HEWLETT PACKARD

Global Code Generation For Instruction-Level Parallelism: Trace Scheduling-2

Joseph A. Fisher
Computer Research Center
HPL-93-43
June, 1993

instruction-level parallelism, compilers, VLIW, superscalar, code generation

Trace Scheduling-2 is a method of code generation for instruction-level parallel architectures. Like its predecessor, *Trace Scheduling*, it relies on estimates of likely program jump directions to select operations to move from basic block to basic block. Unlike trace scheduling, trace scheduling-2 is "nonlinear", that is it allows code to move above a conditional jump from both sides at the same time. Code which falls below the predicted less-likely branch is treated as a first-class citizen. Trace scheduling-2 differs from other techniques with similar objectives in allowing the code generator, rather than a stand-alone phase of the compiler, to make the fundamental choices of code motion. By using an expected value function, the *speculative yield*, trace scheduling-2 can decide whether to move operations between blocks in a manner that is considerate of the cost of speculative execution, and in a manner that interacts naturally with standard scheduling priority functions, such as DAG height. Speculative yield is useful in solving some problems that have appeared in trace scheduling implementations as well.

In order to generate good code for instruction-level parallel systems, accurate jump predictions must be possible. This paper reports on some measures of how realistic it is to expect to have good enough predictions at compile time.

Trace scheduling-2 is now being implemented. This paper considers some of the design issues and challenges of that implementation, reviews trace scheduling and some of its proposed extensions, presents the motivation for and main algorithms of trace scheduling-2, and discusses the issue of data-driven predictions of flow of control, which is central to an accurate speculative yield function.

Internal Accession Date Only
Presented at the *Workshop on Advanced Compilation Techniques for Novel Machine Architecture*, Jerusalem, 1991. To be published by *Springer-Verlag*, London, UK

Global Code Generation For Instruction-Level Parallelism: Trace Scheduling-2

Joseph A. Fisher
Hewlett-Packard Laboratories[1]

Abstract

Trace Scheduling-2 is a method of code generation for instruction-level parallel architectures. Like its predecessor, *Trace Scheduling*, it relies on estimates of likely program jump directions to select operations to move from basic block to basic block. Unlike trace scheduling, trace scheduling-2 is "nonlinear", that is it allows code to move above a conditional jump from both sides at the same time. Code which falls below the predicted less-likely branch is treated as a first-class citizen. Trace scheduling-2 differs from other techniques with similar objectives in allowing the code generator, rather than a stand-alone phase of the compiler, to make the fundamental choices of code motion. By using an expected value function, the *speculative yield*, trace scheduling-2 can decide whether to move operations between blocks in a manner that is considerate of the cost of speculative execution, and in a manner that interacts naturally with standard scheduling priority functions, such as DAG height. Speculative yield is useful in solving some problems that have appeared in trace scheduling implementations as well.

In order to generate good code for instruction-level parallel systems, accurate jump predictions must be possible. This paper reports on some measures of how realistic it is to expect to have good enough predictions at compile time.

Trace scheduling-2 is now being implemented. This paper considers some of the design issues and challenges of that implementation, reviews trace scheduling and some of its proposed extensions, presents the motivation for and main algorithms of trace scheduling-2, and discusses the issue of data-driven predictions of flow of control, which is central to an accurate speculative yield function.

1. Introduction

A machine is said to offer **Instruction-level Parallelism** (or **ILP**) when it can simultaneously execute several machine operations, such as memory loads and stores, integer additions and floating point multiplications. These are normal RISC-style operations executed in parallel, even though the system is handed a single program written for a sequential processor. Most manufacturers of CPUs in the early 1990's appear to have an ILP microprocessor under development or already in the marketplace; machines using ILP have been with us since the CDC-6600 and IBM 390/91 [Thornton 64] [Anderson et al. 67] in

---

[1]Author's address: Hewlett-Packard Laboratories 1501 Page Mill Rd. 1U-15, Palo Alto, CA 94304; jfisher@hpl.hp.com. This paper has been accepted for publication in The Proceedings of the Workshop on Advanced Compilation Techniques for Novel Machine Architectures, Jerusalem, May, 1991. To be published by Springer-Verlag, London, UK.

the early sixties, but are now having a resurgence in the form of Superscalar and VLIW microprocessors. An introductory survey of instruction-level parallelism and the techniques that make it practical can be found in [Fisher and Rau 91/92].

When large amounts of ILP are available in the hardware, we must look far down the instruction stream to find enough data-independent operations to keep the CPU busy. In doing so, we must move operations from basic block to basic block, and in particular move operations up ahead of conditional jumps. But we cannot simply execute data-ready operations as soon as possible, both because of the technical challenges involved in doing operations before we are sure they should be done, and because the density of conditional jumps yields far too many paths that the code might take in the future. We must select operations from the many candidates, and thus we must have good predictions of the directions that conditional jumps are likely to take. Then we can consider only the most likely path as a source of operations, or can consider operations that come from both directions sometimes, and only one direction other times.

Motivated by the above, this paper address two major aspects of compiling for ILP architectures:

- Getting and using flow control predictions as part of code generation for ILP CPUs.
- Extending a class of code generation techniques (techniques in the style of a technique called Trace Scheduling) to be more considerate of code coming from the less likely direction.

An ILP Execution Hardware Subset

The examples in this paper will use a very simple subset of the execution hardware one might find in a typical ILP system, extracting only those features required for the issues discussed here in detail. The execution hardware, shown in Figure 1, consists of two integer functional units and a branch unit connected to a large common register file. The functional units can carry out integer addition and multiplication, and the branch unit can use the result of comparisons to determine the flow of control of the program. This hardware is capable of starting any two operations in its repertoire (addition, multiplication, or conditional jump) every cycle.

All of these operations get their input operands from the register file, which can deliver up to four operands each cycle, and can write back two result values each cycle. One complicating factor is that, while the result of an addition is written back in the cycle in which it was initiated, a multiplication takes four cycles. Thus a multiplication initiated in cycle n writes its results back in cycle n+3, and an operation that reads the results of that multiplication can't be initiated until cycle n+4. Thus there is a forward resource constraint that a code generator (or a hardware scheduler) must consider. Usually ILP CPUs pipeline long operations, thus more operations can be issued even if both functional units are busy doing previously issued multiplications.

A Program Fragment

Figure 2 is a program fragment in low-level sequential code for this execution hardware, and Figure 3 shows the program as it might execute on this hardware. In each case, the data precedence graph (or **DAG**, for directed acyclic graph) for each basic block (or section of straight line code) shows which operations must follow which others for the correct computation to be done.

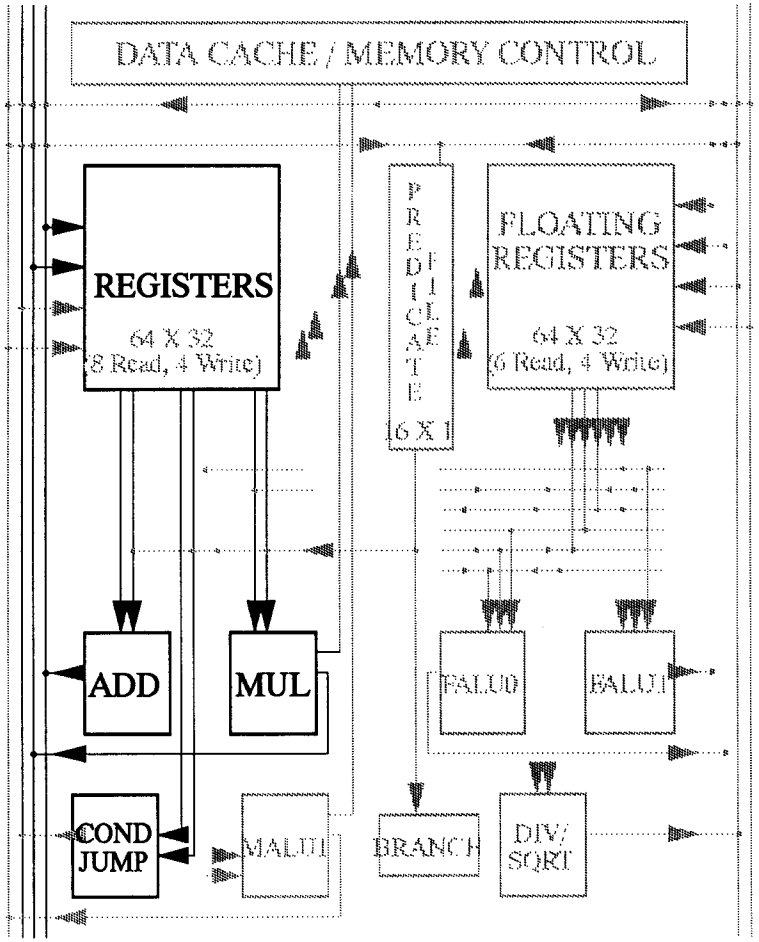


Figure 1: The ILP hardware subset used here (shown as the part with dark lines).

