Technical Summary

Multiflow Computer, Inc.
175 North Main Street
Branford, CT 06405

MULTIFLOW

April 23, 1987

INTRODUCTION

Multiflow TRACE computer systems are general purpose machines designed for use in a wide range of compute-intensive applications. A family of upward compatible processor models deliver extraordinary performance via breakthroughs in computer architecture and compiler technology.

Very Long Instruction Word (VLIW) architecture provides TRACE processors with performance from 53 to 215 VLIW MIPS, with overlapped execution in a single execution stream.

Trace Scheduling compacting compilers exploit fine-grained parallelism without programmer intervention to deliver high performance on unmodified C and FORTRAN applications.

The TRACE/UNIX operating system delivers excellent interactive and computational performance. TRACE/UNIX is based upon 4.3BSD, with significant functional and performance enhancements for engineering and scientific computing. TRACE/UNIX supports demand-paged virtual addressing, with 4 Gigabytes of address space available per process.

This document provides an overview of the technology underlying Multiflow's products.

CHAPTER 1
PARALLEL PROCESSING BACKGROUND

Computer designers have sought to apply parallelism to improve execution speed since the earliest systems were designed. Substantial cost and reliability improvements are obtained when performance can be achieved by using multiple low-cost circuits instead of exotic electronics.

1.1 FINE-GRAINED PARALLELISM

Most modern computers use small scale parallelism to great advantage; for example, instruction *fetch* is commonly overlapped with instruction *execution* in modern "pipelined" superminicomputers. This technique dates back to mainframes of the early 1960's, and is quite effective in delivering a twofold improvement in overall execution speed.

Instruction fetch pipelining, coupled with appropriate instruction set design, helps computers achieve a major design goal: executing one instruction per clock cycle. As RISC design philosophy spreads throughout the industry, this goal is increasingly being achieved by midrange computer systems.

Designers of high-performance computers have achieved a more difficult objective: executing multiple instructions per clock cycle. Today's high-performance mainframe computers incorporate a technique introduced in the late 1960's: *overlapped execution* of multiple program steps. Overlapped execution exploits parallelism among individual scalar operations, such as adds, multiplies, and loads. A *scheduler* examines the relationships among operations of the program; then multiple *functional units* carry out independent computations simultaneously.

This fine-grained parallelism exists throughout all applications, and is independent of the high-level structure of the program. As a result, overlapped execution has been a feature of nearly every high performance scientific and engineering computer built in the last twenty-five years. Examples include the CDC 6600 and all of its descendants; the Cray supercomputers; and the IBM STRETCH, 360/91, and descendants.

Overlapped execution has been universally successful and widely used, but has suffered from a limitation: available speedups have been limited to about a factor of two or three. Characteristics of programs have prevented computer designers from delivering larger speedups due to this low-level approach.

1.2 COARSE-GRAINED PARALLELISM

In response to the limitations of overlapped execution, designers of high-performance scientific computers have looked for other ways to use more hardware in parallel to boost application performance.

Vector architectures perform parallel computations on data aggregates. Vector computers augment a standard scalar architecture with multiple pipelined functional units under rigid lockstep control. Special vector opcodes cause a vector control unit to initiate multiple computations in parallel while accessing registers or memory in a fixed pattern. The regular structure of vector operations allows the construction of computers with high peak performance at relatively low cost. High performance can be delivered on those portions of the computation which exactly fit the structure of the vector hardware.

Multiple processor systems (*multiprocessors*, or *parallel computers*) incorporate multiple independent CPUs with communication and synchronization hardware, and in some cases shared main memory. The overhead cost of fetching and decoding instructions must be paid per CPU; some multiprocessors offset this by exploiting low-cost microprocessor CPUs. The costs of runtime arbitration and synchronization tend to be high, both in hardware and in the time these operations require. Recently, attempts have been made to apply such systems to improve time-to-solution for a single application. Multiple processors can be applied to those portions of an application where large blocks of computation have been identified as being independent of each other, and the program has been expressed as multiple *tasks*.

Both vector and multiprocessor architectures are *coarse-grained*, in that *groups* of independent operations must be identified and expressed, either as vector operations or as tasks, in order for parallel hardware to contribute to performance. Such systems vary widely in their "grain size". Available vector machines have a minimum vector length required for any performance improvement which ranges from five to over 100. Available multiprocessors have a minimum block size required for any parallel benefit ranging from approximately 30 to many thousands of operations. Both approaches require high-level "pattern match" between hardware capabilities and program structure.

Early vector and multiprocessor systems required special programming techniques to use the parallel facilities. Great strides have been made in the automatic identification of loop structures which may be vectorized, eliminating the need for special syntax and languages to use these facilities. Vectorization technology has been adapted to some multiprocessor systems, allowing certain multiply-nested DO loops to be converted to "tasks".

However, significant application restructuring is universally required to achieve a reasonable percentage of vectorization or parallelization. Specific operation patterns and high-level data independence are required for even the most advanced "vectorizing" and "parallelizing" compilers to do their job.

1.3 AMDAHL'S LAW

Compute-intensive engineering and scientific applications vary widely in the extent to which they are *parallelizable*. Only regularly structured code, containing specific operation patterns, can be mapped onto coarse-grained parallel hardware. The percentage of running time spent in such code is extremely application dependent.

High degrees of parallelizability have been found to be rare. Most applications spend between 20 and 60 percent of their running time in code that is potentially vectorizable or parallelizable.

Amdahl's Law points out that for a 50% parallelizable application, even with infinitely fast vector hardware, or infinitely many parallel processors, the maximum achievable speedup is a factor of two.

This phenomenon is responsible for the great disparity between peak and achieved performance observed on coarse-grained parallel systems. Scalar computations tend to dominate the running time on coarse-grained systems, greatly limiting their effectiveness.

CHAPTER 2
A NEW APPROACH

Multiflow's product line is based on a fundamentally new approach to parallel processing. Multiflow's VLIW architecture and Trace Scheduling compacting compilers deliver massive performance improvements through overlapped execution of long streams of operations.

These technologies have yielded a compiler/computer combination which finds and exploits all the parallelism accessible to vector and multiprocessor architectures. It additionally finds parallelism throughout applications, in code which coarse-grained architectures cannot address. Much larger speedups are delivered with low-cost hardware because the "scalar speed bottleneck" has been broken.

2.1 VLIW ARCHITECTURE

Figure 1 shows an idealized VLIW (Very Long Instruction Word) computer. Fields of a wide instruction word directly control the operation of multiple functional units – floating adders, floating multipliers, integer units, memory address units. One field of the instruction word controls program branching.

A single program counter controls the fetching of Very Long Instruction Words. In this idealized example, a single register file holds operands and results for all computations. LOAD and STORE operations move data between registers and main memory.

The functional units are fully *pipelined*. Each functional unit is designed so that it can start a new computation in every clock cycle. An assembly-line or *pipeline* of hardware handles each stage of complex operations, such as floating point arithmetic. While a single floating-point addition may require 3 cycles to complete, new adds may be started every cycle. Note that the memory system is also fully pipelined; new references may be started every cycle, although a single reference may require 3 cycles to complete.

This simple VLIW may be expanded. More functional units can be added, with multiple register files, communicating via buses. Again, a single program counter and a single flow of control directs the fetching of long instruction words which specify the operation of each functional unit in each cycle. All the functional units run in lockstep, each initiating one operation per cycle as directed by its field of the instruction word.

A very wide *instruction cache* holds the instructions which the machine executes. The instruction word width is not related to the width of the data buses used for computation; it is related to how many functional units there are in the machine.

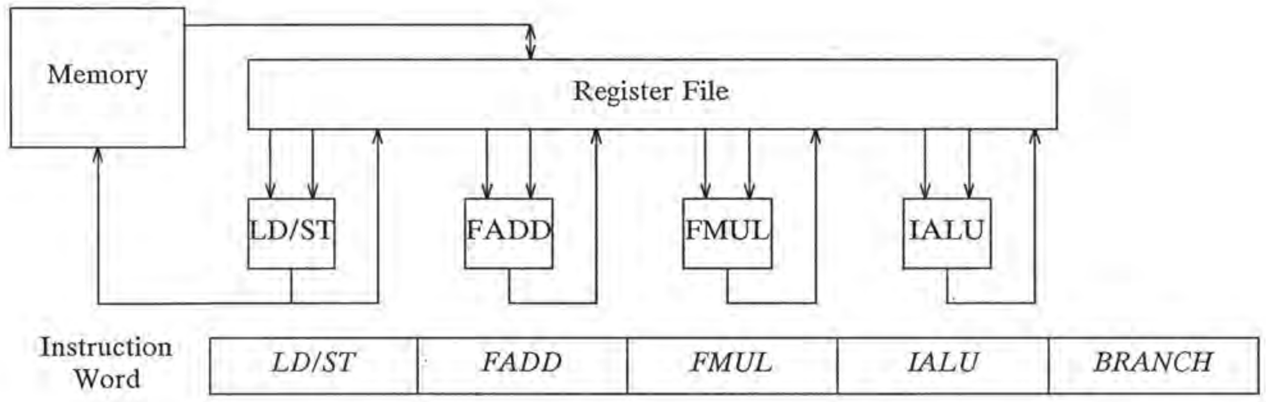

Figure 1: A Simple VLIW

Note that from one standpoint, VLIWs could be regarded as generalized, more efficient vector machines. The pipelined functional units are essentially the same as might be found in a vector machine, but no separation between scalar and vector hardware exists. Hardware control units which count out vector addresses have been replaced by wide instruction words, which specify each computation uniquely. Control hardware has been replaced with memory. Not only is the hardware lower cost to build, but with appropriate software technology the computation units can be used much more efficiently.

Viewed from another standpoint, VLIWs provide overlapped execution in the extreme. The instruction words allow the expression of arbitrary execution overlap among scalar operations, with potentially very large numbers of operations executing simultaneously. Great flexibility is available in how we overlap operations, within a single CPU; one program, one program counter. No runtime synchronization hardware or software is required.

2.2 OVERLAPPED EXECUTION REVISITED

Let's examine overlapped execution in the context of a VLIW. The burden of scheduling sequential code for simultaneous execution has been placed on the compiler which generates VLIW object code; scheduling hardware has been removed. Very high performance is achieved when the compiler identifies independent operations and compacts them into long instructions.

Consider first a single FORTRAN statement.

```
    A = (B + C) * (D + E)
```

On a modern, sequential computer this expands to an assembly language sequence which looks like:

```
    LD      R1, #B
    LD      R2, #C
    FADD    R3, R1, R2
    LD      R4, #D
    LD      R5, #E
    FADD    R6, R4, R5
    FMUL    R1, R6, R3
    STO     R1, #A
```

This sequence performs the computation that our FORTRAN statement specified. B and C are loaded from memory into registers; then they are added. Their sum is held in R3. D and E are loaded, then added. Finally the two sums are multiplied, and the product stored from R1 back into the memory location representing A.

On a modern sequential computer, typical operation costs for the different program steps are:

LD, FADD, FMUL	3 cycles
STO	1 cycle

With these assumptions, the running time for this fragment is 22 cycles. Each operation finishes before the next operation completes.

Consider a VLIW built with the same hardware technology, with the same cycle delays for operations. Let's examine the above fragment, compacted for VLIW execution.

As we compact the code, we must obey two constraints.

First, operations may be scheduled only when their data is ready; we must observe *data precedence*. For example, we cannot schedule the `FADD` step until the two `LD` operations have completed. Since we have a pipelined VLIW which starts a new instruction every cycle, this means that the `FADD` will be, at the earliest, three instructions after the second `LD`.

