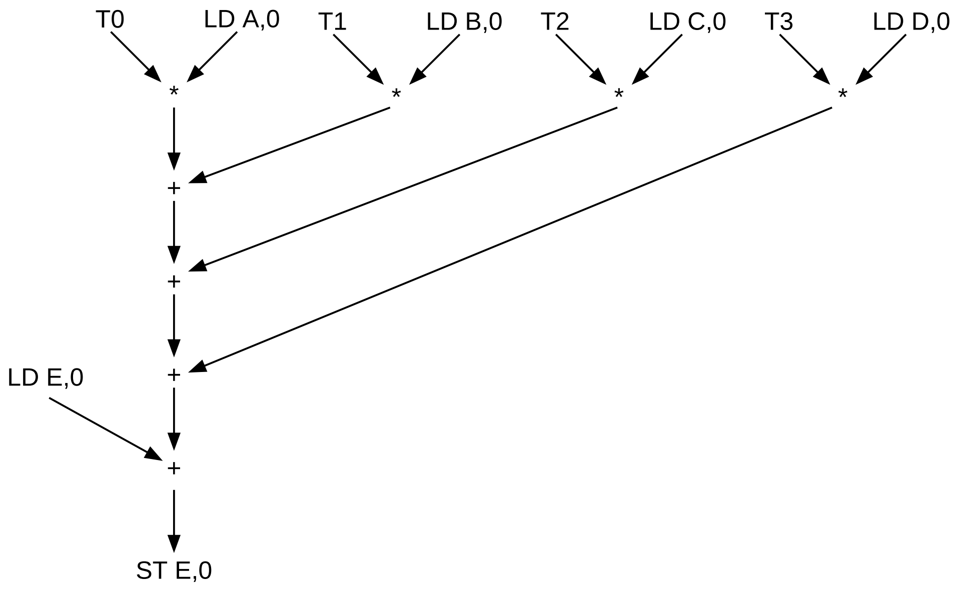


Figure 10-3: One component of DAXPY4

- BUG will not consider how later parts of the computation will fit in the holes created in the schedule. For example, consider an unrolled and interleaved dot product.

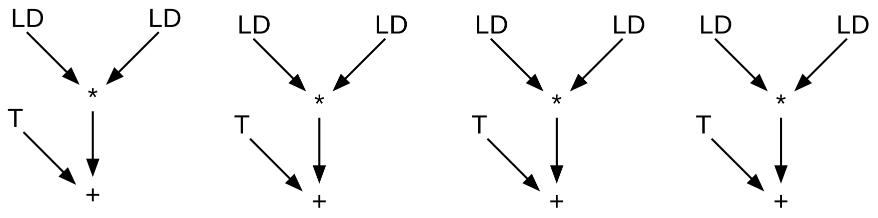


Figure 10-4: Dot product, unrolled 4 times

As the operations in the first tree are assigned, BUG makes the decision to load the two operands of the multiply in parallel, targeting the same register bank. This completely subscribes the register-bank write ports for 7 and 8 beats later. This prevents a multiply (which has a shorter pipeline) issued 4 beats later from targeting the same register bank. A better schedule will only load one operand of each multiply per beat, which leaves a write port available for a later operation.

In addition, BUG does not consider memory-bank conflicts when assigning functional units; this problem is discussed later. In the discussion above, we assume that no memory-bank conflicts are present.

10.2.3 Controlling the Greed of BUG

In order to address these problems we imposed a system of constraints on BUG. The idea is to impose a high penalty on BUG's latency calculation whenever it considers using more than one identical resource for a particular thread of the computation. This keeps the members of the same thread together in the same subset of the machine and achieves parallelism by spreading the threads around the machine. In detail:

- Partition the machine into a number of equivalent *virtual clusters*; a virtual cluster is a subset of a hardware clusters described in section 3. Each virtual cluster contains one functional unit of each kind and represents a simple target with few placement choices.
- Partition the computation into *components* each of which contains relatively little parallelism and a relatively large amount of shared data. We partition the code into components using a depth first search from the terminals of the DPG, traversing only operand edges. A unique component is assigned to each terminal and to each operation encountered for the first time in the search from that terminal. This has the effect of associating common subexpressions with only one of their descendent terminals. In the examples in the previous section, each tree is a component.

We would like to assign the components to the virtual clusters to spread the computation evenly through the machine. The existence of common subexpressions between components makes this nontrivial. Consider Livermore Kernel 12, shown in Figure 10-5. If we assign each component to a different cluster ignoring the inter-component reads, we will have much more data motion than is required. On the other hand, assigning all the components to the same cluster does not utilize the other clusters in the machine. The best solution to this problem is to perform *m* successive components on the same cluster and then switch clusters. This has the effect of reducing the inter-cluster move cost by a factor of *m*.

```
c     kernel 12 first difference.
      do 12 k = 1,n
12    x(k)= y(k+1) - y(k)
```

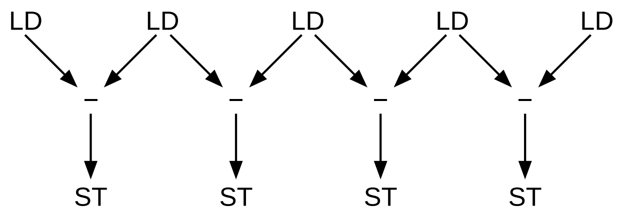


Figure 10-5: LFK 12: First difference, with many common subexpressions

To achieve this assignment, we introduce a third data structure:

- Create a partitioning of the components into *equivalence classes.* Two components are in the same equivalence class if one contains an operation whose result is read by the other. For computations such as Figure 10-5, all the components belong to the same equivalence class. On the other hand, for computations that partition cleanly, as in Figure 10-4, each component has its own equivalence class.

BUG uses this system of virtual clusters, components, and equivalence classes to impose penalties by dynamically associating a virtual cluster with each equivalence class. Initially the association is undefined. Each time a functional unit is chosen for an operation, the equivalence class is assigned to the virtual cluster containing the functional unit. When alternative functional units are considered for an operation and their estimated completion times calculated, a penalty is imposed if the operation's equivalence class has an associated virtual cluster different from that of the given functional unit. This has the desired effect of keeping the members of the same equivalence class in the same virtual cluster until a threshold of opportunity is reached.

10.3 Scheduling and Register Allocation

Once functional-unit and register-bank assignments are complete, we make a second scheduling pass over the data precedence graph. The result of this pass is the schedule returned to the trace scheduler. This pass performs three tasks.

- Schedule machine resources such as functional units, register write ports, and busses, using a complete, detailed model of the machine. The Trace machines are not scoreboarded; resources can be oversubscribed, and register dependencies are not checked by the hardware. The schedule created by the compiler must be accurate.
- Allocate registers, inserting spills and restores as necessary.
- Schedule memory references to avoid card conflicts and minimize bank conflicts.

10.3.1 Instruction Scheduler Design Choices

All instruction scheduler implementors face the same decisions about the basic structure of the scheduler. The choices are:

A. Operation-driven versus cycle-driven. Cycle-driven schedulers maintain a list of data ready operations and consider cycles in turn, trying to fill them with high priority operations from the data ready list. By contrast, operation driven schedulers make a topological sort of the operations (driven by some priority heuristic, usually critical path considerations). They then consider the operations one at a time, trying to find the first instruction cycle in which to place each operation. BUG is an operation driven scheduler. Our scheduling and register allocation pass is cycle-driven.

B. Forward versus backward scheduling. Forward schedulers start by placing the leaves (typically LOADs) of the computation first in the earliest possible cycles of the schedule and moving downward through the data precedence graph to the roots (typically STOREs). Backward schedulers turn this around. Both of our schedulers are forward.

C. Backtracking and how much. To what extent are the decisions made by the scheduler final? BUG does no backtracking. Our scheduling and register allocation pass does a limited amount of backtracking.

D. Separate register allocation and scheduling passes versus integration of register allocation and scheduling. Register allocation and instruction scheduling are often performed in separate passes due to the complexity of each. A separate register allocation pass after scheduling will work well if the target machine has enough registers. But if pipeline depth and issue width increase while the number of registers stays constant, this becomes a more difficult condition to fulfill. Our instruction scheduler integrates scheduling tightly with register allocation.

E. Various priority functions. Most of these will be topological sorts of the DPG, but breadth first (good for parallelism) versus depth first (good for register allocation) can have major performance implications. Our priority functions are depth first. We rely on the clustering heuristics in BUG and the natural breadth first orientation of the list scheduling algorithm to give breadth to our scheduling passes. In BUG, we walk the DPG by component, performing a depth first traversal of each. We visit components in the order they occur on the trace, earliest first; our depth first traversal is guided by the latency-weighted depth of each predecessor in the DPG. For our instruction scheduler, we form a total ordering of the operations by performing a similar walk to BUG; this ordering is used as a priority function. All operations from an earlier component have higher priority than operations from any later component. This ordering gives higher priority to the earlier basic blocks on the trace; in particular, every operation in the first unrolled loop body will have higher priority than the operations in the second, and so forth.

F. Static vs dynamic scheduling priority assignments. Do the priorities of operations change during scheduling? Our scheduling and register allocation pass has a limited amount of dynamic priority assignment.

10.3.2 Integration of Scheduling with Register Allocation

The goal of integrating register allocation with scheduling is to be able to treat registers as a resource on par with the other types of resources. This is important when registers are a scarce resource. On the Trace machines, long pipelines and multiple operations per instruction make floating point registers a critical resource for generating peak performance.

In addition, the Trace machines require a detailed scheduling of machine resources, which is quite expensive in compile time. This provides additional motivation for integrating scheduling and register allocation. Performing register allocation after scheduling requires a second scheduling pass for the spills and restores. Allocating registers before scheduling is not practical for a parallel machine, because it introduces too many constraints into the code.

Cycle-driven scheduling is particularly well suited to integration with register allocation. This organization allows the registers to be treated very much like other resources. At any given point during the scheduling process, each of the registers is either *occupied* or *available*. The process of checking for resources to schedule a particular operation in a given cycle can include a check for a free register for its result. If all the resources are available to schedule the operation, the register, as well as the other resources, is reserved.

There is a major complication with trying to integrate this kind of simple first-come first-served register allocation with operation-driven scheduling [23]. Because operation scheduling does not follow instruction order, it can leave gaps during which a given register is available only for a certain number of cycles. How can the scheduler decide whether an operation can target a register in one of these gaps? In can do so only if the reader(s) of the operation can be made data ready and resource ready in time to read the value before the end of the gap. It is difficult for an operation driven scheduler to make use of these gaps without backtracking.

10.3.3 Spilling

A register allocator must be able to deal with computations that do not fit in the available registers. Our solution ensures that the scheduler always makes progress. Before scheduling an operation, we first check that all required resources are available and then check if a register is free for the result. If there is no free register, we heuristically select a *victim* that is occupying a register. We compare the priority of the operation we are trying to schedule with the priority of the victim. If the victim has lower priority, we schedule a spill of the victim. Otherwise, we delay the operation. It is important that the priority assignments be a topological sort of the data precedence relationship. If the victim has lower priority than the operation we are trying to schedule, the chosen operation does not depend on the victim, and the victim can be spilled. Similarly, if the victim has higher priority, we can delay the operation without risking deadlock.

Our mechanism for picking a victim grew in complexity over time. Operations are assigned a priority before scheduling. When an operation is scheduled, its priority it adjusted to be the priority of its most urgent unscheduled reader. Our first victim choosing strategy chose the least urgent operation that occupied a register of the right type for the unscheduled operation. Later on, we refined the victim choosing strategy to take advantage of our limited backtracking facility. We made the observation that some values are easier to spill than others and, having spilled them, some are easier to restore than others. In particular, loads from memory can be reloaded from their original locations without first requiring a spill if no possibly conflicting store has been scheduled in the interim. Results of immediate moves are also easy to recreate. The result of this strategy is to shorten the live range of memory references in the presence of register pressure. The scheduler greedily fills unused memory bandwidth with loads, and the register allocator removes those that were premature. The effect is similar to what might have been achieved by backward scheduling.

10.4 Limited Backtracking

The scheduler employs a limited form of backtracking that has proved quite useful. Memory operations can be replicated, or *split*, in the DPG, and scheduled multiple times. Splitting occurs as scheduling proceeds; three different situations can trigger it:

A. Register allocation pressure can cause a scheduled load to be replicated in order to reclaim its target register for a higher priority operation. When this happens, the original load is removed from the schedule if none of its readers have yet been scheduled. In some cases, the load is a common subexpression with scheduled readers of more urgent priority and unscheduled readers of lower urgency. Splitting in this case gives an effect similar to *live range splitting* in coloring register allocators [11, 12, 18]. The difference is that our splitting is responsive to the needs of the instruction scheduler.

B. Memory-reference splitting is used to spread values to different register banks. Depending on the balance of memory references to floating point operations, it is sometimes better to load a value several times to different register banks than to load the value once and distribute it with a series (or tree) of register transfers. After a floating point load is initially scheduled, the location requirements of its readers are analyzed. If there are a small number of memory references relative to floating point operations on the trace we will decide to deliver the data to remote register banks by reloading rather than by direct register transfers. This strategy works well in concert with BUG clustering to limit the amount of intercluster register transfers required for code with memory common subexpressions.

C. Splitting can reclaim resources from lower priority operations on the behalf of higher priority ones. One particularly troublesome form of this is due to the mismatch of latencies between floating point and memory operations as shown in Figure 10-6. In this example, two double precision loads issued in cycle 0 completely subscribe the write ports to register bank 1 in cycles 7 and 8; this prevents a floating multiply in cycle 4 from targeting register bank 1. Assume the multiply is of higher priority than the loads, but was not data ready until cycle 4. We will undo one of the lower priority loads to allow the higher priority multiply to be scheduled. The unscheduled load will be returned to the data ready queue.

```
0:    issue 2 loads to reg bank
1:
2:
3:
4:    cannot issue multiply to reg bank 1
5:
6:
7:   reg bank 1 write ports used by loads
8:   reg bank 1 write ports used by loads
```

Figure 10-6: Lockout in scheduling

Unlike unlimited backtracking, our form of limited backtracking does not introduce a large algorithmic time complexity into the scheduler. If we unschedule an operation, we do not refill its issue slot and we guarantee that an operation will never be unscheduled more than once. In practice, very few operations are split.

