HEWLETT PACKARD

Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler

Stefan M. Freudenberger, Thomas R. Gross*
P. Geoffrey Lowney**
Computer Research Center
HPL-93-35
May, 1993

instruction level parallelism, compilation, code generation, trace scheduling, optimization, compiler evaluation

Trace scheduling is an optimization technique that selects a sequence of basic blocks as a trace and schedules the operations from the trace together. If an operation is moved across basic block boundaries, one or more compensation copies may be required in the off-trace code. This paper discusses the generation of compensation code in a trace scheduling compiler and presents techniques for limiting the amount of compensation code: avoidance (restricting code motion so that no compensation code is required) and suppression (analyzing the global flow of the program to detect when a copy is redundant). We evaluate the effectiveness of these techniques based on measurements for the SPEC89 suite and the Livermore Fortran Kernels, using our implementation of trace scheduling for Multiflow Trace 7/300. The paper compares different compiler models, contrasting the performance of trace scheduling with the performance obtained from typical RISC compilation techniques.

There are two key results of this study: First, the amount of compensation code generated is not large. For the SPEC89 suite, the average code size increase due to trace scheduling is 6%. Avoidance is more important than suppression, although there are some kernels that benefit significantly from compensation code suppression. Since compensation code is not a major issue, a compiler can be more aggressive in code motion and loop unrolling.

Second, compensation code is not critical to obtain the benefits of trace scheduling. Our implementation of trace scheduling improves the SPECmark rating by 30% over basic block scheduling, but restricting trace scheduling so that no compensation code is required improves the rating by 25%. This indicates that most basic block scheduling techniques can be extended to trace scheduling without requiring any complicated compensation code bookkeeping.

*Carnegie Mellon University, Pittsburgh, PA,**Digital Equipment Corporation, Hudson, MA
Formerly published by Carnegie Mellon University as CMU-CS-93-145, reprinted with permission from Carnegie Mellon.

Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler

Stefan M. Freudenberger
HP Laboratories
Thomas R. Gross
Carnegie Mellon University
and
P. Geoffrey Lowney
Digital Equipment Corporation

Abstract

Trace scheduling is an optimization technique that selects a sequence of basic blocks as a trace and schedules the operations from the trace together. If an operation is moved across basic block boundaries, one or more compensation copies may be required in the off-trace code. This paper discusses the generation of compensation code in a trace scheduling compiler and presents techniques for limiting the amount of compensation code: avoidance (restricting code motion so that no compensation code is required) and suppression (analyzing the global flow of the program to detect when a copy is redundant). We evaluate the effectiveness of these techniques based on measurements for the SPEC89 suite and the Livermore Fortran Kernels, using our implementation of trace scheduling for a Multiflow Trace 7/300. The paper compares different compiler models, contrasting the performance of trace scheduling with the performance obtained from typical RISC compilation techniques.

There are two key results of this study: First, the amount of compensation code generated is not large. For the SPEC89 suite, the average code size increase due to trace scheduling is 6%. Avoidance is more important than suppression, although there are some kernels that benefit significantly from compensation code suppression. Since compensation code is not a major issue, a compiler can be more aggressive in code motion and loop unrolling.

Second, compensation code is not critical to obtain the benefits of trace scheduling. Our implementation of trace scheduling improves the SPECmark rating by 30% over basic block scheduling, but restricting trace scheduling so that no compensation code is required improves the rating by 25%. This indicates that most basic block scheduling techniques can be extended to trace scheduling without requiring any complicated compensation code bookkeeping.

Authors' addresses: S. M. Freudenberger: Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304; T. R. Gross: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15123; P. G. Lowney: Digital Equipment Corporation, HLO2-3/J03, 77 Reed Road, Hudson, MA 01749-2895. The implementation described in this paper was performed while S. M. Freudenberger and P.G. Lowney were with Multiflow Computer, Inc.; T. Gross was a consultant to Multiflow Computer, Inc. Gathering of the data and writing of the paper was supported by Digital Equipment Corporation, Hewlett-Packard Company, and the School of Computer Science of Carnegie Mellon University.
An earlier version of this paper was presented at the 3rd Workshop on Languages and Compilers for Parallel Processing[12].

1 Introduction

Trace scheduling is an optimization technique that has been used for VLIW, superscalar, and pipelined processors. Whereas a conventional instruction scheduler considers the operations in a single basic block when scheduling instructions, a trace scheduler employs a more global view. The key idea is that the trace scheduler selects a group of operations from a sequence of basic blocks, either based on heuristics or actual frequency information, and schedules these operations as if they were in a single basic block [9]. If operations are moved across basic block boundaries, such optimizations require that the overall program be fixed up to account for the code movements. For example, if an operation that appeared above a branch is moved below this branch, then the code outside of the trace must be fixed up by inserting a *compensation copy* into the off-trace code.

A frequent criticism of trace scheduling is the potential for code explosion due to compensation copies, and the problem of code growth was noted by Fisher in his original presentation of the algorithm [9]. In this paper we show that this is not a practical concern. This paper examines the issue of compensation copies in depth and discusses various techniques for limiting the amount of compensation code generated.

There are two techniques to deal with compensation code: avoidance (restricting code motion so that no compensation code is required) and suppression (analyzing the global flow of the program to detect when a copy is redundant). These techniques are successful; we show that the amount of compensation code generated is small over a variety of programs (the SPEC89 suite [27] and the Livermore Fortran Kernels [20], using the Multiflow Trace 7/300 as the target machine [5]). To evaluate the impact on the execution of programs, we measure the dynamic effects of trace scheduling with three types of code motion:

1. Code motion restricted to basic blocks. This approximates most RISC compilers.
2. Code motion restricted so that no compensation code is required. (This is essentially scheduling over extended basic blocks.[1])
3. Full trace scheduling.

The performance gain of full trace scheduling over basic block scheduling is large (30%); the performance gain of trace scheduling with no compensation code is also large (25%). This result implies that a compiler can get much of the benefit of trace scheduling without the complication of compensation code. However, the incremental performance gain of full trace scheduling over trace scheduling with no compensation code is significant for some programs (10% for programs as diverse as the GNU C compiler, a Lisp interpreter, and the Livermore Fortran Kernels), and the cost in code size is small (an average 6% increase for the SPEC89 suite).

The study presented in this paper is based on the Multiflow compiler; the reader is referred to [19] for a full discussion of Multiflow's trace scheduling compiler. Section 2 summarizes the notations used in this paper; Section 3 describes techniques for avoiding the need for compensation code in a realistic trace-scheduling compiler; and Section 4 describes the incremental algorithm implemented to suppress compensation copies. We discuss the interaction between compensation code suppression and register allocation in detail to allow the reader to understand the issues that a practical implementation has to deal with. Section 5 presents an evaluation of compensation code avoidance and suppression; we compare the different techniques to handle compensation code (avoidance and suppression) for different types of code motion. Section 6 discusses related work.

[1] In this paper, we follow [1] and define an extended basic block to be a sequence of basic blocks with multiple exits. Specifically, an extended basic block is a sequence of blocks $B_1, ..., B_k$ such that for $1 \leq i < k$, B_i is the only predecessor of B_{i+1}, and B_1 does not have a unique predecessor.

2 Background

The Multiflow trace scheduler takes a sequence of operations, called a *trace*, and generates a sequence of machine instructions, called a *schedule*. A trace consists of machine-level operations with symbolic operands; the operations have been subject to the usual set of local and global optimizations.[2] In a wide instruction word machine, each instruction consists of multiple fields, with each field specifying a single *machine operation*. The trace scheduler is then faced with these main tasks:

- allocate machine resources (functional units, busses) to execute the machine operations;
- allocate registers to hold the operands of the machine operations;
- schedule the machine operations in machine instructions.

These tasks are highly interrelated, and one of the challenges of engineering a compiler is to find a workable partitioning that produces reasonable code. The Multiflow compiler divides this work between two modules, a trace scheduler TS and an instruction scheduler IS. TS calls IS, using the following algorithm:

- TS selects a sequence of basic blocks to be scheduled together; this sequence of blocks is called a trace. Traces are limited by several kinds of boundaries; the most important boundaries are routine entry and return, loop back edges (traces do not cross the back edge of a loop), and previously scheduled code.
- TS passes the trace to the instruction scheduler IS, which allocates machine resources and registers, and produces a schedule.
- TS replaces the trace with the schedule in the flow graph. TS adds compensation code, if necessary, to correct for code motion across basic block boundaries. This step is called *bookkeeping*.
- TS repeats these steps until all operations have been scheduled.

For example, consider this C code fragment:

```
if (c != 0)
   { b = a / c; }
else
   { b = 0; }
f = g + h;
```

After the usual local optimizations, the compiler obtains a sequence of intermediate operations as shown in Figure 1.

Figure 2 shows the flow graph for this code segment. Each node represents a single operation. A simple arrows indicates the canonical execution order; dashed arrows indicate control transfers. Notice that 'goto' operations do not show up explicitly in the flow graph.

Let us assume that `(c != 0)` most of the time, so the most likely execution path includes the then-clause of the if-statement as shown in Figure 2. All the operations in the darkly shaded region make up the first trace and are processed by the trace scheduler together; the resulting schedule of instructions for operations A to I is shown in Figure 3. A rectangle represents an instruction, possibly containing more than one operation. For example, instruction 1 consists of the first operation, A (`t1 = LOAD c`), instruction 2 consists of operation C (`t2 = LOAD a`), etc. Notice how operations F and G have moved relative to the other operations in the flow graph.

[2] A trace scheduler can be implemented as an optimization on intermediate code, if an estimate of the final code schedule can be produced at this time in the compiler.

```
START:
        t1 = LOAD c
        branch to ELSE if t1 == 0
THEN:
        t2 = LOAD a
        t3 = t2 / t1
        STORE b, t3
        goto NEXT
ELSE:
        STORE b, 0
        goto NEXT
NEXT:
        t4 = LOAD g
        t5 = LOAD h
        t6 = t4 + t5
        STORE f,t6
END:
```

Figure 1: Intermediate code

The code shown in Figure 3 causes a problem if the branch of operation B *is taken* (that is, `(c == 0)`) and the else-clause is executed. With the current schedule, operation G (`t5 = LOAD h`) cannot be executed without performing the division of the then-clause.

The solution to these problems is part of the core of trace scheduling [8,9,10,6,22,19], but before we describe how trace scheduling deals with these problems, we introduce some terminology.

A trace is an ordered sequence of operations, and each operation in the trace has a unique position, called the *TracePosition*. *TracePosition*(O) is the position of operation O on the trace. Operations that are not part of the trace are called *off-trace*. Instructions in the schedule produced by the instruction scheduler are also ordered, and each instruction is associated with a unique schedule position. *FirstCycle*(O) is the position of the first cycle of operation O in the schedule. *LastCycle*(O) is the the position of the last cycle of operation O in the schedule. We refer to the instantiation of O on the schedule as *scheduled*(O). We refer to a compensation copy of O as *copied*(O), and frequently denote it as O' or O''.

2.1 Splits and joins

The basic block boundaries in the flow graph are determined by branches and branch targets. Since we have to fix up the program by inserting copies whenever an operation crosses a basic block boundary, each copy is associated with either a branch or a branch target.

2.1.1 Split operations

A *split operation*, or *split* for short, is an operation with more than one successor operation (for example, a conditional branch operation or an indirect branch). When selecting the trace, one of the successors is on the trace and is called the *on-trace* successor. The other successors are the targets that are not on the trace; they are called the off-trace successors.

When the instruction scheduler moves an operation below a split on the schedule, the trace scheduler must

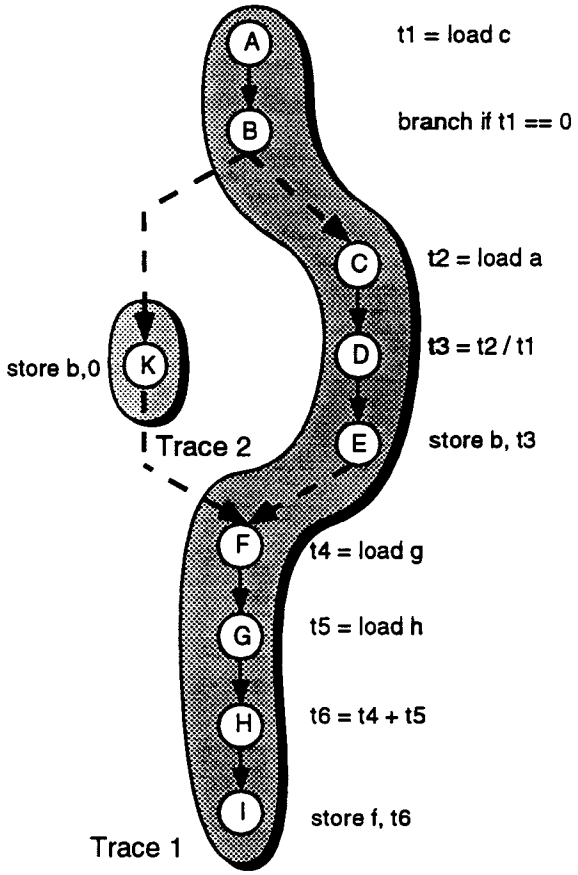

Figure 2: Flow graph for Figure 1

copy this operation onto the off-trace edge. With each split edge, we associate a pseudo-op SP which is the first successor on the off-trace edge of the split. All compensation copies are placed on the split edge in source order (i.e., the order in which the operations appeared on the trace) between the schedule and SP. For example, A is copied in Figure 4. In general, for each split S we determine a tuple of compensation copies $(O'_1, ..., O'_m)$ where *TracePosition*(O_i) < *TracePosition*(S) and *FirstCycle*(O_i) > *LastCycle*(S).

2.1.2 Joins

A *joined operation*, or *join* for short, is an operation on the trace that is the target of a branch operation. With each joining edge, we associate a pseudo-op RP that is the immediate off-trace predecessor. Whenever the trace scheduler moves an operation above a join, it must insert a copy of this operation on the off-trace joining edge, as shown in Figure 5. Otherwise, the moved operation will not be executed if the path from RP on the off-trace path is taken. The copies are placed on the joining edge in source order between RP and the schedule. We refer to RP as the rejoin point for join J.

Before join compensation code is determined, the off-trace branches to joined operations must be adjusted to reflect the scheduling decisions for the trace. The trace scheduler must find an appropriate instruction that can serve as the branch target; just choosing the instruction that contains the joined operation is not correct. For example, a joined operation may have moved relative to other operations, as illustrated by operation C in Figure 5. Therefore, the trace scheduler must find for each join J on the trace a *rejoin instruction* RI so that if a branch operation targeted the join J, the corresponding branch instruction targets instruction RI.

The rejoin instruction RI of a join J must satisfy the constraint that all operations O that appeared prior

0	A	t1 = load c
1	C	t2 = load a
2	F	t4 = load g
3	B	branch if t1 == 0
4	G	t5 = load h
5	D	t3 = t2 / t1
6	E	store b, t3
7	H	t6 = t4 + t5
8	I	store f, t6

Figure 3: Schedule for first trace in Figure 2

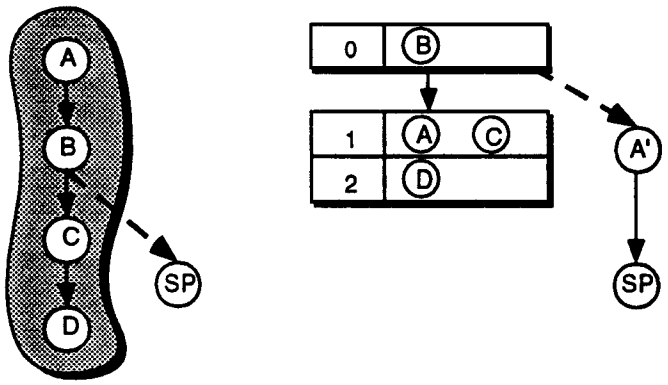

Figure 4: Split compensation code

to J in the trace (i.e., *TracePosition*(O) < *TracePosition*(J)) must be scheduled before instruction RI. For example, in Figure 5, 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. For each join J with rejoin instruction RI there is a tuple of compensation copies (O'_1, ..., O'_m) where *TracePosition*(O_i) $\geq$ *TracePosition*(J) and *FirstCycle*(O_i) < RI.[3]

2.1.3 Copied split operations

If a split is copied onto a joining edge, additional copies may be required. Consider the join to Operation B in Figure 6. The rejoin instruction is 4. The join compensation code is B'', D'', E''. If the program traverses the joining edge, and branches at D'', C is not executed. A copy of C is needed on the off-trace edge of D. In general, all operations that are between the join and the split on the trace and that are not before the rejoin instruction in the schedule must be copied on to the off-trace edge of the copied split. For each split SJ copied on an edge joining the trace at join J with rejoin instruction RI, there exists a tuple of compensation code (O'_1, ..., O'_m), where *TracePosition*(O_i) $\geq$ *TracePosition*(J), *TracePosition*(O_i) < *TracePosition*(SJ), and

[3]When compiling for a non-scoreboarded machine such as the Multiflow Trace, operations that are *bisected* by a join (i.e., *FirstCycle*(O) < RI and *LastCycle*(O) $\geq$ RI) must be treated specially. We do not discuss this here; see the discussion of *partial schedules* in [19] and [6].