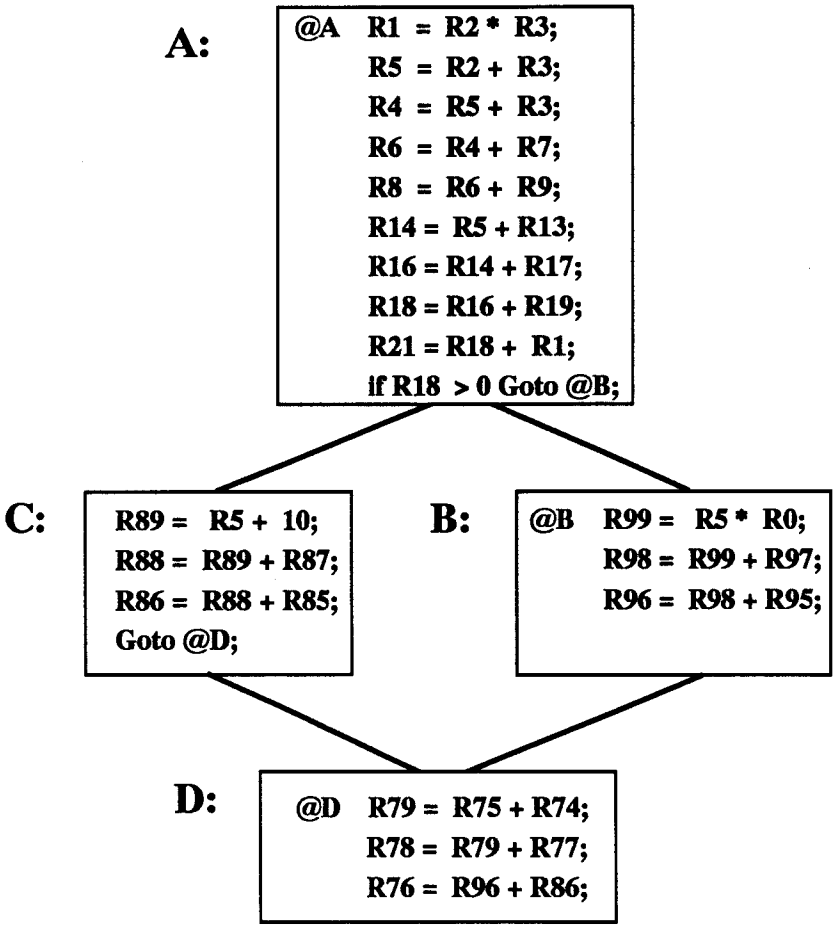


Figure 2: A sample program fragment using the small hardware subset shown to the left.

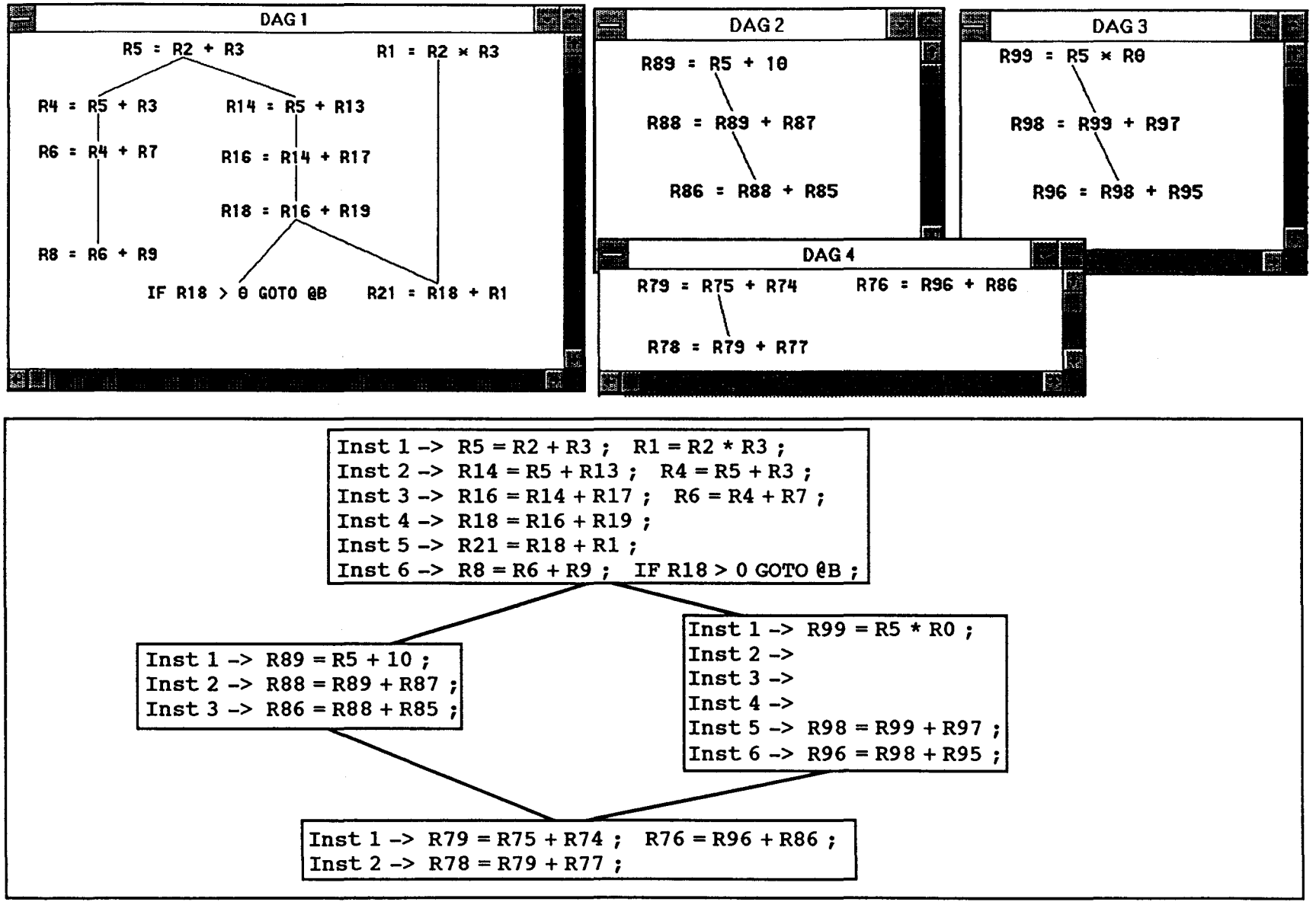


Figure 3: The DAGs and schedules when each of the four basic blocks is compacted separately. Note that the first block uses almost all of the available resources, the others do not.

2. Superscalars, VLIWs and Code Generation

Many different kinds of ILP hardware have been built or proposed. A significant distinguishing feature among them is the timing of the decision to group operations together for simultaneous execution. This decision might be made completely by the hardware, completely by the compiler, or in part by each, but in all cases the result is an execution pattern of the sort shown in Figure 3.

Superscalars. If the hardware determines which operations to overlap as the code runs, the CPU is called **Superscalar** [Johnson 91]. In that case, the hardware is handed sequential code, does the equivalent of building the DAGs, and decides, while the program runs, which operations to do at the same time. Although the question of how far this approach can be pushed before the hardware gets impractical is a very controversial one, machines which issue two operations at once (as in Figure 1) are certainly practical.

VLIWs. When the compiler choreographs the operations completely in advance, the ILP CPU is called a **VLIW**[2] [Fisher 83]. VLIW stands for Very Long Instruction Word machine, because the records of execution shown in Figure 3 look exactly like the object code such a machine requires. Each line is thought of as a single very long instruction consisting of many RISC operations bundled together by the compiler, thus the name.

Generating Code for ILP CPUs

Many CPU designers hope to achieve a great deal of instruction-level parallelism, perhaps finding speedups of a factor of five or more on systems and other codes, and far more than that on highly structured scientific codes. A key factor in achieving this goal is the handling of conditional jumps. Many experiments have shown that unless code is moved across basic block boundaries, there is relatively little parallelism available [Riseman and Foster 72] [Nicolau and Fisher 81] [Wall 91]. To achieve much greater density, operations can be moved between blocks. When they are, the process is referred to as **Global Compaction** and the code motions themselves are called **Global Code Motions**. Global compaction can be done by the compiler beforehand for a VLIW, or by the hardware at run time for a superscalar. It can also be done beforehand by a compiler for a superscalar by having the compiler rearrange the sequential code. Indeed, though there is no universal agreement about this, many people (myself included) believe that extensive global code motion is too complex to expect the hardware to do very much of it at run time. If that is true, the real determinant of how much work must be done by the compiler is the amount of parallelism required, rather than whether one is generating code for a superscalar or a VLIW. In superscalar code generation, the compiler must rearrange the code so that independent operations are closer to each other than they were in the original source, getting them close enough for the hardware to do the final scheduling. While the new program will still be sequential, it will have been optimized for the superscalar in question. Most of the techniques in this paper, though sometimes expressed as if they were meant for a VLIW, map directly into code generation techniques for a superscalar.