10.5 Scheduling and Memory-Bank Disambiguation

As described in section 3, the 300 series has a two level interleaved memory hierarchy exposed to the compiler.

- Card conflicts result in a program error. All memory references in the same beat must be to distinct memory cards; if they are not, the result of the references are undefined.
- Bank conflicts result in a loss of performance. On the 300 series, a memory reference busies a bank for 4 beats; a second reference to the same bank within a 4-beat window of the first will cause the entire machine to stall until the end of the busy time.

Our strategy for managing bank and card conflicts is to divide the memory references in a trace into equivalence classes based on our knowledge of their offsets relative to each other. The equivalence classes are formed by querying the disambiguator while scheduling. Within an equivalence class, we understand the potential bank and card conflicts; between two different equivalence classes, nothing is known. For example, in a simple vector kernel such as vector add (Figure 10-7), we know the relative offsets of the successive references to `a` and the successive references to `b`, but (without interprocedural information) do not understand the relative offset of a reference to `a` and a reference to `b`.

```
subroutine vadd(n,a,b)
double precision a(n),b(n)
do i = 1,n
        a(i) = a(i) + b(i)
end do
end
```

Figure 10-7: Vector Add Subroutine.

In scheduling code for a trace, we schedule a group of references from one equivalence class and then a group from the next, alternating between classes to minimize bank stalls as in Figure 10-8 (b). A schedule which did not batch memory references by equivalence class will risk a bank stall on every reference as in Figure 10-8 (a).

```
(a) Naive Order                     (b) Batched Order

t0 = a[i]                           t0 = a[i]
v0 = b[i]                           t1 = a[i+1]
r0 = t0 + v0                        t2 = a[i+2]
t1 = a[i+1]                         :
v1 = b[i+1]                         v0 = b[i]
r1 = t1 + v1                        v1 = b[i+1]
t2 = a[i+2]                         v2 = b[i+2]
v2 = b[i+2]                         :
r2 = t2 + v2                        r0 = t0 + v0
:                                   r1 = t1 + v1
                                    r2 = t2 + v2
```

Figure 10-8: Memory-reference ordering.

Batching groups of references increases register pressure. If we batch references in groups of 8, the schedule requires 16 registers and risks a bank stall every 8 memory references. The naive schedule order requires only 2 registers and risks a bank stall with every memory reference. This tradeoff requires a heuristic limit on memory-reference batching.

Memory-card conflicts result in program error; we must avoid them when scheduling code. The scheduler keeps track of the memory references scheduled in each cycle, consults the disambiguator for each new candidate reference, and requeues the operation if a definite or possible conflict exists.

Memory-bank conflicts cause a program to slow down; we use our batching strategy to minimize the number of conflicts. The scheduler models the 4-beat bank-stall window by tracking memory references in the cycle being scheduled and the previous 3 cycles. When a memory reference is selected to be scheduled, we call the disambiguator to compare it with the other memory references scheduled in the current bank-stall window. If no bank conflict is found, we schedule it. If a definite bank conflict is found, we requeue the operation for the next cycle. If a possible bank conflict is found, we set it aside, and lower priority operations are considered for the cycle. When the other operations for this cycle have been considered, we revisit the list of possible bank conflicts.

We control the degree of batching by limiting the number of memory references that can be delayed in favor of a lower priority reference. When this limit is exceeded, we reconsider the memory references that we set aside, this time without regard to possible memory-bank conflicts.

The main problem with our approach to managing bank conflicts is that BUG does not consider bank conflicts when performing functional-unit assignment; we only consider bank conflicts when forming the final schedule. This means that BUG frequently assumes a scheduling order that is dramatically different than the one produced by the list scheduler, which sometimes causes poor functional-unit assignments. The scheduler's particular implementation of bank scheduling has other flaws. We do not form the equivalence classes explicitly, but compute them as we schedule each cycle. This means the equivalence classes do not guide our scheduling decisions, but correct them at the last moment. And in the case of vector loops, we use the bank-disambiguation mechanism to reconstruct higher level information about the pattern of memory references which could be communicated to the instruction scheduler more directly.

10.6 Pair mode

The 300 series has a *pair-mode* feature, where 64-bit floating registers can be treated as 2 element single precision vectors, and two results can be computed simultaneously. Coupled with 64-bit loads from memory, 2 element vector operations can be generated. This doubles the peak single precision performance of the machine for applications that can exploit this feature.

The instruction scheduler supports pair mode by making a special pass over the DPG. It pairs two operations in the DPG if they are pairable and are at the same DPG depth. Only single precision floating point operations and memory references are candidates for pairing. The order of search for pairable operations is:

A. Pair memory references. The disambiguator is called to ask if two references (X,Y) are pairable. They are if address(X) = 0 *mod* 8 and address(Y) − address(X) = 4.

B. Pair operations whose readers are paired (e.g., the inputs to a paired store).

C. Pair operations whose operands are paired (e.g., the two adds of two paired loads).

D. Pair opportunistically in the DPG.

This works well if loops are aligned and post-conditioned such that the memory references in the loop iterations could be paired. However, the optimizations for doing this automatically were never completed. The pair mode feature was used to tune benchmarks and libraries in-house, where the appropriate loop-directives could be added by hand. This feature would have been more widely applicable if the system supported 64-bit memory references on 0 *mod* 4 byte boundaries.

11 Machine model

The machine model is the compiler’s view of the architecture. It is the sole source of all machine specific information except for the operation set, which is also encoded in the IL-2. The compiler’s model of the machine is abstracted from the actual hardware. However the absence of hardware resource management in the Trace machines forces the compiler’s machine model to match many hardware aspects precisely. The compiler models all eight machines: three widths for each of three families (two for the 500 series). All models are built into the compiler; a single executable can generate code for all of the machines.

The components of the machine model are machine elements, resources, and connections.

Machine elements are either functional units, register banks, or constant generators. A different functional unit is created for each pattern of resource utilization or set of connections, so there is often more than one machine-model functional unit for each hardware functional unit. For example, on the 300 series, three chips (a floating adder, a floating multiplier, and an integer ALU) execute the floating operation repertoire, but the compiler models the complex as 18 distinct functional units, for there are 18 different resource patterns of use. Of course, these 18 functional units share a common resource so that only one of them can be scheduled in a given beat. A different register bank is created for each set of registers with a unique set of connections to other functional units and register banks. A different constant generator is created for each width of immediate field that can be used.

Resources exist to express limitations on machine elements, such as busses, multiplier cores, and instruction words. The 300 series was modeled using four classes of resources: instruction word limits, which describe the resources of the instruction encoding; functional-unit internal core limits, such as the floating point cores; bus and write port limits, which enumerate the busses and write ports in the machine; and virtual resources introduced for the compiler’s convenience. For example, the compiler introduced a branch resource, even though it could be expressed using the other primitives.

Connections describe paths between machine elements. Paths must start or end at a register bank or a constant generator. There are three types of connections: functional-unit sources and destinations, which describe the paths connecting the functional units, constant generators, and register banks; copies, which describe paths from register bank to register bank, and constant generator to register bank; and procedure linkage, which describe both the paths to the hardware link registers and the virtual paths used to pass values at a procedure call. Resources are associated with connections. Copy connections require a functional unit when scheduled; they are treated specially because the instruction scheduler synthesizes them during scheduling.

12 Calling Sequence

12.1 Register partition

In most modern RISC runtime models, the register file is divided into three partitions: *preserved*, *scratch*, and *other* [18, 57, 36]. Preserved registers are preserved across a call, scratch registers are possibly changed across a call, and other are registers not available for general use (e.g., the stack pointer, the zero register, and the registers reserved for the operating system). Registers used to pass arguments are scratch.

Multiflow uses a pure caller-saves register partition; all register are scratch, except for the registers not available for general use. We originally implemented a pure callee-saves partition with all preserved registers. We were unsatisfied with the performance with this partition, and we switched to caller-saves. This change gave us a performance improvement of over 40% for some procedure call intensive programs, but was typically less than 5% for scientific applications. (We changed the argument passing from in-memory to in-registers at the same time; our in-register design is described below. This change also contributed to the speedup.) The improvement has two explanations.

First, compiling for parallelism requires the use of many more registers than are used in sequential compilation, and we do not want to pay a penalty on procedure entry and exit for their use. These registers do not have to be preserved because the lifetimes of the values in these registers typically do not cross function calls.

Second, the saving and restoring of scratch registers can be intermixed with other operations before and after a call, and on a wide machine with a large amount of memory bandwidth the saves and restores can often be scheduled for free. Trace scheduling selects traces in priority order, so the saving and restoring of registers can be moved out of the most frequently executed paths in the program. In contrast, in a pure callee-saves partition, the saving of some preserved registers must occur on procedure entry before any registers can be written. Similarly, on exit the restoring of some preserved registers must be delayed until the last operand registers are read. These operations can not be intermixed with any others, and they often are on the critical path.

In retrospect, we wish we had kept a few integer preserved registers to hold induction variables and loop invariants referenced in tight loops with function calls. But we did not see a compelling need for preserved floating registers.

12.2 Argument passing

Multiflow uses all of the registers in the first cluster for passing arguments and returning values. 53 integer values and 30 double precision floating values can be passed or returned in registers. Larger argument lists or return values overflow into memory. Aggregates (e.g. C structures) can be passed and returned in registers. The large number of registers available for procedure linkage allowed us to write our N-at-a-time math library in C and have a high performance linkage for 8- and 16-at-a-time functions.

FORTRAN always passes arguments in registers (except for the overflow block). C supports three modes: passing arguments in registers, in memory, or both. C varargs routines require arguments passed in memory, as do some older unportable C programs. Other C routines prefer arguments passed in registers for performance. To provide ease-of-port, our default in C is to pass arguments in both memory and registers; the called procedure reads the arguments from memory (if required) or from registers. Users can request arguments to be passed only in registers, and the linker checks that all caller-callee pairs are compatible.

Passing arguments in both registers and memory has a surprisingly small effect on the performance of C programs. The reason for this is that the access of arguments on procedure entry is often on the critical path, and the arguments can be read directly from registers in all but varargs procedures. The storing of arguments at a call site is often not on the critical path, and it can be scheduled alongside other operations at little cost. In addition, on the 300 series, the load pipeline is long (7 beats), and the store pipeline is short (effectively zero for the data item being stored).

13 Disambiguator

Most work on compile-time memory-reference analysis has been done for vectorizing or parallelizing compilers [5, 58, 73]. These compilers want to know if a loop (or loop nest) can vectorize or parallelize, and thus are interested in knowing if there exists a conflict across any of the iterations of the loop. In contrast, our trace scheduling compiler is looking for fine grained parallelism along a trace and within the body of an unrolled loop. We want to know if the load in one iteration can move above the store in the previous iteration.

The disambiguator is queried in both Phase 2 and Phase 3. In the optimizer, it performs location conflict analysis for common subexpression elimination, loop invariant motion, and the allocation of array elements to register variables in loops. In the instruction scheduler, it is used to analyze both possible location conflicts and possible bank conflicts.

13.1 An example

Consider the following loop, which we have unrolled to approximate at the source a transformation the compiler will make.

```
1.  subroutine test(a,t,j,k,n)
2.  double precision a(100,*),t
3.  do i = 1,n,4
4.          a(i,j) = a(i,j) + t*a(i,k)
5.          a(i+1,j) = a(i+1,j) + t*a(i+1,k)
6.          a(i+2,j) = a(i+2,j) + t*a(i+2,k)
7.          a(i+3,j) = a(i+3,j) + t*a(i+3,k)
8.  end do
9.  end
```

To generate good VLIW code for this loop, the loads in statements 5, 6, and 7 must move above the stores in statements 4, 5, and 6. In particular, the disambiguator must determine if the load from a(i+3,k) will ever refer to the same location as the store to a(i,j) on a given iteration of the loop. To answer this question, it constructs derivations of the addresses referenced by the store and the load and asks if the derivations can ever be equal. The derivations are symbolic equations with terminals corresponding to induction variables or loop invariants.

array reference	*derivation*
a(i,j)	address(a) + 8 * ((i-1) + 100*(j-1))
a(i+3,k)	address(a) + 8 * (((i+3)-1) + 100*(k-1))

We subtract the derivations, and see if the result can equal zero. Simplifying, we get:

100*j - 100*k - 3 =? 0

This is a Diophantine equation. Using the GCD test [42], we can conclude there is no integer solution and the two addresses are never equal. This allows us to move the load above the store when scheduling code.

Notice that we were able to solve this equation because we knew the value of the leading dimension of A. If lda is unknown, we get

```
        lda*j - lda*k - 3 =? 0
```

This equation has a solution, for example, when lda is 3 and j and k differ by 1. But in this case, the program has overstepped the bounds of the first dimension of the array. We lost the dimension information by expanding the index polynomial. To avoid this problem, we construct derivations of each index expression, and perform an index by index check when analyzing a pair of memory references for location conflict. If these checks fail, we then analyze the full addresses.[1] In this case, the index derivations for the first dimension are i and i+3. Their difference is -3, which is clearly not equal to 0.

13.2 Disambiguator Queries

The disambiguator answers the following questions:

- *Location_Conflict(ref1, ref2).* Do *ref1* and *ref2* possibly refer to overlapping memory locations?
- *Bank_Conflict(ref1, ref2).* Do *ref1* and *ref2* possibly refer to the same memory bank?

The answers it returns are: *Yes*, *No*, or *Maybe*.

The questions are answered in terms of the Phase 2 flow graph, before trace scheduling. The disambiguator has the following model: given a sequential execution of the program represented by the Phase2 flow graph, it can answer a question about any pair of dynamic instances of *ref1* and *ref2* that are executed without crossing any loop back edges. To prevent the instruction scheduler from asking a question across a back edge, we do not permit a trace to cross a back edge.