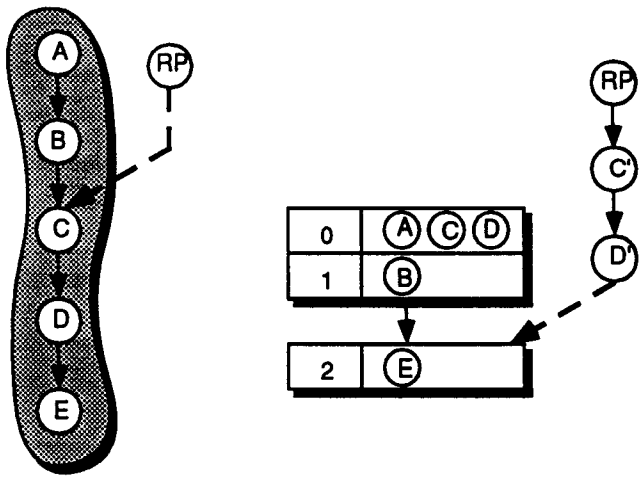

Figure 5: Join compensation code

$FirstCycle(O_i) \geq RI$. We call these operations $Rc - Cj$ copies.

2.1.4 Speculative code motion

Speculative code motion, i.e., moving an operation from below a split on the trace to a position above a split on the schedule, does not produce compensation code. Operation C of Figure 3 provides an example of this transformation. 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 performs such a move only if such a move 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 Multiflow 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 the 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 paths. For operations that write memory or transfer control, the complexity of un-doing them outweighs the potential benefits.

2.2 Instruction scheduling

The instruction scheduler schedules the trace to optimize the performance on the target hardware. In forming its schedule, it respects data-precedence constraints: write-read (true dependence), read-write (anti-dependence), and write-write (output dependence) dependences are respected for both variables and memory locations referenced by the operations on the trace.

An additional data precedence constraint is necessary for scheduling splits from the trace. In the example in Figure 7, we cannot move operation `a=b+c` above the split `if X` since `a` is live on the off-trace path. We associate with each split edge the variables that are live on the off-trace path; the trace scheduler updates this live information incrementally after each trace is scheduled. An operation that writes a variable is constrained by a preceding split for which this variable is *off-live*; this can be viewed as a form of anti-dependence.

The instruction scheduler also assigns register locations to all variables referenced on the trace; these locations are communicated to the remainder of the program by placing *value-location mappings (VLM)* at splits and joins [11,19]. The VLM at a split describes where the instruction scheduler has placed values live at the split; the VLM at a join describes where the instruction scheduler has placed value live at the join. Register allocation is performed incrementally as the compiler schedules each trace.

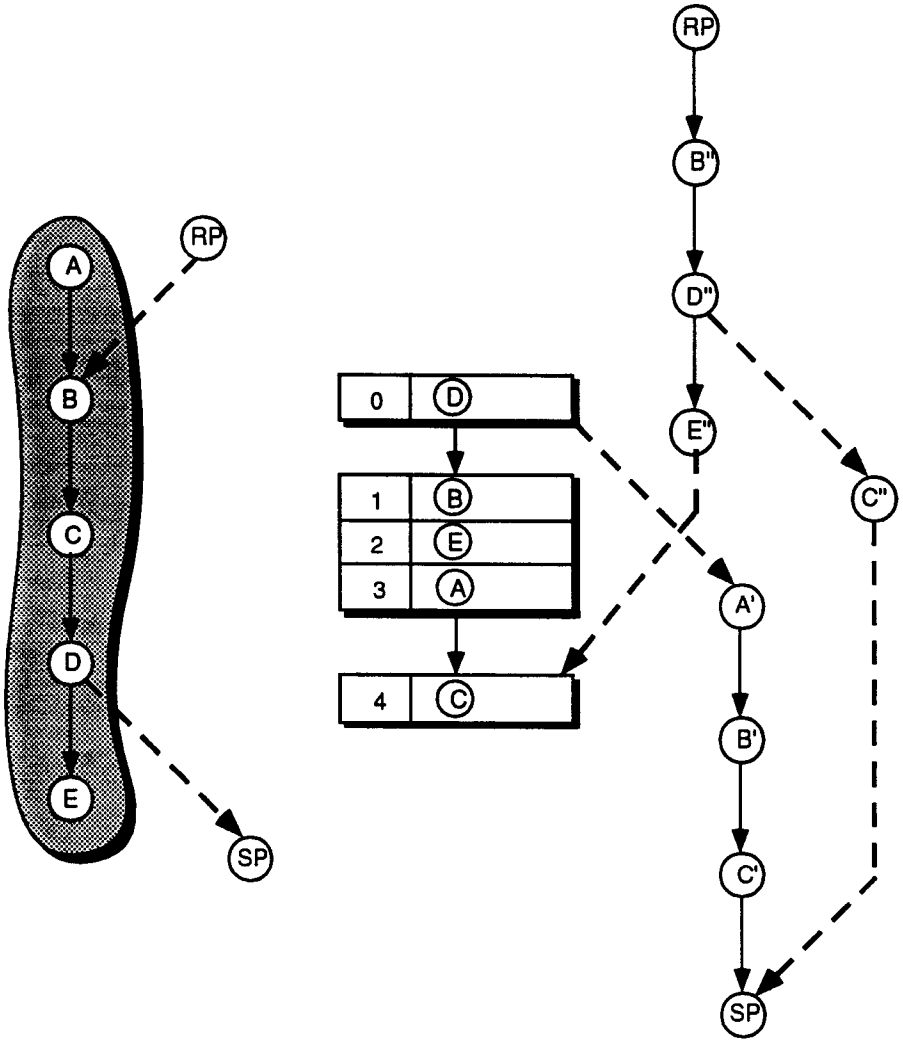

Figure 6: Split copied onto rejoin code

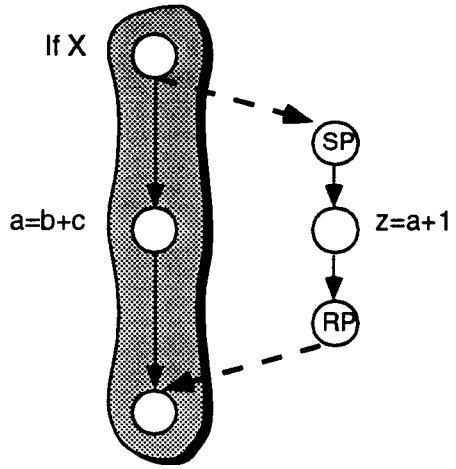

Figure 7: Off live example

3 Avoiding compensation code

The Multiflow compiler places a number of restrictions on trace scheduling to limit the amount of compensation code.

3.1 Avoiding split copies

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. This effectively avoids all split compensation.[4] Note that this restriction requires all splits to be scheduled in source order.

This restriction limits the parallelism available to the scheduler, but has a small effect on performance (see Section 5.2.1). An intuitive explanation is that executing a split early does not typically speed up the execution of the on-trace path; it only speeds up 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 can leave the trace as early as it would with a conventional basic block compiler, and the off-trace path would not be penalized by trace scheduling (though it may not be sped up).

3.2 Avoiding rejoin compensation

Sometimes a joined operation serves as the target of multiple branch operations. In this case a decision must be made if 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. See Figure 8 for an example. 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 on each edge to transfer control to instruction *RI*; no jumps to jumps are required. 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.

3.3 Loops

A trace does not cross a back edge 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 [8]. But it has a number of advantages: it allows the same scheduler to be used for loops and non-looping code and it simplifies the memory reference analysis. In addition, Nicolau relied on this restriction in his proof that trace scheduling terminates [22].

Parallelism between loop iterations is exposed by unrolling the loop and selecting the loop body as a trace. Unlike most compilers, the Multiflow compiler unrolls a loop by duplicating its body, including all exit tests; speculative code motion is relied on to overlap the unrolled bodies when scheduling. A loop could additionally be pre-conditioned or post-conditioned to eliminate some of the exit tests if this was profitable (and possible). This strategy permits *while loops* (i.e., loops with data-dependent exits tests) as well as *counted loops* to be unrolled.

Loops with internal branches can cause a large amount of join compensation code. Figure 9 depicts a loop with branches after unrolling, and Figure 10 shows the corresponding schedule. Notice in this loop that all of this join compensation code is redundant. This was the motivation for compensation copy suppression (see Section 4). Until the compensation copy optimization was implemented, the compiler would not unroll loops with internal branches, except for the highest levels of optimization or when explicitly indicated by the user (on the command line or with a directive).

[4] On machines which permit multiple branches per cycle, such as the Trace 14/300 and 28/300, we permit stores to move below splits to avoid a serialization in the schedule. See [19] for a discussion.

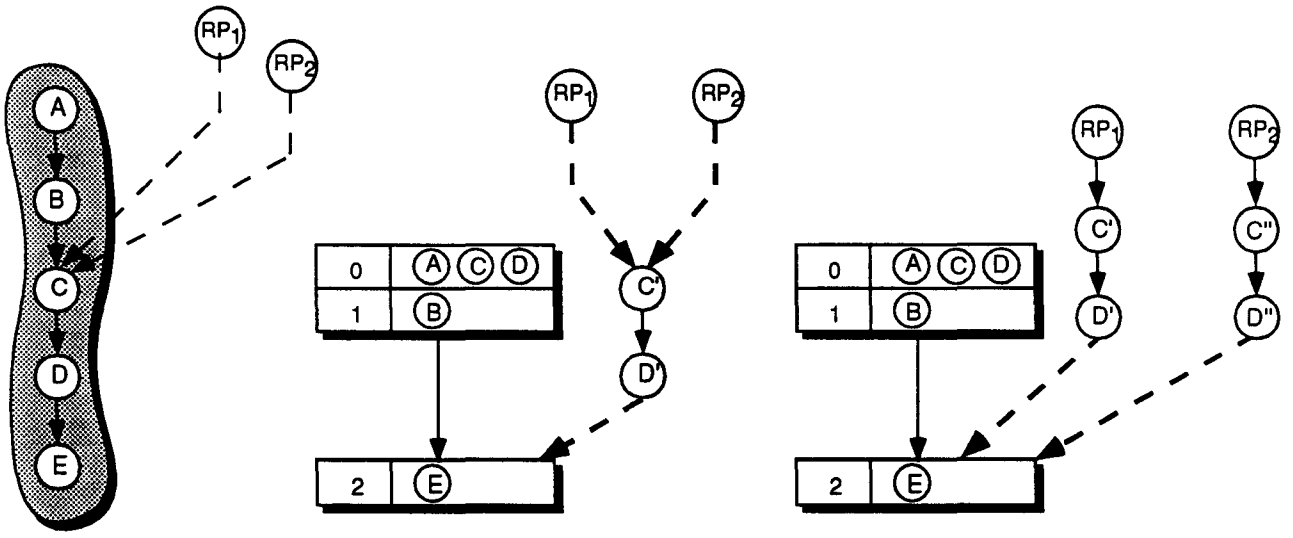

Figure 8: Possible strategies for multiple joins

3.4 Fail safe trace scheduling shutdown

When the number of copies in a program is twice the number of original operations, the trace scheduler no longer permits compensation code to be generated. This ensures that the program then finishes compiling rapidly. This is a fail-safe recovery from worst case copying (such as a heavily unrolled loop with internal branches) that is rarely activated in normal compilation in practice: for the SPEC89 benchmarks discussed in Section 5, fail-safe shutdown is only necessary when all our other copy avoidance techniques are disabled; even then, it is only triggered by six routines.

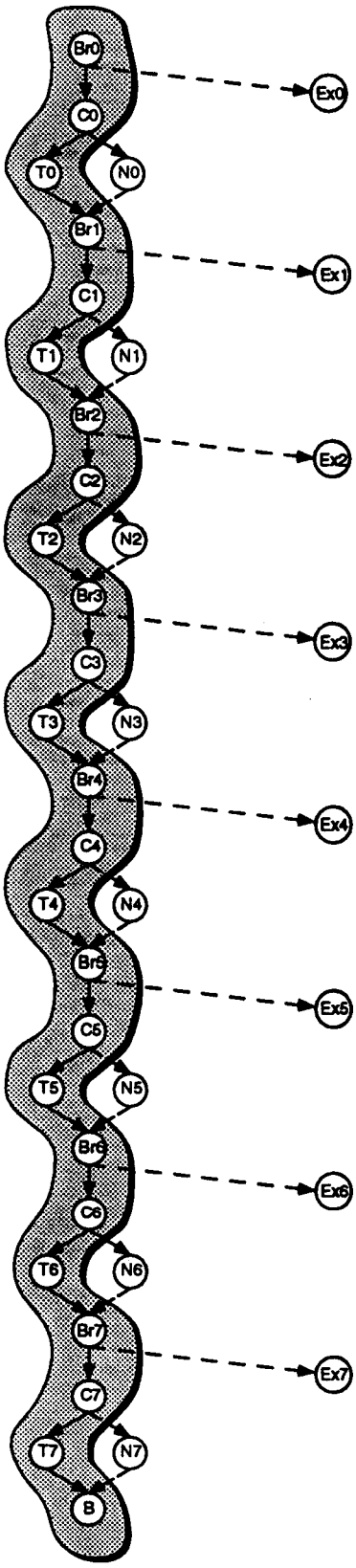

Figure 9: Trace of unrolled loop with branches

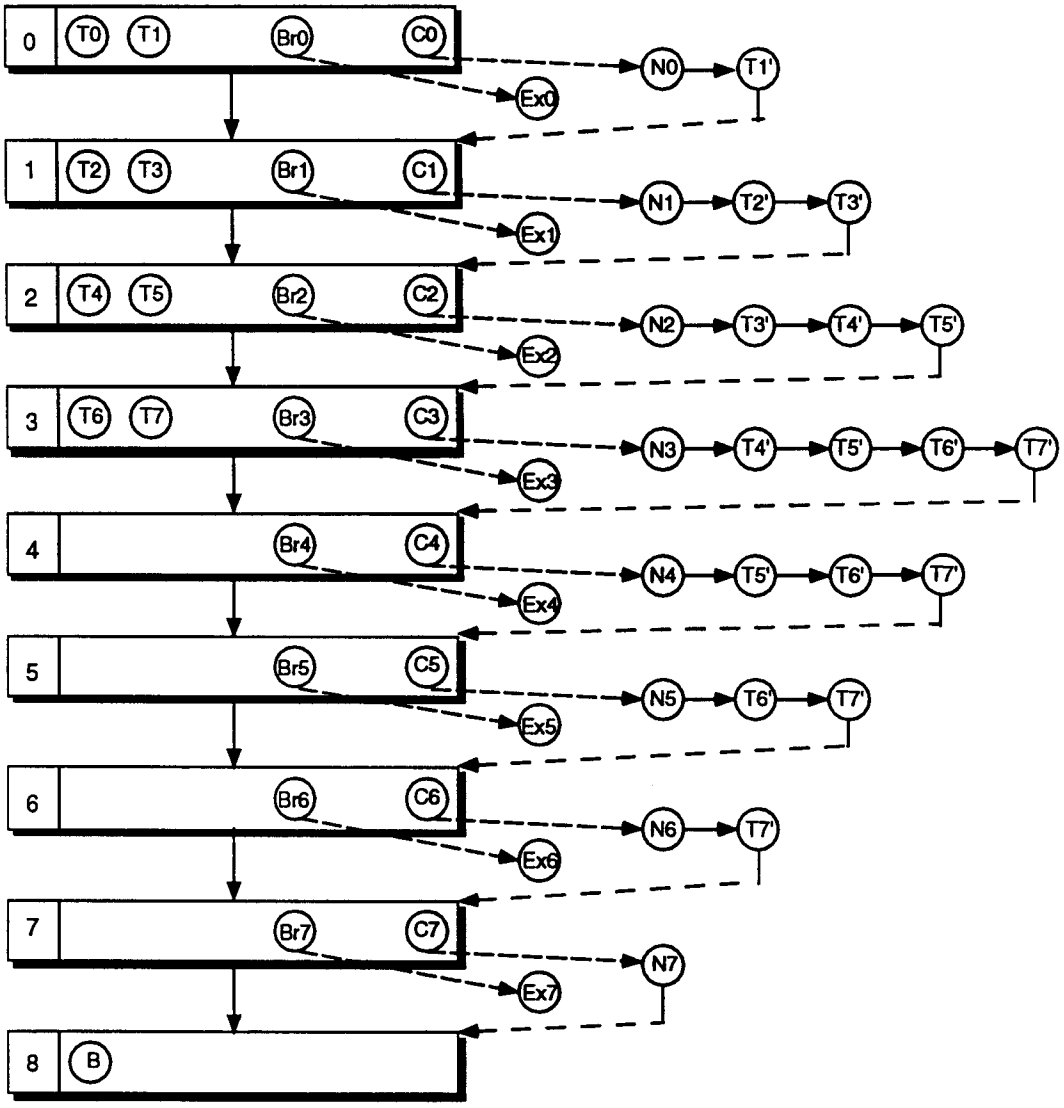

Figure 10: Schedule of trace of Figure 9

4 Suppressing compensation code

If we add the compensation copies required by the code schedule of Figure 3 and schedule the resulting trace, we obtain the following flow graph as shown in Figure 11.