Second, operations may be scheduled only when the required functional unit is available; we observe *resource constraints*. In the simple VLIW we're considering in this example, only one *LD/ST* unit is available, so only one memory reference may be started per instruction. (More powerful VLIWs could potentially start multiple references per instruction.) Similarly, only one `FMUL`, one `FADD`, and one integer operation may be started per cycle. Note that up to five operations may be started in each instruction; due to pipelining, up to 11 operations may be in progress at once. This simple VLIW, whose hardware cost is nearly identical to that of a standard sequential computer using the same electronics, offers the potential of a tenfold performance improvement.

The overlapped code for our fragment, for our idealized VLIW, is shown in figure 2. The running time for this fragment is 13 cycles; it runs 1.7 times faster than it did on the sequential machine which cost the same to build.

LD/ST	*IALU*	*FADD*	*FMUL*	*BRANCH*
LD #B				
LD #C				
LD #D				
LD #E				
		FADD R1,R2		
		FADD R4,R5		
			FMUL R6,R3	
STO #A				

Figure 2: VLIW Compacted Code For Example 1

Notice that we've obeyed the rules stated above. The first `FADD` is scheduled three instructions after the `LD` of `C`. The machine never paused for the memory references; `C` wasn't available in the registers until instruction 3.

By doing this small-scale overlap, on just a single FORTRAN statement, we've improved performance by 1.7 times. What happens as we compact larger fragments of code?

```
A = (B + C) * (D + E)
F = (G * H) + (X * Y)
```

This expands into:

```
LD      R1, #B
LD      R2, #C
FADD    R3, R1, R2
LD      R4, #D
LD      R5, #E
FADD    R6, R4, R5
FMUL    R1, R6, R3
STO     R1, #A
LD      R7, #G
LD      R8, #H
FMUL    R9, R7, R8
LD      R4, #X
LD      R5, #Y
FMUL    R6, R4, R5
FADD    R1, R6, R3
STO     R1, #F
```

with a running time of 44 cycles.

The VLIW version of this fragment is shown in figure 3. This has a running time of 17 cycles, 2.6 times faster than the scalar version.

LD/ST	*IALU*	*FADD*	*FMUL*	*BRANCH*
LD #B				
LD #C				
LD #D				
LD #E				
LD #G		FADD R1,R2		
LD #H				
LD #X		FADD R4,R5		
LD #Y				
			FMUL R7,R8	
			FMUL R6,R3	
			FMUL R4,R5	
STO #A				
		FADD R6,R3		
STO #F				

Figure 3: VLIW Compacted Code For Example 2

Notice that we've freely intermixed steps of the computation from the first and second FORTRAN statements. Since the variables in those statements didn't have anything to do with each other, we were free to do so and still get correct results. In fact, we could have reversed the ordering of many of the steps without affecting the result.

2.2.1 COMPACTION OF LONGER STREAMS

As more operations are considered together for compaction, the performance improvement continues to grow. If we were to increase the size of the above example to four statements, the scalar code would require 88 cycles; the VLIW would be 3.5 times faster, at only 25 cycles. A wider VLIW would be even faster.

Very large amounts of parallelism are found when long streams of operations are compacted, allowing wide VLIWs with many functional units to deliver tenfold and hundredfold performance improvements over scalar execution.

However, a problem arises in trying to compact long streams. Programs are not straight-line streams of operations; they contain control flow statements, or *conditional jumps*. These pose a serious problem for scheduling. How can we overlap operations with prior conditional jumps?

```
         A = (B + C) * (D + E)
         IF (A .GT. 1.0E6) GOTO 5
         F = (G * H) + (X * Y)
    5    CONTINUE
```

In this example, if the the assignment of `F` and its computation occurred before the `IF` test, the program would produce incorrect results whenever `A` was greater than 1.0E6.

Because of correctness issues like this, all previous efforts to overlap execution have overlapped only straight-line sections of code, or "basic blocks." Each conditional jump caused execution to serialize until its test resolved, and the scheduler could know which way to proceed.

Conditional jumps are found every five to eight operations in typical programs. This jump frequency, coupled with basic block compaction, has been the primary obstacle to very large performance gains from overlapped execution in scientific programs. When only small numbers of operations are candidates for overlapped execution, the gains from overlapped execution will be correspondingly small.

In the above example, basic block compaction would perform poorly in the presence of the `IF` test. If the operations below the `IF` cannot be started before the `IF` completes, the running time will be 27 cycles, as opposed to 17 cycles without the `IF`.

However, if we believe that the `IF` test will only rarely be true, and we have some sophistication in our management of registers, we can achieve all the overlap that we did before we added the `IF`. Figure 4 shows this code.

LD/ST	*IALU*	*FADD*	*FMUL*	*BRANCH*
LD #B				
LD #C				
LD #D				
LD #E				
LD #G		FADD R1,R2		
LD #H				
LD #X		FADD R4,R5		
LD #Y				
			FMUL R7,R8	
			FMUL R6,R3	
			FMUL R4,R5	
STO #A		FCGT R1,#1.0E6		
		FADD R6,R3		
				CBR 5
STO #F				

Figure 4: VLIW Compact Code Including Conditional Branch

No extra cycles are required to do the compare and branch. Only the `STO` operation need follow the execution of the branch.

2.3 TRACE SCHEDULING

Multiflow's Trace Scheduling compacting C and FORTRAN compilers overlap execution over long streams of code, going beyond many conditional jumps. The compilers use statistical information about program behavior, and a *compensation* technique, to perform aggressive compaction of long execution paths.

Trace Scheduling is carried out on one program module or subroutine at a time, after the program has been converted to an intermediate representation and after standard optimizations have been carried out.

2.3.1 TRACE SELECTION

Using loop trip count and branch probability information derived from program profiling or from heuristics, the compiler selects the most frequent path, or *trace,* that the code will follow during execution. The path may include multiple conditional jumps (see figure 5).

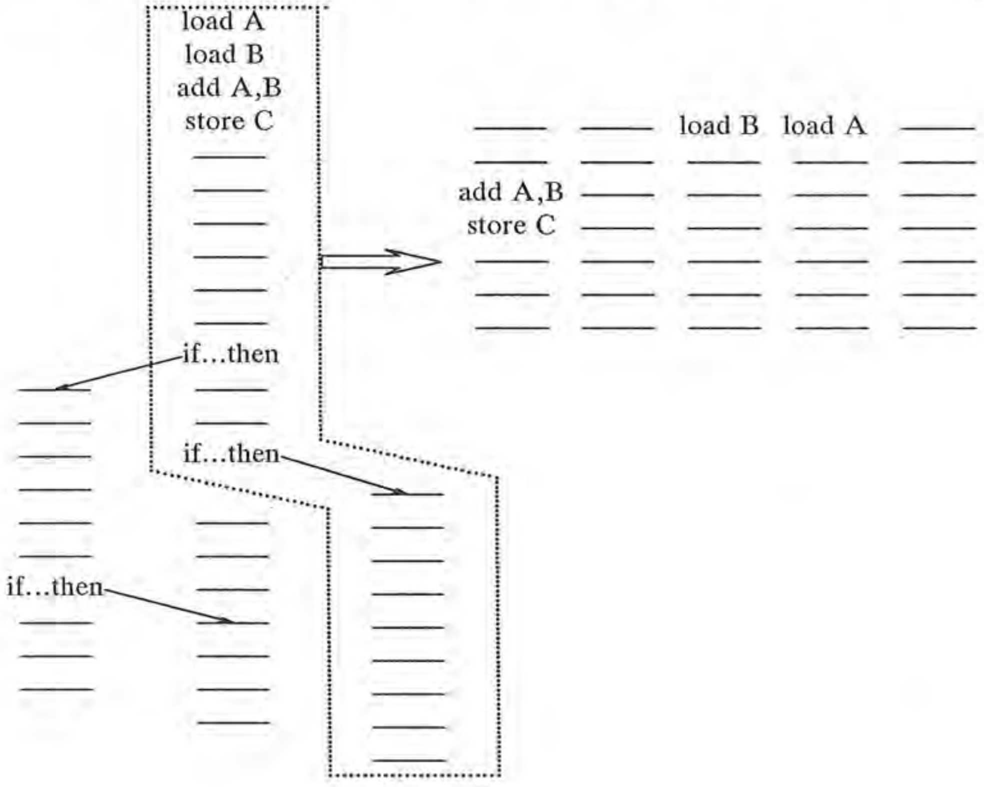

Figure 5: Selecting a Trace; Starting Compaction

This trace is then handed as a whole to a scheduler. The scheduler compacts operations into wide instruction words, taking into account data precedence and hardware resource constraints. These wide instruction words will be directly executed by Multiflow TRACE systems, using multiple functional units. Now, instead of five or eight operations which are candidates for scheduling, hundreds or thousands of operations may be candidates. Many opportunities for parallel execution will be present, and compaction will yield large speedups.

This large-scale overlapping moves operations in ways which could cause logical inconsistencies when a conditional branch goes the "less frequent" direction (see figure 6).

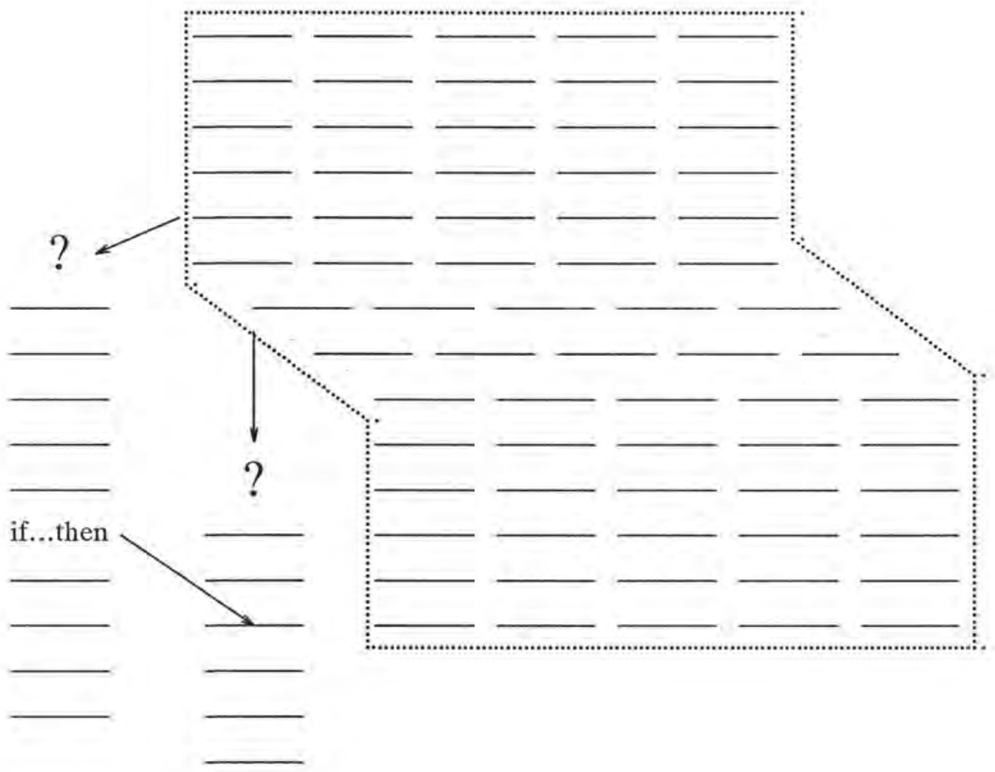

Figure 6: First Trace Compacted; Trouble at Off-Trace Branches

2.3.2 COMPENSATION

Finding a method for handling these inconsistencies after compaction, without touching the compacted code, was the conceptual breakthrough of Trace Scheduling. The compiler adjusts the flow graph of the remaining program to correct the scheduling-generated inconsistencies, and restore correctness for all execution paths (see figure 7).