In the course of doing code motions, one may move an operation above a conditional jump that might otherwise have prevented its execution. This is termed **speculative execution**, and it can be profitable when otherwise idle resources are used to execute operations whose results we might end up using. A system doing speculative operations must be able to ignore the results of those operations when the flow of control does not arrive at their source. Error conditions raised by speculative operations must be ignored

---

[2]There are several alternative ways of building hardware that requires the compiler do all or most of the choreography. These can be very different from VLIWs in important respects, but often require the same sophistication from the compiler that's under discussion here. For a short summary, see [Fisher and Rau 91/92].

then as well, but still handled when the flow does mandate that the operations would have been executed after all. The handling of error conditions in speculative operations is beyond the scope of this paper, but speculative code motions constitute an important class of global compactions, and probably must be handled if one is to achieve a large degree of ILP.

In addition to the techniques considered in this paper, there is a class of techniques called **Software Pipelining** [Rau and Glaeser 81] [Lam 88] for generating code from very tight loops in the source. The goal of software pipelining is to rearrange the pattern of computation inside a loop in such a way that each iteration can be issued at a fixed interval following the previous one, and the code will fit perfectly—the resources used by the new iteration will be available in each cycle as required, and the data it needs to use will be ready on time. The techniques discussed here are instead appropriate for a general flow of control, which software pipelining cannot handle, but will presumably not do as well when presented with a tight loop. Although most of the issues in software pipelining are disjoint from those considered here, there are interesting questions involving how the two classes of techniques can be mixed, and where the boundaries lie. These questions have not, to my knowledge, been addressed in the literature.

A Phase Ordering Problem

Most programmers try to write code that works in phases. To do something as complex as generating ILP code one might consider the following reasonable sequence of tasks:

1. Generate sequential code
2. Schedule each basic block, using the parallel capabilities of the hardware
3. Move operations from block-to-block to increase parallelism

This was, in fact, the method first used to generate ILP code in which the code motions crossed basic blocks [Tokoro et al. 78], and in precursors of software pipelining.

What's Wrong With This Philosophy? Most researchers believe that this approach results in too many arbitrary choices being made that have to be undone to produce a good schedule, although, to my knowledge, no one has ever demonstrated that this is so. This happens because:

1. Operations that should be done in concert are arbitrarily assigned to use the same resources, causing false dependences and unnecessary conflicts.
2. Operations which are not very critical are done early, using up slots that could have been used by a much more critical operation that would have moved from other basic blocks.

Both of these factors can substantially reduce the parallelism in the generated code. The first factor is obvious: if resource allocation, functional unit assignment, et al., do not take the final schedule into consideration there will be desirable combinations of operations that cannot be done in parallel. The second factor is more subtle but also very important: when compacting only a basic block, there is no way to control greed, since there is nothing to be lost in doing noncritical operations in early free cycles. When it comes time to move an operation into that basic block from outside it, the unimportant operations will prevent that motion. Improved schedules can only be arrived at by temporarily generating poorer schedules for the target basic block, demoting the unimportant operations to new points in the schedule. Indeed, because resources will also be bound in advance, it may now be impossible to do the intermediate code motions, since other ops may be occupying the necessary resources.

An example. The example in Figures 2 and 3 was constructed to illustrate this point. The code generator does a very good job on basic block A, but in so doing closes out the multiply at the start of block B. Suppose in practice that the test ***if R18 > 0 Goto @B*** at the end of block A is taken most of the time. In

that case, it would be advantageous to speculatively start executing the multiplication in block B as early as possible. Unfortunately, there are no "holes" in the schedule of basic block A in which to insert the multiplication. Making such a hole is very difficult, requiring a chain of rearrangements which cause the schedule for block A to get worse. The search process that would have to be built into a compiler to find a good solution would be expensive, and since the number of potential code motions between blocks is very large, that approach does not seem desirable. It is this situation that motivated a method of doing the initial code generation in a manner not arbitrarily constrained by block boundaries.

3. Doing All of Code Generation At Once

A technique developed in 1979 addresses the phase ordering problem by combining the code generation tasks. Thus, deciding the ultimate destination block of an operation is done at the same time as generating the instruction-level parallel code, rather than as a separate phase. This technique, called **Trace Scheduling** [Fisher 81], has been followed by several others that have utilized this same philosophy partly or completely, but have attempted to improve the trace scheduling algorithms (see, for example, [Linn 83], [Su, Ding and Jin 84], [Nicolau 85]). All of these methods have, to an extent, followed the trace scheduling solution to the phase ordering problem:

> Gather a large number of candidates at once and start scheduling. These operations originate in many basic blocks. Build a data-precedence graph (DAG) from all of these operations, being willing to do all the code motions as a side effect of the decision to select operations from the DAG for scheduling.

The Trace Scheduling Algorithm

Trace scheduling has been documented extensively, the best reference to date being [Ellis 85]. Here, with the help of Figure 4, is a brief description, required for an understanding of what follows:

1. Using profiling information, the scheduler selects a **trace** (a loop-free, linear selection of code, which might include several basic blocks) from the code not yet scheduled. Figure 4, panel 1, shows such a selection. The compiler can use loop unrolling, inlining, and other techniques to get large sections of code from which to pick traces. The next section of this paper is about the practicality of obtaining appropriate profiling information.
2. By treating the trace as if it were a single basic block, the compiler schedules it in a manner similar to the way it would schedule a basic block. In particular, it builds a DAG containing all the operations on the trace, including conditional jumps. During this process, branches are given no special consideration except that their order is not altered. Register allocation, functional unit selection, and so on, are done only as an operation is being scheduled (though some researchers advocate doing register allocation afterwards).

   Edges are added to the DAG to prevent an operation from moving up above a conditional jump if it would overwrite a live value on the off-trace edge. (A good compiler will use renaming to eliminate many of these edges.) In practice edges are added to preserve the order of conditional jumps, since the engineering involved in inverting jumps is usually deemed not worth the trouble.