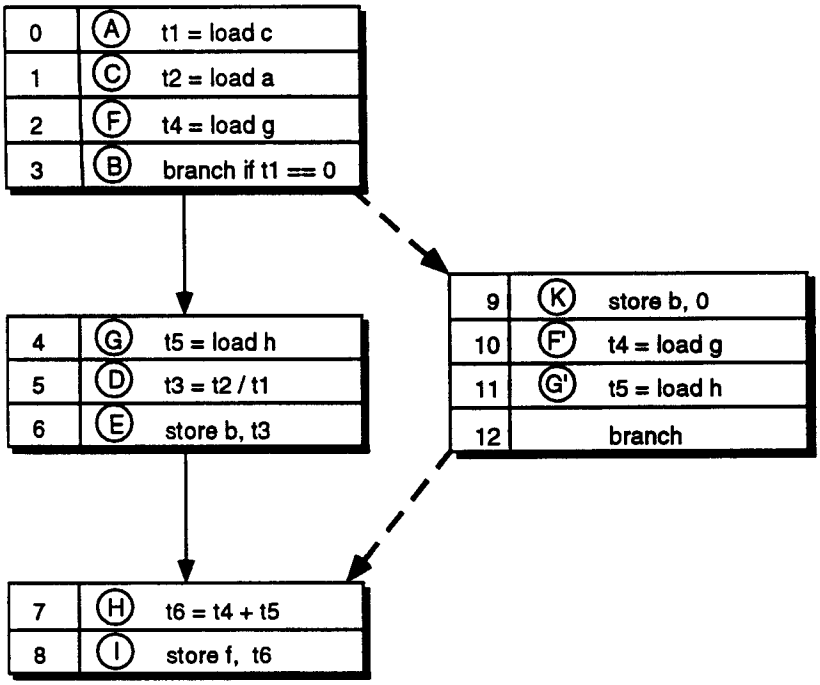

Figure 11: Flow graph after adding compensation code for Figure 2

Clearly, the introduction of compensation code is undesirable. Adding copies of moved operations to the program increases the program size, with its well-known negative impact on cache performance and compilation time. Furthermore, insertion of a compensation copy penalizes the off-trace code. Depending on the operations in the off-trace code, it may not be possible to schedule the compensation copy efficiently with the other off-trace operations.

Furthermore, not all compensation copies are necessary. For example, upon inspection of Figure 11, we notice that the copy of operation F is redundant. This operation has moved so early in the schedule (instruction 2) that it will be executed regardless of which path through the program graph is taken. If the off-trace path with operations K is selected, both operation F and F' are executed.

The effect of inserting compensation code is that the off-trace code contains extra operations. In the case above, these copied operations are executed *twice* whenever the path through the off-trace branch is taken: first, the moved operation is executed, and second, the copied operation is executed. This is a real problem if the off-trace code is executed frequently as well (e.g., in the case of a 60/40 branch that is taken 60% of the time, the off-trace code is nevertheless executed 40% of the time).

A copy O' of operation O is said to be *redundant* and can be suppressed if these two conditions are satisfied:

- *Scheduled*(O) (or another *copied*(O)) is executed on any path that includes the location where the copy O' is to be placed (domination).
- The dominating operation has the same effect as the copy O'.

We present the exact conditions to satisfy these requirements in the following sections. The *join compensation code optimization problem* can then be stated as follows:

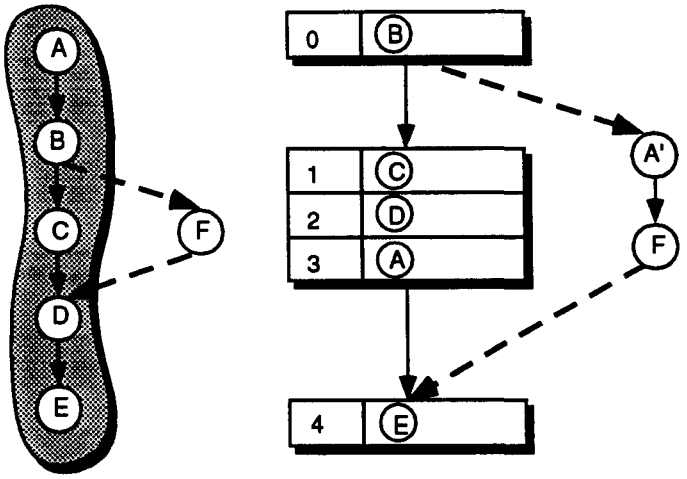

Figure 12: Trace with potential for split copy suppression

Join compensation code optimization problem:

Given a trace T, a join J, an operation O such that *TracePosition*(O) $\geq$ *TracePosition*(J), and a schedule $TS(T)$ for the trace T with rejoin instruction RI for J such that *FirstCycle*(O) $< RI$. Determine if the copy O' is redundant and remove the copy if this is the case.

There exists the dual *split compensation code optimization problem*, which can then be stated as follows:

Split compensation code optimization problem:

Given a trace T, a split operation S, an operation O such that *TracePosition*(O) $<$ *TracePosition*(S), and a schedule $TS(T)$ for the trace T such that *LastCycle*(S) $<$ *FirstCycle*(O). Determine if the copy O' is redundant and remove the copy if this is the case.

However, given the definition of the rejoin instruction presented in Section 2, there will never be any split compensation copies that can be suppressed. Consider operation A in the trace depicted on in Figure 12: if the scheduler moves A below the split (operation B) to instruction 3, then the rejoin instruction (for the arc from operation F) is instruction 4. One can imagine adjusting the rejoin to the schedule not to branch around operations that originally dominated the rejoin on the trace[5]; then redundant split compensation copies could be identified and removed. However, as explained in Section 3, split compensation copies are easily avoided by restricting code motion below splits, and we saw no practical need to consider the suppression of split compensation copies.

4.1 Ellis's algorithm

Ellis described the problem of redundant join compensation code in his thesis [6], and he proposed an algorithm for detecting redundant copies. This algorithm is as follows. Let RP be the pseudo-op for a rejoining edge J. Let O be an operation that follows J on the trace (*TracePosition*(O) $\geq$ *TracePosition*(J)). Let the rejoin instruction for this join be RI, and let *FirstCycle*(O) $< RI$. Then O will be copied onto this rejoin edge, as depicted in Figure 13. The rejoin copy O' is redundant if, after scheduling this trace

- *Scheduled*(O) dominates RP. That is, every path from program entry to RP first executes *scheduled*(O).
- *Scheduled*(O) copy-reaches RP. That is, on each path from *scheduled*(O) to RP, none of the inputs or output variables of O are assigned between the last execution of *scheduled*(O) and RP.

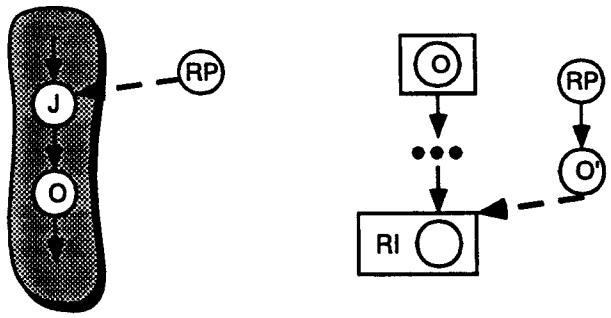

Figure 13: Join compensation copy

Ellis never implemented his algorithm in the Bulldog compiler. In implementing rejoin copy suppression in the Multiflow compiler, we extended Ellis's work in several directions.

- We discovered that the dominance requirement was too restrictive in the case that mattered most for performance: unrolled loops with internal branches. We changed the dominance requirement to be that either *scheduled*(O) or some other copy of O be executed on every path to RP. In other words, the set { *scheduled*(O), all *copied*(O) } dominates RP, i.e., every path from the entry of the flow graph to RP contains some element of this set.
- We developed an algorithm that detects copies after register allocation for the schedule. (Recall that in the Multiflow compiler, register allocation is done incrementally as each trace is scheduled.) Ellis's algorithm will detect most redundant copies only with a maximally-renamed program representation, where there is no unnecessary reuse of variable names. Register allocation introduces many anti-dependences and output-dependences between variables that are not in the program before allocation. These artificial dependences will prevent redundant copies from being detected by Ellis's algorithm.
- We developed an algorithm that works incrementally; it does not require an analysis of the full program between the scheduling of each trace.

4.2 Compiler framework

The Multiflow trace scheduling compiler [19] provided the framework for our implementation of compensation code optimization. The Multiflow compiler is a descendant of the Bulldog compiler developed at Yale [10,6]. Of course, the compiler was rewritten to deal with the complexities of full programming languages and a real machine, but the basic structure of the compiler is similar. In particular, the compiler modules that deal with the compensation code (the *bookkeeper*) closely follow the design described by Ellis in his thesis. The decision to perform this study of compensation code in the framework of this compiler imposed two constraints on our implementation:

- The overall structure of the compiler could not be changed. The instruction scheduler of this compiler schedules a complete trace, and then the bookkeeper inserts join and split compensation copies. The important point is that all operations of the trace are scheduled *before* the compensation copies are inserted. Therefore, it was important that the compensation code optimization be done incrementally after scheduling a trace.
- Including compensation code optimization should not increase the compilation time noticeably. This implies that compensation code optimization should try to reuse global information computed by earlier phases as much as possible.

[5]This will position the rejoin earlier on the schedule and may in fact reduce the number of rejoin copies.

4.3 Limitation of simple domination

A copy O' can be redundant even if it is not dominated by *scheduled*(O). In Figure 14, copy E' is necessary, but copy E'' is redundant if A and B do not affect the operands of E. This type of flow graph, with frequent splits and rejoins from the schedule, is created by unrolling a loop that contains an `if-then-else`. Our dominance requirement is that on every path from entry to RP, either *scheduled*(E) or a *copied*(E) be executed.

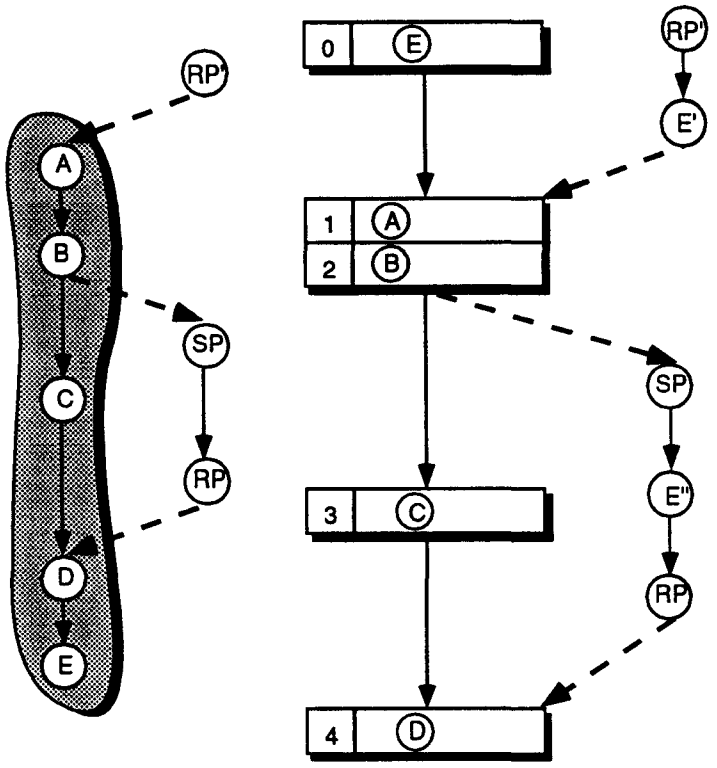

Figure 14: Problem with domination

4.4 Interaction with register allocation

Assigning registers to variables can make redundancy detection more difficult. In Figure 15, assume `t0` is assigned register `r0`. If the only use of `t0` is in F, then the register allocator may reassign `r0` to hold `t1`. The copies E' and F' are clearly redundant (assuming `a` and `b` are not changed by B), but a simple copy-reaching analysis will not detect this.

Our approach is to divide the problem between the trace scheduler (which drives the copy suppression) and the instruction scheduler (which allocates registers).

For a copy candidate O, the trace scheduler will check *(i)* that our dominance criterion is met and *(ii)* that the operands and results of O are not assigned on any off-trace path from the schedule to RP. (Note that if the dominance criterion is met, we know all reverse paths from RP hit the schedule.)

In the instruction scheduler, the on-schedule test for the suppression of the copy of O depends on the schedule position of the operations that read the variable(s) defined by O. We call these operations the readers of O. In Figure 16, O is a candidate for copy suppression, and R is its sole reader.

- If all of the readers of O have also moved to dominate RP, and their copies on this joining edge can be suppressed, then O can be suppressed; it is effectively dead. An example of this situation is shown in Figure 16 (b). (In this example, we assume the copy of R can be suppressed as well.)

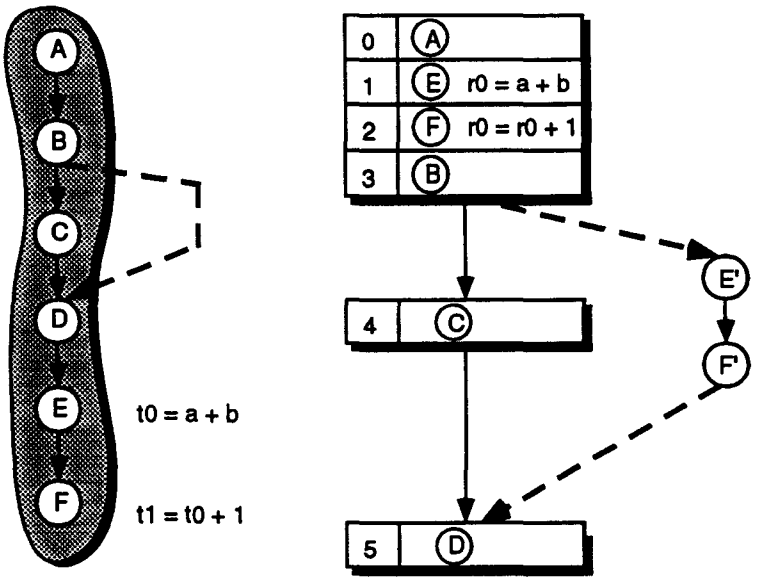

Figure 15: Interaction with register allocation

- If a reader of O is placed after the rejoin instruction, then the instruction scheduler keeps the variable defined by O live on the schedule until after the rejoin instruction. This means the live range for this variable starts at a point that dominates the rejoin and ends after the rejoin. In particular, the variable is live, and hence available in a register, at all of the splits which reach RP. The trace scheduler has tested that the off-trace path does not affect O. So the copy of O can be suppressed. An example of this situation is shown in Figure 16 (c).

- A reader of O is placed between the dominating position of O and the rejoin instruction. The instruction scheduler keeps the variable defined by O live until this reader. We need to extend the live range of this variable to all splits that reach RP; if we can extend the live range to the splits, then we can make sure that subsequent invocations of the trace scheduler keep the variable live until the rejoin point. To extend the live range, we must consider all paths from this O to RP, including the off-trace paths. If we can extend the live range, we can suppress the copy. An example of this situation is shown in Figure 16 (d); the copy of O cannot be suppressed because the live range of `t0` cannot be extended past cycle 3, since the register `r0` is reused to hold `t1`. If `r0` had not been reused, we could extend the live range and suppress the copy.

  Note that these readers will be copied onto the rejoin, for they are below the join on the trace and have not moved to a dominating position on the schedule.