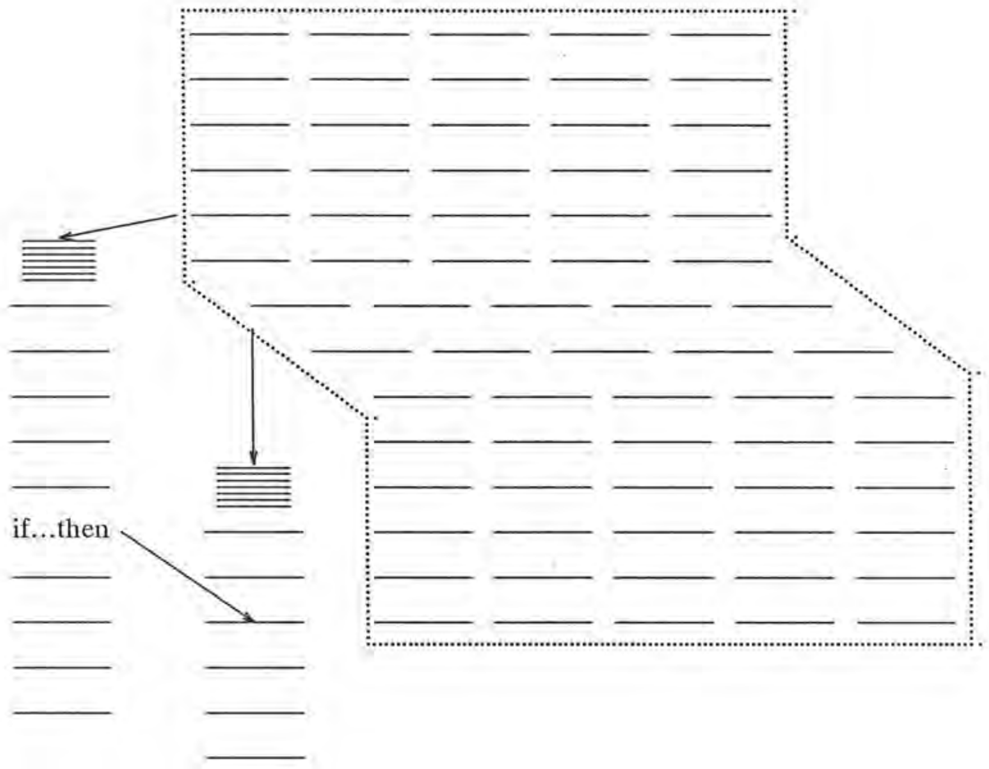

Figure 7: Compensation Operations Correct Off-Trace Paths

For example, if an operation originally above a conditional jump is scheduled below the jump, it is *copied* as part of the compensation for the jump. The copy is made only if the operation "matters" on the off trace path; that is, if there are operations in the off trace code which depend upon the result of this operation.

Similarly, if an operation below a conditional jump is scheduled above the jump, its results are *discarded* as part of the compensation code. This may not involve adding any operations; in most cases, the compiler has multiple temporary values for a variable, and simply uses a previous temporary value.

Similar compensation steps are added at joins into the trace other than at the top of the trace.

Compensation code adds only minor amounts to program size, and less to program runtime. Only a small fraction of the total scheduling decisions made during compaction cause inconsistencies. Only some of these actually require computations to correct.

2.3.3 THE PROCESS REPEATS

The whole process then repeats. The next-most-likely execution path is chosen as a trace and handed to the code generator. This trace may include original operations and compensation code. It is compacted; new compensation code may be generated; and the process repeats, picking paths and compacting them until the entire program has been compiled (see figure 8).

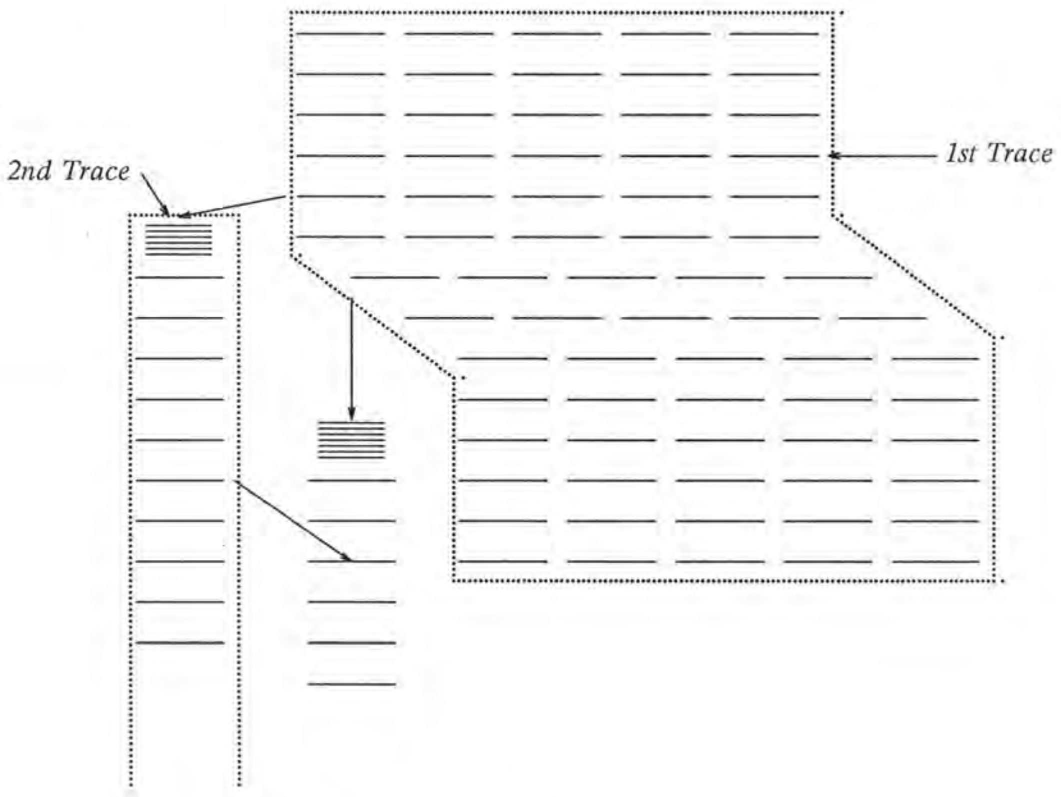

Figure 8: The Next Trace Is Selected For Compaction

2.4 MEMORY REFERENCE DISAMBIGUATION

Array references can pose special problems for compile-time data dependency analysis.

Consider a modification to our earlier fragment:

```
A(I) = (B + C) * (D + E)
F = (A(J) * H) + (X * Y)
```

Getting the best performance here requires compile-time analysis of the possible values of `I` and `J`, so as to be able to decide if the reference to `A(I)` can possibly refer to the same memory element as `A(J)`. If so, the memory load of `A(J)` must be scheduled after the store into `A(I)`, which will reduce parallelism somewhat.

Multiflow's Trace Scheduling compacting C and FORTRAN compilers perform this analysis. The compilers analyze the values which array index expressions can assume, and build symbolic derivations for their values in terms of local loop induction variables and invariant values. They then solve for whether or not the expressions can ever be equal. The process is sophisticated but effective, and automatically resolves memory references to allow maximum parallelism and compaction.

2.5 EXAMPLES

The innermost loop of direct matrix solvers consists of the vectorizable operation `Y = aX + Y`. We include the source, drawn from Argonne National Laboratories' LINPACK, and object code produced for the Multiflow TRACE 7/200. We include this code as a simple example of Trace Scheduling and VLIW treatment of vectorizable operation patterns.

The code:

```
20 DO 50 I = 1,N
      DY(I) = DY(I) + DA*DX(I)
50 CONTINUE
```

Multiflow's Trace Scheduling compacting FORTRAN compiler automatically unrolls this loop, performs a sequence of optimizations, then generates compacted object code, subject to data dependencies and machine resources.

The object code below is for the Multiflow TRACE 7/200. The TRACE 7/200 performs up to seven operations in each instruction (see Chapter 4). In the assembly listing below, each "inst" token marks a new instruction. Individual 32-bit and 64-bit operations are specified by fields of each instruction. Note that pipelined memory and floating point operations are heavily overlapped. Full performance is obtained from pipelined functional units without the rigidity of a vector-style control unit. Any operation pattern can be expressed by Multiflow's Trace Scheduling compacting compilers.

```
?T._daxpy_.1::
L0?3:
inst    ialu0e ld.64 fb0.r2,r3,zero         ialule cgt.s32 lilbb.r3,r32,6#5
        ialu0l ld.64 fb0.r4,r3,6#8                  ialull cgt.s32 lilbb.r4,r32,6#4;
inst    ialu0e ld.64 fb0.r6,r3,6#16         ialule cgt.s32 lilbb.r3,r32,6#3
        ialu0l ld.64 fb0.r8,r3,6#24         ialull cgt.s32 lilbb.r5,r32,6#2
        br true and r3 L1?3;
inst    ialu0e ld.64 fb0.r12,r3,6#32        ialule cgt.s32 lilbb.r6,r32,6#1
        ialu0l ld.64 fb0.r14,r3,6#40        ialull cgt.s32 lilbb.r7,r32,zero;
inst    ialu0e ld.64 fb0.r32,r4,zero        ialule add.u32 lib.r35,r32,6#6
        ialu0l ld.64 fb0.r34,r4,6#8         ialull add.u32 lib.r36,r33,6#48;
inst    ialu0e ld.64 fb0.r38,r4,6#16        ialule add.u32 lib.r37,r34,6#48
        ialull bor.32 lib.r5,zero,r36       falu0 mpy.f64 fb0.r36,r10,r2;
inst    ialu0e ld.64 fb0.r42,r4,6#24        ialule bor.32 lib.r6,zero,r37
        ialu0l ld.64 fb0.r2,r5,zero         ialull add.u32 lib.r33,r36,6#48
        falu0 mpy.f64 fb0.r40,r10,r4;
inst    ialu0e ld.64 fb0.r46,r4,6#32        ialule add.u32 lib.r32,r35,6#6
        ialu0l ld.64 fb0.r4,r5,6#8                  ialull bor.32 lib.r3,zero,r33
        falu0 mpy.f64 fb0.r44,r10,r6;
inst    ialu0e ld.64 fb0.r50,r4,6#40        ialu0l ld.64 fb0.r6,r5,6#16
        falu0 mpy.f64 fb0.r48,r10,r8;
inst    ialu0e ld.64 fb0.r54,r6,zero        ialu0l ld.64 fb0.r8,r5,6#24
        falu0 mpy.f64 fb0.r52,r10,r12       falu1 add.f64 lsb.r0,r32,r36;
inst    ialu0e ld.64 fb0.r36,r6,6#8         ialu0l ld.64 fb0.r12,r5,6#32
        falu0 mpy.f64 fb0.r32,r10,r14       falu1 add.f64 lsb.r2,r34,r40;
inst    ialu0e ld.64 fb0.r40,r6,6#16        ialu0l ld.64 fb0.r2,r5,6#40
        falu0 mpy.f64 fb0.r34,r10,r2        falu1 add.f64 lsb.r4,r38,r44;
inst    ialu0e st.64 sb0.r0,r4,zero         ialule cgt.s32 lilbb.r4,r35,6#5
        ialu0l ld.64 fb0.r42,r6,6#24        falu0 mpy.f64 fb0.r38,r10,r4
        falu1 add.f64 lsb.r6,r48,r42        br false or r4 L2?3;
inst    ialu0e st.64 sb0.r2,r4,6#8                  ialule cgt.s32 lilbb.r3,r35,6#4
        ialu0l ld.64 fb0.r46,r6,6#32        falu0 mpy.f64 fb0.r44,r10,r6
        falu1 add.f64 lsb.r8,r46,r52        br true and r3 L3?3;
inst    ialu0e st.64 sb0.r4,r4,6#16         ialule cgt.s32 lilbb.r5,r35,6#3
        ialu0l ld.64 fb0.r32,r6,6#40        falu0 mpy.f64 fb0.r48,r10,r8
        falu1 add.f64 lsb.r0,r32,r50        br true and r5 L4?3;
inst    ialu0e st.64 sb0.r6,r4,6#24         ialule cgt.s32 lilbb.r6,r35,6#2
        falu0 mpy.f64 fb0.r50,r10,r12       falu1 add.f64 lsb.r2,r54,r34
        br false or r6 L5?3;
inst    ialu0e st.64 sb0.r8,r4,6#32         ialule cgt.s32 lilbb.r7,r35,6#1
        falu0 mpy.f64 fb0.r34,r10,r2        falu1 add.f64 lsb.r4,r36,r38
        br true and r7 L6?3;
inst    ialu0e st.64 sb0.r0,r4,6#40         ialule cgt.s32 lilbb.r4,r35,zero
        ialull add.u32 lib.r34,r37,6#48             falu1 add.f64 lsb.r6,r40,r44
        br false or r4 L7?3;
inst    ialu0e st.64 sb0.r2,r6,zero         ialule bor.32 lib.r4,zero,r34
        falu1 add.f64 lsb.r8,r48,r42        br true and r3 L8?3;
inst    ialu0e st.64 sb0.r4,r6,6#8                  falu1 add.f64 lsb.r0,r46,r50
        br true and r5 L9?3;
inst    ialu0e st.64 sb0.r6,r6,6#16         falu1 add.f64 lsb.r2,r34,r32
        br false or r6 L10?3;
inst    ialu0e st.64 sb0.r8,r6,6#24         br true and r7 L11?3;
inst    ialu0e st.64 sb0.r0,r6,6#32         br false or r4 L12?3;
inst    ialu0e st.64 sb0.r2,r6,6#40;
inst    br true or false L0?3;
```