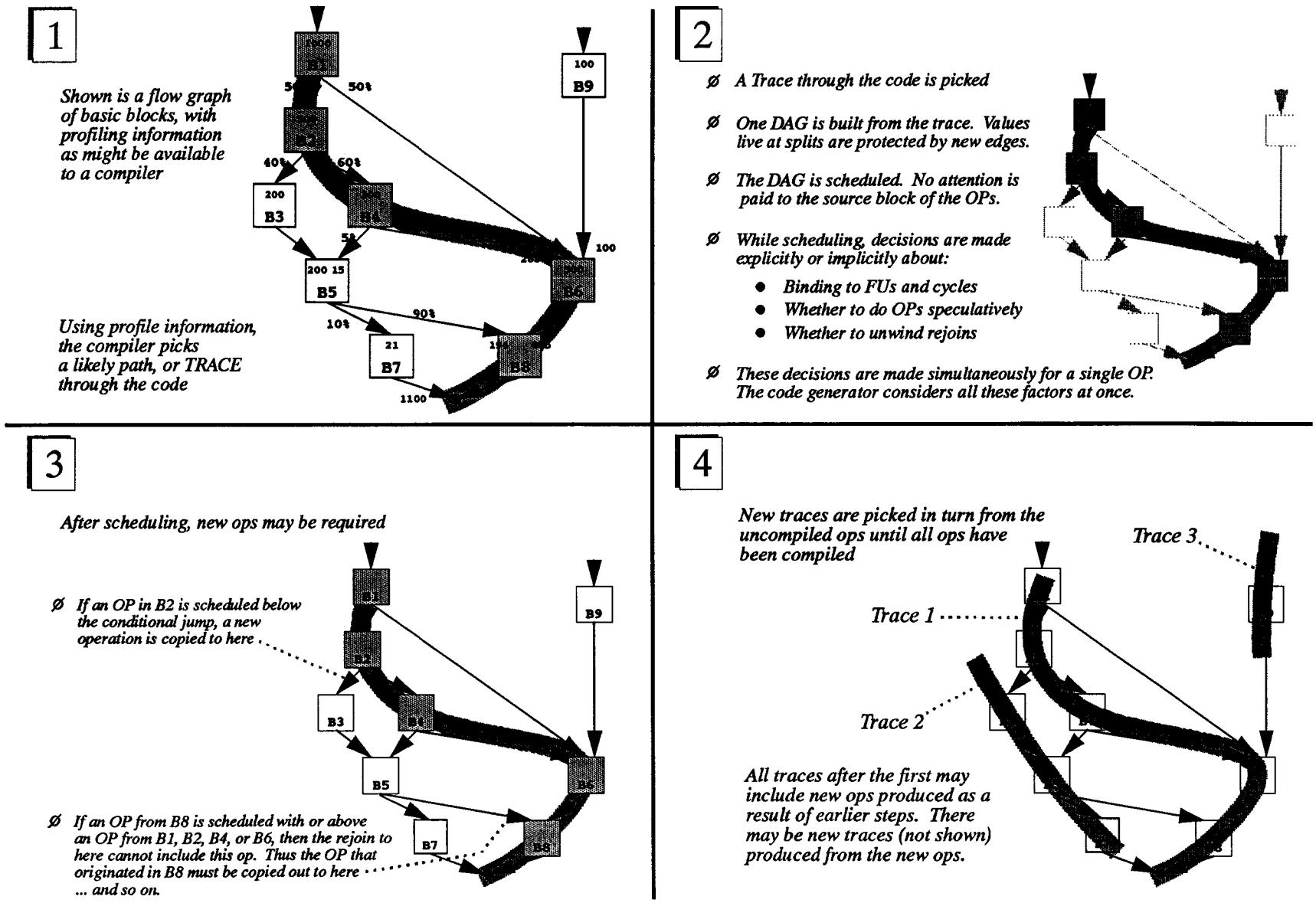


Figure 4: A concise summary of Trace Scheduling. All jumps are passed in the most likely direction only, most code generation decisions concerning an operation are made at the time it is scheduled.

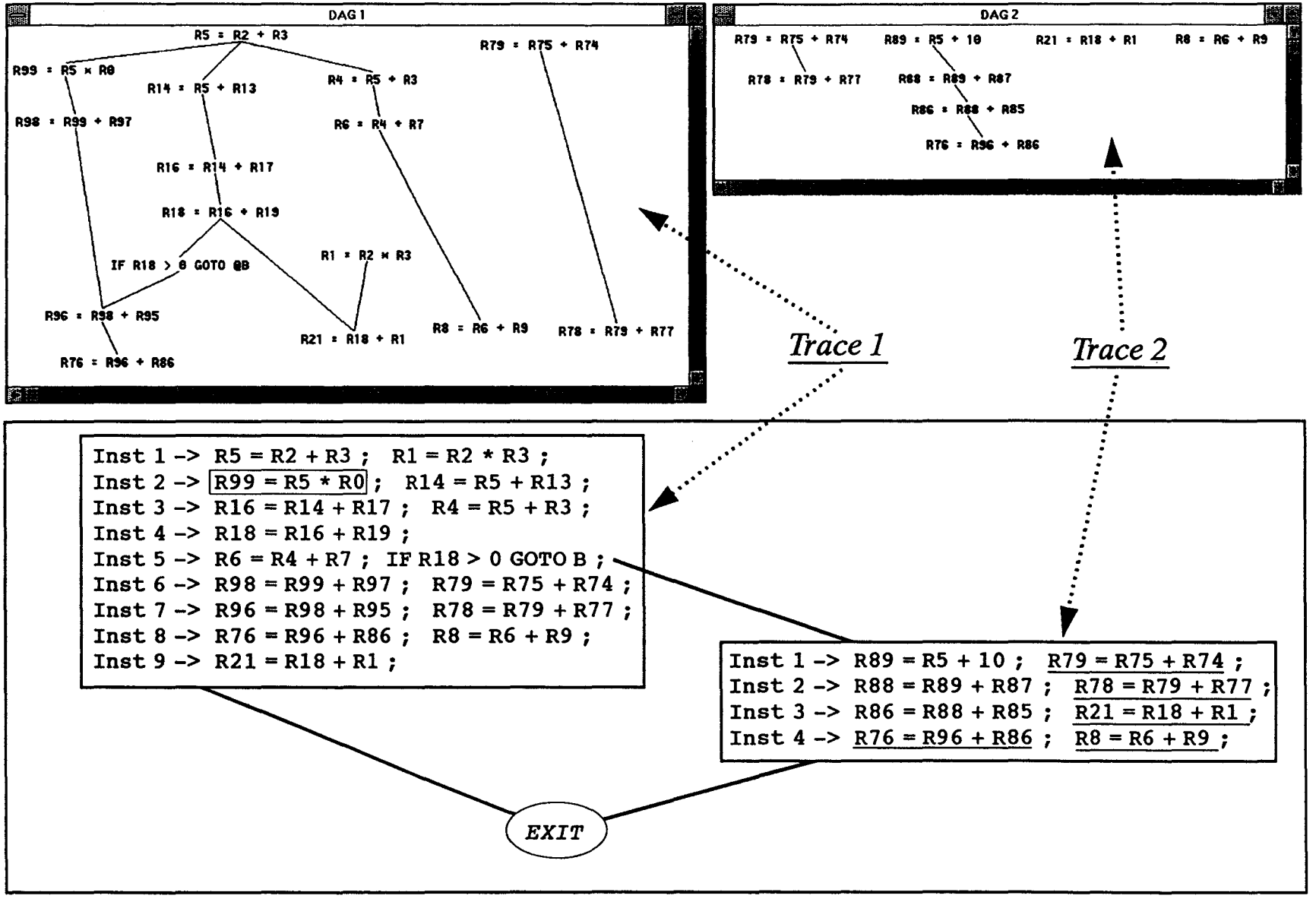


Figure 5: The example trace scheduled. The boxed operation in trace 1 is executed speculatively. The underlined operations in trace 2 are split and rejoin compensation code.

3. After scheduling the trace, the compiler sometimes has to add, off the trace, copies of operations it has scheduled on the trace. This occurs because some operations will have been scheduled after a conditional jump that they used to precede and thus must be duplicated before the off-trace target of the branch. In addition, rejoins that used to jump into the trace now can only jump to the highest point after which may be found only operations that were below the original rejoin. In practice, this code has been minimal, but it is an issue to be dealt with in ILP compilers that do a lot of code motions.
4. The above is done repeatedly until the flow graph has been covered with disjoint traces and no unscheduled operations remain.

This process is shown in Figure 5, working on the code and hardware in figures 1 and 2. In that example, the trace is assumed to go through the basic blocks A-B-D. Using trace scheduling, the path A-B-D finishes in nine cycles, rather than the fourteen cycles required when the blocks are compacted separately. Again, this is a contrived example, but experience has shown trace scheduling to be very effective (see, for example, [Ellis 85], [Colwell et al. 87]).

4. Predicting Flow Control Directions to Cope With Uncertainty

Trace scheduling, and all the methods discussed in this paper, rely upon the accurate prediction of the predicates used to control jump directions. Indeed, some of the experiments discussed above ([Riseman and Foster 72] [Wall 91]) have indicated that the amount of hardware necessary to execute all data-ready operations all the time grows very fast. Thus the strategy of statically predicting the results of conditional jumps is necessary no matter what method one uses to generate code, otherwise one cannot select only the highly-likely-to-be-executed operations and avoid the low-payoff operations. A superscalar CPU might dynamically count the recent history of each jump, but if a compiler has to do a lot of code motion for the superscalar, it must pass jumps carefully. Thus, even for a superscalar, static jump prediction can be necessary.

Although it is possible to examine the source and use heuristic-based predictions, or to ask the programmer to supply some information, many people advocate the use of representative "test runs" and the collection of statistics to predict which way jumps will go. A facility to do this was built into the Multiflow Trace compiler, and, I believe, various other compilers for different reasons.

The Effectiveness of Data Driven Program Flow Prediction

Just how statically predictable are programs, and how realistic is it to expect to use test runs to predict them? A study underway at Hewlett-Packard Labs [Fisher and Freudenberger 92] attempts to answer these questions, and many related ones, using tools developed for the Multiflow Trace (a VLIW with up to 1024 bit instructions) compiler. In this paper I will discuss a few partial results from that study, at least enough to give the sense that jumps are predictable enough to make techniques like those discussed here practical.