4.5 Our algorithm

Our algorithm handles the two issues discussed in the previous two sections; it incorporates the refined definition of dominance and ensures that the information about live variables is maintained correctly. The algorithm processes each join to a trace. It first determines if an operation to be copied is a suppression candidate and then asks the instruction scheduler if the value defined by the operation is available. Note that we do not check for availability of a variable defined by a suppressed copy unless the variable is read by a subsequent operation

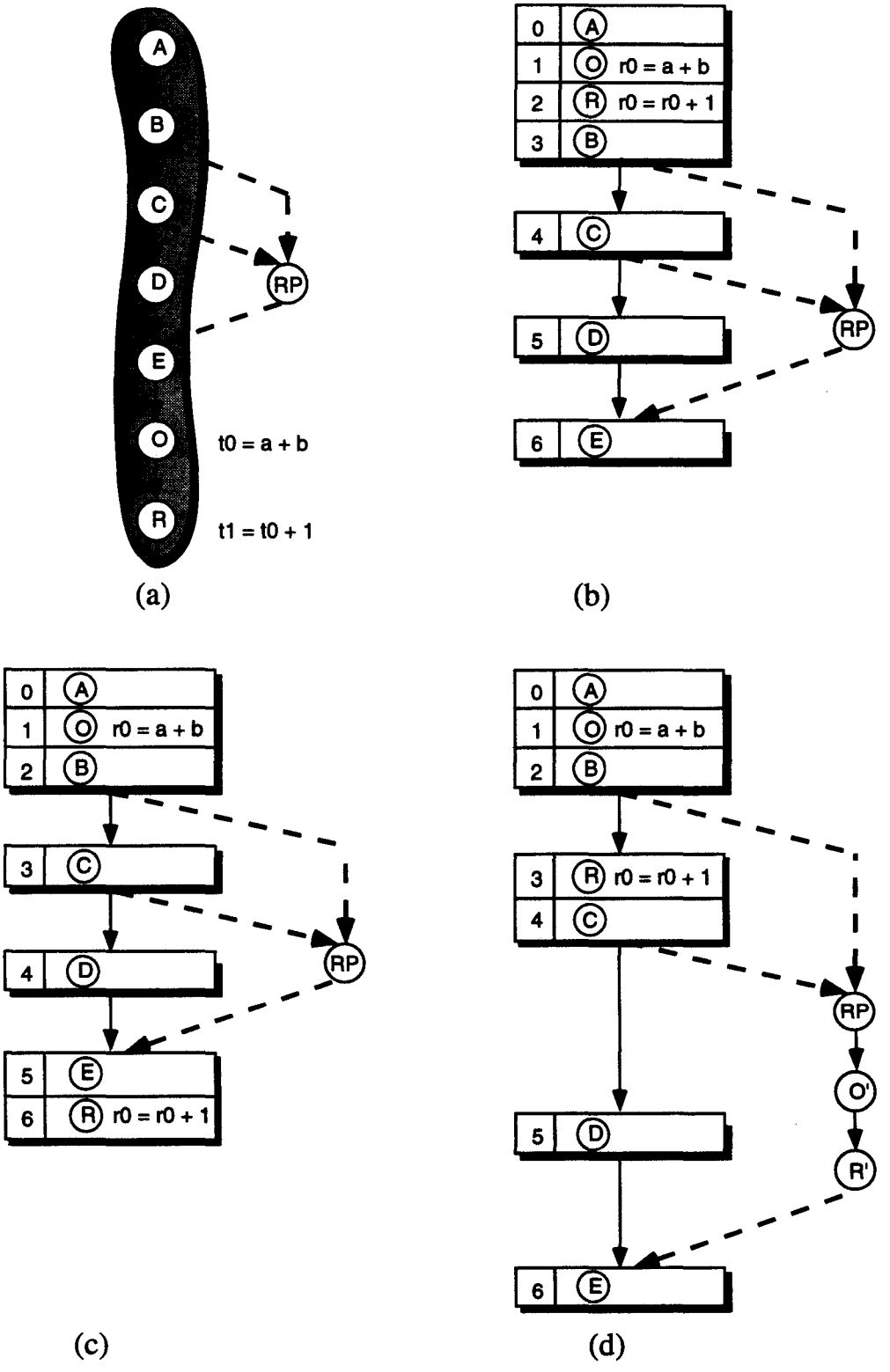

Figure 16: The effect of the placement of readers on copy suppression.

copied onto the joining edge.[6] In presenting the algorithm, we rely on two predicates (*suppression candidate* and *available for copy*); we define these predicates below.

Given a trace T and a schedule $TS(T)$ for this trace. For each join J with pseudo-op RP that joins to J, we consider the joins in trace order.

1. Determine the corresponding join instruction RI.
2. For each operation O in trace order, such that *TracePosition*(O) $\geq$ *TracePosition*(J) and *FirstCycle*(O) $< RI$
   - If O is a *suppression candidate*, mark O as suppression candidate.
   - Else
     - Mark O as copy candidate
     - For each operand V of O (including the off-live, if O is a split (see Section 2.2)), check that V is *available for copy*. If V is not available, mark the suppressed copy C that defines V as a copy candidate, and recursively process each operand NV of C. The recursion is bounded by the number of suppression candidates for this rejoin.
3. Copy the copy candidates onto the joining edge.

In the algorithm, when we determine that a variable is available, we update the value-location mappings for the relevant splits and joins, so that availability of the variable in its location is preserved through later invocations of the trace scheduler.

We now define the two predicates introduced earlier. An operation O is a *suppression candidate* for join J if it meets two tests:

- Dominance test: Either *scheduled*(O) or *copied*(O) is executed on every path from the entry to RP.
- Off-trace test: On every path from RP back to the trace, there is no conflict with the operands of O. A conflict is caused by:
  - an operation that writes any of the inputs or outputs of O,
  - a store conflicting with O (if O is a load),
  - a previously scheduled operation (because we do not want to spend the time to figure out if this code affects the inputs or outputs of O),
  - a call with side effects,
  - a loop (we choose not to analyze this rare case),
  - a back edge of a loop (in doing our availability analysis, we do not want to consider paths that cross a back edge).

The key idea of *available for copy* is to ensure that a variable holds the value computed by an operation C that we suppressed, and that this value is still available in some machine location (i.e., register or spill location). (If this is not the case, we have to insert a copy of C).

Available for copy: Consider a variable V that is read by a join copy O for join J. If V is not defined by a suppression candidate for this join, we always say V is available without further analysis, since our suppression algorithm does not effect V. If V is defined by a suppression candidate, we do not know all of the readers of

[6] We use the term availability of a variable defined by an operation as a shorthand for the availability of the value assigned to the variable by the operation.

V at this point in the compilation (without doing an expensive analysis), so we do not know how long V is kept alive on the schedule. We know V must be kept alive at least until O is scheduled, and therefore check that we can extend the live range of V to all splits that reach RP. A variable V is called *available for copy* if it meets these tests.

The algorithm that we implemented is more complex than described above. We allow values to be copied into multiple machine locations, and we allow values to be recomputed into additional machine locations rather than moved between machine locations (note that this includes spilling). We treat these multiple copies of the same value as one value. This complicates the correct extension of live ranges and requires that *available for copy* starts its analysis with C rather than with O. We also chose not to allow a variable to be made available from a spill location or from a *branch bank*; the branch bank holds the condition codes on the Trace machine[5]. When testing *available for copy*, we give the instruction scheduler the option of saying V is not available and forcing the copy. Similarly, for all variables live at a join, if the variable is defined by a copy suppressed on this join, we permit the instruction scheduler to force a copy.

4.6 Obtaining dominator information

Our copy suppression algorithm is invoked after scheduling each trace; we cannot afford to re-compute the dominator information for the program each time. Instead, we use the dominator information of the flow graph before the trace scheduling begins.

Theorem 1 *Let P be the flow graph before trace scheduling. Let TS be the first application of the trace scheduling algorithm (i.e., pick trace, compact, bookkeep). Let A and B be operations in the flow graph. Let T be the trace selected.*

If A dominates B in P, and A, B are not on trace T, then A dominates B in $TS(P)$ (the flow graph after TS replaced T in P with the scheduled code).

Proof: This follows from Nicolau's proof of the partial correctness of trace scheduling [22]. Nicolau shows that every path through the scheduled code and compensation code for a trace corresponds to a path through the trace before TS. Thus if A and B are not on the trace, the dominator relationship is preserved.□

Lemma 1 *Theorem 1 holds for successive invocations of TS, that is, for $TS^n(P)$. If A and B have not been picked for the first n traces, and A dominates B in P, then A dominates B in $TS^n(P)$.*

Theorem 2 *Let D be an operation picked for trace T. Let D dominate a rejoin point RP to the trace in the flow graph before trace scheduling. Assume D has not been copied (i.e., D has not been on a previous trace). Then the set of operations { scheduled(D), all copied(D) } dominates RP in the flow graph after scheduling trace T.*

Proof: This also follows from Nicolau's proof of the partial correctness of trace scheduling [22]. Nicolau shows that every path through the scheduled code and the compensation code for a trace contains all of the operations on an equivalent path before TS, though the operations may be duplicated and reordered. Since RP is off-trace, D is on-trace, and D dominates RP before TS, we know each path through the scheduled code and compensation code for the trace must contain at least one copy of D. □

Our strategy for obtaining domination information for a copy candidate O is as follows. Let B be the rejoined operation on the trace, and let RP be the rejoin point. We want to find an operation D on the trace such that:

- Before trace scheduling, D dominates B.
- Before processing this trace, D was not copied.

- Whenever *scheduled*(D) or any of *copied*(D) is executed, either *scheduled*(O) or some *copied*(O) is executed.

The first two conditions imply that D dominates RP before scheduling this trace, from Lemma 1. (We compute D dominates B, rather than D dominates RP, because this is easier for our implementation; the RP pseudo-ops are not actually represented in the flow graph.) The third condition implies the dominance information for D can be applied to O, and either O or a copy O' is executed on every path to RP. If the conditions are not met, then we have to insert a copy O'' onto the joining edge.

Theorem 3 *Let D and B be picked for trace T. Let D dominate B in the flow graph before trace scheduling. Assume D has not been copied (i.e., D has not been on a previous trace). Let RP rejoin the trace to B with rejoin instruction RI. Let O be on the trace with TracePosition(O) $\geq$ TracePosition(B). O is not a split.*

If scheduled(D) and scheduled(O) are in the same basic block, then scheduled(O) is executed whenever scheduled(D) is executed, and copied(O) is executed whenever copied(D) is executed if:

- *D is not copied on this trace;*
- *D is copied on this trace onto a split S, but TracePosition(S) > TracePosition(O);*
- *D is copied on this trace onto a rejoin J_i (with rejoin instruction RI_i), but there are no splits S such that TracePosition(S) $\geq$ TracePosition(D), TracePosition(S) < TracePosition(O), and FirstCycle(S) < RI_i.*

Proof: First note that *TracePosition*(D) < *TracePosition*(O), since D dominates B, and we do not pick traces across back edges.

There are three cases to consider:

- D is not copied on this trace. Clearly D and O will always be executed together.
- D is copied onto a split edge but the split follows O on the trace. We know *TracePosition*(D) < *TracePosition*(O). Since the split follows O on the trace (*TracePosition*(O) < *TracePosition*(S)), and D is copied onto the split edge, O must have been copied as well. Clearly, whenever D or D' is executed, O or O' is. Figure 17 depicts this situation (recall that we try to suppress O'').
- D is copied onto another joining edge J_i and there are no splits between D and O on the trace that are also copied. We know *TracePosition*(D) < *TracePosition*(O). Since D has been copied onto the joining edge J_i, *TracePosition*(J_i) $\leq$ *TracePosition*(D). Hence *TracePosition*(J_i) < *TracePosition*(O), and O will have been copied onto the joining edge for J_i. Clearly, whenever D or D' is executed, O or O' is. Figure 18 depicts this situation.

□

If the conditions of Theorem 3 hold, we have an operation O such that either O or a copy of O is executed with D or a copy of D. Theorem 2 guarantees that the members of { *scheduled*(O), all *copied*(O) } together dominate the rejoin point.

Theorem 3 required that D and O appear in the same basic block in the schedule so that we can use the dominance information about D for O. In practice, we do not need to require that D and O be in the same basic block on the schedule. If D and O are separated by a split on the schedule, the dominance information for D cannot be applied to O. However, if they are separated by a join, it usually can. There are two cases to consider:

- D precedes O on the schedule, and there is a join between D and O on the schedule.

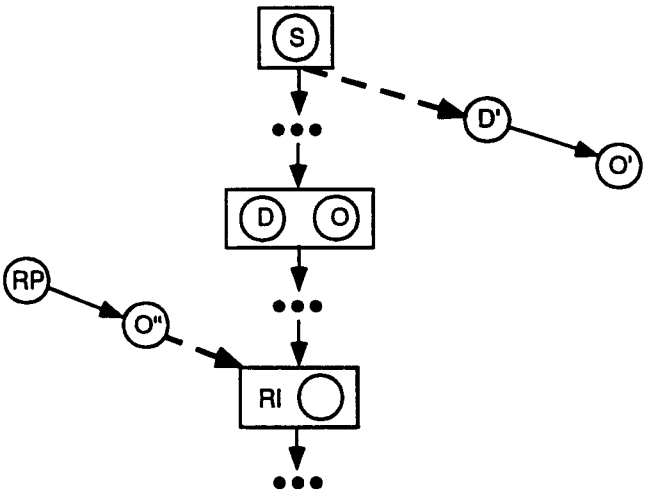

Figure 17: Example for case 2 of Theorem 3

- If D is not copied onto the joining edge, then the join was between D and O on the trace. Whenever *scheduled*(D) is executed, *scheduled*(O) will be.
- If D is copied onto the joining edge, then, as in Theorem 3, if no split between D and O on the trace is copied, we can treat D and O as if they are in the same basic block.

- O precedes D on the schedule, and there is a join between O and D on the schedule. O must be copied onto the joining edge, since O was below D (and the join) on the trace. If there is no split copied onto the joining edge after O, we can treat D and O as if they are in the same basic block. If a split is copied onto the joining edge after O, then D will be in the $Rc - Cj$ compensation for the split (see Section 2.1.3), so we can again treat O and D as if they are in the same basic block.

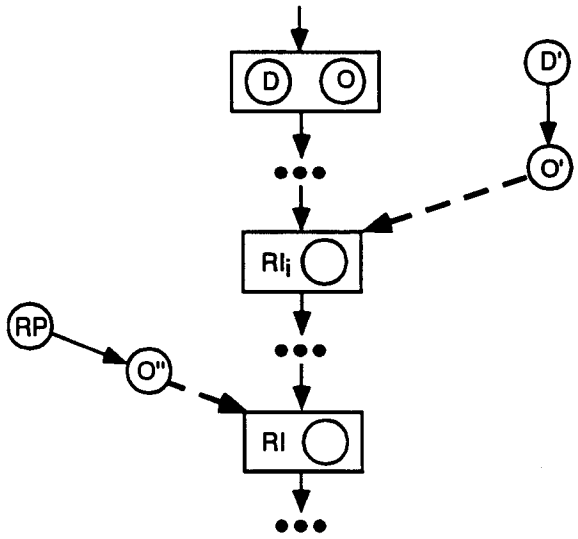

Figure 18: Example for case 3 of Theorem 3

5 Evaluation

To evaluate the effectiveness of trace scheduling and our compensation code optimizations, we ran a series of experiments, compiling and running the SPEC89 benchmarks on the Multiflow Trace 7/300. These experiments indicate that:

- Trace scheduling gives significant performance improvements over the compilation techniques used for most RISC processors. Much of this performance gain can be achieved without requiring any compensation code.
- The code growth due to trace scheduling and the loop unrolling used to support it is small. It is comparable to the code growth due to the loop unrolling and pre- or post-conditioning done by most RISC compilers.
- The amount of code growth due to compensation code is very small. Split compensation code can be avoided successfully with a minimal performance penalty. Avoiding join compensation code is more difficult, and our copy suppression algorithm is only partially successful.

We also measured the Livermore Fortran Kernels, heavily unrolled and tuned for the Multiflow Trace 7/300. These experiments show that trace scheduling with compensation code can deliver significant performance improvements. The code growth here is larger, but the component due to compensation code is smaller than the component due to loop unrolling.

5.1 Experimental Framework

5.1.1 Hardware

The Multiflow Trace 7/300 is a VLIW computer; it encodes up to 7 operations in a single long 256-bit 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. There is no data cache; the memory system is interleaved. The hardware provides support for suppressing or deferring the exceptions generated by speculative operations. There is also a 3-input, one output select operation (`a = b ? c : d`). This permits many short forward branches to be mapped into straight line code.

The Trace 7/300 is roughly comparable to a 3.5 issue superscalar machine. On the 7/300, 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.

The Trace 7/300 requires two classes of operations that are not frequently executed on a RISC machine. First, each functional unit has its own register bank, and moves between register banks are sometimes required. Second, the machine is not scoreboarded, and a *multi-beat nop* is sometimes required to delay the issue of a subsequent instruction; see [5] for a discussion.

5.1.2 Compiler

To model different compiler technology, we vary the loop unrolling heuristics and the code motions permitted by the instruction scheduler. In addition, we selectively apply the techniques for avoiding and suppressing compensation code discussed in Sections 3 and 4. All other optimizations and compiler options are not changed. In particular, we always use the full trace scheduling algorithm (pick trace, schedule, bookkeep), and we model basic block scheduling by constraining the code motions performed by the scheduler. This ensures that the

other benefits of our implementation of trace scheduling (e.g., priority-driven register allocation, adjacent code placement of frequently executed paths, etc.) are constant across all experiments.

We use two different trace picking heuristics in all of our experiments. The first uses the default heuristics in the Multiflow compiler (assume loop exits are not taken, that other conditional branches are most likely taken); the other uses feedback on branch probabilities from a previous run of the program on the same input data. We call the first strategy *static prediction* and the second *dynamic prediction.*

We model four loop unrolling heuristics.

- None. Loops are not unrolled at all.
- RISC-style. We model the unrolling used by most RISC compilers as follows.

  We attempt to unroll all inner loops that meet the following criteria:

  - The loop must be a counted loop.
  - The loop body must be a single basic block.
  - The loop body must contain no function call except for certain intrinsic functions (e.g., sin).

  If the loop body contains at most 16 machine-level operations, we unroll by 4; if it contains at most 32 machine-level operations, we unroll by 2; else we do not unroll the loop.

  When we unroll the loop by n, we post-condition it: we remove all but 1 loop exit, adjust the trip count increment to be n, and add $n-1$ copies of the loop body to execute the remaining *trip-count* **mod** n iterations. For example, when we unroll by 4, we make 7 copies of the loop body, 3 in the post-amble, and 4 in the body of the new loop, as depicted in Figure 19 (c). Note that the code would be smaller if we generated the post-loop code as a loop, but this code would have less parallelism.

  If the trip count is known at compile time, exactly *trip-count* **mod** n copies of the loop body are made on loop exit. For example, if we unroll by 4, and the trip count of the loop is known to be 81, we would add 1 copy of the loop body on loop exit.

  An alternative strategy is to pre-condition the loop, by executing *trip-count* **mod** n iterations before entering the unrolled loop body, as shown in Figure 19 (d). Post-conditioning and pre-conditioning have essentially the same effects on run-time performance and code size.[7]

- Multiflow-style. The Multiflow compiler does not pre- or post-condition loops (except for certain special situations). Rather it unrolls a loop by duplicating the loop body, leaving the loop exits in, as depicted in Figure 19 (b). This permits the compiler to unroll *while-loops* (i.e., loops with data dependent loop exits) and also avoids the overhead on loop exit introduced by post-conditioning. Speculative execution and scheduling across basic blocks are required to obtain parallelism across unrolled loop iterations if the loop exits are not removed.