This code runs at 7.7 million floating point operations per second on the TRACE 7/200.

2.5.1 ANOTHER EXAMPLE

Trace Scheduling and VLIW architectures perform equally well on irregular code. From the Lawrence Livermore National Laboratories computational kernels:

```
C
C******************************************************************************
C***  KERNEL 15     CASUAL FORTRAN.  DEVELOPMENT VERSION.
C******************************************************************************
C
C
C       CASUAL ORDERING OF SCALAR OPERATIONS IS TYPICAL PRACTICE.
C       THIS EXAMPLE DEMONSTRATES THE NON-TRIVIAL TRANSFORMATION
C       REQUIRED TO MAP INTO AN EFFICIENT MACHINE IMPLEMENTATION.
C
        DO 45  L = 1,Loop
               NR= 7
               NZ= n
               AR= 0.053
               BR= 0.073
   15   DO 45  j = 2,NR
        DO 45  k = 2,NZ
               IF( j-NR) 31,30,30
   30     VY(k,j)= 0.0
                   GO TO 45
   31          IF( VH(k,j+1) -VH(k,j)) 33,33,32
   32           T= AR
                   GO TO 34
   33           T= BR
   34          IF( VF(k,j) -VF(k-1,j)) 35,36,36
   35           R= MAX( VH(k-1,j), VH(k-1,j+1))
                S= VF(k-1,j)
                   GO TO 37
   36           R= MAX( VH(k,j),   VH(k,j+1))
                S= VF(k,j)
   37     VY(k,j)= SQRT( VG(k,j)**2 +R*R)*T/S
   38          IF( k-NZ) 40,39,39
   39     VS(k,j)= 0.
                   GO TO 45
   40          IF( VF(k,j) -VF(k,j-1)) 41,42,42
   41           R= MAX( VG(k,j-1), VG(k+1,j-1))
                S= VF(k,j-1)
                T= BR
                   GO TO 43
   42           R= MAX( VG(k,j),   VG(k+1,j))
                S= VF(k,j)
                T= AR
   43     VS(k,j)= SQRT( VH(k,j)**2 +R*R)*T/S
   45     CONTINUE
C
C...................
```

For this code, the compiler produces a series of traces. TRACE 7/200 object code for trace number 7, again produced by Multiflow FORTRAN release 1.4:

```
L12?3:
inst ialu0e ld.64 fb0.r52,r3,6#8          ialu1e bor.32 lib.r4,zero,r34
     ialu1l cgt.s32 lilbb.r3,r32,6#2      falu0 div.f64 lfb.r6,r20,r4;
inst ialu0e ld.64 fb0.r56,r4,6#8          ialu1e cge.s32 lilbb.r6,r33,zero
     ialu0l ld.64 fb0.r60,r5,17#5664      ialu1l cge.s32 lilbb.r4,r32,6#2;
inst ialu0e ld.64 fb0.r32,r5,17#5656      ialu1e cgt.s32 lilbb.r7,r32,6#1
     ialu0l ld.64 fb0.r34,r7,17#25864     ialu1l cge.s32 lilbb.r5,r33,zero;
```

```
inst ialu0e ld.64 fb0.r40,r8,6#8           ialu01 ld.64 fb0.r36,r12,17#5656;
inst ialu0e ld.64 fb0.r38,r12,17#5664      ialu01 ld.64 fb0.r4,r9,zero;
inst ialu0e ld.64 fb0.r54,r5,6#8           ialu01 ld.64 fb0.r58,r6,6#8
     falu1 sub.f64 lfb.r42,r52,r56;
inst ialu0e ld.64 fb0.r8,r7,17#31520       ialu01 ld.64 fb0.r48,r8,17#5664;
inst ialu0e ld.64 fb0.r10,r9,6#8           ialu01 ld.64 fb0.r44,r8,17#5672
     falu1 sub.f64 lfb.r52,r60,r32;
inst ialu0e ld.64 fb0.r12,r7,17#31528      ialu01 ld.64 fb0.r14,r9,6#16
     falu1 cgt.f64 li0bb.r4,r42,zero;
inst ialu0e ld.64 fb0.r32,r3,6#16          ialu01 ld.64 fb0.r16,r7,17#31536
     falu1 sub.f64 lfb.r56,r34,r40;
inst ialu0e ld.64 fb0.r18,r9,6#24          ialu01 ld.64 fb0.r20,r12,zero
     falu1 clt.f64 li0bb.r5,r52,zero;
inst ialu0e ld.64 fb0.r22,r5,17#5672       ialu01 ld.64 fb0.r26,r5,6#16
     falu0 mpy.f64 fb0.r34,r10,r10         falu1 cle.f64 lf1bb.r1,r38,r36;
inst ialu0e ld.64 fb0.r28,r6,6#16          ialu01 ld.64 fb0.r10,r7,17#25872
     falu0 mpy.f64 fb0.r52,r12,r12         falu1 cge.f64 li0bb.r6,r56,zero;
inst ialu0e ld.64 fb0.r12,r8,6#16          ialu01 ld.64 fb0.r6,r4,6#24
     falu0 mpy.f64 lsb.r0,r6,r2            falu1 cgt.f64 lfb.r50,r58,r54;
inst ialu0e ld.64 fb0.r24,r5,17#5680       ialu01 ld.64 fb0.r42,r5,17#5664
     falu0 mpy.f64 fb0.r56,r14,r14         falu1 cgt.f64 lib.r10,r44,r48;
inst ialu0e ld.64 fb0.r2,r5,6#24           ialu01 ld.64 fb0.r16,r8,17#5680
     falu0 mpy.f64 fb0.r46,r16,r16;
inst ialu01 ld.64 fb0.r18,r3,6#24          falu0 mpy.f64 fb0.r14,r18,r18;
inst ialu0e st.64 sb0.r0,r7,zero;
inst ialu01 st.64 sb0.r2,r0,6#(?2.1?2loc_stg + 0x1c)
     falu1 bor.64 lsb.r2,zero,r56;
inst ialu01 st.64 sb0.r0,r0,6#(?2.1?2loc_stg + 0xc)
     falu1 bor.64 lsb.r0,zero,r42;
inst ialu0e ld.64 fb0.r56,r5,17#5672       falu0 bor.64 lsb.r4,zero,r2
     falu1 bor.64 lsb.r2,zero,r32;
inst ialu01 ld.64 fb0.r14,r8,17#5672       falu0 bor.64 lsb.r0,zero,r14
     falu1 bor.64 lsb.r6,zero,r46;
inst ialu0e ld.64 fb0.r46,r6,6#24          falu0 bor.64 lsb.r8,zero,r8
     falu1 bor.64 lsb.r10,zero,r52;
inst ialu1e bor.32 lfb.r8,zero,32#1068180176
     ialu11 bor.32 lfb.r9,zero,32#3758096384
     ialu01 ld.64 fb0.r38,r4,6#16          falu0 mpy.f64 fb0.r32,r4,r4
     falu1 slct.64 lfb.r2,r38,r36;
inst falu0 bor.64 lfb.r42,zero,r24         falu1 bor.64 lsb.r12,zero,r56;
inst ialu01 st.64 sb0.r4,r0,6#(?2.1?2loc_stg + 0x34);
inst ialu0e st.64 sb0.r8,r0,17#(?2.1?2loc_stg + 0x5c)
     ialu01 st.64 sb0.r10,r0,6#(?2.1?2loc_stg + 0x14);
inst ialu0e st.64 sb0.r6,r0,6#(?2.1?2loc_stg + 0x24)
     ialu01 st.64 sb0.r12,r0,17#(?2.1?2loc_stg + 0x3c);
inst ialu0e st.64 sb0.r0,r0,6#(?2.1?2loc_stg + 0x2c)
     ialu01 st.64 sb0.r2,r0,6#(?2.1?2loc_stg + 0x4);
inst ialu0e ld.64 fb0.r4,r0,17#(?2.1?2loc_stg + 0x5c);
inst;
inst;
inst;
inst falu0 mpy.f64 fb0.r36,r4,r4;
inst;
inst;
inst ialu01 ld.64 fb0.r4,r8,6#24;
inst ialu01 ld.64 fb0.r52,r7,17#25880      br true or false L13?3;
```

This code runs at 3.5 MFLOPs on a TRACE 7/200.

CHAPTER 3
COMPILERS

Multiflow's product line includes fully optimizing Trace Scheduling compacting C and FORTRAN compilers which generate code for the TRACE series of VLIW computers.

Multiflow's Trace Scheduling compacting compilers exploit fine-grained parallelism to generate high-performance object code for the TRACE series of VLIW computer systems. The compilers pack multiple operations into each wide instruction word, allowing the TRACE's hardware functional units to execute those operations simultaneously.

Multiflow's TRACE series of computers were designed specifically to match the requirements and capabilities of Multiflow's Trace Scheduling compilers. This design approach – software technology directing processor development – has produced extremely efficient, cost-effective computer systems.

3.1 FORTRAN LANGUAGE

Multiflow FORTRAN fully implements ANSI FORTRAN-77 (ANSI X3.9-1978), including features specified in the DoD Supplement (MIL-STD-1753). It is compatible with the older FORTRAN 66 standard (ANSI X3.9-1966). Multiflow FORTRAN includes extensions from the proposed FORTRAN 8X standard(ANSI X3J3/S8).

VAX FORTRAN compatibility features provide easy program portability from VAX/VMS environments.