There are 12 programs discussed here: the familiar SPEC suite, along with the utilities *compress* (the familiar Unix compression routine) and *spiff* (a program that reports the differences but ignores the near differences between file contents); these are a subset of the programs used in the full study. The part of the study discussed here is an attempt to answer the question "are programs statically predictable?" Although the results are applicable to several domains, the goal here is to measure our ability to look at earlier runs

of a program and decide, for each jump in the source, which way it will go what percentage of the time. One might hope that two different things would be true: that jumps tend to go in one direction most of the time, and that this direction doesn't usually change when the data does. It is important to note that both of these are dynamic statements; we're interested in jumps in proportion to how frequently they are executed. Thus:

> Suppose that for each conditional jump appearing in the source text of the program, the best possible prediction of which way the jump goes, given a particular data set, is made before the program runs. If, after the program runs, the dynamic count of the number of times those predictions were correct, summed over all jumps, is near 50%, we say that the program/data set combination is **inherently unpredictable.**

For example, suppose a program has three jumps in its source, and for some run of the data, the following occurs:

JUMP #	# EXECUTED	# TAKEN	BEST PREDICTION
JUMP #1:	1000	450	550
JUMP #2:	800	200	600
JUMP #3:	2000	1100	1100
TOTAL:	**3800**		**2250 (59%)**

Intuitively, these jumps are each very near 50-50, and the best possible prediction can't do very well. Notice that jump #2 is more predictable than the others, but we're interested in dynamic results, and jump #2 isn't executed as often. Alternatively, a program might be inherently predictable, but different datasets might cause very different behavior at jumps:

> Suppose that for each conditional jump appearing in the text of the program, we use the results of a run of the program using some predictor dataset to predict which way the jump goes before the program runs. If the program then runs with a target dataset and the dynamic count of the percentage of times those predictions were correct, summed over all jumps, is near 50%, we'd say that the program + 2 data sets were **dataset unpredictable.**

For example, consider again a program with three jumps in its source, but now with a prediction having been made, based on previous runs, of whether each jump is or isn't likely to be taken:

JUMP #	PREDICTION	# EXECUTED	# TAKEN	PREDICTED RIGHT
JUMP #1:	yes	4000	3700	3700
JUMP #2:	yes	3200	50	50
JUMP #3:	no	2000	1100	900
TOTAL:		9200		4650 **(50.5%)**

Here, we would say that the program + dataset are inherently predictable, since the best possible predictor would be correct 86.4% of the time. But the data we're using for prediction isn't good enough.

Of course, the predictor datasets might be expected to be the sum of several "reasonable datasets," not just one. One important factor is the completeness of previous runs: if the dataset being predicted exercises a

jump that the previous ones exercised lightly or not at all, any prediction of the behavior of that jump would likely be meaningless. The program *compress* provides an easy example of this. In some systems, *compress filename* compresses a file, and *compress -d filename* decompresses it. In fact, the first thing this program does is call one of two disjoint subroutines based on the presence or lack of the -d switch! When decompress data is used to predict the compress case, it is essentially random.

To see the extent to which these codes are inherently- and dataset- unpredictable, consider Table 1. Next to each sample program name, column 2 shows the number of datasets collected for it. Note that in the case of FPPPP, Spice and Doduc there are multiple datasets, but they are so similar in spirit they might as well be considered a single dataset, and GCC (in this partial study) has only one dataset represented. Tomcatv, Matrix300 and Nasa7 have only one dataset collected, but they represent programs whose flow of control is so rigid that changing the data does not make a big difference (though with a rigid flow of control a program can still be inherently unpredictable, if the flow is 50-50 in many places).

	No. of Data-sets	Correctly Pred. Jumps (Best Possible)	Correctly Pred. Jumps (Using Datasets)	Ops/ Jump	Ops/Wrong Jump (Best)	Ops/Wrong Jump (Data)
TOMCATV	1	99.6%	N/A	28	7461	N/A
MATRIX300	1	99.7%	N/A	17	5123	N/A
NASA7	1	99.4%	N/A	21	3400	N/A
FPPPP	2	83.1%	N/A	154	913	N/A
GCC	1	87.8%	N/A	9	74	N/A
SPICE	2	92.0%	N/A	12	148	N/A
DODUC	3	93.1%	N/A	18	260	N/A
COMPRESS	8	86.3%	86.2%	10	72	72
SPIFF	4	88.8%	88.0%	13	116	108
XLISP	5	84.4%	84.3%	8	50	50
ESPRESSO	8	83.9%	79.8%	7	45	36
EQNTOTT	8	86.6%	86.4%	5	37	36

Table 1: The number operations between mispredicted jumps, using the best possible static prediction for each program + dataset, and using other datasets.

The third and fourth columns of Table 1 show the percentage of times the predictions are right. Column 3 shows the inherent unpredictability, that is each dataset is used to predict itself; the number shown is the average over all datasets. Column 4, where applicable, shows what happens when you use all of the other datasets to predict a given dataset; again the number shown is the average of applying that process to each dataset separately. These numbers are in line with the conventional wisdom, at least as I've heard it. In addition, the similarity between the columns shows that for this data *dataset unpredictability is not an issue*. Only *espresso* shows a meaningful, though still small, difference, when other datasets are used to

predict the target dataset. (In the full study, more varied datasets and other factors may weaken this statement somewhat, but it appears to be a fair generalization nonetheless.)

Predictability—what measure to use?

The literature typically measures the percentage of jumps that go in the predicted directed—the figure of merit used in columns 3 and 4. If one were interested in the behavior of jumps in isolation, that would be an interesting measure. Perhaps there is a reason for being interested in jumps in isolation, but here what we care about is the behavior of programs, not jumps. As such, the percentage that jumps are correctly predicted is virtually useless. That is evident if one looks at the SPEC programs *FPPPP* and *XLISP*. A glance at the code makes it clear that *FPPPP* is a very predictable program—it spends most of its time calculating gigantic expressions. *XLISP* is a Lisp interpreter—it spends most of its time questioning what's going on and is obviously far less predictable. Yet by this measure *XLISP* is more predictable than *FPPPP* ! The problem lies in the fact that "percentage correct" ignores the density of jumps. Indeed, dynamically, *XLISP* contains 20 times *FPPPP*'s jump density. This swamps the small differences in jump predictability found in columns 3 and 4.

From the above discussion, it becomes evident that a more interesting number is **(dynamic) ops per mispredicted jump**. This can be computed over several different datasets as the average of the number of ops there are for each jump, divided by the average percentage of jumps that are mispredicted.

This seems to be the right measure for scheduling techniques like those discussed here: a correctly predicted jump will be passed by speculatively executed operations as if it weren't a jump at all, and that will turn out to have been the correct assumption. In that sense, correctly predicted jumps "go away". Any hardware used by operations that passed the jump will have been profitably used. A mispredicted jump, however, implies a waste of hardware used, and a total loss in having chased down the mispredicted path. The question is how far can one go, on average, before a mispredicted jump. It's easy enough to correct the figures in Table 1 to reflect this improved measure: column 5 shows the average ops/jump[3]. Dividing that by the percentage of jumps that are mispredicted, via the measurements in columns 3 and 4, gives us columns 6 and 7 respectively. Now we see a very dramatic, more than 200-fold difference in the effect of jumps.

One important caveat about this partial study: there are breaks in the flow of control besides mispredicted jumps that also imply an end to available instruction-level parallelism. These include indirect jumps, switch statements, subroutine calls, and others. The fuller study referred to above discusses and measures these effects in more detail. These are complex issues, but their effects can often be eliminated: the net effect is small, but only if one does relatively sophisticated compiler optimizations. In any case, this does not seem to detract significantly from the provisional conclusions one can draw from this part of the larger study:

1. For the benchmarks and datasets studied, other datasets could predict jumps almost as well as they could be predicted (that is, while there are varying degrees of *inherent unpredictability*, these programs and datasets showed very little *dataset unpredictability*).

---

[3]For various reasons, the Multiflow Trace VLIW compiler produces extraneous operations, but does about the same number of conditional jumps as any other RISC machine. Ops/jump from the Trace would not be a suitable number to use, so this column is calculated using the MIPs Pixie utility sometimes, and more often using the counts in [Cmelik et. al. 91].

2. Although there is an enormous range, the most unpredictable programs have surprisingly large numbers of operations between mispredicted jumps[4].

Whatever conclusion one draws from the above data, it is clear that jump prediction must play a large role in code generation. This study implies that jump prediction will allow one to go considerably beyond conditional jumps before paying the price of mispredicted jumps.

5. Extending Trace Scheduling to Nonlinear Code Motions

As described, trace scheduling will usually miss code motions which:

- Are speculative, and
- Move ops from blocks in one trace to blocks in another.

Trace scheduling can be thought of as a linear process: it moves operations along a straight line of code. The motions trace scheduling misses by only going in one direction at conditional jumps can be called **Nonlinear Code Motions** and techniques that emphasize such motions can be called **Nonlinear Compaction** methods.

Suppose that in Figure 5 the trace had been chosen as A-C-D, instead of A-B-D, and suppose further that the probability of the flow A-B and A-C are both very near 50%. In that case, the same argument used to motivate trace scheduling in the first place points out a potential weakness of trace scheduling: again, the long latency operation ***R99 = R5 * R0***, at the start of block B, should be started early. Even though it is only 50% likely to be profitable, it speeds up block B so much that the gamble is worth it.