If the two references would not be executed on any dynamic path, the disambiguator cannot give a meaningful answer. Consider the following example where the two references have identical derivations, but clearly always refer to different memory locations for the same value of j.

```
        if (cond1) i = j + 1
        if (.not.cond1) i = j - 1
        if (cond1) a(i) = y
        if (.not.cond1) x = a(i)+1.0
```

13.3 Derivations

A derivation is a symbolic equation of the address used in the memory reference; the terminals in the equation are constants and definitions (or sets of definitions) of variables in the program. Derivations are constructed by walking recursively up the chain of reaching definitions [1] for the address. For example, consider the following code fragment.

```
    1.      subroutine test(a,n)
    2.      double precision a(n)
    3.      i = n+1
    4.      j = n/3
    5.      if (i.gt.100) then
    6.              m = n+1
    7.      else
    8.              m = n-1
```

[1] This is the compiler's default behavior. For those FORTRAN programs that reference arrays out-of-dimension we provide the ability to disable the index by index check.

```
9.      end if
10.     p = m+1
11.     k = i + j + p
12.     x = a(k)
```

The load of a(k) in statement 12 refers to address(a)+8*(k-1). To construct the derivation for a(k), we walk up the tree of reaching definitions for k, stopping when we reach a leaf of the tree, or when we reach an operation that we will not be able to analyze with our equation solver, or when we reach a variable with multiple reaching definitions (as suggested by [23]). In this example, we stop at statement 1 (the definition of n), since the subroutine entry has no predecessors, and at statement 4 (the definition of j), since we cannot analyze division, and at statement 10, since m has multiple reaching definitions. We substitute the equation defining k into the equation for the address of the load, and get the following derivation (where [x:y] represents the definition of y in statement x).

```
address(a) + 8 * (([1:n]+1 + [4:j] + {[6:m],[8:m]}+1) - 1)
```

Each derivation is normalized to a canonical form, so that arithmetic operations can be easily implemented. The derivation is expressed as a sum of products; terminals are reaching definitions, sets of reaching definitions, packet seeds, and constants.

To prevent the derivation from wrapping around the back-edge of a loop, we insert derivation fences at the loop head. A derivation fence is a self-assignment that terminates the recursive walk of the chain of reaching definitions. We create a derivation fence for all variables that are both defined in the loop and live on entrance to the loop head. This includes all induction variables. Once derivations are computed, the derivation fences are removed from the flow graph.

13.4 Packet checks

When asked about a location conflict between *ref1* and *ref2*, the disambiguator first performs a heuristic check based on the referenced packets. As defined in section 5, a packet represents a group of variables with a language-defined storage relationship. If the packet check is unsuccessful, it uses its equation solver to compare the derivations. For bank conflicts, the relevant packet information is in the packet seed, which is incorporated into the derivation, so we skip the packet check, and query the equation solver directly.

The packet checks test simple facts about the memory references that can rule out a location conflict. For example, direct references to distinct packets cannot conflict, nor can references to the different non-overlapping variables within the same packet. A pointer reference cannot refer to a variable whose address was never taken. Two template packets representing by-reference FORTRAN arguments cannot reference the same location if they are both referenced on a dynamic path through the flow graph, and there is a store to one of them, due to restrictions placed by the FORTRAN standard [6].

Actually, FORTRAN array arguments frequently do reference the same location in practice. To handle this, we provide a way to disable the ANSI restriction. However, aliased array arguments often refer to exactly the same portions of an array, and we added an equal-or-disjoint option to handle this case. When true the compiler will assume by-reference FORTRAN array arguments either refer to the same array, or are completely disjoint; it assumes they do not to refer to different offsets within the same array. The equal-or-disjoint option was sufficient for all of the applications we encountered.

13.5 The equation solver

The equation solver is algorithmically simple. For a location conflict, we want to know if

```
address(ref1) - address(ref2) =? 0
```

To answer the question, we subtract the two derivations, normalize the result D, find the GCD of the coefficients of the non-constant terms in D, and check if the GCD divides the constant term of D. If it does, there is a possible conflict.

For a bank conflict, we want to know if the two addresses are possibly equal modulo the number of banks B times the number of bytes per word, which is 4. We write this as:

```
address(ref1) - address(ref2) + i*4*B =? 0, for some integer i.
```

To answer this question, we compute D as before (ignoring the i*4*B term). We then fold 4*B into the computation of the GCD of the coefficients of the non-constant terms in D, effectively treating i*4*B as an additional term in D whose coefficient is 4*B. As before, if GCD divides the constant term of D there is a possible conflict.

13.6 Size and alignment

If the size or alignment of the two references differ, they may have different addresses, but still conflict in memory. For example, consider the following FORTRAN equivalence statement:

```
double precision a(10)
integer m(10)
equivalence (a(2),m(2))
```

m(1) overlaps with the second half of a(1), though the difference of the two addresses is 4.

To answer this, we define a window of possible conflict, based on the size and alignment of the two references, and check if the two references could possibly be located in the same window. If they could, the disambiguator returns *maybe*, indicating that the two references may conflict. This applies to both location and bank conflict.

13.7 Assertions

13.7.1 An example

For some pairs of references, the disambiguator cannot answer questions based on the derivations alone. The following example sets up an array for a call to an FFT routine. Performance of this code on the Trace 14/300 is disappointing; in the inner trace after loop unrolling, the loads from iteration i+1 cannot move above the stores on iteration i. This forces a sequential evaluation.

```
subroutine unpack(a,n)
real a(*)
...
 do i = 2, n_half

     afront_r = a(2*i)
     afront_i = a(2*i+1)
     aback_r = a(2*(n-i))
     aback_i = a(2*(n-i)+1)

     r_even = afront_r + aback_r
     i_even = afront_i + aback_i
     r_odd = afront_r - aback_r
     i_odd = afront_i - aback_i
```

```
          temp_1 = -twp1_r * i_even - twp1_i * r_odd
          temp_2 = twp1_r * r_odd - twp1_i * i_even

          a(2*i) = r_even + temp_1
          a(2*i+1) = i_odd + temp_2
          a(2*(n-i)) = r_even - temp_1
          a(2*(n-i) +1) = temp_2 - i_odd
      end do
      return
      end
```

The potential location conflicts are between the stores in loop-body i and the loads at the top of loop-body i+1. When comparing the first store in body i with all of the loads at the top of body i+1, the disambiguator gives the following answers. (Remember that i is incremented between the two bodies; we have substituted the incremented value of i in the second column.)

Body i	*Body i + 1*	*Disambiguator*
a(2*i) =	= a(2*i+2)	NO CONFLICT
a(2*i) =	= a(2*i+3)	NO CONFLICT
a(2*i) =	= a(2*(n-i)-2)	POSSIBLE CONFLICT
a(2*i) =	= a(2*(n-i)-1)	NO CONFLICT

In analyzing the memory reference in row 3, the disambiguator is trying to solve the following equation:

```
 (address(a) + 4*((2*i)-1)) - (address(a) + 4*((2*(n-i)-2)-1)) =? 0
```

Simplifying, we get:

```
 2*i - n + 1 =? 0
```

This has an integer solution when n = 2*i +1. However, the programmer knows that i is always less than n/2. Our compiler provides an assertion facility that allows a programmer to tell the compiler facts about the relationships between program variables. In this example, we would assert that 2*i is less than or equal to n at the loop head.

```
subroutine unpack(a,n)
real a(*)
...
do i = 2, n_half
c!mf! assert le(2*i,n)
      ...
end do
return
end
```

When the loop is unrolled, the assertion is duplicated with all of the other statements. The path from the first store in body i to the the third load in body i+1 now looks like the following:

```
a(2*i) =                        ; store from body i
...
i = i+1                         ; increment i (end of body i)
...
assert(2*i <= n)                ; assertion (beginning of body i+1)   ...
= a(2*(n-i))                    ; load from body i+1
```

Now, if we substitute for the increment of i, we have

```
a(2*i) =
...
assert (2*i+2 <= n)
...
= a(2*n-2*i-2)
```

The assertion can be rewritten as

```
 2*i - n + 1 < 0
```

This directly answers the question we had above. We know that 2*i - n + 1 cannot equal zero, and there is no location conflict.

If the compiler knew that n_half = n/2, it could generate the assertion itself. We implemented this optimization shortly before Multiflow closed, but we have no experience with it.

13.7.2 Definition

There are 8 types of assertions: EQ, NE, GT, GE, LT, LE, EQ_MOD, NE_MOD. The first 6 assert a relationship between two integer or pointer expressions. The last two assert a modulus relationship between two expressions; the base of the modulus must be a constant integer expression, restricted to powers of 2. The modulus assertions are useful in asserting about bank conflicts.

Derivations are constructed for assertions as well as memory references. To normalize an assertion, we construct the derivation of each side, subtract them, and factor out the GCD of the non-constant terms. For example, the assertion 4*i + 4*j < 1 is equivalent to 4*i + 4*j < 4 (since i and j are integers), which in turn is equivalent to i + j < 1, which is what the disambiguator uses. We discard useless assertions. (An assertion is useless if it is trivially false or trivially true.) The GT, GE, LT, LE are all mapped into GT, by commuting the operands and adding one if necessary.

Induction variable simplification introduces deriv_assign pseudo-ops to map the strength reduced induction variables back to the original subscript expressions. This allows derivations to refer to the original induction variables, which in turn permits assertions to be applied after IVS is performed. (Deriv_assigns also help loops without assertions, in that they allow different induction variables to be compared when otherwise they would be regarded as unrelated to each other.)

The meaning of an assertion is defined in terms of an execution of a program. The asserted relationship is assumed to be true at the point the assertion would be executed in the program. This means an assertion can be used in answering a question about two memory references if it dominates either reference. (If it dominates one of the two references, it must be executed if the reference is.) We do not actually generate code to execute the assertion. To allow for program portability, the assertions are represented as FORTRAN or C comments.

To keep track of which assertions apply to which memory references, we chain together the assertions that apply at a given point in the program. These chains form a tree that mirrors the dominator tree for the flow graph.

To use an applicable assertion, it must be an exact match: We subtract the derivation of the assertion from the derivation of the equation we are trying to solve, and the result must be a constant. The assertion is then used by testing the resulting constant. This is much faster than a theorem prover, and we have not encountered real-life examples where this restriction has been a problem.

13.8 Using the disambiguator during trace scheduling

The instruction scheduler queries the disambiguator in two contexts. When building its data precedence graph (DPG) for the trace, it asks the disambiguator about location conflicts between <*load,store*>, <*store,load*>, and <*store,store*> pairs on the trace. To minimize the number of questions, all other edges in the DPG are constructed first and the disambiguator is queried only about pairs of references that are not already constrained.

When scheduling code, the instruction scheduler asks about bank conflicts. The instruction scheduler models the 4-beat bank-stall window, and before placing a memory reference in it, queries the disambiguator about possible bank conflicts with other references in the window. Because the window is greater than one instruction, the instruction scheduler may need to perform bank conflict analysis across adjacent traces, i.e., when a split from one trace rejoins to another. To permit this, a record describing the bank-stall window is associated with any split from or rejoin to the schedule for a trace.

13.8.1 Motion above splits

The instruction scheduler often moves code above splits. We can location-disambiguate loads that have moved above a split because of restrictions in the instruction scheduler. The instruction scheduler will not move an operation above a split if it has a side effect that affects the off-trace path. A store can never move above a split; a load L can move above a split only if the register variable written by L is dead on the off-trace path. The disambiguation for location conflicts involving the load L may be incorrect for executions which take the off-trace path, but it is in precisely these situations that we do not care about the value returned by the load.

The bank-stall feature of the TRACE allows the instruction scheduler to use the disambiguator as a heuristic for bank conflicts when a memory reference has been moved above a split. The instruction scheduler assumes that the on-trace path is the most likely one to be followed at run-time. If it is, all bank-disambiguation questions are well-defined and the answers will be correct. If the program jumps off the trace, it may take some unexpected bank stalls. Experience has shown that the disambiguator's answer is usually correct when the off-trace branch is taken.

13.8.2 Tracking motion above splits

In his thesis [54], Nicolau proved that after one iteration of the trace scheduling algorithm (pick a trace, schedule, and update the flow graph) on a flow graph *P,* each path in the new flow graph *P'* corresponds to a path in *P.* A new path may contain some extra operations whose results are eventually discarded; an examination of Nicolau's proof shows that these operations are the result of motion above splits. By induction, we can see that each trace corresponds to a path in the Phase 2 flow graph, with some extra operations which have moved above a split. By tracking motion above splits, we can identify operations which may not be disambiguated correctly.

An operation can move above a split during instruction scheduling or when the trace scheduler creates compensation code. If it happens during instruction scheduling, the instruction scheduler is aware of the motion when it queries the disambiguator; it also records this fact in the record it makes describing the bank-stall window around a split or a join. If an operation moves above a split while creating compensation code, the trace scheduler flags this operation as having moved above a split; this information is preserved when the operation is passed to the instruction scheduler or copied again.

14 Competitive Evaluation

In this section we compare the Trace 14/300 to two contemporary systems: the Convex C210 and the MIPS M/1000. First, we summarize the hardware characteristics of all three systems. Second, we look at the relative performance of the 14/300 and C210 on mini-supercomputer workloads. Third, we compare all three systems on the common ground of synthetic scientific benchmarks. We would prefer to compare the systems on real applications, or at least on large standard benchmarks like the SPEC suite. But the systems were sold into largely non-overlapping markets with different important applications, and Convex has not published SPEC results for the C210. Finally, we present an analytical model of the 14/300 and M/1000; this model captures the ability of the compilers and the systems to exploit operation level parallelism.

These comparisons lead us to conclude that a VLIW architecture with a trace scheduling compiler is superior to a vector architecture for single stream scientific problems. A VLIW can equal a vector machine on vector codes and outperform it on floating point and integer scalar codes. For systems applications, there is instruction level parallelism to be exploited by a modest scale VLIW. However, it was not sufficient to overcome the architectural and organizational advantages of a RISC system tuned for these types of applications.

14.1 Competitive System Overview

The C210 and M/1000 are quite different processors, designed for different markets, and optimized for different workloads. Figure 14-1 summarizes their characteristics and compares them with the 14/300.