  We attempt to unroll all inner loops that contain no function calls (except for certain intrinsic functions, e.g., sin) and that contain no internal branches. Note that we do not require the loop body be a single basic block; the loop body may have several exit branches. We attempt to unroll loops in C programs up to 4 times and loops in Fortran programs up to 8 times. For both languages, we will attempt to unroll the loop until the sum of the unrolled bodies consists of 64 operations. For example, if a Fortran loop body consists of 9 operations, it is unrolled 7 times.

[7] Our compiler implements post-conditioning for two reasons. First, post-conditioning does not change the alignment of data on entry to the main loop. This is important for single precision data on the Trace machines, which permit aligned pairs of single precision data to be loaded with one operation. Second, post-conditioning does not require a divide to determine the trip count of the main loop; this gives more flexibility in choosing the unroll amount. Note that this divide can be a shift if the unroll amount is a power of 2; pre-conditioned unrollings are typically constrained to be a power of 2.

- Multiflow-style with internal branches. This is Multiflow-style loop unrolling, extended to permit loops with internal branches to be unrolled. We will unroll a loop, subject to the limits described above, until the sum of the unrolled bodies contains 4 internal branches.

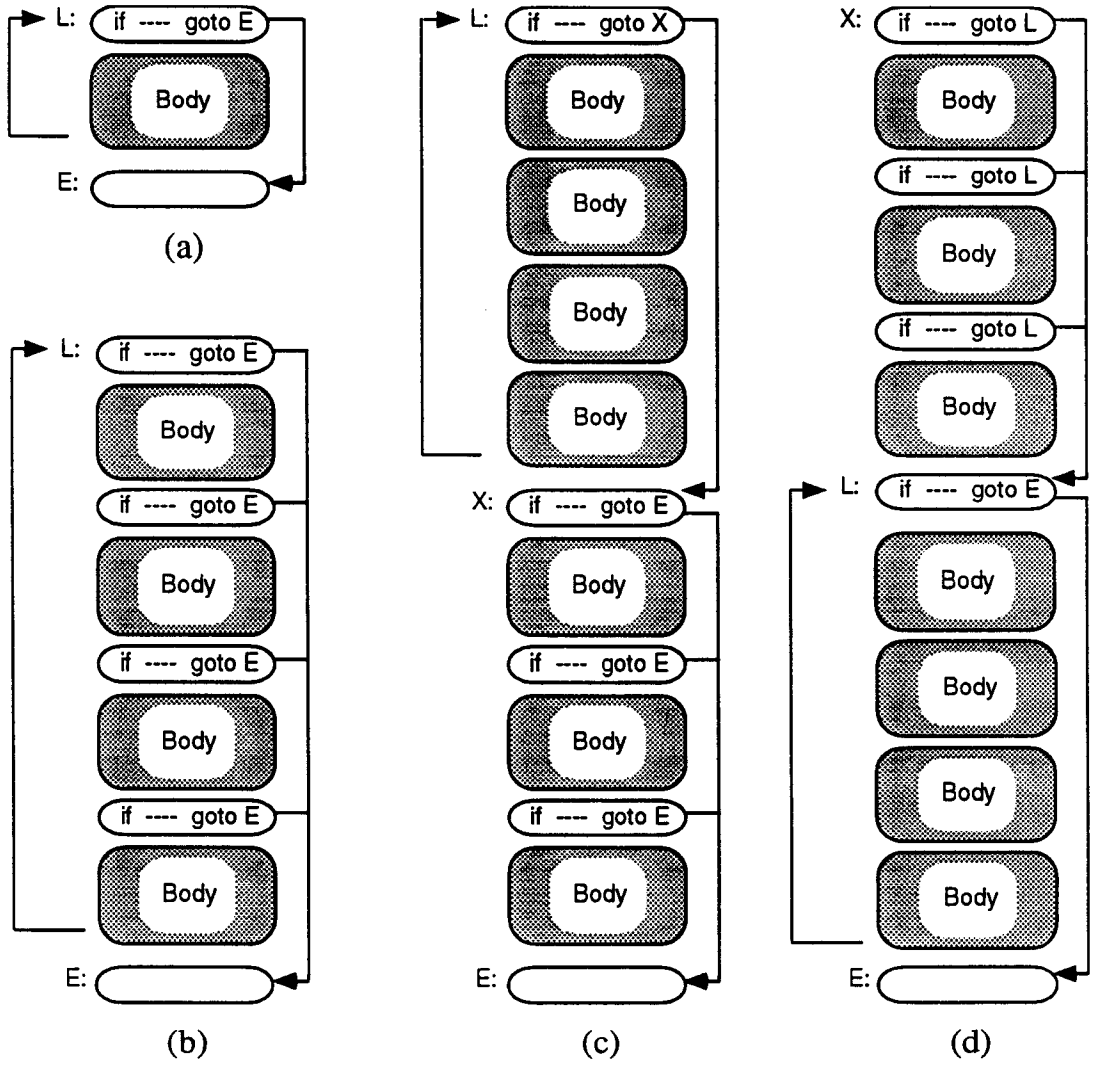

Figure 19: Styles of loop unrolling: (a) simple loop: (b) unrolled Multiflow-style, (c) unrolled RISC-style with post-conditioning, (d) unrolled RISC-style with pre-conditioning.

We model 5 variations on the code motions permitted by the instruction scheduler. In all cases, we pass the instruction scheduler full multi-basic-block traces and place restrictions on the code motions the instruction scheduler can perform.

- Basic block scheduling. We restrict the code motion in the instruction scheduler to basic blocks.
- Extended basic block scheduling. We restrict the code motion in the instruction schedule to extended basic blocks (a single-entry, multiple-exit sequence of basic blocks). In addition, we do not permit operations to move below a split. This prevents any split compensation code from being required; the restriction to extended basic blocks prevents any join compensation code from being required. Note that we do permit motion above splits (speculative execution), as described in Section 2.1.4.

- Trace scheduling with no avoidance techniques. We permit code motion across all basic blocks on the trace. The only restriction to the scheduler is that branches must remain in trace order.
- Trace scheduling with no join avoidance techniques. We permit code motion across all basic blocks on the trace. We restrict motion according to the split compensation code avoidance techniques we presented in Section 3 (no motion below splits).
- Trace scheduling with all avoidance techniques. We permit code motion across all basic blocks on the trace. We restrict motion according to the compensation code avoidance techniques we presented in Section 3 (no motion below splits, no motion above a join with 4 predecessors).

5.1.3 Models

The four loop unrolling heuristics and the five code motion techniques span a space of 20 different compilation strategies. We selected 7 compiler models that represent realistic choices for our domain.

- Model 1. Basic block scheduling; no loop unrolling. This models RISC compilers that do not unroll loops.
- Model 2. Basic block scheduling; RISC-style loop unrolling. This models most modern RISC compilers.
- Model 3. Extended basic block scheduling; Multiflow-style unrolling. This models trace scheduling without compensation code.
- Model 4. Trace scheduling with all avoidance techniques; Multiflow-style unrolling. This models the compiler techniques used in the production Multiflow compilers before the implementation of copy suppression.
- Model 5. Trace scheduling with no avoidance techniques; Multiflow-style unrolling with internal branches. This models unrestricted trace scheduling.
- Model 6. Trace scheduling with no avoidance techniques; Multiflow-style unrolling with internal branches; copy suppression. This models our copy suppression algorithm.
- Model 7. Trace scheduling with no join avoidance techniques; Multiflow-style unrolling with internal branches; copy suppression. This models our split compensation code avoidance and our copy suppression algorithm.

5.2 Experiments with the SPEC89 benchmarks

5.2.1 Performance

Tables 1 and 3 present the results of our 7 compiler models for the SPEC89 benchmarks. Table 1 presents the results for static trace prediction; Table 3 presents the results for dynamic trace prediction. Tables 2 and 4 show the same results as a percentage improvement over the base model (Model 1). For the combined SPECmark, the results with dynamic prediction are up to 10% faster than the results for static prediction; the variance is larger for individual benchmarks (from 10% slower to 30% faster).

From the combined SPECmark, which is the geometric mean of the individual benchmarks, we can see the following. For both static and dynamic prediction, Model 2 (basic block scheduling with RISC-style loop unrolling) outperforms Model 1 (basic block scheduling with no loop unrolling) by approximately 20%. Model 3 (trace scheduling without compensation code) provides an additional speedup over Model 2. The speedup is 22% for static trace prediction and 25% for dynamic trace prediction; dynamic information gives slightly

better performance. The models of trace scheduling with compensation code give modest improvements, if any, over Model 3. For dynamic prediction, the best improvement is for Model 6 (no avoidance techniques, unroll loops with internal branches, copy suppression); this gives a 4% improvement over the Model 3. For static prediction, Model 4 (trace scheduling with all avoidance techniques) gives equivalent performance to Model 3; Models 5-7 give small performance degradations (up to 3%).

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	SPECmark
Model 1	5.77	6.08	7.43	12.36	22.06	5.35	6.92	13.85	14.24	21.78	10.12
Model 2	5.91	6.25	7.04	12.80	36.62	5.36	6.90	37.05	14.13	29.93	12.16
Model 3	5.74	7.23	10.00	16.05	40.10	6.00	12.98	61.27	13.93	29.53	14.86
Model 4	5.77	7.47	10.22	15.97	39.17	6.19	12.96	61.10	12.96	29.82	14.85
Model 5	5.76	7.63	10.69	15.24	40.67	5.72	13.02	63.15	11.76	30.04	14.74
Model 6	5.88	7.66	10.47	14.30	40.78	4.85	13.19	61.07	11.76	30.00	14.38
Model 7	5.87	7.51	10.52	15.20	39.57	5.68	12.84	61.26	12.80	29.97	14.72

Table 1: Absolute performance of compiler models on SPEC89. Static trace prediction.

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	SPECmark
Model 1	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Model 2	102.42	102.73	94.82	103.51	166.01	100.16	99.67	267.61	99.21	137.39	120.10
Model 3	99.59	118.93	134.71	129.85	181.82	112.17	187.61	442.53	97.83	135.56	146.81
Model 4	100.10	122.90	137.67	129.18	177.57	115.77	187.30	441.28	91.03	136.91	146.75
Model 5	99.98	125.47	143.99	123.29	184.39	106.92	188.28	456.06	82.57	137.91	145.64
Model 6	101.92	126.02	140.96	115.66	184.90	90.78	190.60	441.04	82.57	137.72	142.11
Model 7	101.73	123.46	141.63	122.96	179.39	106.26	185.53	442.41	89.88	137.58	145.43

Table 2: Relative performance of compiler models on SPEC89 (base = Model 1). Static trace prediction.

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	SPECmark
Model 1	6.34	6.08	8.15	12.85	22.05	5.14	6.93	13.84	13.41	21.88	10.26
Model 2	6.34	6.25	7.23	12.94	36.64	5.47	6.91	37.65	12.68	29.41	12.18
Model 3	6.64	7.44	10.59	17.45	39.66	6.06	12.62	61.18	13.68	29.62	15.26
Model 4	7.40	7.85	10.48	15.66	39.17	6.66	13.14	60.80	14.06	29.82	15.56
Model 5	7.37	8.07	9.98	17.64	40.81	6.20	13.20	63.67	12.96	29.45	15.59
Model 6	7.30	8.14	11.28	17.81	40.81	6.43	12.97	63.62	12.69	30.07	15.83
Model 7	7.25	8.22	11.29	14.69	39.48	6.05	12.85	60.43	13.57	29.89	15.39

Table 3: Absolute performance of compiler models on SPEC89. Dynamic trace prediction.

To gain more insight into the performance data, we classify the SPEC89 benchmarks into four groups.

- Scalar integer (Sint): gcc, espresso, and li. These three are integer benchmarks that are not dominated (on the Trace 7/300) by vectorizable loops.
- Vector integer (Vint): eqntott. This is an integer benchmark, dominated by a loop which compares two vectors. The loop has a data dependent exit test; speculative execution is required to exploit parallelism between the iterations. There are no other dependences across iterations.

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	SPECmark
Model 1	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Model 2	99.99	102.73	88.79	100.70	166.17	106.27	99.70	272.08	94.55	134.41	121.01
Model 3	104.63	122.32	129.98	135.77	179.90	117.74	182.03	442.12	102.00	135.36	148.78
Model 4	116.71	128.97	128.57	121.89	177.68	129.46	189.57	439.33	104.84	136.29	151.71
Model 5	116.27	132.56	122.50	137.29	185.09	120.49	190.39	460.10	96.67	134.58	152.01
Model 6	115.14	133.79	138.47	138.56	185.12	125.03	187.09	459.71	94.62	137.45	154.30
Model 7	114.30	135.16	138.52	114.32	179.06	117.62	185.36	436.69	101.22	136.60	150.00

Table 4: Relative performance of compiler models on SPEC89 (base = Model 1). Dynamic trace prediction.

- Scalar floating point (Sfloat): spice2g6, doduc, and fpppp. These three are floating point benchmarks that are not dominated by vectorizable loops.
- Vector floating point (Vfloat): nasa7, matrix300, and tomcatv. These three are floating point benchmarks dominated by vectorizable loops.

Tables 5 and 6 present the results. For each group of benchmarks, we compute the geometric mean of the individual SPEC programs. Note that the performance of the vector integer and vector floating benchmarks is essentially identical for static and dynamic prediction. The effect of dynamic prediction is seen in the scalar benchmarks, where the branch direction is not as successfully predicted by the compiler's heuristics.

Figures 20 and 21 depict the incremental performance improvement of each compiler model (i.e., comparing Model 2 to Model 1, Model 3 to Model 2, and so on). From the data presented, we can make several observations.

- Model 2 (basic block compaction, RISC-style unrolling) provides a significant speedup over Model 1 only for the vector floating benchmarks; however, for these benchmarks, the speedup is large (82%).
- Model 3 (trace scheduling with no compensation code, Multiflow style unrolling) provides significant speedups over Model 2 for all of the benchmark categories; the speedups range from 8% to 88%.
- The speedup for trace scheduling with compensation code is only significant for the scalar integer benchmarks with dynamic prediction; for these benchmarks, Model 4 (trace scheduling with avoidance techniques) is 9% faster than Model 3, and Models 5-7 are roughly equivalent to Model 4.
- Comparing Models 5 and 6, we can see that our copy suppression algorithm has a small positive effect on performance for the scalar integer and the floating benchmarks with dynamic prediction, as well as on the vector integer benchmarks with static prediction, but it has a negative effect otherwise.[8]
- There is a small benefit to removing the split avoidance techniques for the vector floating benchmarks; Models 5 and 6 run faster than Models 4 and 7. This became a more noticeable phenomenon on the wider Multiflow machines. To avoid this problem while still avoiding most compensation code, we permit stores to be copied onto loop exits when compiling for these machines (see [19] for a discussion).

Since the numbers for static prediction do not differ substantially from the number for dynamic prediction, we concentrate in the rest of the paper on the data gathered for dynamic prediction.

[8]Keep in mind that the compilation of a program is influenced by a number of factors. The trace picker forms traces based on several parameters, and the traces formed for the compilation with static prediction can be slightly different from the traces for the compilation with dynamic prediction. Furthermore, suppressing compensation copies for one trace will influence the selection of subsequent traces. None of these effects is expected to be major but they can explain small variations between static and dynamic prediction.

	Sint	Vint	Sfloat	Vfloat
	SPECmark			
Model 1	5.72	6.92	10.94	18.81
Model 2	5.82	6.90	10.84	34.37
Model 3	6.29	12.98	13.08	41.71
Model 4	6.44	12.96	12.84	41.48
Model 5	6.31	13.02	12.42	42.57
Model 6	6.02	13.19	12.07	42.12
Model 7	6.30	12.84	12.70	41.72
	% Speedup			
Model 1	100.00	100.00	100.00	100.00
Model 2	101.76	99.67	99.12	182.76
Model 3	109.93	187.61	119.61	221.77
Model 4	112.51	187.30	117.42	220.55
Model 5	110.28	188.28	113.59	226.35
Model 6	105.25	190.60	110.42	223.94
Model 7	110.10	185.53	116.11	221.85

Table 5: Summary performance of compiler models on SPEC89. Static trace prediction.

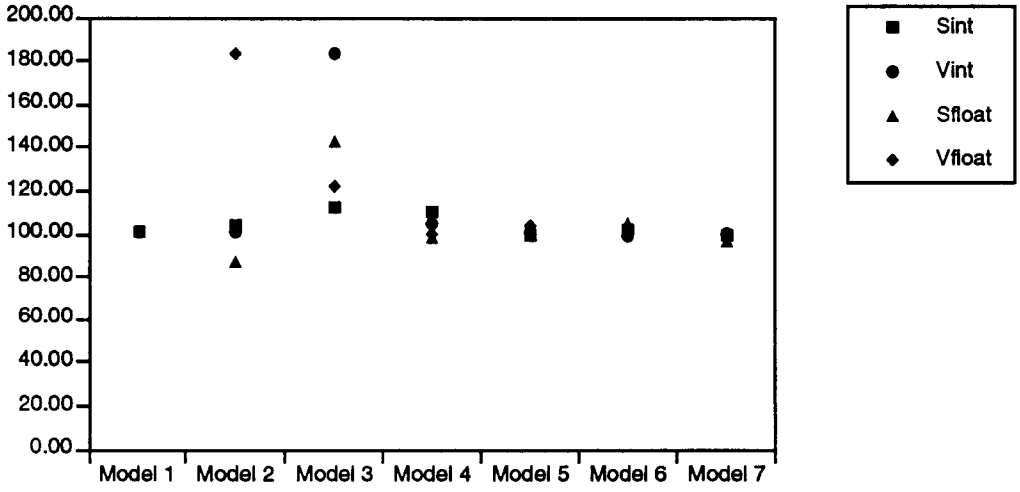

Figure 20: Incremental comparison of compiler models, static trace prediction.