I don't know of any experiments which have been done that measure how much of the available parallelism is thrown away by doing only linear code motions. The experiment in [Wall 91] measures something close to this difference, but seems not to account for an important class of speedups, which may (or may not) have a large effect on the results. In that experiment, the setting of one of the parameters allows a choice between moving operations up past jumps from only the most likely direction, and moving operations up from both directions. The former alternative is put forward as a measure of the parallelism available to trace scheduling, and at first glance it is. But suppose in Figure 2 the trace is chosen to be A-B-D, and an operation, ***@D R79 = R75 + R74***, say, is moved from D up into A. Then, even when the flow of control goes "the wrong way," through A-C-D, ***@D*** will have already been done in A, and trace scheduling will have found the parallelism. However, [Wall 91] wouldn't catch that parallelism, since when the jump goes the "wrong" way, it presumably flushes all operations which moved up past the incorrectly predicted jump, and would thus unnecessarily flush ***@D***. Code motions from D to A can be termed **dominator code motions**, that is, backward code motions from a block to a block that it postdominates (D postdominates A means that every path through A must ultimately get to D). Such motions are never speculative, although they may go past conditional jumps. (Of course, one must be sure that the code motion can be made on all paths from the dominated block to the dominator block.)

This distinction is potentially an important one. Compiler transformations and hardware support can turn speculative code motions into dominator code motions, allowing trace scheduling, and other techniques that go one way at conditional jumps, to find more of the available parallelism. This goes farther than the

---

[4] I say surprisingly because it has been my experience that, when asked, many architecture/hardware/compiler experts reason that most C programs have a jump about every 5 operations, and most of those jumps are nearly 50-50, yielding much smaller numbers.

distinction between one-sided and two-sided jump passing, it is also a question of speculative vs. dominator parallelism. Speculative parallelism is a difficult process, somewhat because of the potential for the wasted use of functional units which might have been used profitably elsewhere, but mostly because of the problem of false exceptions. By moving operations from a postdominator block, motions that pass jumps do not have to be speculative. The hardware used to execute them is always used profitably, and exceptions they raise can and should be handled as they occur.

Previous Methods of Going Both Ways at Jumps

Despite the lack of experimental evidence that linear traces are limiting, the intuition that moving operations past splits from both directions is beneficial is very strong. Thus there have been several attempts to generalize trace scheduling to do that. The main ones I know of are:

- The original formulation of trace scheduling
- *Percolation Scheduling*
- *SR-DAG Scheduling*

Original Trace Scheduling. Trace scheduling as it was originally put forward [Fisher 81] contained a technique for moving some operations past the less-likely jumps. When a trace is selected, operations that are data ready at the top of the trace could be "lifted" up into holes in already compacted traces which jump to this trace. Unfortunately, this suffers again from the phase-ordering shortcoming that motivated trace scheduling in the first place: the already scheduled trace is, obviously, already scheduled. Small changes in the scheduling of the earlier trace might have made it possible to lift critical operations, but, it's too late. So while trace scheduling's approach to the phase ordering problem between scheduling and code generation is to do both at the same time, it doesn't follow the same philosophy with respect to nonlinear code motions. The Yale and Multiflow implementations of trace scheduling didn't attempt this task lifting.

	ACTION TAKEN AT REJOINS	WHEN NONLINEAR MOTIONS BOUND
ORIGINAL EXTENSION TO TRACE SCHEDULING	Lift Ops To Scheduled Trace After Scheduling	After Scheduling Previous Traces
SR-DAG COMPACTION	No Compaction Above Rejoins	During Scheduling
PERCOLATION SCHEDULING	Code Motion OK At Rejoins	Before Scheduling Begins
TRACE SCHEDULING - 2	Code Motion OK At Rejoins	During Scheduling

Table 2. Distinctions among proposed nonlinear global compaction methods.

Percolation Scheduling. Alex Nicolau, one of the participants in the Yale Bulldog compiler project, developed a technique called Percolation Scheduling [Nicolau 85] to overcome this limitation of trace scheduling. Percolation scheduling works as follows:

1. Refer to a master catalogue of all possible code motions that might increase the opportunity for ILP. The code motions can move from any block to any other, as long as they do not cross a back edge.

2. Make the most desirable of these code motions, use heuristics to pick among them, though it is implied in the references that the general goal is to move operations up as high as possible.
3. Schedule, leaving operations in the blocks that step 2 placed them in.

Percolation scheduling backs off on the motivating concept behind trace scheduling. Code motions done in isolation from the scheduling phase can lead to the sort of arbitrary choices that completely scheduling each block in isolation will lead to: there is no way of knowing which block an operation should be placed in unless one knows what scheduling decisions are possible. But percolation scheduling does provide a catalog of code motions, useful for the implementer of a nonlinear scheduling algorithm.

An implementation of percolation scheduling at Carnegie-Mellon [Breternitz 91] goes beyond the original suggestions for percolation scheduling, and reinstates the phase-ordering idea present in trace scheduling. There, the catalog of code motions is referred to and updated as scheduling proceeds.

SRDAG Compaction [Linn 83]. This technique closely follows the spirit of trace scheduling, and is similar to the starting point for the methodology developed in this paper. SRDAG stands for "Singly-rooted directed acyclic graph". Here, the DAG does not refer to the data precedence graph, but rather the basic blocks from which operations are selected for code motions. At jumps, the algorithm permits consideration from both directions but, at rejoins, only one direction is allowed. Like trace scheduling, a data precedence DAG is built before any code motions are considered. Code is then scheduled from the DAG, and code motions are made as implied by the scheduling decisions. Rejoins are handled as in trace scheduling. I don't believe SRDAG compaction has ever been implemented.

A Program Dependence Graph Approach [Bernstein and Rodeh 91]. Concurrent with the implementation described below, a nonlinear implementation underway at the IBM Israel Scientific Center (and described elsewhere in this volume) uses the *Program Dependence Graph* approach [Ferrante, et al. 87] to represent the program. As such, that implementation offers an approach to dominator parallelism that is similar in effect to the one described below. In addition, speculative code motions are allowed, with the number of conditional jumps passed being used a heuristic, rather than speculative yield. This implementation has as its focus a superscalar architecture (the IBM System/6000), and is thus less aimed at environments in which a great quantity of ILP is desired. The ideas expressed in [Bernstein and Rodeh 91] could be extended to a broader architectural base with interesting results.

6. Trace Scheduling-2

Trace Scheduling-2, currently being implemented, is an attempt to broaden trace scheduling to nonlinear compaction. Trace scheduling-2 preserves, to the extent possible, the engineering advantages of trace scheduling, in particular:

- Decisions about scheduling, motions, code generation are all done at once, when practical.
- There are no artificial limitations on motions past rejoins.
- Trace scheduling fits into the customary model of compiling.

In addition to nonlinear code motions, trace scheduling-2 enhances trace scheduling by:

- Allowing dominator parallelism to occur without the generation of extra operations.
- Adding a speculative yield function (explained below) to guide trace picking and scheduling.

Both of these related features should have been in trace scheduling in the first place.

The trace scheduling-2 Algorithm

Trace scheduling-2 works as follows (this process is illustrated in Figure 6):

1. Assume a flow graph, and assume that conditional jumps have static probabilities attached to them.
2. Pick a *cluster* of operations, consisting of many basic blocks and the edges connecting them, not including backedges. A good candidate is any maximal set of operations that can be connected without backedges.
3. Compute the data necessary for scheduling: live-dead, data dependence, disambiguation, etc.
4. Generate the speculative yield function (explained below) for this cluster.
5. Topologically sort the blocks, start scheduling the first on the list.
6. Continue scheduling until the jump defining the end of the block is scheduled. Pick the next block on the sorted list and repeat.

Notice how similar to trace scheduling this is. Instead of picking a trace to generate the candidate set, we now have a cluster, which is a partially ordered acyclic graph (i.e. a DAG) of basic blocks instead of a totally ordered (i.e. linear) acyclic set of blocks. Otherwise it seems familiar; indeed, a trace probably should still picked as the order in which the blocks are first compacted. But there are very significant differences, including:

- The live-dead relationship changes as the operations are scheduled. We will see shortly why this happens.
- Operations which are ready to be scheduled can become unready during scheduling and must be treated accordingly.
- The priority functions used must take many more factors into account.
- It is more difficult to locate the appropriate place to place compensation code.

and many more. Figure 7 shows trace scheduling-2 applied to the example program in Figure 2. Note that in this example, original trace scheduling does slightly better than trace scheduling-2: A-C-D takes the same 9 cycles in both cases, but A-B-D takes 1 cycle longer. (Although this was a contrived example, it wasn't my intention to make this happen!) This might or might not happen often, and probably wouldn't have happened in this case had speculative yield been used as part of the priority heuristic. But it is a good illustration of a genuine difficulty in balancing scheduling priorities: one wants to start operations from both sides of a jump according to their height and speculative yield, but then operations from both sides must share the available hardware, ultimately slowing down the operations from whichever side the jump does take. Thus one might suggest executing jumps as quickly as possible, but those who've implemented trace scheduling compilers know that life is not that simple! If the jump is done quickly, many operations from above the jump will have to be copied out into both branches, causing too much extra code, poor instruction cache performance, and so on. This is representative of a difficult tradeoff faced in trace scheduling, nonlinear methods are sure to make it worse.