The Convex C210 is a vector processor [13]. It was the mini-supercomputer market leader in 1988-91 and a direct competitor with the Trace 14/300. The C210 is a hardware technology generation ahead of the 14/300, with 150% faster gate speed and a 63% faster cycle time. The C210 has higher peak megaflops but lower peak memory bandwidth. The C210 has a data cache; it is used for scalar references and bypassed by vector references. Convex has excellent vector compiler technology [51]. The C210's scalar performance is difficult to characterize; Convex has not released the standard scalar benchmarks (e.g., the integer SPECmarks). Within a year of its introduction, the C210 could be extended into a multiprocessor system with up to four processors (the C240). We are concerned with single processor performance, and we will focus on the C210.

The MIPS M/1000 is based on the MIPS R2000/R2010 microprocessor chipset [40], the fastest circa-1987 RISC processor. MIPS also has excellent compiler technology [16]. The M/1000 was marketed as a network server. It was designed to maximize price/performance on general purpose workloads within the constraints of a 2-chip CPU. The 14/300 and the M/1000 have comparable cycle times. The Trace does not have a data cache; the Mips does not have an interleaved memory system. The load latency of the Trace is much longer than the latency of a cache hit on the Mips. The floating latencies are roughly comparable, but the Mips floating units are not fully pipelined. The peak megaflops and memory bandwidth of the M/1000 are much lower than the 14/300. Unfortunately, we do not have complete performance data on the M/1000, and in Section 14.3 we substitute performance data for the MIPS M/120-5. The M/120-5 was released shortly after the M/1000; it too is R2000/R2010 based and has a similar cache subsystem, but it cycles at 60ns rather than 66.7ns.

Processor	*Trace 14/300*	*Convex C210*	*MIPS M/1000*
Year of Introduction	1987	1988	1987
Architecture	VLIW	RISC?+Vector	RISC
Cycle Time	65ns	40ns	66.7ns
Peak Operation Issue Rates (per cycle)			
All ops	7	6[1]	1
Memory ops	2	1	1
Floating ops	2	2	0.5-0.2
Bandwidth			
Memory (Mb/sec)	246	200	60
Floating (Mflop/sec)	30	50	3-7.5
Latencies (cycles)			

[1] Measures the C210's vector unit.

Load	7	2/12[1]	2
Flop	4	2	2-5
Branch	2	2	2
Instruction Cache Size			
Bytes	512K	8K	64K
Instructions	8K	4K	16K
Operations	112K	64K[1]	16K
Data Cache (bytes)	—	4K	64K
Vector Registers (bytes)	—	8K	—
Number of Banks	64	32	—
Bank-Stall Penalty (beats)	4	?	—

Figure 14-1: Summary of the Trace 14/300, Convex C210, and MIPS M/1000

14.2 Performance on Mini-Supercomputer Workloads

Mini-supercomputers are a class of computer systems designed in the 1980's primarily to run the simulation component of computer aided design applications (CAD). These simulations embody the theories and techniques of disciplines including physics, mechanical engineering, electrical engineering, chemistry, physics, mathematical signal processing, and geology. The codes that expressed these simulations have been under development for some time (thirty years in some cases) and have grown to be quite large, complex, and functional computer programs. When Multiflow was founded in 1984, it was widely believed that the increase in use of CAD would result in a tremendous increase in the demand for high speed computers specifically designed to run these simulations. By early 1987, the market for such systems had attracted roughly 30 companies, each intent on delivering compute servers designed to sell in the $200,000-$500,000 price range. Market analysts divided the machines produced by these companies into two classes: the *mini-supers* that could run the existing simulation codes (also known as *dusty decks*) with little or no modification, and the *parallel processors* that would require the development of new simulation codes.

Convex was the mini-super market leader, followed by Alliant, SCS, FPS, and a host of others. Multiflow was late to the market, yet managed to break in and grow to become the third largest vendor. While an exhaustive exploration of the reasons for Multiflow's relative success would have to consider many factors that aren't relevant to this paper (e.g., relative financial viability), we believe that the essential reason Multiflow managed to do so well is technical, and thus quite relevant. Multiflow was able, using its architecture and its compiler technology, to exploit fine grained parallelism in the simulation codes in a way that the other vendors were not; this ability fostered the company's success, albeit a limited and temporary success.

Figure 14-2 contains a set of benchmark times for the 14/300 and C210 drawn from the mini-supercomputer workload of mechanical engineering, computational chemistry, and signal processing applications. Nastran and Ansys are large commercial packages (approximately 500,000 source lines) for performing structural analysis using the finite element method. Though similar, they are worth considering separately because their respective developers have quite different policies for allowing modifications by computer systems vendors: the vendor of Ansys, Swanson Analysis, allows vendors virtually unlimited time for a port and will consider quite extensive modifications and extensions to source (Convex replaced Ansys's simultaneous equation solver with its own 10,000 line package); the vendor of Nastran, MacNeal-Schwendler, typically allows vendors almost no access to the application source code, and thus accepts only limited modifications. The relative performance of the 14/300 is better on Nastran than Ansys, due to the dusty deck nature of Nastran.

Gaussian86 is a computational chemistry package of roughly 200,000 lines that does *ab initio* molecular modeling. Amber is a molecular mechanics and dynamics program, Mopac is a semi-empirical electronic structure calculation program, and Prolsq performs X-ray crystallography calculations. The 14/300 outperforms the C210 on all of the chemistry codes.

The CFFT application is a signal processing kernel that performs single precision complex fast Fourier transforms of varying sizes. Both the 14/300 and C210 use handcoded subroutine libraries for this application, but the VLIW architecture is a better match to the needs of the FFT algorithm and the 14/300 fully utilizes its memory and floating point bandwidth [65].

Benchmark Name	*Test Case*	*Trace 14/300*	*Convex C210*
Mechanical Engineering (seconds)			
Nastran	BCELL3	9.00	9.14
	BCELL4	17.00	19.62
	BCELL5	36.00	38.29
	BCELL6	71.00	72.52
	BCELL8	261.00	220.58
	BCELL9	345.00	369.00
	BCLL10	568.00	612.00
	BCLL14	4687.00	3587.61
Ansys	S-1	134.00	114.00
	S-2	733.00	575.00
	S-3	1849.00	1713.00
	S-4	3760.00	3811.00
	S-5	3927.00	3637.00
	M-1	14.00	13.00
	M-2	129.00	164.00
	M-3	915.00	1020.00
Computational Chemistry (seconds)			
Gaussian86	Malon2	1879.00	2193.00
Amber	Case1	339.00	420.00
	Case2	732.00	738.00
	Case3	515.00	580.00
Mopac	SCF	2.50	2.90
Prolsq	Columbia	3846.00	4038.00
Signal processing (milliseconds)			
CFFT	1024 elem	0.90	2.25
	256 elem	0.21	0.55
	64 elem	0.06	0.18

Figure 14-2: Performance on Mini-Super Workload: 14/300 vs. C210.

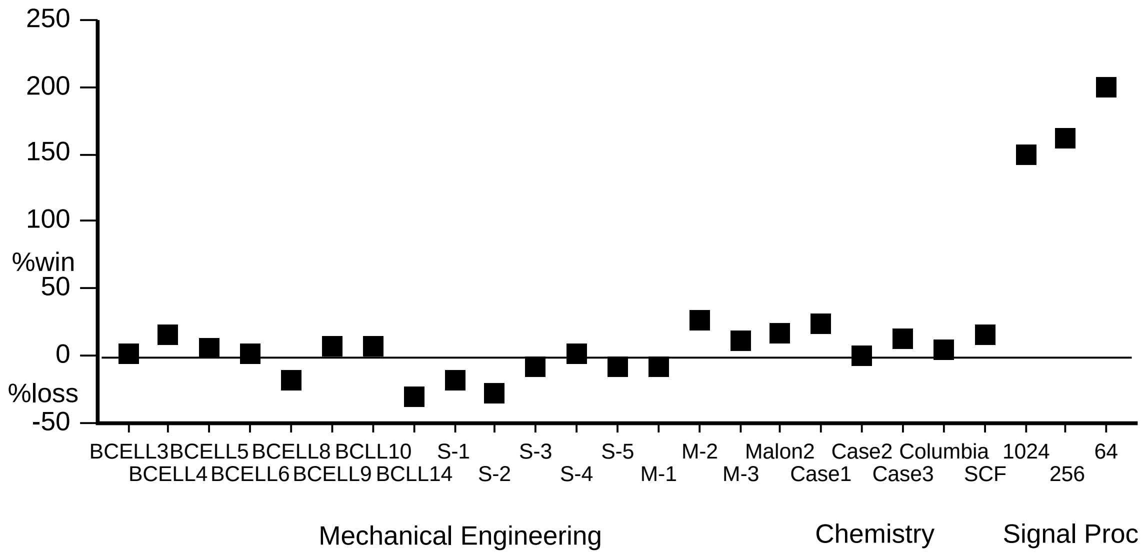


Figure 14-3: Relative performance of 14/300 vs. C210.

Figure 14-3 graphically summarizes the performance data of Figure 14-2. The benchmarks are listed along the abscissa and a square is plotted at that point along the ordinate that represents the percentage by which the 14/300 is faster or slower than the C210. The percentages are computed symmetrically. Suppose, for example, that the 14/300 ran an application in 100 seconds: if the C210 ran it in 200 seconds the percentage would be +100%; if the C210 ran it in 50 seconds the percentage would be –100%.

The 14/300 is a hardware technology generation behind the C210 and manages to eke out rough single processor performance parity. Performance parity was no guarantee of market success. Multiflow's 500 series, due out in late 1990, was expected to achieve a 4.5x performance step; Convex's C380 series, built out of gallium arsenide and delivered in late 1991, has achieved a 2.5x performance step on vector codes. A 500 series processor would have been about twice the performance of a C380 processor.

Many of the mini-supercomputer applications are well-suited for multiprocessing. The lack of multiprocessing was a weakness of the 14/300, particularly in competitive situations with Convex. The 500 series would have introduced a dual processor capability, where a fully configured machine could run either as a dual processor 14-wide or a single processor 28-wide.

14.3 Performance on Synthetic Scientific Workloads

Linpack [22] and the Livermore FORTRAN kernels [50] are the standard scientific kernel benchmarks for both mini-supercomputers and RISC-based systems. These benchmarks are small and have received a large amount of attention from each vendor, so they are indicative of peak system performance on tuned code. Figure 14-4 shows the circa-1988 results of these benchmarks for the Trace 14/300, the Convex C210, and the MIPS M/120-5.

Benchmark Name	*Test Case*	*Trace 14/300*	*Convex C210*	*MIPS M/120-5*
Livermore	1	25.14	29.00	2.95
FORTRAN	2	9.05	2.70	2.55
Kernels	3	24.04	20.60	2.97
(LFK)	4	14.79	11.00	2.98
	5	2.87	1.90	2.04
	6	3.32	4.00	1.93
	7	23.78	36.60	3.88
	8	10.92	26.00	3.53
	9	24.82	33.80	3.58
	10	9.74	10.00	1.41
	11	2.35	1.50	1.48
	12	8.25	7.80	1.50
	13	3.89	1.10	0.87
	14	4.92	2.30	0.92
	15	3.00	4.00	1.05
	16	1.73	1.60	1.51
	17	4.35	3.70	2.55
	18	16.61	19.90	3.16
	19	3.09	3.20	2.88
	20	4.56	2.10	2.67
	21	14.12	30.40	2.30
	22	11.18	12.30	1.52
	23	5.94	4.00	3.36
	24	5.24	12.70	0.94
LFK means	Average	9.90	11.75	2.27
	Geometric	7.26	6.74	2.06
	Harmonic	5.38	3.94	1.84
Linpack	100x100	17.00	17.00	2.10
	1000x1000	42.00	44.00	3.60

Figure 14-4: Synthetic Scientific Workload: 14/300, C210, and M/120-5

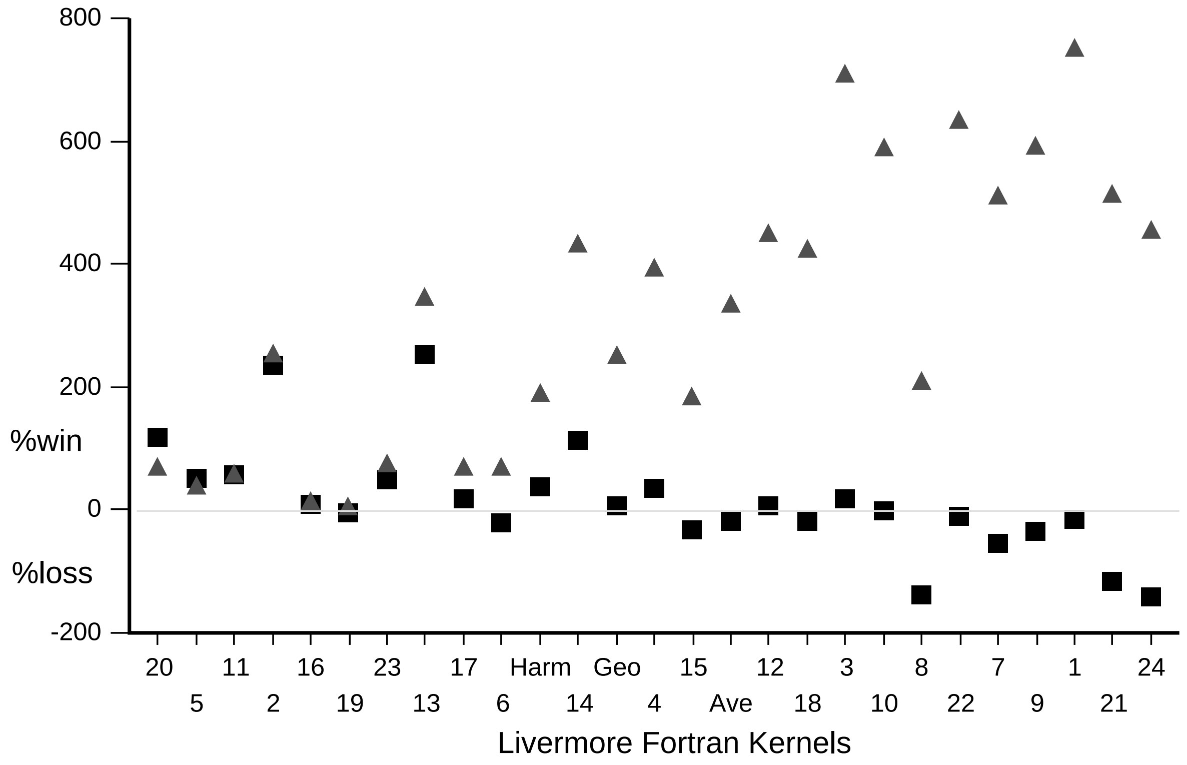


Figure 14-5: Comparison of LFK performance of 14/300 vs C210 (square) and 14/300 vs M/120-5 (triangle). Comparison ordered left to right by relative performance of C210 vs M/120-5.