Format and syntax extensions include:

- 32-character variable names
- End-of-line comments
- INCLUDE statement
- DATA statements may appear in executable code
- Constant syntax includes Hollerith, hex, and octal
- Tab character formatting

Functional extensions include:

- Direct system call interface from FORTRAN
- Extended-range DO loops
- DO WHILE and END DO constructs

- Extended I/O format descriptors
- ACCEPT and TYPE statements
- Bitwise intrinsic functions

Data types include INTEGER*2, INTEGER*4, LOGICAL, REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16. All data types are directly supported by TRACE hardware facilities.

Directives, contained in comment fields, allow the user to control compiler optimizAation. Automatic mechanisms for inline substitution, loop unrolling, memory reference disambiguation, and branch prediction may be overruled by the programmer without introducing unportable changes to the source code.

3.2 C LANGUAGE

Multiflow C includes the complete UNIX System/V standard C language, with the Berkeley standard extensions.

The C compiler front end was derived from the System/V *pcc* front end, with Berkeley extensions added. Syntax, semantics, and error messages are completely compatible with industry standard C implementations.

Runtime data formats are exactly compatible with runtime data formats found on 68020-based workstations. Pointers, integer representations, and byte ordering are all compatible. Single and double precision IEEE 754 floating point formats are used.

3.3 STANDARD PROGRAM EXECUTION

FORTRAN and C programs execute on Multiflow TRACE computers just as they do on conventional "scalar" computers. Programs run in the standard, sequential, predictable way that scientific users expect. Conventional program development and debugging techniques are used.

This is a major benefit to users familiar with standard computer systems. Multiflow's Trace Scheduling technology is uniquely able to deliver parallelism without user involvement, because VLIW overlapped execution is *below the level of the language*. Optimization and compilation proceeds automatically, without user guidance and without requiring special programming techniques.

3.4 COMPILER STRUCTURE AND ORGANIZATION

The compiler does its job in three phases. Phase 1 is a language-specific front end; Phase 2 is an enhanced optimizer; and Phase 3 uses Trace Scheduling compaction to build TRACE wide instruction words.

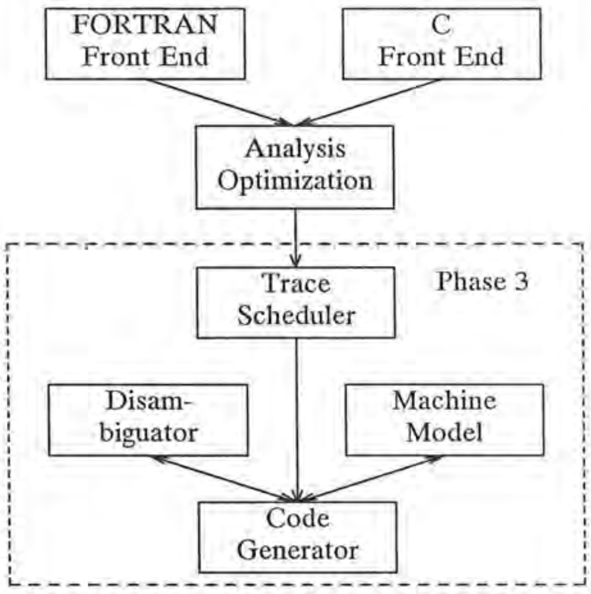

Figure 9: Multiflow Trace Scheduling Compacting Compiler Organization

Independent FORTRAN and C front ends feed a common Phase 2. This strategy guarantees compatibility between code produced for FORTRAN and C subroutines and guarantees that FORTRAN and C programs receive the benefits of full optimization.

3.5 PHASE 2: ANALYSIS AND OPTIMIZATION

Phase 2 performs a wide range of optimizations on the program's intermediate representation. These optimizations have two primary goals:

- Reducing run-time computation
- Reducing data dependency between operations (to increase parallelism).

As a first step, Phase 2 builds a *flow graph* of the program. The flow graph represents all possible control flows through the program, with straight-line code sequences, or *basic blocks*, as the nodes in the graph.

It then performs a series of analyses on the flow graph. These analyses will be used during optimization. They include analysis for:

- Loop structures
- Live variables (those that will be used more than once)
- Reaching definitions (a mapping from a statement defining a variable to statements where that variable may be used)
- Reaching uses (a mapping from a statement using a variable to statements where that variable may be defined)
- Reaching copies (tracing variable assignments).

Following these analyses, Phase 2 performs optimization. Optimizations include:

• **Induction Variable Simplification.** Simplifies calculations within loops by replacing repetitive multiplications in terms of an induction variable with iterative addition. For example, the two loops below are equivalent:

```
                         K=0
      DO 10 I=1,J        DO 10 I=1,J
      K=I*10             K=K+10
10    CONTINUE           CONTINUE
```

This optimization greatly simplifies address calculations for arrays; multidimensional array index multiplications can be replaced by additions. Often an induction variable can be entirely eliminated. This is a special case of *Strength Reduction*.

• **Common Subexpression Elimination.** Simplifies calculations by eliminating expressions that are common to two or more statements. For example, the following sequences of code are equivalent:

```
                        temp=A*4
      X = A*4+B*3       X=temp+B*3
      Y = A*4+D*3       Y=temp+D*3
```

The code on the right is significantly faster because the computation A*4 is only performed once.

• **Copy propagation.** Eliminates needless assignments. The following sequences of code are equivalent:

```
     A = B          A = B
     Z = S * A      Z = S * B
     Y = A + 4.0    Y = B + 4.0
```

Since A and B are not changed between the statement A=B and the use of A, the uses of A can be replaced by uses of B. This is preferable because Z=S*B is independent of the assignment A=B; more compaction will be possible. Later, if the first assignment is no longer used, Dead Code Removal will eliminate it.

• **Constant folding.** Evaluates constant expressions during compilation; replaces variables being used as constants by the constants themselves.

• **Global dead code removal.** Removes all code that is unreachable by any legitimate path from the start of the program, produces no result that is ever used, or for any other reason has no effect on the outcome of the program. This optimization "cleans up" after other optimization steps have been performed, removing stubs which become unreachable after other optimizations.

• **Register Variable Detection.** TRACE systems have large numbers of general registers. Register Variable Detection identifies variables that can reside in registers instead of main memory, eliminating memory references.

• **Loop Invariant Motion.** Removes unchanging expressions from loops by moving them outside, where they can be calculated just once.

• **Variable renaming.** Introduces new names for disjoint uses of the same variable. For example, the following code sequences are equivalent:

```
A = Z * X          A1 = Z * X
B = A * 4          B = A1 * 4
A = Q * Y          A = Q * Y
```

The first and second use of the variable A are disjoint. The renamed version of the code (on the right) has equivalent functionality, but more parallelism available to trace scheduling.

• **Inline substitution.** At the points where the source program calls a subroutine, automatically decides whether or not to insert the subroutine into the program directly rather than generate a subroutine call. This optimization removes procedure call overhead and allows the machine code for the called subroutine to be overlapped in parallel with the code for the calling routine. Multiflow's Trace Scheduling FORTRAN compiler is unique in that it does inline substitution automatically. Compiler heuristics determine, based on call frequency, number of call sites, and object code expansion, when to substitute subroutine bodies inline. The inline substitution process generates code that in every respect matches the behavior of the original subroutine. Other than performance improvements, there are no programmer visible changes; the behavior of local SAVED variables, COMMON references, etc. are completely preserved.

• **Loop Unrolling.** Modifies the program's inner loops by replicating their bodies several times. This optimization exposes parallelism between loop iterations, allowing Phase 3 to generate more efficient code through compaction of longer traces. Compiler heuristics determine loop unrolling amounts, based on loop body size, expected iterations, and object code expansion.

3.6 PHASE 3

Phase 3 compacts the program. It takes the optimized sequential representation of the program as input, and produces compacted Very Long Instruction Word object code for a TRACE series processor as its output.

Four major modules comprise Phase 3:

The *Trace Scheduler*, which builds a flow graph of the program and picks execution paths, or *traces*, for compaction;

The *Code Generator,* which compacts the operations in each trace into wide instructions;

The *Machine Model,* which represents the resources and architectural rules for each TRACE processor; and

The *Disambiguator*, which solves array index expressions to distinguish memory addresses and allow parallel execution of memory references.

3.7 COMPACTION

Operations are scheduled using information about program control flow, data dependencies among operations, and hardware resources.

• **Control flow**. Control flow analysis, performed by the trace scheduler, allows scheduling of operations drawn from the source program beyond basic block boundaries. Statistical information about program execution is gathered from sample runs of the application or generated by compiler heuristics. These statistics guide the selection of long execution paths, or *traces*, for compaction. The selection process begins with the most frequent execution path, compacting it for highest performance, then repeats, picking traces and compacting them until the entire program has been compiled. As each trace is picked, the trace scheduler calls the code generator to generate machine code for the trace.

Each trace is compacted as a whole. The code generator has access to parallelism throughout the trace. Instead of the small number of operations available within straight-line "basic blocks", hundreds or thousands of operations become candidates for overlap. Many opportunities for parallel execution are present, and compaction yields large speedups.

This large-scale overlapping moves operations in ways which could cause logical inconsistencies when the program branches off the chosen trace. The trace scheduler adjusts the flow graph of the remaining program, adding small amounts of *compensation code* to correct for scheduling-generated inconsistencies and ensure correctness for all execution paths.

For example, if an operation is scheduled after a branch which it originally preceded, a *compensation copy* will be made of the operation on the off-trace branch path, if any operations along the off-trace path depend upon the operation.

• **Data dependencies**. Data dependencies are managed while scheduling operations into wide instructions. Extensive analysis and optimization is performed to eliminate "surface" dependencies which result from the expression of the program, rather than from the algorithm itself. Memory references are exhaustively analyzed, using index expression derivation analysis and symbolic evaluation, to minimize conflicts among array references.

The code generator first builds a graph showing every operation on the trace, and the data dependencies among operations. It then uses this graph to assign operations to functional units, to assign registers, and to generate an efficient sequence of wide instruction words.

In placing operations, the code generator takes advantage of the exposed nature of the hardware, deciding which functional unit should perform each operation.

Memory reference analysis and detailed compiler knowledge of the TRACE memory structure further allows compile-time management of memory banks. This provides high memory bandwidth via an interleaved, pipelined memory system without "stunt boxes" or hardware memory reference schedulers.

• **Hardware resources.** Multiflow compilers incorporate a detailed model of the TRACE hardware which includes functional unit opcodes, pipeline depths, resource requirements, and datapath interconnect. The compilers completely control the operation of the hardware on a cycle-by-cycle basis, and manage system hardware resources such as buses, functional units, memory banks, and register write ports. Control and scheduling hardware has been replaced by compiler management.

CHAPTER 4
TRACE COMPUTER SYSTEMS

Multiflow TRACE computers are an upgradable, compatible processor family offering from 53 to 215 VLIW MIPS and from 30 to 120 MFLOPs single precision, and from 15 to 60 MFLOPs double precision. Floating point computation conforms to IEEE standard 754. Up to 28 fast functional units operate simultaneously, in a single, synchronous execution stream, under the control of a Very Long Instruction Word. The central processor includes multiple high-performance integer/logical units, memory reference units, floating point multiply/divide units, floating point add/logic units, and a floating point square root unit.

TRACE systems are designed for integrity and reliability using high-speed, low-power CMOS VLSI componentry with advanced Schottky TTL support logic.

TRACE systems feature large, high-bandwidth, low-cost main memory, with sustained performance to 492 Megabytes per second and capacity to 512 Megabytes. Memory is demand-paged and virtually addressed, with 4 Gigabytes per user process.

Unlike coarse-grained parallel systems, no high level regularity in the user's code is required to make effective use of the hardware. VLIW instructions can express any pattern of parallel execution. Unlike multiprocessor systems, there is no penalty for synchronization or communication. All functional units run completely synchronized, directly controlled in each clock cycle by the compiler. No queues, recognizers, or interrupt mechanisms are required to move data about the processor.