The Speculative Yield Function

We have seen that speculative operations are costly in wasted hardware resources and false exceptions. But the scheduler cannot simply ignore speculative operations, or even, as has been suggested, only schedule one when there is no nonspeculative operation to do. Sometimes an operation is critical, and extremely likely, though not certain, to be **profitable** (i. e. that flow of control will pass or has already passed to its block of origin). Thus it is important to give a scheduler the following tool:

> The **Speculative Yield Function** is defined on pairs of basic blocks. $SY(B_i , B_j) = p$ means that an operation originating in block B_i, but scheduled in block B_j, has probability p of being profitable.

Figure 6, panel 3 shows the speculative yield function for the flow graph in panel 1. Note that I will more commonly refer to the speculative yield of an operation, rather than the speculative yield function of B_i x B_j. By this I mean that B_i is the source block of the given operation, and B_j is the block currently being scheduled.

The speculative yield function is a component of the priority function used to select the operations to be scheduled. How it should be used is certainly implementation-dependent, but the basic idea behind it is that it is the "expected value" of doing an operation. In some sense, the usual priority function of DAG height, used in most schedulers, is also a somewhat linear value function on candidate operations, in that doing a high operation early frees up later operations. Thus multiplying DAG height by the speculative yield function to determine priority has some intuitive appeal. This same concept would have been useful in original trace scheduling, both as a scheduling priority component and to help guide trace picking.

Updating Scheduling Information on the Fly

A very big engineering change in moving from trace scheduling to a nonlinear compactor is that many quantities which had remained constant during scheduling now can change when any operation is scheduled. Thus aspects of the engineering task must now be mixed when they had been separated.

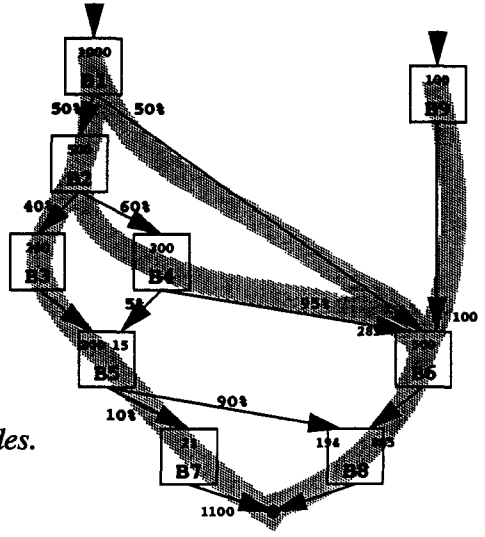
Live-Dead Analysis. In doing code motions, live dead analysis is required because operations that move between basic blocks must not move past a jump when they could kill a value live in the other direction. In trace scheduling, live-dead information is computed once for the trace, and then maintained during the scheduling of that trace. In trace scheduling-2, live-dead analysis is more complex: when a value moves up past a branch, it changes what is live on that branch. Operations that could have moved up from the other side (which wouldn't have occurred in linear trace scheduling) now might not be allowed to, or operations that couldn't before, now can. Thus scheduling must stop while an incremental live-dead analysis is done and the implications of it are assessed.

Changing Code Motion Paths. An operation may have several paths on which it could move from block to block. Whether an operation can move up along a given path is a dynamic question, which can change as each earlier operation is scheduled. The code generator must know, for several reasons, exactly which paths operations can move along. Thus this information must be maintained as scheduling progresses.

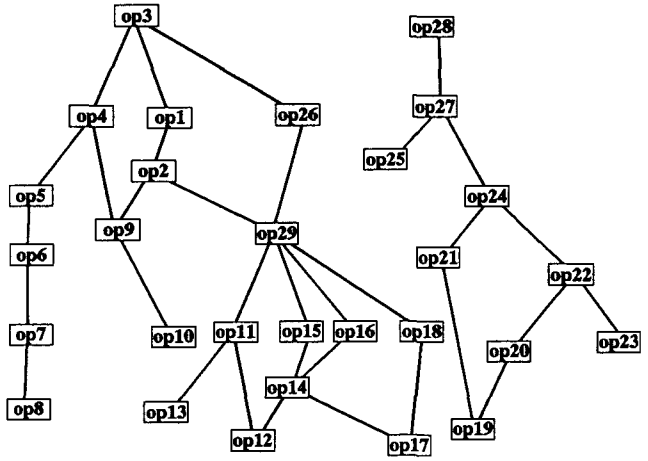
Adding Compensation Code. In trace scheduling, compensation code is added all at once after a trace is scheduled. Here, the compensation code itself is part of the cluster under consideration and must be added during scheduling. Finding where to add the compensation code, a relatively straightforward matter in trace scheduling, is considerably more difficult here.

1

Pick a large group of basic blocks, called a "cluster". Go past jumps in both directions, where desirable. Go past rejoins, possibly from both sides.



2 *Build one DAG from the entire cluster of operations:*



3 *Compute the "Speculative Yield" Function*

	B1	B2	B3	B4	B5	B6	B7	B8	B9
B1	1	1	1	1	1	1	1	1	0
B2	.5	1	1	1	1	1	1	1	0
B3	.2	.4	1	0	1	0	1	1	0
B4	.3	.6	0	1	1	1	1	1	0
B5	.215	.43	1	.05	1	0	1	1	0
B6	.785	.57	0	.95	0	1	0	1	1
B7	.021	.043	.1	.005	.1	0	1	0	0
B8	.979	.957	.9	.995	.9	1	0	1	1
B9	0	0	0	0	0	1	0	1	1

4 *Form schedules from the entire DAG. Use speculative yield to help make speculative placement decisions. Again, make corrections at each jump and rejoin.*

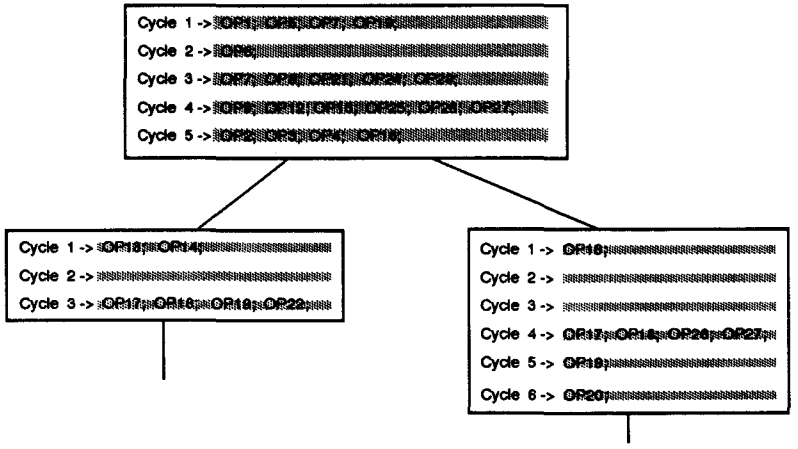


Figure 6: A summary of Trace Scheduling 2. As in Trace Scheduling, a large group of operations is considered at once. Now operations can move from anyplace in a cluster, but code generation is still done along with movement.