Figure 14-5 graphically compares these performance results. The relative performance varies with the traditional scalar/vector mix; to help clarify this trend, the LFK results have been sorted from left to right based on the vector C210's performance vs the scalar M/120-5. The left side of the chart shows the relative performance of the 14/300 vs the C210 and M/120-5 on more scalar codes, the right hand side compares the performance on more vectorizable codes. If the twelve least vectorizable LFK's are taken to measure scalar floating point performance and the twelve most vectorizable are taken to measure vector floating point performance, the scalar and vector floating point harmonic mean mega-flops are as follows:

	14/300	*C210*	*M/120-5*
Scalar FP	3.71	2.31	1.78
Vector FP	9.80	13.33	1.90

Comparing the systems pairwise shows the 14/300 to be a faster scalar floating point processor than either the C210 or M/120-5, a much faster vector floating processor than the M/120-5, but not quite as good a vector processor as the C210.

	14/300 vs C210	*14/300 vs M/120-5*	*C210 vs M/120-5*
Scalar FP	+ 60%	+ 108%	+ 30%
Vector FP	− 36%	+ 418%	+ 601%

If the differences in implementation technologies are factored out by normalizing all three machines linearly to the same cycle time, one sees the following relative performance:

	14/300 vs C210	*14/300 vs M/120-5*	*C210 vs M/120-5*
Scalar FP	+ 161%	+ 126%	− 15%
Vector FP	+ 22%	+ 460%	+ 400%

The normalized 14/300 is a substantially faster scalar floating point processor than either the C210 or M/120-5, a much better vector processor than the M/120-5, and a slightly better vector processor than the C210. While the C210 far outperforms the M/120-5 on vector floating point, it is not as fast a scalar floating point engine.

14.4 Analyzing Parallelism on General Purpose Workloads

In the fall of 1989, two of the authors worked with Joel Emer of Digital Equipment Corporation and Richard Lethin of Multiflow Computer to create a performance analysis of the Trace machines. We developed a performance model and used it to compare the Trace 14/300 with the Mips M/1000. The analysis measured the achieved parallelism of the Trace and Mips across a set of benchmarks. We discovered that the Trace compiler found significant parallelism in all code, averaging 2 to 11 scheduled operations in flight across runs of entire programs.

14.4.1 The performance model

The performance model defines the execution time of a program as the product of cycles_executed and cycle_time. Cycles_executed is partitioned into scheduled_cycles, which is the compiler's static schedule of the CPU pipeline and dynamic_cycles, which are dynamic stalls introduced by the memory system.

$$T = (\text{scheduled\_cycles} + \text{dynamic\_cycles}) * \text{cycle\_time}$$

This equation can be rewritten as follows, where the dynamic cycles are expressed as an overhead factor, and the scheduled cycles are expressed in terms of scheduled parallelism.

$$T = O/P * (1 + X) * C$$

T	Execution time
O	Number of operations
P	Average scheduled parallelism: Ops/scheduled_cycle Cycles include all "compile-time" interlocks but ignore dynamic stalls due to memory system
X	Execution overhead I-cache stall + D-cache stall + Bank stall
C	Cycle time

P, the scheduled parallelism, can be rewritten as a ratio of scheduled operations-in-flight to average scheduled latency, using Little's law [48]. By exposing the latency, we can measure the parallelism that the compiler is actually finding, the average number of scheduled operations in flight.

$$P = D/L$$

D	Scheduled operations in flight
L	Average scheduled latency

Substituting for P, we get a final equation, which we use in the analysis below

$$T = O * L/D * (1 + X) * C$$

This equation identifies the key components of single processor performance.

- C, the cycle time.
- O, the number of operations executed.
- L, the average latency of operations, ignoring the effects of the memory system.
- D, the instruction level parallelism.
- X, the overhead of the memory system.

It also highlights the fact that both the operation issue rate and the latency of operations contribute equally to parallelism, and in particular, as the latency of operations increases, the compiler must find more parallelism for the performance of the system to remain the same. This is described as *supersymmetry* between *superscalar* and *superpipelining* in [38].

14.4.2 Gathering the data

We used the following techniques to gather the data.

- Time.
  - T, the execution time, was measured by running the programs on stand-alone systems.
  - C, the cycle time, was known for each machine, and confirmed with simple timing loops.
- Scheduled cycles. We used the MIPS pixie tool [35] to instrument the MIPS executable and measure the scheduled cycles. We wrote a similar tool for the Multiflow 14/300. These tools can measure the total number of scheduled cycles (ignoring memory effects), the total number of operations executed, and the dynamic distribution of operations by opcode:
  - O, the number of operations, is a direct output of pixie.
  - P is O/scheduled_cycles, and scheduled cycles is also measured by pixie.
  - L = Σ (op_latency * op_count)/O, where op_count, the dynamic count of each type of operation, is a direct output of pixie, and op_latency, the latency of each type of operation, is published in the architecture manuals for the machines.
  - D = P*L
- Dynamic cycles.
  - X can be computed from T, C, and scheduled_cycles

    X = (T – C*scheduled_cycles) / C*scheduled_cycles
  - On the Multiflow Trace we could measure X's component parts. Hardware counters measured the stall due to bank conflicts and instruction cache miss.
  - On the MIPS, we could not partition the dynamic cycles into data cache miss and instruction cache miss.

14.4.3 The programs

We measured four classes of programs, as shown in Figure 14-6. These are most of the SPEC89 benchmarks [66], with the addition of Linpack, which is a well-known vector benchmark [22], and grep, which is a version of the Unix utility tuned for the Multiflow machine. For Linpack, we modeled in the source a blocking transformation that the Multiflow optimizer was performing, but the MIPS optimizer was not, and compiled and ran the blocked Linpack for both machines.

Vector Floating		
	Linpack	100x100 Gaussian elimination
	Tomcatv	Fluid flow analysis
Scalar floating		
	Doduc	Nuclear calculations
	Fpppp	Quantum chemistry
	Spice2g6	Circuit simulator
Vector Integer		
	Grep	Unix pattern search
	Eqntott	CAD tool: convert boolean equations to truth table
Scalar Integer		
	Li	Lisp interpreter
	Gcc	C compiler
	Espresso	CAD tool: PLA generator and optimizer

Figure 14-6: Benchmark programs

14.4.4 The results

A complete set of results are shown in Figure 14-7, 14-8, 14-9, and 14-10. Operations O are reported in millions, cycles C are reported in nanoseconds, and time T is reported in seconds.

Benchmark	*O(M)*	*D*	*L*	*1+X*	*C(ns)*	*T(s)*
espresso	1627.27	2.40	2.14	1.11	65.00	104.59
gcc	167.35	1.86	2.22	1.54	65.00	19.94
li	1269.18	1.98	2.50	1.05	65.00	109.76
eqntott	1226.03	5.49	2.40	1.40	65.00	48.73
grep	2.36	3.52	1.99	1.02	65.00	0.08
doduc	1368.57	3.36	2.40	1.72	65.00	109.00
fpppp	500.83	7.12	3.33	1.61	65.00	19.60
spice2g6	3469.02	2.88	2.67	1.33	65.00	277.00
linpack	61.96	10.86	3.55	1.26	65.00	1.66
tomcatv	1249.22	11.94	4.00	1.32	65.00	35.90

Figure 14-7: 14/300 Benchmark Performance

Benchmark	*O(M)*	*D*	*L*	*1+X*	*C(ns)*	*T(s)*
espresso	1029.28	1.23	1.39	1.12	66.70	87.10
gcc	117.88	1.25	1.46	1.83	66.70	16.80
li	926.10	1.20	1.55	1.36	66.70	107.80
eqntott	1062.71	1.30	1.52	1.73	66.70	143.30
grep	2.16	1.47	1.58	1.94	66.70	0.30
doduc	1451.67	1.39	2.31	1.31	66.70	211.00
fpppp	538.68	1.55	2.06	2.41	66.70	114.70
spice2g6	3229.00	1.22	1.55	1.38	66.70	379.00
linpack	91.07	1.66	2.02	2.59	66.70	19.10
tomcatv	1673.07	1.78	2.06	2.68	66.70	345.00

Figure 14-8: M/1000 Benchmark Performance

Benchmark	*O(M)*	*D*	*L*	*1+X*	*C(ns)*	*T(s)*
espresso	0.63	1.96	0.65	1.01	1.03	0.83
gcc	0.70	1.49	0.66	1.19	1.03	0.84
li	0.73	1.64	0.62	1.29	1.03	0.98
eqntott	0.87	4.22	0.63	1.23	1.03	2.94
grep	0.92	2.40	0.79	1.90	1.03	3.60
doduc	1.06	2.42	0.97	0.76	1.03	1.94
fpppp	1.34	4.58	0.62	1.50	1.03	5.85
spice2g6	0.93	2.37	0.58	1.04	1.03	1.37
linpack	1.47	6.53	0.57	2.05	1.03	11.51
tomcatv	1.34	6.69	0.51	2.03	1.03	9.61

Figure 14-9: Speedup from M/1000 to 14/300

Benchmark	*1+X*	*IC miss*	*Bank Stall*
espresso	1.11	0.08	0.03
gcc	1.54	0.50	0.04
li	1.05	0.05	0.00
eqntott	1.40	0.21	0.19
grep	1.02	0.02	0.00
doduc	1.72	0.71	0.01
fpppp	1.61	0.49	0.12
spice2g6	1.33	0.10	0.22
linpack	1.26	0.13	0.13
tomcatv	1.32	0.06	0.26

Figure 14-10: 14/300 decomposition of X

We aggregate the results by normalizing each program to 100 seconds of runtime on the MIPS M/1000; this prevents a bias towards vectors programs which run much faster on the Multiflow machine. Figure 14-11 presents the aggregate summary. The first two rows present the model data for the Mips and the Trace, and the bottom row presents the ratio of their performance. On average, the Trace machine was 1.8x faster than the Mips. The biggest contributor to the speed was D, the scheduled-ops-in-flight. On average, the Trace maintained 3 scheduled operations in flight. The average operation latency L was less for the Mips, but the Multiflow machine had much less dynamic overhead in the memory system (1+X).

	O(M)	D	L	1+X	C(ns)	T(s)
M/1000	7175.45	1.36	1.69	1.69	66.67	1000.00
14/300	8283.86	2.99	2.48	1.26	65.00	562.19
Speedup	0.87	2.20	0.68	1.34	1.03	1.78

Figure 14-11: Normalized aggregate for all ten benchmarks

This comparison yields dramatically different results depending on the workload.

Workload	*Speedup*
Scalar integer	0.88
Vector integer	3.30
Scalar floating point	2.12
Vector floating point	10.47

Figure 14-12: Speedup from M/1000 to 14/300 by workload

Figure 14-13 presents the results of the vector programs. The Trace is 10.5x faster than the Mips. The largest factor is due to scheduled operations in flight (D). The Trace averages 11.44 scheduled operations in flight across these benchmarks. This is a significant achievement, for we are not just measuring the inner loops, but the entire run of the program. The other large factor in the performance differential is the memory system overhead, where the 14/300 is twice the speed of the M/1000. This advantage is largely offset by the shorter scheduled latencies on the Mips, which are due to the use of a data cache. The large number of extra operations performed by the Mips is due to the lack of double precision load and store operations.

	O(M)	D	L	1+X	C(ns)	T(s)
M/1000	961.75	1.72	2.04	2.63	66.70	200.00
14/300	686.49	11.44	3.79	1.29	65.00	19.10
Speedup	1.40	6.65	0.54	2.03	1.03	10.47

Figure 14-13: Aggregate for vector floating (linpack, tomcatv)

Figure 14-14 presents the results of the scalar floating programs. Here the Trace outperforms the Mips by a factor of 2. All of the performance differential is due to the parallelism achieved by the compiler (D). Figure 14-15 presents the results of the vector integer programs. The Trace compiler also finds a significant amount of scheduled parallelism here, which again accounts for the 14/300's factor of 3 performance advantage.

	O(M)	D	L	1+X	C(ns)	T(s)
M/1000	2009.62	1.36	1.93	1.58	66.70	300.00
14/300	1913.38	3.50	2.70	1.48	65.00	141.83
Speedup	1.05	2.57	0.71	1.07	1.03	2.12

Figure 14-14: Aggregate for scalar floating (doduc, fpppp, spice2g6)

	O(M)	D	L	1+X	C(ns)	T(s)
M/1000	1461.60	1.38	1.55	1.83	66.70	200.00
14/300	1642.24	4.42	2.20	1.14	65.00	60.67
Speedup	0.89	3.20	0.70	1.60	1.03	3.30

Figure 14-15: Aggregate for vector integer programs (grep, eqntott)

Figure 14-16 presents the results of the scalar integer programs. The 14/300 is 14% slower than the M/1000. This is largely due to two reasons. First, the Trace executes a large number of additional operations (O). This is partially explained by two architectural differences between the two machines: the Multiflow Trace does not have 8-bit or 16-bit memory reference operations or a branch-on-equal. These tasks require two operation sequences on the Trace and can be done in one operation on the Mips. Second, the data cache on the Mips significantly lowers the average scheduled latency of operations (L). This offsets the scheduled parallelism (D) found by the Trace compiler.