	Sint	Vint	Sfloat	Vfloat
	SPECmark			
Model 1	5.83	6.93	11.20	18.83
Model 2	6.01	6.91	9.61	34.36
Model 3	6.69	12.62	13.62	41.58
Model 4	7.29	13.14	13.21	41.41
Model 5	7.17	13.20	13.17	42.45
Model 6	7.26	12.97	13.66	42.74
Model 7	7.12	12.85	13.11	41.47
	% Speedup			
Model 1	100.00	100.00	100.00	100.00
Model 2	102.97	99.70	85.79	182.47
Model 3	114.64	182.03	121.65	220.81
Model 4	124.90	189.57	118.00	219.94
Model 5	122.92	190.39	117.58	225.46
Model 6	124.42	187.09	121.99	227.00
Model 7	122.03	185.36	117.03	220.23

Table 6: Summary performance of compiler models on SPEC89. Dynamic trace prediction.

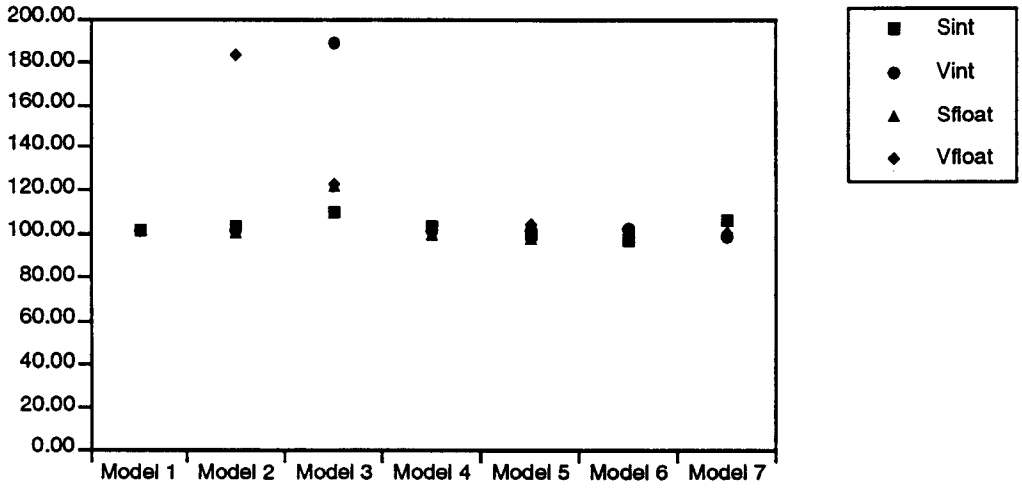

Figure 21: Incremental comparison of compiler models, dynamic trace prediction.

5.2.2 Code size

We present three measures of code size for compiling the SPEC89 benchmarks with our 7 compiler models. The first, in Table 7, is the number of VLIW instructions. This is an unsatisfactory measure, since a VLIW instruction may hold up to 7 operations, and the size of the instruction will vary with the number of operations it contains [5]. This measure understates the code growth, because the compiler is successful in packing the additional operations into proportionally fewer instructions. In Table 8, we present the number of machine operations. This measure is biased by operations unique to the Multiflow architecture (e.g., movement between various register banks, the *multi-beat nop*). And last, in Table 9, we present the number of internal compiler intermediate operations; these operations are machine-level, but they do not include the extra overhead operations necessary to map the program onto the machine. These overhead operations are not candidates for compensation copies. We include the data in Table 9 to give a machine independent view of the code growth.

The overhead operations we ignore in Table 9 fall into two classes. The first class consists of the familiar operations necessary for register allocation and procedure calls (e.g., spills and restores required for register allocation, the creation of constants, the mapping of arguments to the appropriate locations, setting up a frame, etc.). The second are the unique machine-specific operations mentioned above.

The data in all three tables is normalized for each benchmark to the size of Model 1, and a simple (not weighted) average is computed. The last column (Incr) shows the percentage change in the average from one compiler model to the next. We present the data for dynamic prediction only; the data for static prediction is nearly identical when averaged, though there are some variations on the individual benchmarks.

On average, by all three measures, trace scheduling with no compensation code (Model 3) produces code that is roughly the same as the RISC compilers with unrolling (Model 2). Note that this is true even though we are unrolling more aggressively for the Fortran programs in Model 3; we attempt to unroll by 4 in Model 2, and by 8 in Model 3. This is due to the code-size overhead of post-conditioning loops, required in the RISC model. Trace scheduling with compensation code, using all avoidance techniques (Model 4) is only 4-6% larger than trace scheduling with no compensation code. It is not significantly larger than RISC compilers with unrolling (Model 2). Removing the avoidance techniques (Model 5) causes a noticeable increase in code size (16-23%). Our copy suppression algorithm (Model 6) reduces this by 1.5-2.5%, and adding split avoidance (Model 7) reduces the size by another 5.1-7.5%.

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	Avg	Incr
Model 1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.00	100.00
Model 2	1.017	1.096	1.037	1.125	1.494	1.007	1.050	1.200	1.052	1.672	1.17	117.50
Model 3	1.019	1.091	1.068	1.094	1.287	1.031	1.075	1.114	1.053	1.391	1.12	95.52
Model 4	1.053	1.107	1.175	1.263	1.308	1.046	1.088	1.132	1.076	1.381	1.16	103.63
Model 5	1.130	1.181	1.624	1.715	1.374	1.070	1.177	1.122	1.177	1.468	1.30	112.13
Model 6	1.101	1.175	1.624	1.613	1.365	1.064	1.158	1.102	1.170	1.468	1.28	98.48
Model 7	1.096	1.161	1.230	1.317	1.310	1.060	1.141	1.094	1.078	1.381	1.19	92.42

Table 7: VLIW instructions, normalized (base = Model 1), dynamic prediction.

To gain more insight into the data, we group the benchmarks into four categories, as before. Table 10, Table 11, and Table 12 present the results. We normalized the data to Model 2 and computed a simple average for each group. We also present the incremental change for each compiler model. In summary, we find:

- For scalar and vector integer programs, the size increase due to trace scheduling is never larger than 32% relative to Model 2. A quarter or more of that increase is due to the more aggressive loop unrolling of Model 3; only three quarters of the increase is due to compensation code (roughly 25% increase relative to Model 2).

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	Avg	Incr
Model 1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.00	100.00
Model 2	1.025	1.134	1.045	1.158	1.572	1.009	1.071	1.226	1.062	2.051	1.24	123.51
Model 3	1.078	1.167	1.126	1.187	1.506	1.074	1.176	1.222	1.075	1.827	1.24	100.70
Model 4	1.146	1.202	1.274	1.450	1.564	1.100	1.204	1.273	1.129	1.833	1.32	105.93
Model 5	1.268	1.303	1.795	2.007	1.634	1.137	1.321	1.267	1.231	1.925	1.49	113.00
Model 6	1.227	1.293	1.803	1.854	1.618	1.130	1.301	1.242	1.219	1.925	1.46	98.15
Model 7	1.220	1.276	1.361	1.514	1.563	1.125	1.283	1.233	1.129	1.833	1.35	92.64

Table 8: Machine operations, normalized (base = Model 1), dynamic prediction.

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	Avg	Incr
Model 1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.00	100.00
Model 2	1.035	1.189	1.066	1.191	1.688	1.017	1.093	1.370	1.068	2.168	1.29	128.84
Model 3	1.105	1.251	1.184	1.244	1.583	1.120	1.233	1.410	1.084	1.896	1.31	101.76
Model 4	1.208	1.306	1.378	1.503	1.644	1.151	1.276	1.447	1.126	1.896	1.39	106.27
Model 5	1.373	1.444	2.054	2.008	1.797	1.204	1.442	1.483	1.261	2.000	1.61	115.30
Model 6	1.300	1.426	2.051	1.840	1.784	1.193	1.410	1.451	1.246	2.000	1.57	97.74
Model 7	1.291	1.402	1.457	1.561	1.738	1.188	1.380	1.438	1.518	1.914	1.49	94.81

Table 9: Intermediate operations, normalized (base = Model 1), dynamic prediction.

- For scalar floating point programs, there is a potential for significant code expansion (up to 75%) for trace scheduling with no split avoidance techniques (Model 5 and 6), but with our avoidance and suppression techniques, the operation growth is less than 30%.
- For vector floating point programs, all variants of trace scheduling produce code that is the same size or smaller than the post-conditioned unrolling of Model 2, even though we are attempting to unroll twice as much.

5.2.3 Compensation copies

Table 13 presents the number of compensation copies introduced by trace scheduling. We use dynamic prediction; the data for static prediction is similar. We present the percentage of compensation copies of the total intermediate operations for the compilation of the benchmark; we also average the data, weighting each benchmark equally.

On average, trace scheduling with all avoidance techniques (Model 4) introduces 6% additional operations as compensation code. Full unrestricted trace scheduling introduces 15%. Our copy suppression algorithm (Model 6) improves this ratio by 12%, and our split avoidance techniques combined with copy suppression reduces the proportion of copies by about half.

Table 14 presents the data for the benchmarks grouped into categories. To average the data, we normalize each benchmark by the number of intermediate operations in the Model 2 compilation. For scalar integer, vector integer and vector floating, the number of copies is relatively small for all models. Scalar floating has a larger percentage of copies for all models, and for Models 5 and 6, where no avoidance techniques are used, the number is significant (roughly 50%).

Figure 22 classifies the compensation copies according to type and makes the effects of our avoidance and suppression techniques clear. Model 5 presents unrestricted trace scheduling; all data in the table is presented

	Sint	Vint	Sfloat	Vfloat
	SPECmark			
Model 1	0.971	0.952	0.947	0.679
Model 2	1.000	1.000	1.000	1.000
Model 3	1.002	1.024	1.014	0.865
Model 4	1.031	1.036	1.117	0.877
Model 5	1.103	1.121	1.502	0.916
Model 6	1.080	1.103	1.482	0.910
Model 7	1.073	1.086	1.163	0.874
	Incremental % Change			
Model 1	100.000	100.000	100.000	100.000
Model 2	102.935	105.012	105.651	147.347
Model 3	100.236	102.397	101.398	86.527
Model 4	102.901	101.150	110.208	101.337
Model 5	106.937	108.242	134.433	104.469
Model 6	97.887	98.350	98.623	99.307
Model 7	99.389	98.532	78.481	96.123

Table 10: VLIW instructions, normalized (base = Model 2), dynamic prediction.

	Sint	Vint	Sfloat	Vfloat
	SPECmark			
Model 1	0.959	0.934	0.935	0.637
Model 2	1.000	1.000	1.000	1.000
Model 3	1.048	1.098	1.057	0.954
Model 4	1.106	1.125	1.205	0.987
Model 5	1.215	1.234	1.643	1.026
Model 6	1.183	1.215	1.618	1.016
Model 7	1.174	1.198	1.270	0.982
	Incremental % Change			
Model 1	100.000	100.000	100.000	100.000
Model 2	104.248	107.068	106.986	157.019
Model 3	104.802	109.809	105.707	95.392
Model 4	105.548	102.417	113.972	103.417
Model 5	109.804	109.695	136.406	103.961
Model 6	97.362	98.512	98.453	99.031
Model 7	99.298	98.590	78.500	96.723

Table 11: Machine operations, normalized (base = Model 2), dynamic prediction.

	Sint	Vint	Sfloat	Vfloat
	SPECmark			
Model 1	0.943	0.915	0.913	0.585
Model 2	1.000	1.000	1.000	1.000
Model 3	1.067	1.128	1.079	0.937
Model 4	1.152	1.167	1.248	0.967
Model 5	1.299	1.319	1.749	1.045
Model 6	1.242	1.290	1.710	1.037
Model 7	1.230	1.262	1.296	1.003
	Incremental % Change			
Model 1	100.000	100.000	100.000	100.000
Model 2	106.009	109.318	109.578	170.880
Model 3	106.686	112.828	107.901	93.652
Model 4	108.024	103.453	115.637	103.203
Model 5	112.672	112.982	140.213	108.085
Model 6	95.626	97.829	97.714	99.253
Model 7	99.095	97.848	75.826	96.770

Table 12: Intermediate operations, normalized (base = Model 2), dynamic prediction.

	gcc	espresso	spice	doduc	nasa	li	eqntott	matrix	fpppp	tomcatv	Avg
Model 4	8.48	4.19	13.21	16.55	3.79	2.73	3.34	2.50	3.85	0.00	5.86
Model 5	16.91	8.64	37.70	35.97	11.70	5.65	9.83	4.88	13.22	5.21	14.97
Model 6	12.26	7.49	37.85	30.24	11.07	4.80	7.82	2.77	12.19	5.21	13.17
Model 7	11.61	5.90	15.41	16.74	8.46	4.38	5.80	1.68	5.42	0.00	7.54

Table 13: Copies as percentage of intermediate operations, dynamic prediction

	Sint	Vint	Sfloat	Vfloat
Model 4	5.86	3.90	14.00	2.11
Model 5	13.20	12.96	49.64	7.51
Model 6	10.01	10.09	44.60	6.48
Model 7	8.85	7.32	16.11	3.48

Table 14: Copies of intermediate operations

as a percentage of the compensations copies from the Model 5 compilation of the benchmark. Split copies are 29% of all copies, and they can be completely eliminated with our split avoidance techniques. Rejoin copies are 70% of all copies. Model 4 includes our rejoin copy avoidance techniques. These reduce the rejoin copies by more than half, to 33%. Model 6 uses our copy suppression algorithm; it removes roughly 20% of the total rejoin copies. It is more effective when combined with the split avoidance techniques in Model 7; combined, these two techniques reduce the number of rejoin copies by 40%. $Rc - Cj$ copies are less than 2% of the total; they are eliminated by our split avoidance techniques.

Figure 23 provides the same information for the programs grouped as before. We see the scalar integer programs have almost no split compensation code, but overall, compensation code optimization leaves the largest percentage of intermediate operations for this group of programs. However, the impact on the number of VLIW instructions is much lower (see Table 10), and the execution time is improved as well (see Table 6).

5.2.4 Copy suppression

Table 15 presents the effectiveness of our copy suppression algorithm across the compilation of the all the benchmarks, using dynamic prediction. This table shows the aggregate count of the operations that are considered by step 2 of our compensation suppression algorithm introduced in Section 4.5. Operations listed as "copied" are inserted as rejoin copies (step 3). The algorithm succeeds in suppressing roughly 23% of the possible rejoin copies in Model 6 and in suppressing roughly 28% in Model 7. Note the beneficial effect of Model 7, which reduces compensation copies that cannot be suppressed.

	Operations			
	Copied	Suppressed	% Copied	% Suppressed
Model 6	43828	13008	77.1	22.9
Model 7	34185	13245	72.1	27.9

Table 15: Success of copy suppression

Table 16 presents data on why the algorithm failed. 90% of the failures are due to the placement of the code at a location that does not dominate the rejoin. Less than 5% is due to the variable not being available. The remaining failures are due to limitations in our implementation. We chose not to allow a variable to be made available from a spill location or from a *branch bank*; the branch bank holds the condition codes on the Trace machine. This data indicates that positioning of the code by the instruction scheduler is the major reason for the failure of the algorithm. The availability of the operands and the interactions with our register allocator are not a significant problem.

	No dominance	Value not available	Implementation restrictions	% No dominance	% Value not available	% Implementation restrictions
Model 6	39614	1622	2592	90.4	3.7	5.9
Model 7	29937	1593	2655	87.6	4.7	7.8

Table 16: Why copy suppression fails

5.3 Experiments with the Livermore Fortran Kernels

We performed a similar experiment with the Livermore Fortran Kernels. Individual kernels provide examples of significant performance improvements of trace scheduling with compensation code (more than 100%), and also an example of a very large amount of compensation code generated by unrestricted trace scheduling (almost a 2500% increase in operations).