The true cost of every operation is exposed at the instruction set level, so that the compiler can optimize operation scheduling. Pipelining allows new operations to begin on every functional unit in every instruction. Exposed concurrency allows the hardware always to proceed at full speed, since the functional units never wait for each other.

Pipelining also improves the system clock speed. Judiciously used small scale pipelining of operations like register-to-register moves and integer multiplies, together with the more traditional floating point calculation and memory reference pipelines, allows a fast clock rate for TRACE CPUs with low-cost, reliable electronics.

There is no microcode. Hardware directly executes the Very Long Instruction Words, without the overhead of intermediate interpretation or decoding steps.

Over the last two decades, the cost of computer memory has dropped much faster than the cost of logic, making the construction of a VLIW, which replaces scheduling logic with instruction-word memory, practical and attractive.

VLIWs yield a type of computation that users are accustomed to. Unlike multiprocessor systems, VLIW execution patterns are fully static; the fine-grained execution pattern does not depend on how many users are using the machine concurrently. Programs run the same way every time, improving software testability and problem isolation.

TRACE systems are designed for large-scale applications. Functional units are optimized for 64-bit floating-point intensive computations, with high performance integer and 32-bit floating computations. Excellent multi-user Unix performance is achieved; many hardware features improve performance in a multiple-process interactive environment.

4.1 SYSTEM ARCHITECTURE

TRACE computers are modular, expandable machines. The core circuitry is implemented in low-power, reliable 2 micron CMOS VLSI, with advanced Schottky TTL support logic. Main memory is implemented with high density, low-cost Dynamic RAMs, using pipelined and interleaved design techniques for very large capacity and performance at low cost.

Six basic module types make up the system: Integer Units, Floating-Point Units, Memory Controllers, Memory Modules, I/O Processors, and a Global Controller. The core of the Integer and Floating Point units is built in 8000-gate 2-micron CMOS gate arrays.

Compact packaging provides reliability and minimal space requirements. A single 24 inch backplane acommodates CPU and memory in all configurations.

4.2 INTEGER UNIT

The TRACE integer instruction set comprises over 80 operations, including arithmetic, logical, and compare operations; high performance primitives for 32-bit and 64-bit multiplication; shift, bit-reverse, extract, and merge operations for bit and byte field manipulations; and pipelined 32-bit and 64-bit load and store operations for referencing memory.

Each Integer Unit contains two Arithmetic/Logic Units units (ALU0 and ALU1) associated with a register bank of 64 general-purpose 32-bit registers. The register bank incorporates multiple read and write ports and a bus-to-bus crossbar among its twelve bus ports. During one instruction, eight reads, eight writes, and eight bus-to-bus data moves can be accommodated.

Each instruction executes in two 65 nanosecond minor cycles, or "beats." Each ALU performs a new 32-bit operation during each beat; four separate integer or address computations are performed on each Integer Unit during each instruction. Integer operations complete immediately, without pipeline delay.

A Program Address unit provides target branch addresses. Prioritized conditional branch operations are based on the results of comparison operations.

Substantial support is provided for injecting immediate constants into computation. Each ALU can use a 6-bit, 17-bit, or 32-bit immediate constant, under the control of the instruction word. A 32-bit immediate field of the instruction is flexibly shared between ALU0, ALU1, and the Program Address unit.

The Integer Unit also contains a virtual to physical address translation buffer (TLB). The TLB contains 4K process-tagged entries, so that the TLB need not be flushed at every context switch and provides a high multi-user hit rate. Operating system software manages TLB contents and handles "misses". TRACE systems support 4 gigabytes of data address space per process.

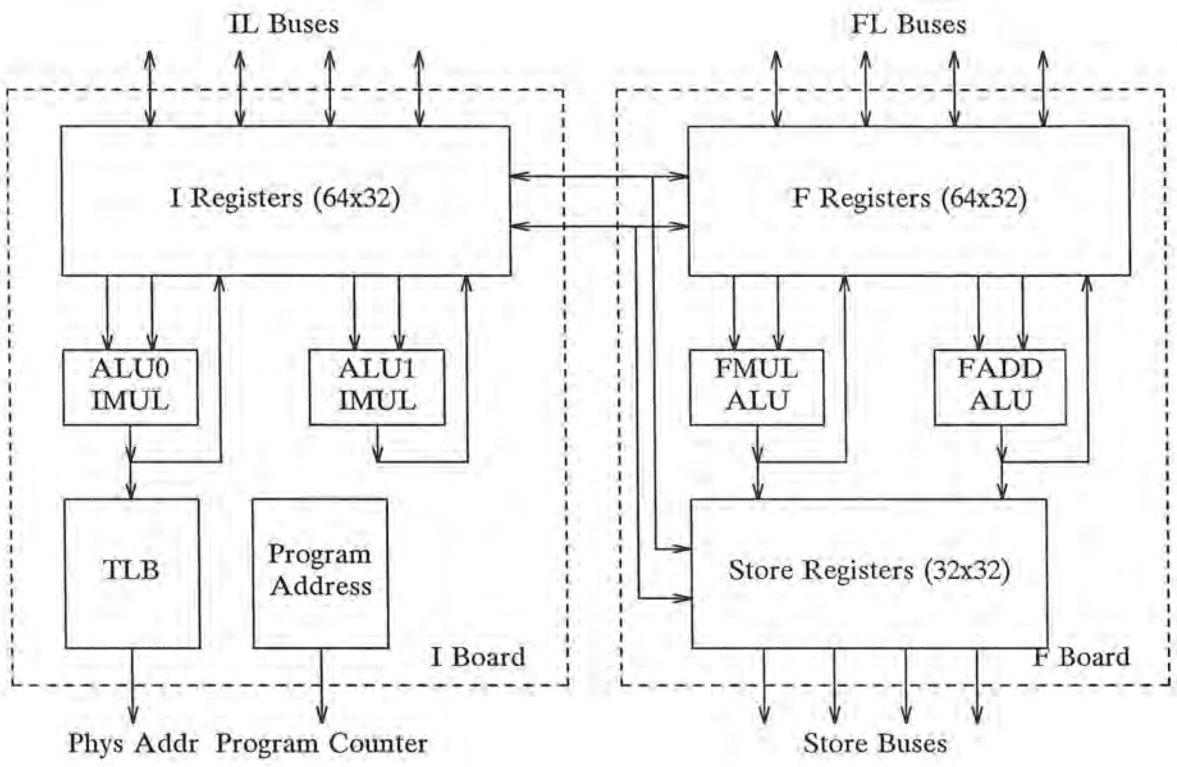

Figure 10: TRACE 7/200 Major Datapaths

4.3 FLOATING-POINT UNIT

Like the Integer Unit, the Floating-Point Unit contains a bank of 64 general-purpose 32-bit registers with a bus-to-bus crossbar. 32-bit registers are used in pairs to hold 64-bit values. Functional units include a floating-point multiplier/divider (FMUL), a floating-point adder (FADD), and two integer ALUs. An additional register bank of 32 "Store" registers expands register bandwidth and improves memory reference performance.

The floating operation suite includes the integer opcodes, plus floating opcodes for addition, subtraction, multiplication, division, type conversion, and comparison.

Exception handling hardware provides several modes of operations, including full compliance with IEEE 754 exception processing.

Pipelined design techniques allow the multiplier, the adder, and the integer ALUs to initiate a new operation with every instruction regardless of the previous instruction. 64-bit floating addition has a pipeline depth of six beats, or 390 ns; 64-bit floating multiplication has a seven beat pipeline.

Word 0: I 0 ALU 0, Early beat.

31 25	24 19	18 16	15 13	12	11 7	6 1	0
opcode	dest	dest_bank	branch_test		src1	src2	Imm

Word 1: Immediate constant 0 (early).

31 0
immediate constant (early)

Word 2: I 0 ALU 1, Early beat.

31 25	24 19	18 16	15 13	12	11 7	6 1	0
opcode	dest	dest_bank	branch_test		src1	src2	Imm

Word 3: F 0 FA/ALUA control fields.

31 25	24	23	22 17	16	15 11	10 6	5	4	3 1	0
opcode	64		dest		src1	src2			dest_bank	

Word 4: I 0 ALU 0, Late beat.

31 25	24 19	18 16	15 13	12	11 7	6 1	0
opcode	dest	dest_bank			src1	src2	Imm

Word 5: Immediate constant 0 (late).

31 0
immediate constant (late)

Word 6: I 0 ALU 1, Late beat.

31 25	24 19	18 16	15 13	12	11 7	6 1	0
opcode	dest	dest_bank			src1	src2	Imm

Word 7: F 0 FM/ALUM control fields.

31 25	24	23	22 17	16	15 11	10 6	5	4	3 1	0
opcode	64		dest		src1	src2			dest_bank	

Figure 11: TRACE 7/200 Instruction Word Format

Figure 11 shows the instruction word for a TRACE 7/200. The representation as 32-bit words corresponds to the way instructions are stored in main memory; the entire 256-bit instruction is fetched and executed simultaneously.

The instruction word for a TRACE 28/200 has this format replicated four times, with separate fields of the instruction controlling the operations of multiple Integer Units and Floating Units.

4.4 THE MEMORY SUBSYSTEM

Unique cooperation between Multiflow's compilers and the TRACE hardware architecture allows the construction of a memory system which can sustain high bandwidth without the limitations of data caches or the costs of hardware memory-reference schedulers.

The speed of the CPU/memory interconnection in a computer system is a major determining factor in overall system performance. Every modern computer design has to handle the mismatch in speed between the dynamic RAMs used to build the memory and the (much faster) logic used to build the processor.

For standard sequential processors which cannot express or tolerate parallelism among memory references, the overall memory bandwidth `B = 1/L`, the latency for a single reference. Sequential processors frequently include *cache memories* for data references. Caches provide lower latency than main memory when there is reasonable locality of reference in the access pattern. Shared-memory multiprocessors depend upon caches to lower the memory bandwidth requirements of each processor, so that multiple processors can share a bus to memory.

Data caches, however, work poorly in many scientific applications, where very large arrays of data are repeatedly accessed. During matrix solves and other large-scale computations, cache hit rates fall off rapidly. System performance degrades to the performance of the main memory.

High-end supercomputers employ a different approach. Through overlapped scalar execution, and through vector memory references, parallelism among memory references can be identified. In this case, overall memory bandwidth is potentially `B = N / L`, where N is the number of references pending simultaneously. The memory system is interleaved and pipelined. Memory addresses are spread across multiple independent banks of RAMs. While one or more new references may be initiated in each cycle, each reference requires multiple cycles to complete.

One problem in high-end supercomputer design is managing the status of several simultaneously outstanding references. In a traditional scalar or scalar/vector supercomputer, a hardware bank scheduler, or "stunt box", is required to track the busy status of each bank of memory, watch each memory address, and prevent conflicts by temporarily suspending execution. Conflicts would arise if a single bank of RAMs were accessed while still busy processing an earlier reference.

Multiflow TRACE systems achieve the consistently high memory performance of large-scale supercomputers without the costs and complexities of hardware scheduling. The TRACE architecture incorporates a software-managed interleaved memory system. Memory addresses are spread across multiple independent banks of RAMs. The memory system is pipelined; up to eight new references may be initiated in each instruction. Multiple RAM banks cycle simultaneously to provide massive bandwidth. Multiflow's compilers incorporate knowledge of the TRACE memory bank structure, and generate code that executes correctly and at high performance without requiring hardware scheduling.

Multiflow's compilers schedule memory references by analyzing array index expressions and the compiler's placement of data in memory. The compiler builds derivation trees for array index expressions and solves for whether or not references can conflict.