	O(M)	D	L	1+X	C(ns)	T(s)
M/1000	2742.48	1.22	1.46	1.38	66.70	300.00
14/300	4041.76	2.11	2.26	1.21	65.00	340.59
Speedup	0.68	1.72	0.64	1.14	1.03	0.88

Figure 14-16: Aggregate for scalar integer programs (espresso, gcc, li)

Figure 14-17 presents the scheduled operations in flight measured for all of these benchmarks. For all of the programs measured, the Trace compiler finds a significant amount of parallelism relative to a RISC machine.

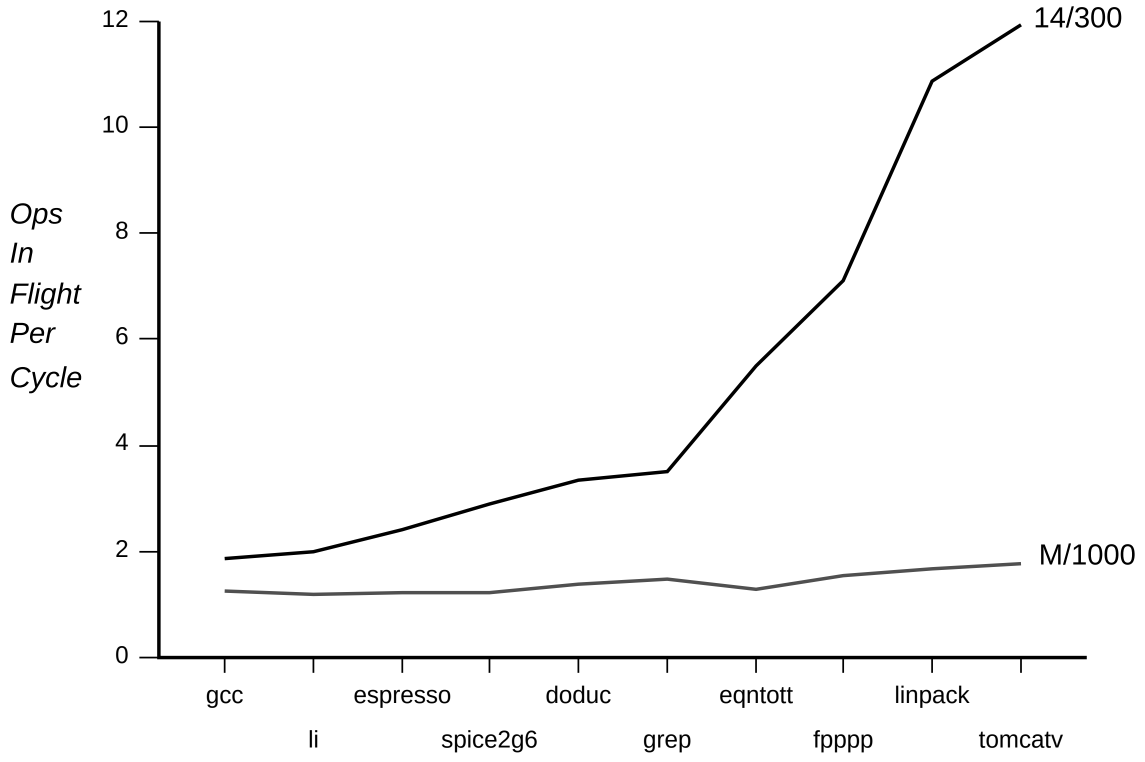


Figure 14-17: Operations In flight per cycle

15 Compiler Evaluation

15.1 Displaying the DPG

In the course of developing the compiler we used a directed acyclic graph drawer to display the DPG constructed by the instruction scheduler. These drawings graphically display the parallelism exposed by the compiler. In the DAGs, nodes are operations, solid lines are operand edges, and dotted lines are either constraining edges or memory conflicts. Inputs to the DPG are drawn at the top of the DAG, and outputs are at the bottom.

By looking at various compilations of daxpy, we can illustrate the effectiveness of our loop unrolling strategy. In Figure 15-1, we show the inner trace of daxpy, with no loop unrolling. There is very little parallelism within a single loop body. Figure 15-2 shows the inner trace of daxpy, postconditioned by 8. The graph clearly illustrates the parallelism of this vector kernel. Figure 15-3 shows daxpy unrolled by 8 with the loop exits left in, as the Trace compiler does by default. It shows an equal amount of parallelism to the postconditioned daxpy, with the exception of the constraint between stores and the preceding loop exit. Figure 15-4 shows the same daxpy unrolled by 8, but with only "traditional" optimizations applied; none of the Multiflow optimizations to eliminate dependencies between unrolled loop bodies have been performed. The amount of parallelism is heavily constrained. On vector kernels the Multiflow compiler effectively eliminates all of the dependencies between loop bodies due to induction variables; the resulting unrolled loop has as much parallelism as if it had been postconditioned.

```
do i = 1,n
dy(i) = dy(i) + da*dx(i)
end do
```

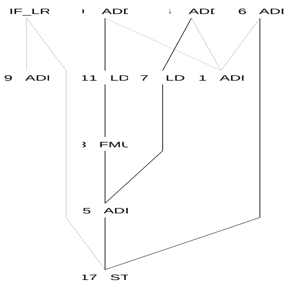


Figure 15-1: Inner trace of DAXPY with no unrolling

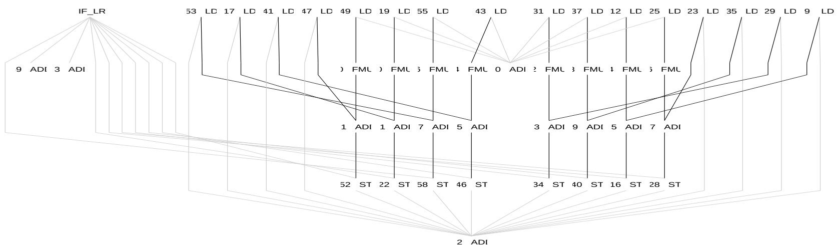


Figure 15-2: Inner trace of DAXPY postconditioned by 8

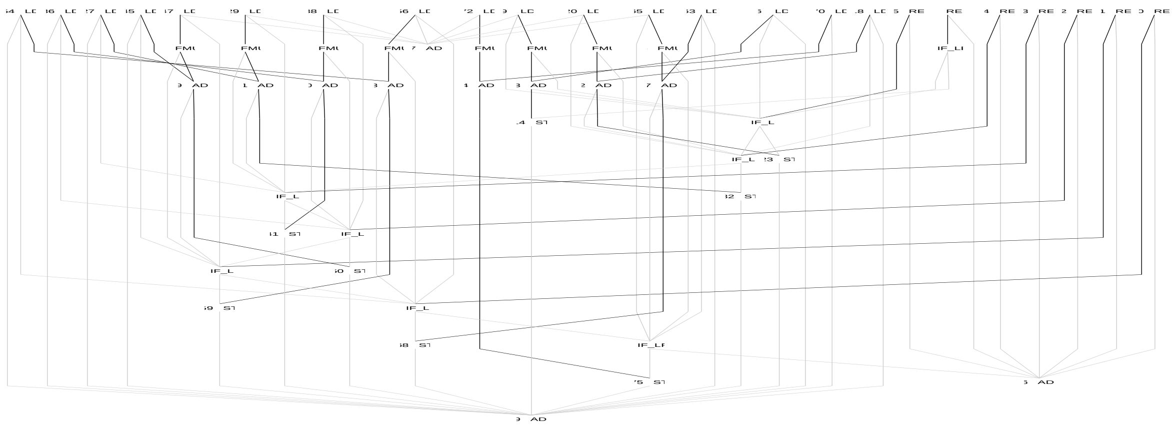


Figure 15-3: Inner trace of DAXPY unrolled by 8

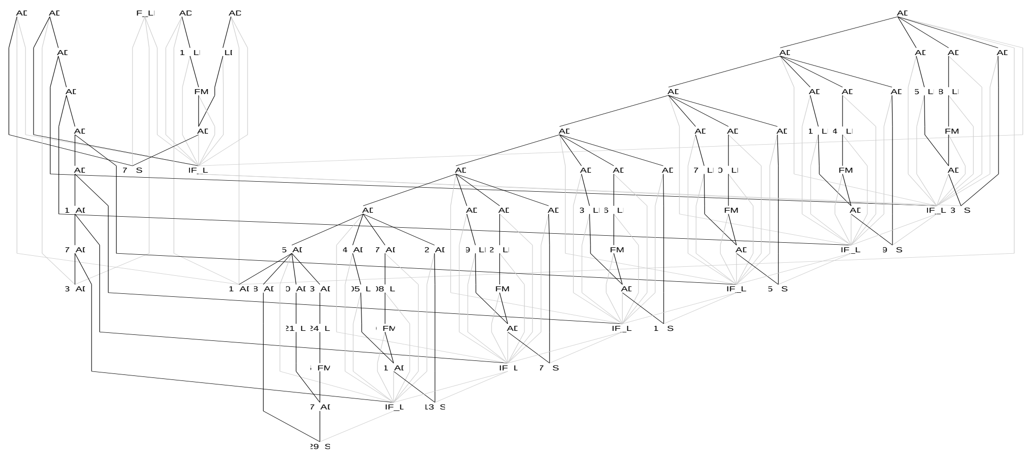


Figure 15-4: Inner trace of DAXPY unrolled by 8 with only traditional optimization

Figure 15-5 shows the large amount of parallelism in a vectorizable kernel from the Livermore loops. It is unrolled 16 times, which is the default for the 14/300. Stores constrained by exit tests are the only limitation to parallelism. Figure 15-6 shows a surprising amount of parallelism from the inner trace of a Livermore kernel 16, unrolled 4 times by the compiler. The control flow in the loop body appears to inhibit parallelism, but with feedback from previous runs, the compiler can unroll the loop and select the dominant path.

```
      do 1 l = 1,loop
      do 1 k = 1,n
1     x(k)= q + y(k)*(r*zx(k+10) + t*zx(k+11))
```

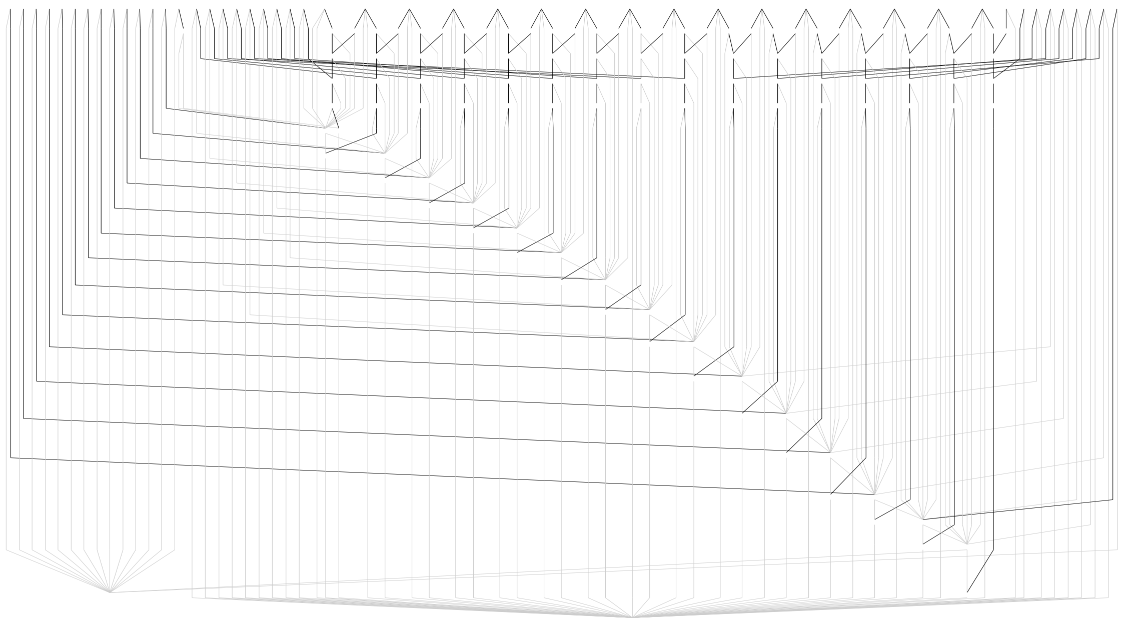


Figure 15-5: LFK Kernel 1

```
        ii= n/3
        lb= ii+ii
        k2= 0
        k3= 0
        c
        do 485 l= 1,loop
        m= 1
405     i1= m
410     j2= (n+n)*(m-1)+1
        do 470 k= 1,n
        k2= k2+1
        j4= j2+k+k
        j5= zone(j4)
        if( j5-n ) 420,475,450
415     if( j5-n+ii ) 430,425,425
420     if( j5-n+lb ) 435,415,415
425     if( plan(j5)-r) 445,480,440
430     if( plan(j5)-s) 445,480,440
435     if( plan(j5)-t) 445,480,440
440     if( zone(j4-1)) 455,485,470
445     if( zone(j4-1)) 470,485,455
450     k3= k3+1
        if( d(j5)-(d(j5-1)*(t-d(j5-2))**2+(s-d(j5-3))**2
  .     +(r-d(j5-4))**2)) 445,480,440
455     m= m+1
        if( m-zone(1) ) 465,465,460
460     m= 1
465     if( i1-m) 410,480,410
470     continue
475     continue
480     continue
485     continue
```