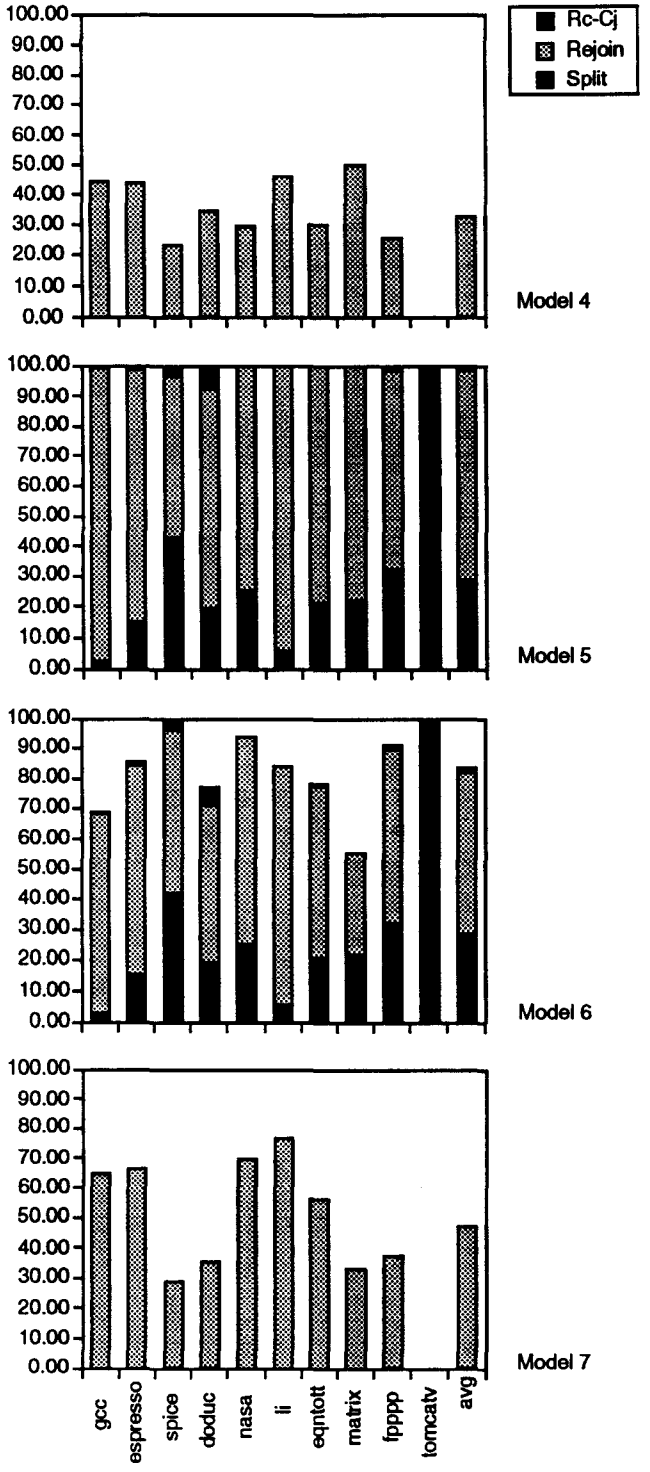

Figure 22: Classification of compensation code, all programs, relative to Model 5.

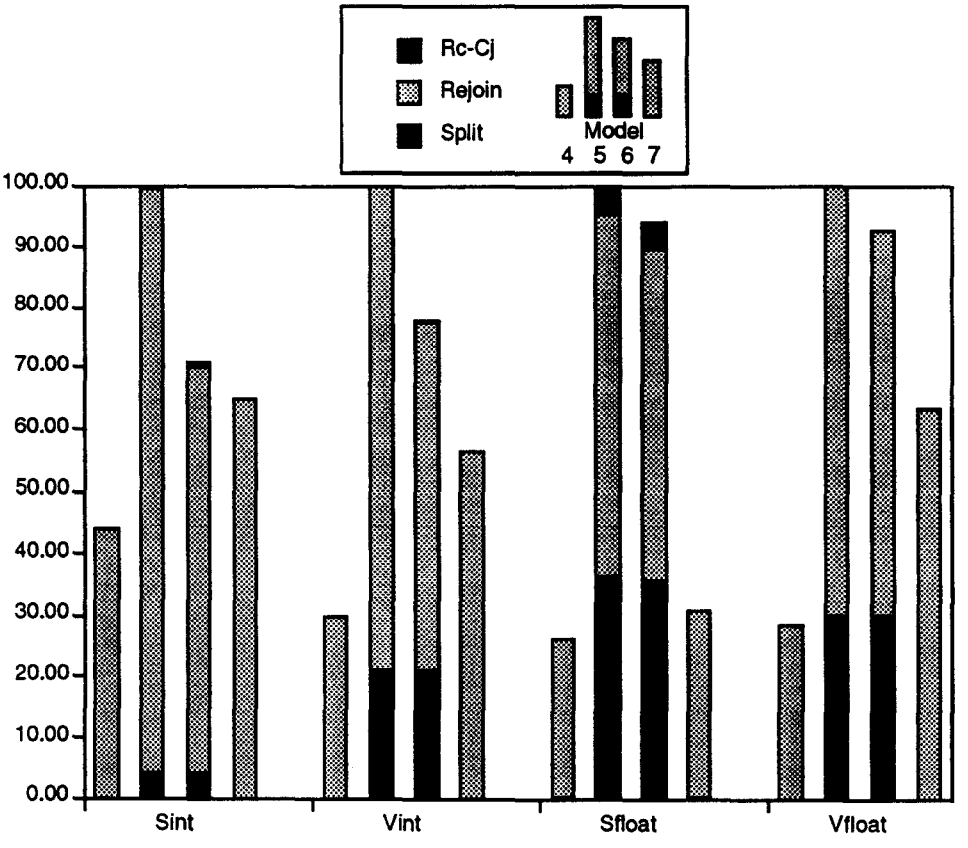

Figure 23: Classification of compensation code, summary, relative to Model 5.

There are 24 Livermore kernels; five have internal branches (kernels 15, 16, 17, 20, 24). Unrolled loops with internal branches exercise heavily the compensation code suppression and avoidance techniques we have described. We present detailed results for kernels 15, 16, 17, and 20, and for the 24 kernels as a whole. We do not study kernel 24 in detail since the Multiflow compiler translates the internal branch into a select.

In this experiment, when compiling with the trace scheduling models (Models 3-7), we use explicit unroll directives in the source to deliver peak machine performance. The loops are unrolled heavily, up to 32 times; the optimal unrolling was selected for each kernel. Models 3-7 unroll the kernels identically and vary only in their treatment of compensation code. For the RISC models (Models 1-2), we do not use the unrolling directives; we use the unrolling strategies described in Section 5.1.3.

In this experiment, we use *static prediction*, except that in the trace scheduling models (Models 3-7), we add explicit branch probability directives to the 5 kernels that have branches.

5.3.1 Performance

Table 17 presents the performance of the Livermore Fortran Kernels for our 7 compiler models run on the Multiflow Trace 7/300. The data is measured megaflops for the long vector case. We present the harmonic mean for all 24 kernels and individual results for 4 kernels. Figure 24 depicts the speedup of each compiler model relative to Model 1; Figure 25 shows the incremental contribution of each compiler model.

It is interesting to compare this data with the SPEC89 vector floating-point performance presented in Tables 5 and 6. In both cases, we see that Model 2 (RISC-style unrolling, basic block scheduling) provides large speedups over Model 1 (approximately 80%), and that Model 3 (Multiflow-style unrolling, trace scheduling with no compensation code) provides a substantial speedup over Model 2 (22-25% for SPEC89, 46% for the kernels).[9] However, for the SPEC89 vector floating-point benchmarks, trace scheduling with compensation code (Models 4-7) provided no substantial performance improvement. For the Livermore Kernels, trace scheduling with compensation code provides a significant improvement (9.5%). This is due to the 4 kernels with internal branches, whose performance is presented in detail.

Looking at the performance data for the 4 individual kernels, we can make the following observations.

- Model 2 (RISC-style unrolling, basic block scheduling) provides no speedup over Model 1. These loops are not unrolled, using the RISC-unrolling criteria.
- Model 3 (Multiflow-style unrolling, trace scheduling with no compensation code) provides a substantial performance improvement over Model 2 (17-41%).
- For three of the kernels, Model 4 (trace scheduling with compensation code) provides a signficant improvement over Model 3 (10-22%). For kernel 16, it causes a 7% drop in performance.
- For kernel 15, Model 5 (unrestricted trace scheduling) provides a very large performance improvement (79%). Note that most of this performance gain is lost in Model 7, when we reinstate our split avoidance techniques. This is an example where restricting code motion to avoid split compensation code has a large effect on performance. For kernel 16, Model 5 also gives a significant performance boost, which is preserved across Models 6 and 7; this is evidently due to the elimination of our join avoidance techniques. For the other kernels, Model 5 provides no noteworthy speedup.
- Comparing Models 5 and 6, we see that our copy suppression algorithm does not have a large effect on performance. It speeds up kernel 16 by 6%, speeds up kernel 15 by 2%, and leaves the other two unchanged.

[9]Note that for each benchmark we are using the averaging technique most commonly associated with it: geometric mean for SPEC89 and harmonic mean for the Livermore Fortran Kernels. Thus the percentage improvements for the averages for the two sets of benchmarks are not directly comparable, but the trends are. See [25] for a discussion.

- Comparing Models 4 and 7, we see that replacing join avoidance techniques with copy suppression does have a positive effect on performance. Kernel 16 shows a significant improvement (45%), and kernel 15 is also noticeably affected (15%). The only difference between these two models is that Model 4 uses join avoidance techniques described in Section 3.2 and Model 7 uses copy suppression; for the Livermore Kernels, we unroll all the trace scheduling models (models 3-7) identically.

	Kern15	Kern16	Kern17	Kern20	LFK
Model 1	2.3885	1.3699	2.5087	3.2981	1.5496
Model 2	2.3854	1.3682	2.5015	3.2986	2.8099
Model 3	2.9643	1.6009	3.5339	4.4183	4.1147
Model 4	3.2475	1.4932	4.3144	5.0726	4.2540
Model 5	5.8251	2.1289	4.3301	5.0899	4.4794
Model 6	5.9655	2.2485	4.3277	5.0972	4.5022
Model 7	3.7524	2.1713	4.3293	5.0705	4.4759

Table 17: Performance of compiler models on LFK kernels (megaflops).

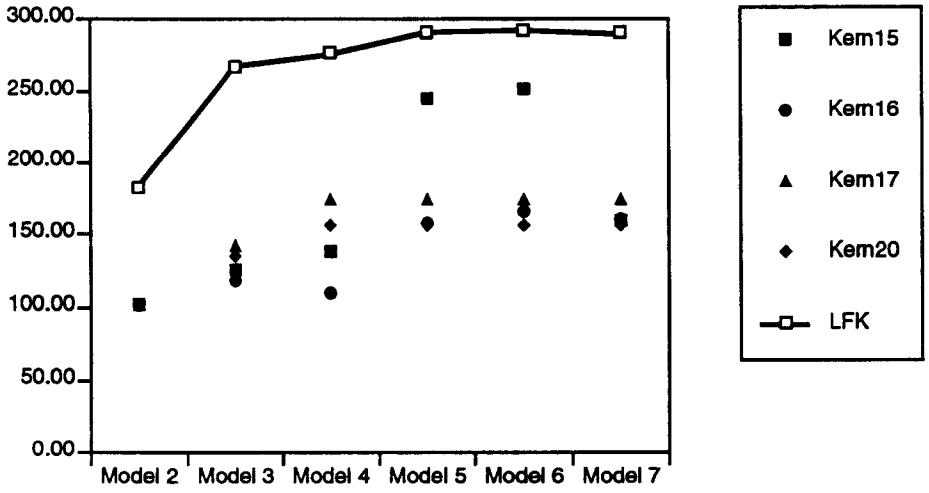

Figure 24: Percentage speedup for LFK kernels (relative to Model 1)

5.3.2 Code size

As described in Section 5.2.2, we present three measures of code size: VLIW instructions in Table 18, machine operations in Table 19, and compiler intermediate operations in Table 20. The data in each table is normalized to the size of Model 1. For each table, we present the size of all 24 kernels (without the driver or any of the timing code), the size of the 4 individual kernels we are examining in depth, and the average of the 4 kernels.

Looking at the data for all 24 kernels, we see for each measure, that the RISC unrolling done in Model 2 causes a large increase in size (65-75%) over Model 1, and that the heavy unrolling done in Model 3 (up to 32 times), causes a larger increase (360-512%) over Model 2. The size increase due to compensation code when using our avoidance techniques (Model 4) is relatively small (15-26% larger than Model 3); this is less than half the percentage increase due to RISC unrolling. Using our copy suppression algorithm as a substitution for join avoidance techniques (Model 7) gives roughly the same size. However, unrestricted trace scheduling, with or without copy suppression (Models 5 and 6), give very large code growth (about 300% larger than Model 4).

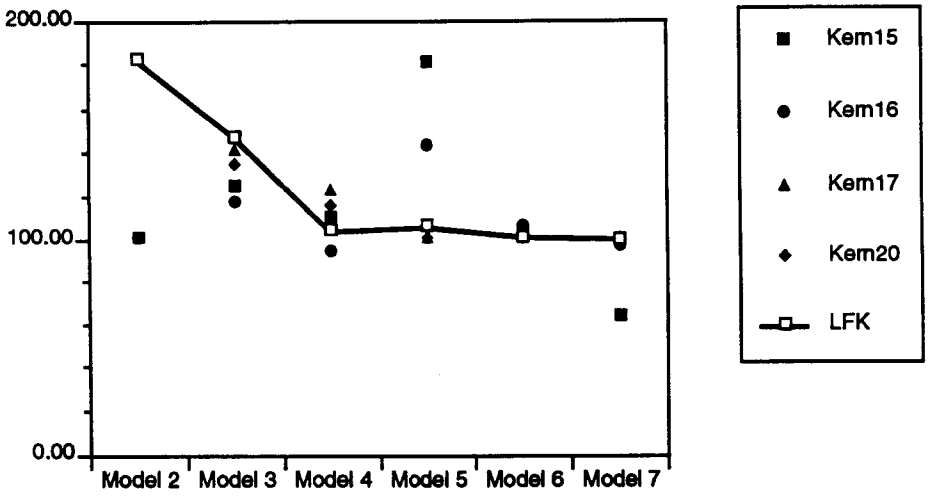

Figure 25: Incremental % speedup for LFK kernels.

Looking at the individual kernels, we see that the very large code growth for unrestricted trace scheduling is largely due to kernel 15. Unrestricted trace scheduling delivers large speedups (79% over Model 4), but the large number of code motions relative to splits and joins causes an explosive growth in the number of intermediate operations (2812% over Model 4). This growth is effectively limited by our avoidance and suppression techniques, at some cost in performance.

	Kern15	Kern16	Kern17	Kern20	Avg.	Incr (Avg)	LFK	Incr (LFK)
Model 1	1.00	1.00	1.00	1.00	1.00	100.00	1.00	100.00
Model 2	1.00	1.00	1.21	1.00	1.05	105.19	1.65	165.10
Model 3	2.86	5.52	3.13	5.08	4.15	394.41	5.94	359.96
Model 4	5.62	14.70	3.11	6.00	7.36	177.35	6.82	114.81
Model 5	143.96	12.33	3.68	6.00	41.49	563.93	20.33	297.93
Model 6	147.29	11.09	3.68	4.88	41.74	100.59	20.46	100.66
Model 7	4.90	11.67	3.21	4.94	6.18	14.81	6.97	34.05

Table 18: VLIW instructions, normalized to Model 1.

	Kern15	Kern16	Kern17	Kern20	Avg.	Incr (Avg)	LFK	Incr (LFK)
Model 1	1.00	1.00	1.00	1.00	1.00	100.00	1.00	100.00
Model 2	1.00	1.00	1.06	1.00	1.02	101.52	1.71	171.01
Model 3	2.69	6.10	2.76	4.80	4.09	402.62	8.33	487.37
Model 4	8.72	22.39	3.22	6.85	10.30	251.93	10.49	125.85
Model 5	223.27	20.54	3.70	7.03	63.63	617.99	30.87	294.31
Model 6	233.23	16.25	3.70	5.80	64.74	101.74	31.32	101.47
Model 7	7.69	17.46	3.26	5.74	8.54	13.19	10.74	34.30

Table 19: Machine operations, normalized to Model 1.

	Kern15	Kern16	Kern17	Kern20	Avg.	Incr (Avg)	LFK	Incr (LFK)
Model 1	1.00	1.00	1.00	1.00	1.00	100.00	1.00	100.00
Model 2	1.00	1.00	1.17	1.00	1.04	104.17	1.73	172.91
Model 3	2.51	6.19	3.05	4.77	4.13	396.36	8.86	512.18
Model 4	8.39	19.01	3.44	7.44	9.57	231.76	10.94	123.53
Model 5	244.33	19.95	3.97	7.56	68.95	720.59	30.87	282.17
Model 6	244.72	15.55	3.97	5.43	67.42	97.77	30.46	98.66
Model 7	7.24	17.15	3.55	5.43	8.34	12.37	11.33	37.21

Table 20: Intermediate operations, normalized to Model 1.

5.3.3 Compensation copies

Table 21 presents the percent of compensation copies introduced by trace scheduling. We present the percentage of compensation copies of the total intermediate operations for the compilation of the benchmark. We present the results for all 24 kernels, and for the four kernels we are examining in detail. For the 4 kernels, we also average the data, weighting each benchmark equally.

For the 24 kernels, trace scheduling with all avoidance techniques (Model 4) introduces 16% additional operations as compensation code. Trace scheduling with copy suppression and split avoidance (Model 7) introduces only slightly more (19%). However, unrestricted trace scheduling introduces 70% additional operations (Model 5), and copy suppression with no avoidance techniques (Model 6) has a very small effect in reducing them (0.5%).