Under this approach, software guarantees that conflicts cannot occur. This allows TRACE computers to provide uniformly high speed access to very large main memories at very low cost.

The memory subsystem consists of up to eight Memory Controllers, each of which carries up to four Memory Modules. Sixteen 32-bit data and address buses interconnect the TRACE CPU and Memory Controllers, for high sustainable performance. Memory references, including virtual address translation, have a seven beat pipeline depth.

Each Memory Module carries two independent banks of RAM. Memory addresses are interleaved among controllers and banks. A fully populated system is 64-way interleaved, with a capacity of 512 Megabytes, using 1 Megabit DRAM technology.

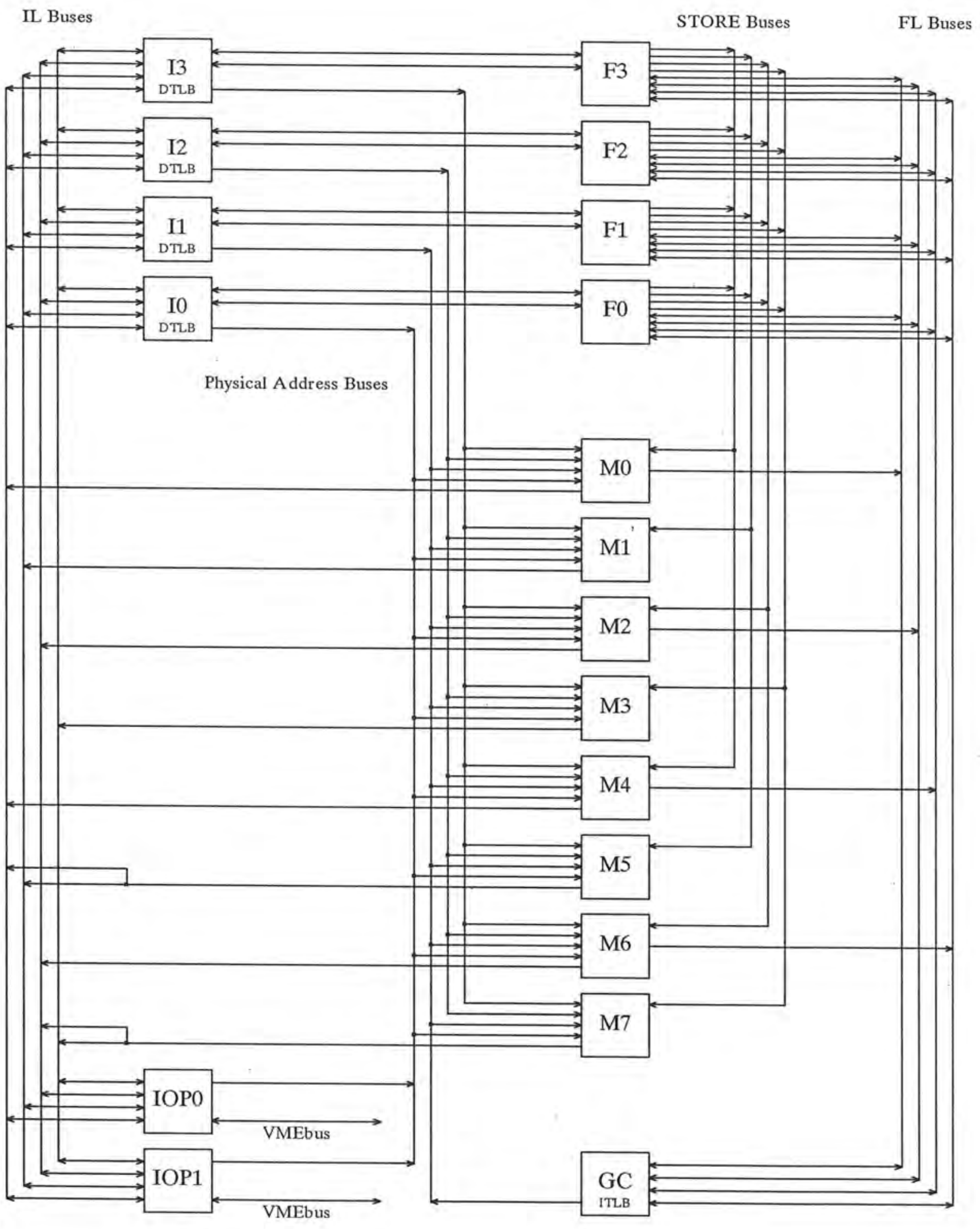

Figure 12: TRACE System Bus Architecture

4.5 INSTRUCTION CACHE

The CPU's Global Controller coordinates instruction fetch and execution and controls the Instruction Cache. The Instruction Cache is virtually addressed and process-tagged, for excellent multi-user performance. Due to process tagging, cache "cold starts" occur rarely, not at every process context switch. The Instruction Cache accomodates 8K instructions in all processor configurations. In a fully configured system, this corresponds to 1 megabyte of instruction cache with a cache bandwidth of 984 megabytes per second.

Instruction virtual addresses are translated to physical addresses during cache refill through a dedicated instruction-stream translation buffer (ITLB). The ITLB has 4K process tagged entries, with an addressing scheme which optimizes multiple-process hit rates.

Because TRACE instructions are directly executed without decoding or microcode dispatching, instruction fetching and branching is fully overlapped with execution and never stalls or restrains the processor except on cache misses. Conditional branch operations never delay computation.

4.6 INSTRUCTION COMPACTION

Not every very long instruction word in every compiled program will be fully packed with operations. Yet each instruction in a fully configured machine is 1024 bits long. The unused parts are filled with no ops, which would take up a significant amount of memory if stored there. To save main memory space, TRACE systems store instructions in memory in a compressed form. An encoding is used such that empty instruction fields take up no physical space in the main memory. This allows maximum expressible parallelism with minimal object program size.

4.7 MULTI-WAY JUMP

The TRACE architecture incorporates a multi-way conditional jump, used by Multiflow compilers to compact multiple source program conditional jump operations into single instructions. Up to four conditional jumps can be executed in each instruction. A software controlled priority mechanism selects one of five target addresses at which execution will continue. Multiflow compilers use this mechanism to allow simultaneous execution of multiple conditional jumps.

4.8 HARDWARE SQUARE ROOT UNIT

The Global Controller includes a high-performance floating-point Square Root Unit. The Square Root Unit extracts a 32-bit square root in 4.8 microseconds, and a 64-bit square root in 7.8 microseconds.

4.9 I/O PROCESSOR

I/O transactions on the TRACE system are handled by I/O Processors (IOPs). Up to two IOPs are currently configurable per system.

The CPU fully overlaps execution with I/O, and is relieved of interrupt-handling, optimization, and device management tasks. Each IOP includes:

- A Motorola MC68010, with 2 megabytes of local RAM
- A 512 Kbyte multiported, high-bandwidth buffer memory
- A DMA engine which reads and writes main memory at 123 MB/s
- A 6 megabyte per second interface to the VMEbus
- Interrupt facilities to/from the CPU
- A diagnostic-channel interface to the CPU

An intelligent channel protocol is used to direct I/O operations. Each IOP carries out I/O operations, stealing small bursts of time for block data transfers to and from main memory. The CPU is involved only when each high-level transaction completes. Due to the high bandwidth of the DMA engines, the CPU is slowed by at most 6% when both IOPs are fully active.

4.10 CONFIGURATIONS

Three upwards-compatible, field-upgradable processor models are currently included in the TRACE series:

The TRACE 7/200 features:

- One Integer Unit
- One Floating-Point Unit
- 256-bit instructions
- 7 operations per instruction

- 256 KByte instruction cache
- 984 Megabytes per second register bandwidth
- 53 VLIW MIPs
- 30 MFLOPs single precision
- 15 MFLOPs double precision
- 123 Megabytes per second memory bandwidth
- 32 to 512 Megabyte memory capacity

The TRACE 14/200 features:

- Two Integer Units
- Two Floating-Point Units
- 512-bit instructions
- 14 operations per instruction
- 512 KByte instruction cache
- 1968 Megabytes per second register bandwidth
- 107 VLIW MIPs
- 60 MFLOPs single precision
- 30 MFLOPs double precision
- 246 Megabytes per second memory bandwidth
- 32 to 512 Megabyte memory capacity

The TRACE 28/200 features:

- Four Integer Units
- Four Floating-Point Units
- 1024-bit instructions
- 28 operations per instruction
- 1024 KByte instruction cache
- 3692 Megabytes per second register bandwidth
- 215 VLIW MIPs
- 120 MFLOPs single precision
- 60 MFLOPs double precision
- 492 Megabytes per second memory bandwidth
- 32 to 512 Megabyte memory capacity

The family is upward compatible; object code compiled for a smaller model will run unchanged on a larger one. Recompilation is required to obtain higher performance on the larger system.

4.11 INTEGRITY, RELIABILITY AND SERVICEABILITY

TRACE systems are intended to handle enormous computations with large numbers of hardware functional units. Accordingly, Multiflow has taken great care in the system design to assure computational integrity, and to eliminate undetected transient errors.

All general registers, buses, and major datapaths incorporate parity checking. Parity hardware verifies that data is correctly transmitted and stored. Byte parity is computed on-chip when functional units generate results, and checked on-chip when functional units accept operands. Parity flows throughout the system and is stored in registers, for end-to-end protection. To avoid impacting the system cycle time, parity is transmitted and checked one clock period following data.

Main memory is protected by error detection and correction logic which automatically corrects all single-bit errors and detects all double-bit errors. Seven extra bits are stored for each 32-bit field stored in memory, providing enough redundancy to allow all single-bit errors to be transparently corrected, and all double-bit errors to be detected. A memory scrubbing protocol ensures that all of main memory is accessed and rewritten with corrected data on a regular basis. Error correction significantly improves system reliability, as it compensates for the primary source of errors in modern computer systems: soft errors (dropped bits) in dynamic RAMs.

The Instruction Cache and address translation buffers detect and correct single-bit errors. Byte parity is stored with the cache data; parity errors are treated as cache misses, and the cache entry is transparently reloaded from the backing store. This technique has a similar impact on overall system uptime; occasional static RAM errors can be tolerated.

The master IOP runs a small support operating system, MDX (Multiflow Diagnostic Executive), and is responsible for power-on test and system bootstrap. It accesses a dedicated disk partition with an independent file system containing bootstrap programs, diagnostics, and error logs.

MDX provides a single-process UNIX environment running on the IOP, including shell and system calls. This allows for rapid development and deployment of diagnostics and CPU monitoring tools. MDX shares device drivers with TRACE/UNIX, and can operate concurrently with TRACE/UNIX.

The master IOP further supports the TRACE's comprehensive diagnostic facility. The IOP includes a special-purpose diagnostic channel to probe every part of the TRACE system. Through the diagnostic channel, diagnostic programs have direct access to every signal on the CMOS VLSI arrays as well as access to cache, registers, and logic throughout the system. This provides major benefits in fault isolation. The diagnostic channel also configures every board; no switches or jumpers are present on CPU or memory boards. This allows full remote reconfiguration and test, and improves reliability.

The TRACE Environmental Processor (EP) monitors operating conditions and enforces safety limits. The EP communicates with the operator console, front panel, master IOP, and remote diagnosis modem. It monitors system DC voltages, temperature, air flow, and AC power. The EP controls AC power on/off, and power supply DC voltage settings .

Via the EP and the diagnostic channel, diagnostic software can exercise the system at high/low voltage margin conditions and high/low clock rates. Systems are tested in manufacturing and at customer sites at these margin conditions to ensure reliable operation, with a substantial "safety zone" beyond normal operating conditions.

The EP allows all this to be done remotely, via telephone lines and a built-in modem. Under customer keyswitch and password control, Multiflow service engineers can remotely check out any system, control system power, measure and margin voltages, run diagnostics, and study the results.