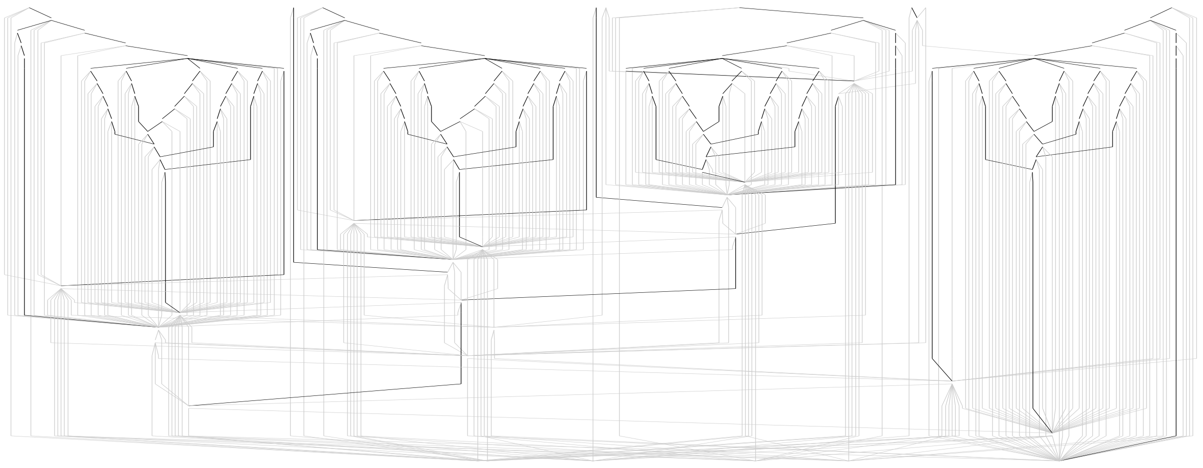


Figure 15-6: LFK Kernel 16

Figure 15-7 shows the parallelism in the inner trace of a recursive C program, with no inlining It is not as large as in kernels above, but there is a significant amount.

```
 /* The eight queens problem */
Try(i, q, a, b, c, x)
int i, *q, a[], b[], c[], x[];
{
        int j;
        j = 0;
        *q = false;
        while ( (! *q) && (j != 8) ) {
                j = j + 1;
                *q = false;
                if ( b[j] && a[i+j] && c[i-j+7] ) {
                        x[i] = j;
                        b[j] = false;
                        a[i+j] = false;
                        c[i-j+7] = false;
                        if ( i < 8 ) {
                                Try(i+1,q,a,b,c,x);
                                if ( ! *q ) {
                                        b[j] = true;
                                        a[i+j] = true;
                                        c[i-j+7] = true;
                                        }
                        }
                        else *q = true;
                }
        }
}
```

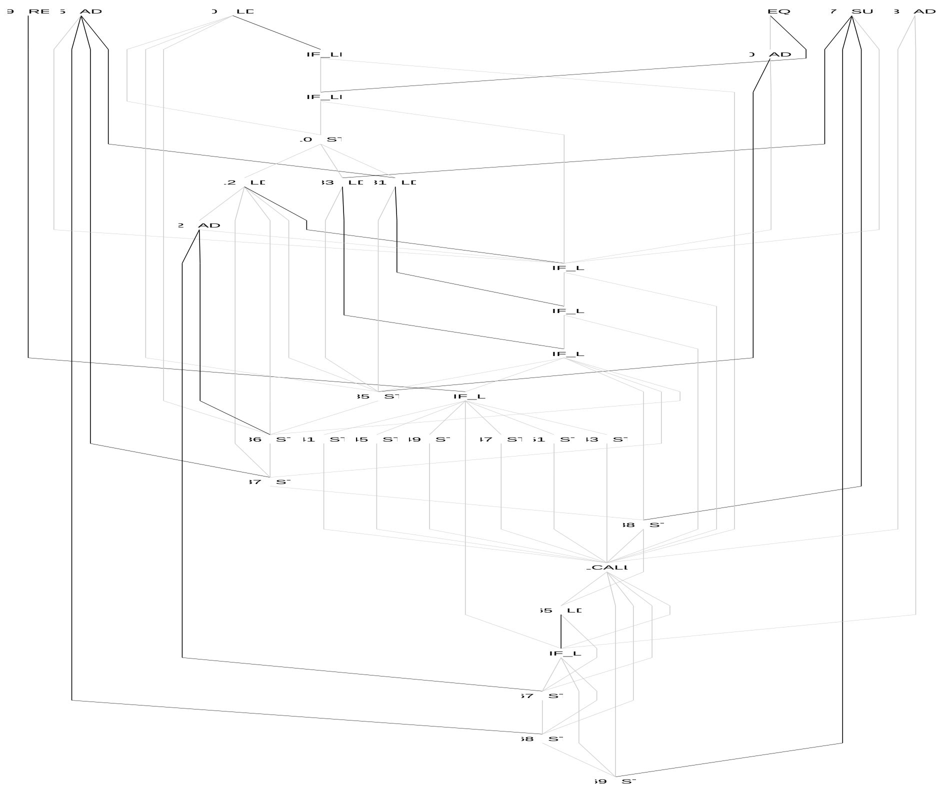


Figure 15-7: Eight Queens

15.2 Compile speed

The Multiflow compiler is slow. To measure the speed of our compiler we developed a compile speed metric, which is a 5855 line FORTRAN program composed of routines culled from various applications, combined into a single file. All comments and blank lines have been stripped from the file, so that every line is a source line. The results of compiling the metric are shown in Figure 15-8. The chart presents the speed of compiling the metric with the Multiflow FORTRAN compiler, running on a Multiflow 14/300, targeting the three widths of the 300 series, with the optimizer on and off. It also presents the speed of the native Mips compiler, running on a DECstation 3100. The 3100 is based on the Mips R2000, and is slightly faster than an M/1000. As we demonstrated in section 14, a Mips M/1000 is a somewhat faster than a 14/300 for systems applications.

	Compile Speed (Lines Per Minute)			
	7/300	*14/300*	*28/300*	*Mips*
optimized	815	620	390	3220
checkout	2080	—	—	8610

	Performance Ratio			
	7/300	*14/300*	*28/300*	*Mips*
optimized	3.95	5.19	8.26	1.0
checkout	4.14	—	—	1.0

Figure 15-8: Compile speed of FORTRAN metric

Many factors contribute to the slow speed of the compiler. Foremost was a design decision in the implementation: because of the ambitious nature of the project, we separated the major phases of the compiler with narrow functional interfaces. As a result, the source for a program built with the Multiflow compiler passed through six distinct representations, with information often recomputed at each stage:

A. the pcc trees in the front end,

B. the flow graph of IL-1 passed to the optimizer,

C. the flow graph of IL-2 produced by the optimizer,

D. the flow graph built by the trace scheduler,

E. the DPG built by the code generator,

F. the machine level representation built to emit code.

The overhead of the interfaces and multiple representations shows up in the speed of our checkout compiler, which is four times slower than the checkout compiler for the DECstation 3100. The checkout compiler bypasses the trace scheduler and substitutes its own code generator, so a program compiled with it has only 5, rather than 6, representations.

The next factor in compile speed is the amount of unrolling the compiler performs and the width of the target machine. The ratio of the optimized DECstation 3100 compile speed to the 7/300 compilation is 4.15, roughly the same as the ratio of the checkout compiler speeds. Arguably, the 7/300 compile speed problem is entirely explained by our interface overhead. However, the 14/300 is 30% slower than the 7/300, and the 28/300 is 50% slower than the 14/300. This is because we unroll much more heavily for the wider machines, and, in the instruction scheduler, we have many more functional units and register banks to consider.

The third factor in compile speed is the algorithms used in the compiler. Compared to most other compilers, our optimizer performs more repeated analysis and optimizations and uses more expensive algorithms. In particular, our common subexpression and loop invariant implementation is much slower than the partial redundancy algorithm used in the Mips compiler [14, 52], or the Reif and Tarjan algorithm [62, 63, 64]. In addition, our instruction scheduler schedules much longer sections of code for a much wider machine than the Mips compiler, and makes two scheduling passes. Attention was paid to the complexity of the scheduling algorithms, but the constant factors are large. The trace scheduler is also not implemented as efficiently as it could be; its flow graph nodes are individual operations rather than basic blocks.

The final factor in the compile speed is the coding style used in the compiler. A high-level object-oriented style is used throughout, and, particularly in the trace scheduler and code generator, a Lisp-like style is used. This high-level style enabled us to write a large program with a small number of engineers, but it contributed to the memory usage and speed of the result.

16 Conclusions

In retrospect, we have formed the following opinions.

Fine grained parallelism is practical. A fine-grained parallel processor can be effectively scheduled by a compiler with relatively little input from the applications programmer. Multiflow developed a good system for exploiting fine grained parallelism in scientific applications. It is not a particularly good one for systems applications, nor for applications written in a style or language that makes compile-time memory dependency analysis difficult. Most of these limitations are due to the design goals of the system. The machines were designed explicitly to provide the largest possible potential for parallel speedup. Many design tradeoffs arose in which cost, latency, and performance on sequential code were traded away for issue and execute bandwidth, and the ability to achieve high speeds on very parallel codes. The compiler is able to find significant amounts of static parallelism in systems applications, and we believe relatively simple implementations of current RISC instruction set architectures can be designed to exploit this parallelism.

On the Trace architecture and implementation.

- A fast cycle time is important.
- The 7-wide cluster is a nice balance of functional units that can be exploited by a compiler. Many of the authors are now pursuing similar designs, though at considerably faster cycle times.
- The value of the wider machines is questionable. Poor connectivity makes them difficult to use effectively except for very parallel code. It appears these codes can be as effectively addressed with more traditional vector and parallel computers. However, a wide machine with better connectivity may be more widely applicable. Better connectivity, particularly in the register files, is important for the narrower machines as well.
- Reducing the latency of operations always helps. One of the primary incentives of the 300 series was to reduce the latency of floating point operations. A data cache would have been nice, but it was not possible to implement one that would be coherent for four highly correlated memory access streams.
- It is wise to limit the amount of processor organizational detail defined in the architecture. The compiler should schedule for the underlying organization in order to maximize performance, but it seems clear that a relatively narrow VLIW organization can be “papered-over” with a simple, stable, superscalar-style, RISC-like architecture.
- Speculative execution is important. To reach wider acceptance, a better solution to exceptions for speculative operations will need to be discovered. However, no Multiflow customer complained about the exception behavior of his optimized program.
- The select operation is a good idea. More predicated features, like the conditional store and conditional flop operations of the 500 series [20], should be added to future machines.
- Compiler scheduling of bank conflicts works reasonably well. Bank stalls will not be a major drag on performance when the memory system is highly interleaved as long as the compiler batches references to avoid risking a stall with every reference.
- The Multiflow card conflict scheme works poorly, in that it limits the usable memory bandwidth for vectors of unknown stride. This is fixed in the 500 series [39, 20]; in this design, card conflicts cause a one beat stall, not a program error.
- The Multiflow Trace machines were designed using a compiler to model the machine. This was very successful, in that for scientific applications, which were the focus of our design, the compiler and machine formed an effective system for delivering performance.

On the Trace compiler.

- Trace scheduling worked well. The algorithm is simple, and excellently described in [23]. We are surprised it has not made its way into other compilers. We have not measured its effectiveness without speculative execution. An extension to consider multiple control flow paths, as suggested in [46, 47], is a good idea, but the trace algorithm as described is an improvement over basic block schedulers.
- Feedback directed compilation is very profitable.
- The integrated register allocation and instruction scheduling strategy implements two important ideas. The first is the high priority given to allocating registers in the most frequently executed regions of code; the recently described hierarchical coloring [10] captures this idea. The second is that registers are a critical resource that need to be scheduled like other machine resources. A question is whether the three pass schedule/allocate/schedule algorithms used in the IBM and HP compilers [55, 56] can extend to more parallel machines.
- Machines with large amounts of instruction level parallelism can effectively utilize a calling-sequence register partition with a large number of scratch registers.
- Greedy schedulers do not scale to high degrees of parallelism.

- A weakness of our loop strategy is that the unrolling is done independently of the instruction scheduler. This independence, coupled with the complicated topology of the wide machines, makes the performance of the compiler somewhat fragile: unroll one too many times, and performance for a loop may drop by 10%. A software pipeliner may be able to address this problem more directly. However, it is not clear that software pipelining is practical on a machine with as many constraints as the wide Trace machines, particularly with multiple register banks, end-of-pipeline resource contention, and compiler-managed memory interleaving.
- The compiler management of the memory system works reasonably well, though we feel we could have done a better job designing our implementation. Our approach seems directly applicable to the cache memory systems found on most high-performance RISC machines, regardless of whether they execute instructions in parallel. Rather than batch references to avoid conflicts, on these machines we want to group references to the same cache line to avoid possible thrashes.
- The disambiguator is very effective for FORTRAN programs, but less so for C programs with pointers. It is not a significant drag on compile speed, which we originally feared. The assertion mechanism we designed is very difficult to use; customers wanted an IVDEP, which we eventually provided.
- The Multiflow compiler does not do the high-level loop transformations (loop interchange, loop splitting, loop fusion, outer loop unrolling, unroll-and-jam [73, 4, 9]) performed by the best vector compilers. This is a major weakness of the product. It is sometimes difficult to find enough parallelism to fill up the wide machines without these restructurings. The widespread use of the Kuck and Associates pre-processor demonstrates the value of these transformations for all machines; the Trace would have benefited as well.
- For highly vectorizable codes, the Multiflow compiler can deliver the peak speed of the machine, but it often requires more tuning than a vector compiler. For non-vectorizable codes with potential parallelism, the Multiflow compiler does much better than any other compiler (or handcoder) for instruction-level parallel machines.