Most of the copies with unrestricted trace scheduling come from kernels 15 and 16. In kernel 15, over 98% of the operations are copies when no avoidance or suppression techniques are used, and copy suppression by itself does not reduce the number of copies. However, copy suppression and split avoidance reduce the number of copies to 65%, which is better than what is achieved with all of our avoidance techniques.

	Kern15	Kern16	Kern17	Kern20	Avg	LFK
Model 4	69.58	66.31	11.45	35.02	45.59	16.35
Model 5	98.35	68.05	23.28	36.01	56.42	69.94
Model 6	98.38	59.34	23.28	10.88	47.97	69.56
Model 7	64.64	63.72	14.10	10.88	38.33	19.37

Table 21: Copies as a percentage of intermediate operations

Figure 26 depicts a classification of the compensation copies according to type and makes the effects of our avoidance and suppression techniques clear. Model 5 presents unrestricted trace scheduling; all data in the table is presented as a percentage of the compensations copies from the Model 5 compilation of the benchmark. Looking at the data for all 24 kernels, we see that split copies are 19% of all copies, and they can be completely eliminated with our split avoidance techniques. Rejoin copies are 73% of all copies. Roughly 85% of them can be removed by either our avoidance techniques (Model 4), or copy suppression together with our split avoidance techniques (Model 7). Note that our copy suppression algorithm by itself is not very successful in removing copies (Model 6). 8% of the copies are $Rc - Cj$ copies; they are eliminated with our split avoidance techniques.

5.3.4 Copy suppression

Figure 27 presents another view of the effectiveness of our copy suppression algorithm. For each program, this figure shows what percentage of the operations considered by step 2 of our algorithm are actually suppressed

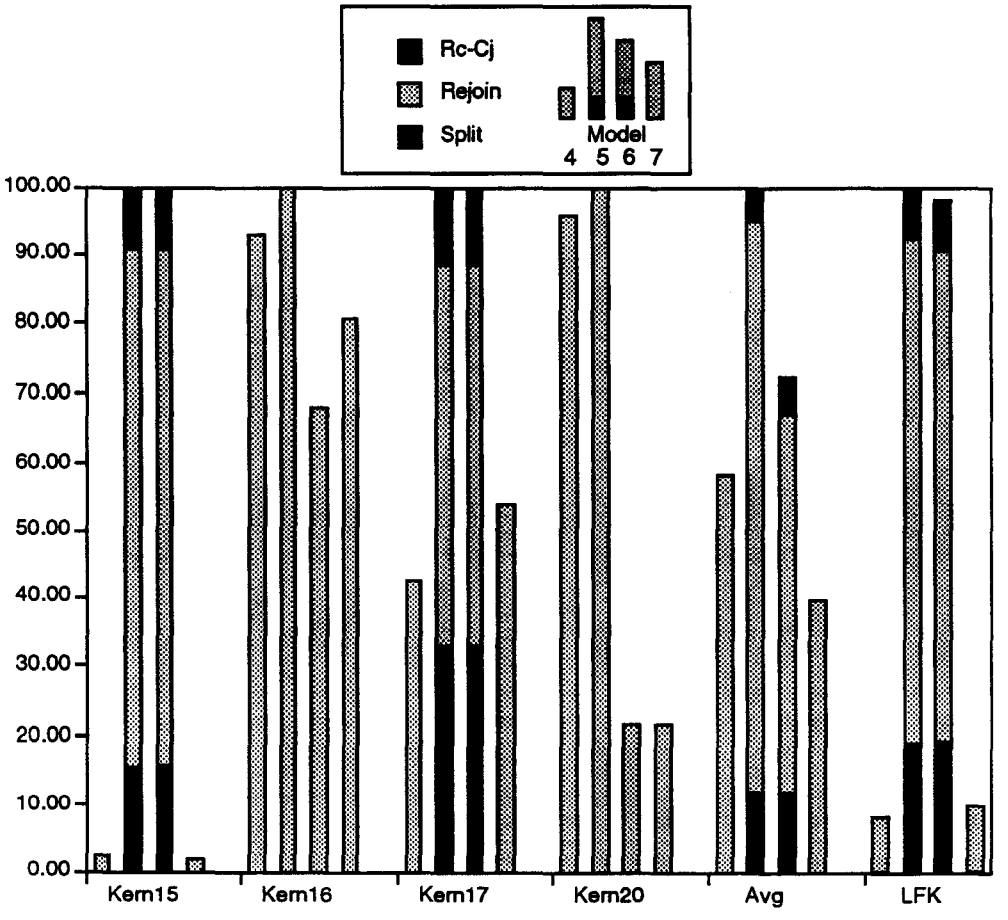

Figure 26: Classification of compensation code (base = Model 5)

(not copied in step 3). For Model 6, our algorithm succeeds in suppressing 5% of the copies in all 24 kernels, and for Model 7, 19% are suppressed. Note that there is considerable variation from kernel to kernel: in kernel 17, no copies are suppressed; in kernel 20, 79% are.

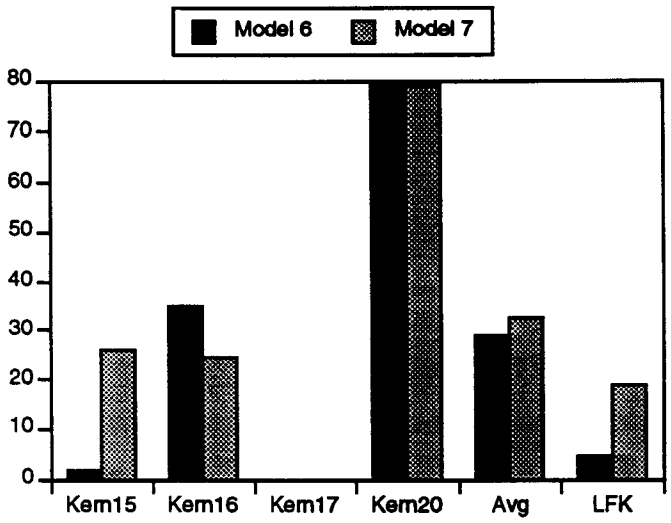

Figure 27: Success of copy suppression

Figure 28 presents data on why the algorithm failed. As for the SPEC89 benchmarks, almost all of the failures are due to the placement of code at a location that does not dominate the rejoin. In kernel 20, our implementation restrictions prevent us from removing a significant percentage of the remaining copies; however, for this kernel, our algorithm is successful in removing almost 80% of the copies (see Figure 27).

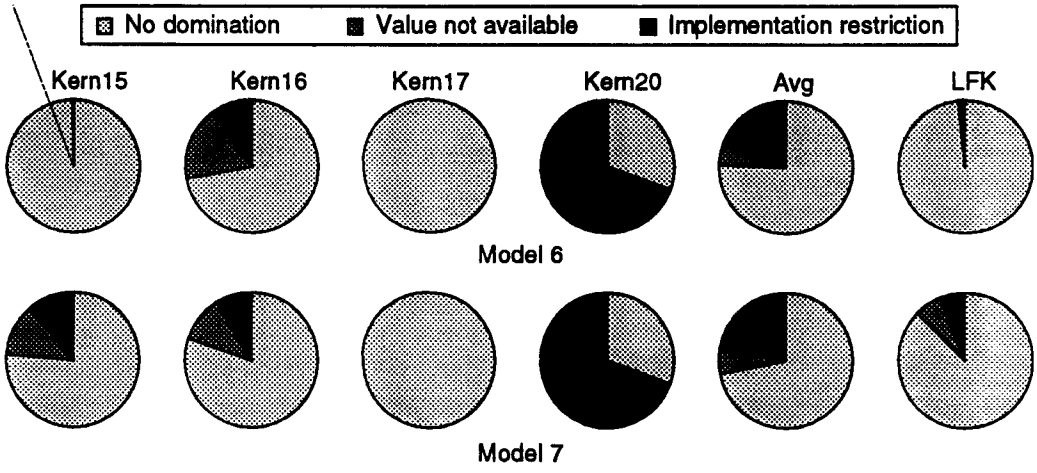

Figure 28: Why copy suppression fails

6 Comparison with other work

6.1 Bulldog compiler

Ellis measured the amount of compensation code generated by Bulldog compiling for the ELI-512 and found it ranged from 12%-144% of the number of operations in the flow graph before trace scheduling [see [6], p 244]. He measured small, heavily unrolled kernels, similar to the tuned Livermore Fortran Kernels. Our results show that compensation code for heavily unrolled kernels can be limited to 20% of the number of operations before trace scheduling. For whole programs, such as the SPEC89 benchmarks, we have shown compensation code can be limited to 6% of the number of operations before trace scheduling. In addition, Ellis found a predominance of split compensation copies, where our experiments show that split compensation code can be effectively controlled by the compiler.

6.2 Improvements to trace scheduling

Numerous improvements to trace scheduling have been suggested since Fisher first published his algorithm [17,18,16,23,15,13]. Recently, three approaches have been suggested that have different strategies for creating compensation code: superblock scheduling [14,4], Bernstein and Rodeh's global instruction scheduling [3,2], and Smith's global scheduling [26]. Our work was done before these three, but not published until now.

All three approaches constrain the scheduler to prevent motion below splits; this is the same as our split avoidance heuristic. Thus there is no split compensation code. The three approaches vary on join compensation code.

Superblock scheduling produces the maximal amount of join compensation code. A superblock is constructed by first identifying a trace, and then transforming the trace into a single-entry, multiple-exit extended basic block by performing tail duplication to eliminate any rejoins to the body of the trace [14,4]. Tail duplication repositions each join to the body of the trace to the successor of the trace, and requires that all operations between the join to the trace and the end of the trace be copied onto the rejoining edge.

Superblock scheduling can result in a very large code expansion. Hwu, et al., report that some programs increase in size by a factor of over 4 [14]. Trace scheduling augmented with the simple avoidance techniques described in section 3 will always produce less compensation code then superblock scheduling.

Global instruction scheduling proposed by Bernstein, et al., constrains the scheduler so that any join copy has the *single copy on a path* property. Their scheduler works on single-entry acyclic regions of the flow graph, and, loosely speaking, this restriction means that on every path through the region, only one copy of a duplicated operation will be executed.

Redundant join copies do not have a major effect on performance. In Table 3 we saw that our copy suppression algorithm (Models 6 and 7) had minimal contribution to program performance. Intuitively, this is because redundancy on less frequently traveled paths is not important. Our data indicates that the single copy on a path restriction is not appropriate.

Global scheduling proposed by Smith, et al., is very similar to trace scheduling with the avoidance and suppression techniques described in Sections 3 and 4. Their scheduler schedules a trace, but performs compensation code analysis as each operation is scheduled, rather than after the entire trace has been scheduled. Within a trace they identify *control equivalent* blocks, i.e., blocks that dominate and post-dominate each other, respectively.[10] For a given operation, they also identify *data equivalent* blocks; two blocks are data equivalent for an operation if the operands of the operation are not written on any path between the two blocks. To move an operation up the trace when scheduling, they move it from block to block up the trace. They move directly between control and data equivalent blocks; otherwise they copy the operation on the joining edges of any block with multiple predecessors. They simplify their scheduler by scheduling all of the operations in the current

[10]These are often called hammocks [7] in the control flow graph.

basic block before considering operations from successor blocks; this can be a poor choice for traces with long latency operations later in the trace.

Their test for control equivalence is comparable to our test for dominance. Our test is more powerful, in that they also perform a check for post-dominance. (The check for post-dominance is not motivated by compensation code; it is to identify speculative code motion.) Their test for data equivalence is comparable to our off-trace test and availability test. They have not integrated their approach with register allocation; their scheduler runs after register allocation, or assumes infinite registers.

6.3 Partial redundancy elimination

Our algorithm for removing redundant join compensation copies is an incremental elimination of partial redundancies introduced by trace scheduling. An alternative is to perform a global partial redundancy elimination after trace scheduling is complete [21,24]. This approach could detect some redundant copies which we do not.

However, removing code after trace scheduling would require the rescheduling of altered traces for best performance. (The Multiflow Trace was not scoreboarded, so this would also be a requirement for correct performance.) Moreover, our register allocation was integrated with our scheduler [11]; scheduling operations that would later be deleted would adversely effect our allocation decisions. Given the compile-time penalty of rescheduling, the effect on register allocation, and the complexity of engineering an additional global optimization late in the compiler backend, we felt our incremental algorithm to be the better choice in practice.

7 Conclusions

The most important conclusion of our measurements is that compensation code is not really that important. Compensation copies do not cause code explosion or uncontrolled code growth in our trace-scheduling compiler for the Trace computers. Our compiler did not unroll loops with internal branches prior to this implementation of compensation code suppression. In light of the data collected, this strategy was overly conservative. Allowing code motion that requires the insertion of compensation code improves the performance somewhat, but most of the gains of trace scheduling can be obtained without the code motion strategies that result in the insertion of compensation code. Since compensation code is not a major issue, a compiler can be more aggressive in code motion and loop unrolling.

If a compiler allows code motion that results in compensation copies, it is important to choose a set of restrictions on legal code motions. Avoiding compensation code is far more important than suppressing it. A few restrictions keep the actual percentage of compensation copies tolerable without incurring any loss of performance. Split copies, which are hard to optimize, are controlled effectively that way. A simple incremental algorithm can optimize away a sizeable fraction of the rejoin compensation copies; those that our algorithm does not catch are most likely to stay.

Compensation code suppression can yield large percentage improvements for individual kernels, but the performance on small kernels is of limited importance. Concentrating on the complete programs of the SPEC89 suite, it is interesting to note that the most significant speedups for scheduling with compensation code are for the hard-to-optimize scalar integer programs (see Table 6). This is encouraging and perhaps with more work, futher improvement is possible.

Acknowledgments

Many people contributed to the development of the Multiflow compiler. Mike Ward made major contributions to the copy suppression algorithm; he did much of the detailed design and implementation. We appreciate discussions with Cindy Collins, Tom Karzes, Woody Lichtenstein, and John Ruttenberg during the design and implementation of the compensation code optimizations. Bob Nix made some valuable suggestions about our experiments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] D. Bernstein, D. Cohen, and H. Krawcyzk. Code Duplication: An Assist for Global Instruction Scheduling. In *Proceedings of MICRO24*, pages 103 – 113. IEEE Computer Society, Nov 1991.

[3] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241 – 255. ACM, June 1991.

[4] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software Practice and Experience*, 21(12):1301 – 1321, Dec 1991.

[5] R. P. Colwell, R. P. Nix, Papworth D. B. O'Donnell, J. J., and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Trans. on Computers*, 37(8):967–979, August 1988.

[6] J. R. Ellis. Bulldog: A Compiler for VLIW Architectures. Technical Report DCS/RR-364, Yale Univ., Feb. 1985. published by MIT Press, Cambridge MA., 1986.

[7] J. Ferrante, K. J. Ottenstein, and J. W. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319 – 349, July 1987.

[8] J.A. Fisher. The Optimization of Horizontal Microcode Within and Beyond Basic Blocks. Technical Report COO-3077-161, Courant Mathematics and Computing Laboratory, New York University, October 1979.

[9] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers*, C-30(7):478–490, July 1981.

[10] J.A. Fisher, J.R. Ellis, J.C. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler & A Dumb Machine. In *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, Montreal, June 1984. ACM.

[11] S. Freudenberger and J. Ruttenberg. *Phase Ordering of Register Allocation and Instruction Scheduling*. In *Code Generation - Concepts, Tools, Techniques*, Giegerich, R. and Graham, S. L. (Eds), pages 146–170, Springer Verlag, 1992.

[12] T. Gross and M. Ward. The Suppression of Compensation Code. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 260–273. Pitman/MIT Press, Irvine, CA, 1990.

[13] R. Gupta and M. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421 – 431, April 1990.

[14] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1,2), March 1993.

[15] Ebcioglu. K. and A. Nicolau. A Global Resource-Constrained Parallelization Technique. In *Proceedings of the Third International Conference on Supercomputing*, pages 154 – 163. 1989.

[16] J. Lah and D. Atkins. Tree Compaction of Microprograms. In *Proceedings of the 16th Annual Microprogramming Workshop*, pages 23 – 33. IEEE Computer Society Press, October 1983.

[17] J. L. Linn. Srdag Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information. In *Proceedings of the 16th Annual Microprogramming Workshop*, pages 11 – 22. IEEE Computer Society Press, Oct 1983.

[18] J. L. Linn. *Horizontal Microcode Compaction*, in *Microprogramming and Firmware Engineering Methods*, Habib, S. (Ed), pages 381 – 431. Van Nostrand Reinhold, New York, N. Y., 1988.

[19] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1,2):51–142, March 1993.

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

[21] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96 – 103, Feb 1979.

[22] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness Issues for a Trace Scheduling Compiler.* PhD thesis, Yale University, June 1984.

[23] A. Nicolau. Percolation Scheduling: A Parallel Compilation Technique. TR 85-679, Department of Computer Science, Cornell University, Ithaca, N. Y., May 1985.

[24] B. K. Rosen, M. N. Wegman, and Zadek F. K. Global Value Numbers and Redundant Computations. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, pages 12 – 27. ACM, Jan 1988.

[25] J. Smith. Characterizing Computer Performance with a Single Number. *CACM*, 31(10):1202–1206, Oct 1988.

[26] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient Superscalar Performance Through Boosting. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248 – 259. ACM, Oct 1992.

[27] J. Uniejewski. Spec Benchmark Suite: Designed for Today's Advanced Systems. *SPEC Newsletter*, 1(1), Fall 1989.