CHAPTER 5
TRACE/UNIX OPERATING SYSTEM

Multiflow's TRACE systems run the TRACE/UNIX operating system, a version of the Berkeley 4.3 BSD UNIX operating system that has been extended and enhanced for scientific applications.

TRACE/UNIX harnesses the power of TRACE computer systems and delivers it to the user. TRACE/UNIX provides a kernel which manages the machine, a powerful and flexible command language, an easy-to-use hierarchical file system, and a large suite of utility programs for text processing, program development, and communications.

TRACE/UNIX includes functional and performance enhancements for the technical computing environment. Optional packages provide transparent distributed file capabilities (NFS), communications with DEC systems (DECnet compatibility), inter-vendor remote procedure call (Network Computing System), and communications with IBM systems.

Multiflow's TRACE/UNIX provides unprecedented operating system performance. All programs running on the TRACE, including utilities and the TRACE/UNIX kernel itself, run at high speed via VLIW overlapped execution. This approach differs sharply from other high-performance systems which run the operating system on an external, slower processor. In the TRACE/UNIX environment, the operating system does not become a bottleneck that limits performance.

5.1 ONE ENVIRONMENT

UNIX has become the industry standard for technical computing environments. Compatible UNIX systems span the range from personal computers to high-end supercomputers. A robust set of networking, file sharing, and remote graphics tools allows Multiflow TRACE systems to fully participate in this continuum. Multiflow systems can act as compute servers for large networks of UNIX-based systems without user retraining.

5.2 SUPPORT FOR LARGE-SCALE FORTRAN COMPUTATION

The primary goal of TRACE/UNIX is high-quality support for execution and development of large-scale scientific and engineering FORTRAN applications.

TRACE/UNIX supports very large programs and data. All the tools involved in software development – compilers, debuggers, linkers, profilers, configuration managers, and filesystem tools – have been tuned to handle programs many hundreds of thousands of lines long, contained in hundreds or thousands of source files. TRACE/UNIX handles extremely large files, not bounded by the size of disk volumes.

TRACE/UNIX includes a simple, complete batch system to manage large-scale compute-intensive jobs. Network batch service allows convenient use of TRACE systems as network compute servers.

5.3 COMMAND LANGUAGES

A command interpreter, or *shell*, provides operating system services to the user. TRACE/UNIX offers two shells which provide simple, consistent, powerful command of the system. Both the AT&T-developed Bourne shell (*sh*) and the Berkeley-developed C shell (*csh*) are fully supported by TRACE/UNIX.

The two shells perform similar functions and accept similar command languages. Both offer the full power of Unix, including multiple process management, and simple I/O retargeting. They differ in their programming facilities: loop constructs, string handling, and the like.

The interactive user sees the shell as his command environment. The shell accepts input from the terminal, and finds system or user programs and executes them as the user specifies.

The shell, in cooperation with the Unix operating system, can direct output from one program to be the input to another. Shell commands can be combined in simple ways and be stored in command files, so that many routine programming tasks can be accomplished without writing C or FORTRAN programs.

5.4 SYSTEM EFFICIENCY

A number of Multiflow extensions increase TRACE/UNIX system efficiency in handling large scientific programs.

TRACE/UNIX pages programs from the executable file itself, using backing store ("swap space") only for the program's private data. This greatly speeds starting large programs, and increases the amount of virtual memory supportable by a given configuration.

TRACE/UNIX provides "copy-on-write" process creation. When a new process starts, its address space is shared with its parent. Pages are copied only when the new process attempts to modify shared pages. This eliminates large amounts of data copying at process start-up.

A shared library facility boosts system performance by maintaining only one copy of widely used subroutines, paging this library on a system-wide basis. This facility improves disk usage efficiency, eliminating copies of widely used routines in every program image.

High-accuracy timing facilities assist in program measurement and tuning. The TRACE hardware provides counters which measure CPU time, accurate down to the clock cycle. These timers are made available to TRACE/UNIX and directly to user code, for low-overhead precise time measurement.

5.5 A SUPERCOMPUTER FILE SYSTEM

The TRACE/UNIX file system provides an easy-to-use hierarchy of directories and files spanning multiple physical devices. A single, consistent naming strategy allows specification of files anywhere throughout the local and network file system. Files are byte sequences. TRACE/UNIX imposes no internal structure on files, such as records or blocking. User programs have complete control over file contents.

TRACE/UNIX incorporates significant file system enhancements beyond other UNIX implementations.

TRACE/UNIX manages its file system using up to 32 KByte disk blocks, a data cache of up to 32 Megabytes, directory look-up caching, file write-behind and read-ahead, and a tuned file allocation algorithm. This provides for very high filesystem performance across a broad range of applications, including general timesharing and program development. The filesystem has been tuned for very large files.

TRACE/UNIX includes a facility for explicit, asynchronous, physical I/O. This provides large-scale applications the ability to explicitly overlap I/O and computation, with I/O proceeding directly from buffers in program address space. I/O intensive applications can directly manage I/O strategies for best performance.

The TRACE/UNIX file striping facility allows filesystems to be transparently spread across multiple disk units and controllers, providing configurable file system bandwidth. High-throughput file access is available transparently to the application program.

5.6 HIGH PERFORMANCE I/O

TRACE/UNIX, in concert with the TRACE system architecture, provides high performance I/O to provide balanced system throughput.

I/O driver modules run primarily on the TRACE system I/O Processors, offloading the CPU and minimizing overhead. IOPs access main memory in bursts at 123 megabytes per second and interrupt the CPU only when I/O transactions complete.

Devices are configured in TRACE/UNIX system at boot time. No "sysgen" or custom kernel linking is required to handle site-specific device configurations. A configuration file, managed by the MDX operating system running on the IOP, specifies the correspondence between physical and logical devices to the operating system. This makes device upgrades and system configuration flexible and simple.

5.7 HIGH PERFORMANCE WITH MULTIPLE PROCESSES

TRACE systems were designed to run large numbers of user processes at high speed; the design goal was to support as many users as would be comfortable on a modern, large supermini, but to support order-of-magnitude larger computations than current superminis could support.

Context switch speed is an important factor in the effectiveness of a computer system in handling many users. Context switch time is often considered to be simply the cost of saving and restoring registers. However, the full cost of a context switch also includes the interrupt time, scheduling overhead, and any penalty for cache purging and cold-start. On many machines, the cost of purging the virtual address translation (TLB) and instruction caches is far higher than the time required for register saving. The TRACE provides very large instruction and translation caches, which are process-tagged with an 8-bit "Address Space ID". No purging of the cache or TLBs is necessary on a context switch.

Updating the ASID registers is cheap, and the high available memory bandwidth allows a complete context switch in 300 microseconds. This figure makes it possible for TRACE systems to comfortably support large numbers of users.

5.8 SOFTWARE PRODUCTIVITY TOOLS

TRACE/UNIX includes a range of utilities for software development, including compilers, debuggers, profilers, and source code management tools.

Software configuration and revision control is accomplished via the *make* utility and the Revision Control System (*RCS*).

Using *make*, a developer declares the structure of his application once. As code is developed, *make* ensures that the latest changes are incorporated each time a program is built, and that all configuration requirements have been met. *Make* eliminates manual steps in software development, reducing the potential for errors.

RCS is a source-code management tool designed to aid in managing large programming projects. It allows the maintenance of all versions of a program in a recoverable form; eliminates problems stemming from multiple programmers modifying the same code at once; and provides a complete audit trail for revisions.

Two debuggers are provided: *adb* and *dbx*. *Adb* is an assembly-language debugging utility. *Dbx* is a source-language debugging utility for C and FORTRAN programs, providing breakpointing, source-level code and data examination, and a wide range of powerful features to speed application debugging. TRACE/UNIX is unique in its support for "address break". *Dbx*, in conjunction with special address break hardware built into the TRACE, allows the user to suspend execution whenever a specified memory location is accessed.

TRACE/UNIX includes three profilers: a branch-probability profiler integrated with Multiflow's C and FORTRAN compilers for improving trace-picking heuristics, the *prof* profiler which provides program-counter histogramming, and the *gprof* profiler which further provides call-graph information. Together, these tools provide the developer accurate information about program behavior to aid in performance tuning.

The shell provides symbolic traceback on program errors. Program problems are pinpointed with line number and call-stack information.

5.9 SYSTEM UTILITIES

TRACE/UNIX includes the complete set of 4.3BSD utilities. These provide technical professionals a toolkit for a wide range of tasks beyond numerically intensive computation. Major applications include:

- Interactive screen editors. Three editors (plus an optional EDT-compatible fourth) provide session logging, intelligent formatting, and compatibility with a wide range of terminals and workstations.

- File and filesystem management. Many tools for file comparison, sorting, searching, directory and filesystem management are included.

- Personal utilities. Extended-precision desk calculators, calendar utilities, and electronic mail.

- Text processing. Spelling checkers, dictionaries, and typesetting tools including table and equation processors.

- Online documentation. Interactive tutorials for TRACE/UNIX and screen editors, complete online documentation, and keyword search facility.

5.10 DEC COMPATIBILITY

The TRACE/UNIX VMS compatibility package provides the VMS-knowledgeable user a bridge to the power of TRACE systems without retraining.

Multiflow's compatibility package offers compatibility at three levels:

- User skills. We don't want to impose retraining requirements for engineers and other technical professionals who use large-scale computation in support of their jobs. We offer an EDT-compatible text editor and DCL-compatible command interpreter, so that the environment the user sees is a familiar and productive one without retraining.

- FORTRAN programs. Multiflow FORTRAN includes extensions for compatibility with VAX FORTRAN. We will make it easy to move large FORTRAN applications from VAX systems without modification.

- Network Environment. Multiflow TRACE systems can operate as end nodes in DECnet Phase IV networks, allowing a TRACE to be as transparently integrated as another VAX. No hardware, software, or user training changes are required to use TRACE systems as network resources.

CHAPTER 6
RESULTS

The combination of VLIW architecture and Trace Scheduling compacting compilers yields a substantially more cost-effective computing system than any other approach. Multiflow's technologies obsolete vectorization and other coarse-grained parallel approaches to large-scale computing.

Multiflow systems are cost-effective not only in CPU price/performance, but in user effort as well. Multiflow's technologies deliver full performance without application restructuring. Unlike vectorization or "parallelization", Trace Scheduling compacting compilers require no special loop structures to detect and exploit parallelism. Every loop in every program benefits from overlapped execution. Straight-line, loop-free code also receives the full performance benefit of VLIW overlapped execution. At Multiflow, we don't talk about what percentage of the loops we've "parallelized"; our number is always 100%. Instead, we measure how successful our global optimizations are on any given application.

Multiflow is delivering performance and value to users today. Early performance results reflect the "breakthrough" nature of Trace Scheduling and VLIW technologies. Figure 13 presents early performance results for Multiflow's entry-level TRACE 7/200. The TRACE 7/200 is, by a large margin, the lowest cost system listed in the table.

	TRACE 7/200	DEC 8700	Convex C-1 XP	Alliant FX/8-8	IBM 3090-200	Cray X-MP/12
Industry Standard Benchmarks: (Units)						
Compiled Linpack Full Precision (MFLOPs)	**6.0**	0.97	3.0	7.6	6.8	24.0
Whetstone (Double Precision KWHETs)	**12605**	3953	4200	3630	25000	35000
Livermore Loops (Double Precision, 24-kernel MFLOPs)	**2.3**	N/A	1.2	1.6	N/A	9.8
Dhrystone (DHRYs)	**14195**	8500	7000	6500	31250	17857

Figure 13: Initial Multiflow TRACE 7/200 Performance Results