17 Afterword

The Multiflow compiler was the work of many people. John Ellis was never an employee of Multiflow, but his Bulldog compiler got us off to a running start, and his coding style set a high standard for us to live up to. The engineering team was led by John Ruttenberg and Geoff Lowney, with Ruttenberg focusing on the instruction scheduler and Lowney focusing on the trace scheduler and the early phases of the compiler. The IL was designed by Tom Karzes, Geoff Lowney, Mike Ward, and Stefan Freudenberger, and implemented by Karzes and Freudenberger. Joe Rodrigue wrote an IL interpreter that was used in the inital compiler debug. The C front end was implemented by Mike Ward. The FORTRAN front end was implemented by Mike Ward, Cindy Collins, Jose Oglesby, Marshall Presser, and Bob Nix. The Fortran libraries were designed by Cindy Collins and implemented by Collins and Ellen Colwell. The *n-at-time* math library was designed and implemented by Woody Lichtenstein, Barinder Malik, and Lee Campbell. Chandra Joshi, Doug Gilmore, David Papworth and Chris Ryland wrote some of the other math functions. Tom Karzes designed and implemented the optimizer; he was later assisted by Ray Ellis. Woody Lichtenstein contributed to the design of the reductions. Stefan Freudenberger designed and implemented CSE and the code expansions. Geoff Lowney designed and implemented the trace scheduler; Thomas Gross and Mike Ward added the copy suppression optimization. John Ruttenberg and Stefan Freudenberger designed and implemented the instruction scheduler and machine model. Woody Lichtenstein worked on the extensions necessary to get good performance on the wider machines. Cindy Collins contributed to the implementation of pair mode and to the optimization of partial schedules. Cindy Collins did the original retarget of the compiler to the 300 series, and David Papworth did the original retarget to the 500 series; Stefan Freudenberger polished both. Tom Karzes invented the delayed binding algorithm, with contributions from John Ruttenberg, who also did the actual implementation. Tom Karzes designed and implemented the disambiguator. Jim Radigan wrote the checkout code generator. He also enhanced the linker. Ben Cutler wrote the assemblers and the instruction set simulators for all eight Trace models. He also wrote the trap code. Chris Genly wrote the debugger, and Ray Ellis and Brian Siritzky wrote the profiling tools. Dan Kerns wrote a static resource checker. Pat Clancy implemented an icache optimizer in the linker. Joe Rodrigue implemented the DAG drawer used to draw the figures in section 15; it was enhanced by Stefan Freudenberger and Bob Nix. Rich Lethin wrote mfpixie and did the experiments reported in section 14.4. Chani Pangali taught us the honest way to cheat at Linpack. Bob Nix and John O'Donnell led a company-wide "performance war" that gave us detailed feedback on the quality of the compiler. Cindy Collins designed our test system; Joe Rodrigue wrote many of the original tests. Neda Hajimohamadi managed testing and releases.

The Multiflow compiler technology has been purchased by Intel, Hewlett-Packard, Digital Equipment Corporation, Fujitsu, Hughes, HAL Computer, and Silicon Graphics.

Acknowledgements. Josh Fisher encouraged us to write this paper. Rich Lethin, John Ellis, and Michael Adler provided detailed reviews of early drafts.

18 Bibliography

[1] Aho, Alfred V., Jeffery D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Mass., 1977.

[2] Aiken, Alexander, and Alexandru Nicolau. Optimal Loop Parallelization. Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. Atlanta, Georgia, June 1988, pp. 308-317.

[3] Allen, F.E., J.L. Carter,J. Fabri,J. Ferrante, W.H. Harrison, P.G. Loewner,L.H. Trevillyan,The Experimental Compiling System, IBM Journal of Research and Development, 24(6),November 1980, pp 695-715.

[4] Allen, John R., and Ken. Kennedy. Automatic Loop Interchange. In Proc. ACM SIGPLAN '84 Symposium on Compiler Construction (Montreal, June 1984). ACM, New York, 1984, pp. 233-246

[5] Allen, Randy, Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. ACM Transactions on Programming Languages and Systems, 9(4), October 1987, pp .491-542.

[6] American National Standard Programming Language Fortran, American National Standards Institute, New York, New York, 1978

[7] Bannerjee, U. Data Dependence in Ordinary Programs. M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 76-837, November 1976.

[8] Bannerjee, U. Speedup of Ordinary Programs. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 79-989, October, 1979.

[9] Callahan, David, Steve Carr, Ken Kennedy, Improving Register Allocation for Subscripted Variables. ACM SIGPLAN90 Conference on Programming Language Design and Implementation. White Plains, NY June 1990. pp 53-65.

[10] Callahan, D., and Koblenz, B. Register allocation via hierarchical graph coloring. In Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation (Toronto, June 91). ACM, New York, 1991, pp. 192-203.

[11] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. Register allocation via coloring. Comput. Lang. 6(1) (January 1981), pp. 47-57.

[12] Chaitin, G. J. Register allocation and spilling via graph coloring. In Proc. ACM SIGPLAN '82 Symposium on Compiler Construction (Boston, June 1982). ACM, New York, 1982, pp. 98-105

[13] Chastain, Mike, Gary Gostlin, Jim Mankovich, Steve Wallach. The Convex C240 Architecture. IEEE 1988 Supercomputing Conference, pp. 321-329.

[14] Chow, Frederick C. A Portable Machine-Independent Global Optimizer -- Design and Measurements. Technical Report 83-254, Computer Systems Laboratory, Stanford University, 1983.

[15] Chow, F., S. Correll, M. Himelstein, E. Killian, L. Weber. How Many Addressing Modes are Enough? Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II). Palo Alto, CA, October, 1987.

[16] Chow, F., M. Himelstein, E. Killian, L. Weber. Engineering a RISC Compiler System. IEEE Compcon. 1986

[17] Chow, Frederick and John Hennessy. Register Allocation by Priority-Based Coloring. In Proc. ACM SIGPLAN '84 Symposium on Compiler Construction (Montreal, June 1984). ACM, New York, 1984, pp. 222-232.

[18] Chow, F. C., and Hennessy, J. L. The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. 12(4) (October 1990), pp. 501-536.

[19] Colwell, R.P., R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, A VLIW Architecture for a Trace Scheduling Compiler, IEEE Trans. Comput., Vol 37, No. 8, pp. 967-979, August 1988.

[20] Colwell, R. P., W. E. Hall, C. S. Joshi, D. B. Papworth, and P. K. Rodman, Architecture and Implementation of a VLIW Supercomputer, Proceedings, IEEE/ACM Supercomputing 90, October 1990, pp. 910-919.

[21] Dehnert, James C., Peter Y.-T. Hsu, Joseph P. Bratt. Overlapped Loop Support in the Cydra 5. Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, April 1989. pp. 26-38.

[22] Dongarra, Jack J. Performance of Various Computers Using Standard Linear Equations Software. CS-89-85, Computer Science Department, University of Tennessee, September 5, 1991.

[23] Ellis, John R. Bulldog: A Compiler for VLIW Architectures. MIT Press, Cambridge, MA, 1986. Also Ph.D. Thesis, Yale Univ., New Haven, Conn., February 1985.

[24] Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization, ACM Transactions on Programming Languages and Systems, July 1987, 9(3), pp. 319-349.

[25] Feldman, Stuart I. Implementation of a Portable F77 Compiler Using Modern Tools, Proceedings of the SIGPLAN Symposium on Compiler Construction, August 1979, pp. 98-106.

[26] Fisher, Joseph A. The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources. Ph.D. Thesis, New York Univ., New York, N.Y., 1979.

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

[28] Fisher, Joseph. A., Ellis, John R., Ruttenberg, John. C., and Nicolau, Alexandru. Parallel processing: A smart compiler and a dumb machine. In Proc. ACM SIGPLAN '84 Symp. Compiler Construction (Montreal, June 1984). ACM, New York, 1984, pp. 37-47.

[29] Foster, C. C. and E. M. Riseman, Percolation of code to enhance parallel dispatching and execution, IEEE Trans. Comput., vol C-21, pp. 1411-1415, 1972

[30] Freudenberger, S. M. and Ruttenberg, J. C. Phase Ordering of Register Allocation and Instruction Scheduling. Code Generation - Concepts, Tools, Techniques, edited by Robert Giergerich and Susan L. Graham. Springer-Verlag, London, 1992.

[31] Gibbons, Phillip B. and Steven S. Munchnik, Efficient Instruction Scheduling for a Pipelined Architecture. Proceedings of the SIGPLAN86 Symposium on Compiler Construction. Palo Alto, CA, June 1986, pp. 11-16.

[32] Gross, Thomas. Code Optimization of Pipeline Constraints. Technical Report No. 83-255. Computer Systems Laboratory, Stanford University, December 1983.

[33] Gross, T. and Ward, M., The Suppression of Compensation Code. Proceedings of the Third Workshop on Languages and Compilers for Parallel Processing, Irvine, August 1990, pp 260-273.

[34] Hennessy, J. L. and T. R. Gross. Postpass Code Optimization of Pipeline Constraints. ACM Transactions on Programming Languages and Systems 5(3), July 1983.

[35] Himelstein, Mark I. Compiler Tail Ends. Tutorial Notes from SIGPLAN'91 Conference on Programming Language Design and Implementation, June 1991.

[36] IBM RS/6000 Assembler Language Reference, Order Number SC23-2197-1, IBM.

[37] Johnson, S. C. A tour through the portable C compiler. AT&T Bell Laboratories, Murray Hill, NJ, 1979.

[38] Jouppi, Norman P., David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, April 1989. pp. 272-282

[39] Joshi, Chandra S., Brad A. Reger, John R. Feehrer. Memory System for a Statically Scheduled Supercomputer. 1991 International Conference on Parallel Processing, Vol I, pp. 196-203.

[40] Kane, Gerry. MIPS R2000 RISC Architecture. Prentice Hall, Englewood Cliffs, NJ. 1987.

[41] Kernighan, Brian. W., and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Inc, Englewood Cliffs, N.J., 1978.

[42] Knuth, Donald E. Knuth. Seminumerical Algorithms. The Art of Computer Programming, Volume 2, Second Edition. Addison-Wesley Publishing Company, Reading, Mass. 1981.

[43] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. Eighth Annual ACM Symposium on Principles of Programming Languages. Williamsburg, Virginia, January 1981, pp. 207-218.

[44] Lam, Monica. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. Proceedings of the SIGPLAN'88 Conference on Programming Design and Implementation. Atlanta, Georgia June 22-24, 1988. pp. 318-328.

[45] Lam, Monica S., A Systolic Array Optimizing Compiler, Kluwer Academic Publishers, Boston, Ma., 1989.

[46] Linn, Joseph L. SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information. Proceedings of the 16th Annual Microprogramming Workshop, October, 1983. pp. 11-22.

[47] Linn, Joseph L. Horizontal Microcode Compaction. In Microprogramming and Firmware Engineering Methods, edited by Stanley Habib. Van Nostrand Reinhold, New York, 1988, pp. 381-431.

[48] Little, J. D. C., A proof of the queuing formula L = DW, Opns. Res. 9,3 (May 1961), pp. 383-387. Cited in Edward G. Coffman, Jr., Peter J. Denning. Operating Systems Theory, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[49] Lubeck, Olaf M. Supercomputer Performance: The Theory, Practice, and Results. LA-11204-MS, Los Alamos National Laboratory, Los Alamos, New Mexico, January 1988.

[50] McMahon, Frank H. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. UCRL-53745, Lawrence Livermore National Laboratory, University of California, Livermore, California, December 1986.

[51] Mercer, Randall. The CONVEX FORTRAN 5.0 Compiler. Proceedings of Supercomputing '88, International Supercomputing Institute, St Petersburg, Florida, 1988.

[52] Morel, E., C. Renvoise. Global Optimization by Suppression of Partial Redundancies, Communications of the ACM, 22(2), February 1979, pp 96-103.

[53] Nicolau, Alexandru, and Joseph Fisher. Using an oracle to measure parallelism in single instruction stream programs. The 14th Annual Microprogramming Workshop, October 1981, pp. 171-182.

[54] Nicolau, Alexandru. Parallelism, memory anti-aliasing and correctness for trace scheduling compilers. Ph.D. Thesis, Yale Univ., New Haven, Conn., 1984.

[55] O'Brien, Kevin, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, Mathew Zaleski, Advanced Compiler Technology for the RISC System/6000 Architecture. IBM RISC System/6000 Technology, IBM, 1990, pp. 154-161.

[56] Odnert, Daryl, Robert Hansen, Manoj Dadoo, Mark Laventhal. Architecture and Compiler Enhancements for PA-RISC Workstations. IEEE Compcon, Spring 1991, pp. 214-218.

[57] PA-RISC Procedure Calling Convention Reference Manual, Order number 09740-90015, Hewlett-Packard, 1991.

[58] Padua, David A., Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. Communications of the ACM 29(12), December 1986, pp 1184-1201.

[59] Pettis, Karl, Robert C. Hansen, Profile Guided Code Positioning. ACM SIGPLAN90 Conference on Programming Language Design and Implementation. White Plains, NY June 1990. pp 16-27.

[60] Rau, B.R., C.D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proceedings of the Fourteenth Annual Microprogramming Workshop, October 1981, pp. 183-198.

[61] Rau, B. Ramakrishna, David W.L. Yen, Wei Yen, Ross A. Towle. The Cydra 5 Departmental Supercomputer Design Philosophies, Decisions, and Trade-offs. IEEE Computer 22(1), January 1989, pp. 12-34.

[62] Reif, John H. Combinatorial Aspects of Symbolic Program Analysis, TR-11-77, Center for Research in Computing Technology, Cambridge, Massachusetts, 1977.

[63] Reif, John H., Code Motion. SIAM Journal of Computing, 9(2), May 1980, pp. 375-395.

[64] Reif, John H., R. E. Tarjan. Symbolic Analysis in Almost Linear Time. SIAM Journal of Computing 11(1), February, 1982

[65] Rodman, Paul. High Performance FFTs for a VLIW architecture. 1989 Symposium on Computer Architecture and Digital Signal Processing, Hong Kong. IEE, Hong Kong Centre, GPO Box 10007, Hong Kong.

[66] SPEC Newsletter, 1(1), Fall 1989, c/o Waterside Associates, Fremont CA.

[67] Thornton, J. E., Design of a Computer: The Control Data 6600. Glenview, IL: Scott, Foresman, 1970.

[68] Tirmulai P., M. Lee, M. Schlansker. Parallelization of Loops With Exits on Pipelined Architectures. Proceedings of Supercomputing’90, New York, November 1990. IEEE Computer Society Press, Los Alimitos, CA, 1990. pp. 200-212.

[69] Tjaden, G. S. and M. J. Flynn, Detection and parallel execution of independent instructions. IEEE Trans. Comput., vol C-19, pp. 889-895, Oct. 1970.

[70] Tomasulo, R. M. , An efficient algorithm for exploiting multiple arithmetic units, in Computer Structures: Principles and Examples. New York: McGraw-Hill, 1982, pp. 293-305.

[71] Wall, David W. Limits of Instruction-Level Parallelism. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April 1991. pp. 176-188

[72] Warren, H. S., Instruction scheduling for the IBM RISC System/6000 processor. IBM Journal of Research and Development, 34(1) January 1990, pp 85-92.

[73] Wolfe, Michael. Optimizing Supercompilers for Supercomputers. The MIT Press, Cambridge, Massachusetts, 1989.