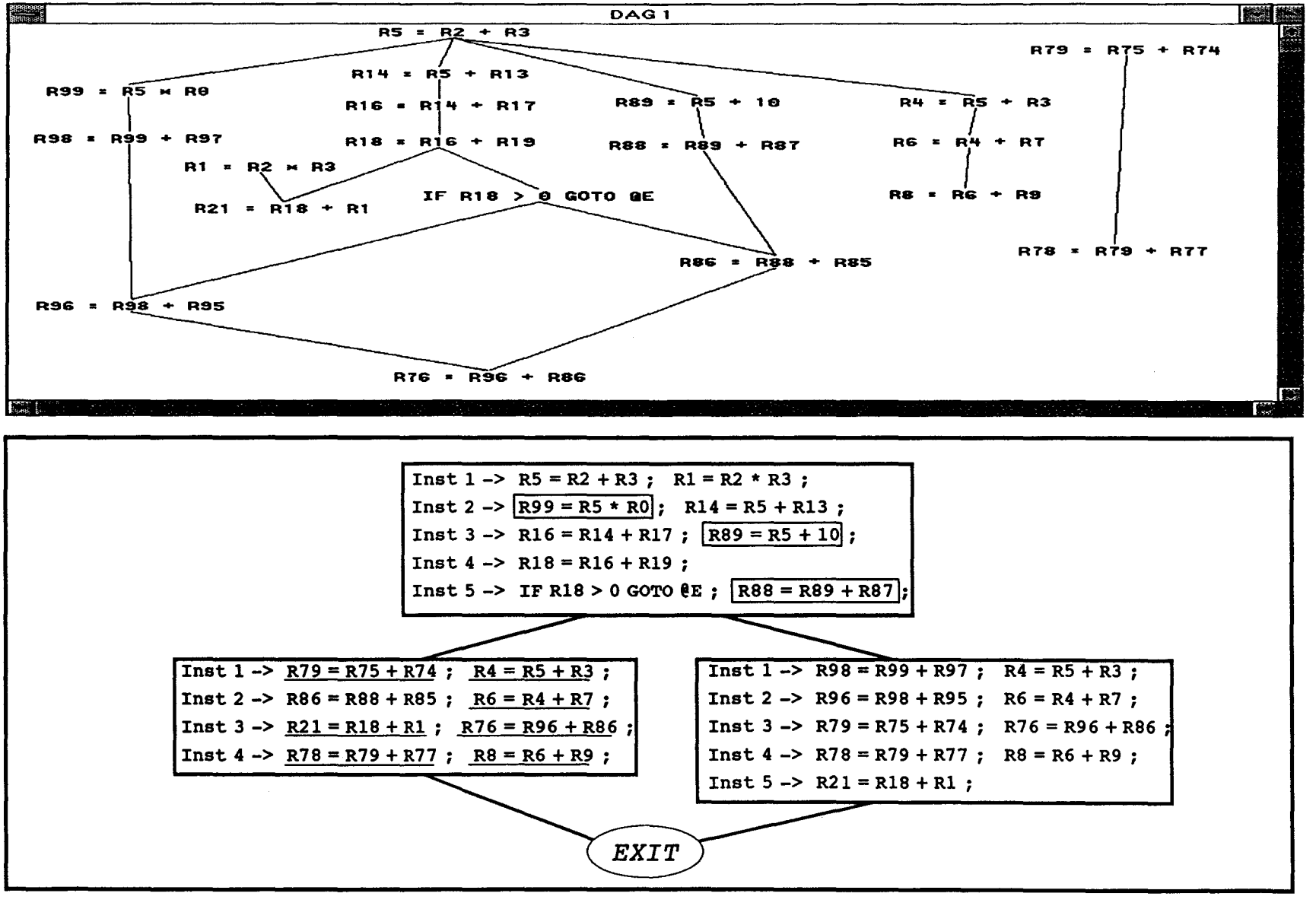


Figure 7: The DAG and schedule after carrying out trace scheduling-2. Boxed ops are speculative, the underlined ops on left are compensation copies from blocks A and D (if we arbitrarily consider the righthand copies to be the originals).

Changing Speculative Yield Values. As a side effect of the above, speculative yield values can change as each operation is scheduled, not just when a branch is passed.

An Implementation of Trace Scheduling-2

Part of the implementation of trace scheduling-2 underway (and which generated the DAGs and schedules above) adds dominator parallelism and a speculative yield function to trace scheduling, and compares the performance of:

1. Trace scheduling, vs.
2. Trace scheduling with dominator parallelism and a speculative yield function, vs.
3. Trace scheduling-2.

Are Nonlinear Scheduling Techniques Worth The Trouble? Although intuition says that the added advantage of nonlinearity will be great, there are arguments in the other direction:

- Jumps are very predictable, and tend to go in only one direction.
- If the hardware has support for select statements, in which simple if-then-else "hammocks" are converted to single statements, nonlinear control flow can often be linearized.
- If the hardware contains predicates, in which operations can be turned on or off dynamically according to a previously set condition, again nonlinear control flow can often be linearized.
- A good nonlinear scheduling algorithm may give back some of its advantage by slowing down one or the other branch sometimes, as in Figure 7.
- Finally, we have seen that trace scheduling-2 is potentially far more complex to engineer. Having code along a single linear path allows the compiler to use familiar code generation techniques and algorithms that divide into clean-cut phases to a far greater extent than seems possible with nonlinear code generation. A time-honored principle is that complex engineering has its cost: there will be capabilities added to the simpler system that won't go into the more complex one for simple lack of time. Of course, this trade-off could apply to all complex engineering, trace scheduling included.

The implementation underway is being done with as much attention as possible being paid to yielding a meaningful measurement of the performance differences among the alternatives.

Acknowledgments: Conversations with Bob Rau, Stefan Freudenberger, Rajiv Gupta and Vinod Kathail helped clarify many of the concepts in this paper.

8. References

[Anderson et al. 67]

D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, "The System/360 Model 91: Machine philosophy and instruction handling" *IBM Journal of Research and Development*, Vol. 11, pp. 8-24, January 1967.

[Bernstein and Rodeh 91]

D. Bernstein and Michael Rodeh, "Global instruction scheduling for superscalar machines", *Proceedings of the Conference on Programming Language and Design*, Toronto, Ontario, Canada, ACM Sigplan, June, 1991.

[Breternitz 91]

M. Breternitz, "Architecture synthesis of high-performance application-specific processors", Carnegie-Mellon University Research Report No. CMUCDS-91-5, April, 1991.

[Cmelik et al. 91]

R. Cmelik, S. Kong, D. Ditzel, E. Kelly, "An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems,* Santa Clara, California, ACM and IEEE Computer Society, April 1991.

[Colwell et al. 87]

R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler", *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 180-192, Palo Alto, California, ACM and IEEE Computer Society, October 1987.

[Ellis 85]

J. R. Ellis, *Bulldog: A Compiler For VLIW Architectures*, The MIT Press, Cambridge, MA, 1985.

[Ferrante, el al. 87]

J. Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems*, 9(3), pp. 319-349, July 1987.

[Fisher 81]

J. A. Fisher, "Trace scheduling: A technique for global microcode compaction", *IEEE Transactions on Computers*, 30(7), pp. 478-490, July 1981.

[Fisher 83]

J. A. Fisher, "Very long instruction word architectures and the ELI-512", *10th Annual International Symposium on Computer Architecture*, pp. 140-150, Stockholm, Sweden, ACM and IEEE Computer Society, June 1983.

[Fisher and Freudenberger 92]
J. A. Fisher and Stefan Freudenberger, Provisional title: "Predicting conditional jump directions from previous runs of a program," Hewlett-Packard Laboratories Technical Report, Palo Alto, California, to be published 1992.

[Fisher and Rau 91/92]
J. A. Fisher and B. Ramakrishna Rau, "Instruction-level parallelism", *Science,* 253(5025), pp. 1233-1242, September 1991. A slightly longer and more detailed version is: "Instruction-level parallelism," Hewlett-Packard Laboratories Technical Report, Palo Alto, CA, HPL-92-02, January 1992.

[Johnson 91]
W. Johnson, *Superscalar Microprocessor Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Lam 88]
M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines", *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pp. 318-327, June 1988.

[Linn 83]
J. L. Linn, "SRDAG compaction--a generalization of trace scheduling to increase the use of global context information", *Proceedings of the 16th Annual Workshop on Microprogramming*, pp. 11-22, ACM and IEEE Computer Society, October 1983.

[[Nicolau 85]
A. Nicolau, "Uniform parallelism exploitation in ordinary programs", *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 614-618, ACM and IEEE Computer Society, August 1985.

[Nicolau and Fisher 81]
A. Nicolau and J. A. Fisher, "Using an oracle to measure parallelism in single instruction stream programs", *Proceedings of the 14th Annual Microprogramming Workshop*, pp. 171-182, ACM and IEEE Computer Society, October 1981.

[Rau and Glaeser 81]
B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", *Proceedings of the 14th Annual Workshop on Microprogramming*, pp. 183-198, ACM and IEEE Computer Society, October 1981.

[Riseman and Foster 72]
E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps", *IEEE Transactions on Computers*, 21(12), pp. 1405-1411, December 1972.

[Su, Ding and Jin 84]
B. Su, S. Ding and L. Jin, "An improvement of trace scheduling for global microcode compaction", *Proceedings of the 17th Annual Workshop on Microprogramming*, pp. 78-85, ACM and IEEE Computer Society, October 1984.

[Thornton 64]
J. E. Thornton, "Parallel operations in the Control Data 6600", *Proceedings of the AFIPS Conference*, Vol 26, pp. 33-40, 1964.

[Tokoro et al. 78]
M. Tokoro, T. Takizuka, E. Tamura and I. Yamaura, "A technique of global optimization of microprograms", *Proceedings of the 11th Annual Workshop on Microprogramming*, pp. 41-50, ACM and IEEE Computer Society, November 1978.

[Wall 91]
D. W. Wall, "Limits of instruction-level parallelism", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, ACM, April 1991.