Limits of Instruction-Level Parallelism

David W. Wall

Digital Equipment Corporation
Western Research Laboratory

Abstract

Growing interest in ambitious multiple-issue machines and heavily-pipelined machines requires a careful examination of how much instruction-level parallelism exists in typical programs. Such an examination is complicated by the wide variety of hardware and software techniques for increasing the parallelism that can be exploited, including branch prediction, register renaming, and alias analysis. By performing simulations based on instruction traces, we can model techniques at the limits of feasibility and even beyond. Our study shows a striking difference between assuming that the techniques we use are perfect and merely assuming that they are impossibly good. Even with impossibly good techniques, average parallelism rarely exceeds 7, with 5 more common.

1. Introduction

There is growing interest in machines that exploit, usually with compiler assistance, the parallelism that programs have at the instruction level. Figure 1 shows an example of this parallelism.

r1 := 0[r9]	r1 := 0[r9]
r2 := 17	r2 := r1 + 17
4[r3] := r6	4[r2] := r6
(a) parallelism=3	*(b) parallelism=1*

Figure 1. Instruction-level parallelism (and lack thereof).

The code fragment in 1(a) consists of three instructions that can be executed at the same time, because they do not depend on each other's results. The code fragment in 1(b) does have dependencies, and so cannot be executed in parallel. In each case, the parallelism is the number of instructions divided by the number of cycles required.

Architectures to take advantage of this kind of parallelism have been proposed. A superscalar machine [1] is one that can issue multiple independent instructions in the same cycle. A superpipelined machine [7] issues one instruction per cycle, but the cycle time is set much less than the typical instruction latency. A VLIW machine [11] is like a superscalar machine, except the parallel instructions must be explicitly packed by the compiler into very long instruction words.

But how much parallelism is there to exploit? Popular wisdom, supported by a few studies [7,13,14], suggests that parallelism within a basic block rarely exceeds 3 or 4 on the average. Peak parallelism can be higher, especially for some kinds of numeric programs, but the payoff of high peak parallelism is low if the average is still small.

These limits are troublesome. Many machines already have some degree of pipelining, as reflected in operations with latencies of multiple cycles. We can compute the degree of pipelining by multiplying the latency of each operation by its dynamic frequency in typical programs; for the DECStation 5000,* load latencies, delayed branches, and floating-point latencies give the machine a degree of pipelining equal to about 1.5. Adding a superscalar capability to a machine with some pipelining is beneficial only if there is more parallelism available than the pipelining already exploits.

To increase the instruction-level parallelism that the hardware can exploit, people have explored a variety of techniques. These fall roughly into two categories. One category includes techniques for increasing the parallelism within a basic block, the other for using parallelism across several basic blocks. These techniques often interact in a way that has not been adequately explored. We would like to bound the effectiveness of a

* DECStation is a trademark of Digital Equipment Corporation.

technique whether it is used in combination with impossibly good companion techniques, or with none. A general approach is therefore needed. In this paper, we will describe our use of trace-driven simulation to study the importance of register renaming, branch and jump prediction, and alias analysis. In each case we can model a range of possibilities from perfect to non-existent.

We will begin with a survey of ambitious techniques for increasing the exploitable instruction-level parallelism of programs.

1.1. Increasing parallelism within blocks.

Parallelism within a basic block is limited by dependencies between pairs of instructions. Some of these dependencies are real, reflecting the flow of data in the program. Others are false dependencies, accidents of the code generation or results of our lack of precise knowledge about the flow of data.

Allocating registers assuming a traditional scalar architecture can lead to a false dependency on a register. In the code sequence

```
r1 := 0[r9]
r2 := r1 + 1
r1 := 9
```

we must do the second and third instructions in that order, because the third changes the value of r1. However, if the compiler had used r3 instead of r1 in the third instruction, these two instructions would be independent.

A smart compiler might pay attention to its allocation of registers, so as to maximize the opportunities for parallelism. Current compilers often do not, preferring instead to reuse registers as often as possible so that the number of registers needed is minimized.

An alternative is the hardware solution of *register renaming*, in which the hardware imposes a level of indirection between the register number appearing in the instruction and the actual register used. Each time an instruction sets a register, the hardware selects an actual register to use for as long as that value is needed. In a sense the hardware does the register allocation dynamically, which can give better results than the compiler's static allocation, even if the compiler did it as well as it could. In addition, register renaming allows the hardware to include more registers than will fit in the instruction format, further reducing false dependencies. Unfortunately, register renaming can also lengthen the machine pipeline, thereby increasing the branch penalties of the machine, but we are concerned here only with its effects on parallelism.

False dependencies can also involve memory. We assume that memory locations have meaning to the programmer that registers do not, and hence that hardware renaming of memory locations is not desirable. However, we may still have to make conservative assumptions that lead to false dependencies on memory. For example, in the code sequence

```
r1 := 0[r9]
4[r16] := r3
```

we may have no way of knowing whether the memory locations referenced in the load and store are the same. If they are the same, then there is a dependency between these two instructions: we cannot store a new value until we have fetched the old. If they are different, there is no dependency. *Alias analysis* can help a compiler decide when two memory references are independent, but even that is imprecise; sometimes we must assume the worst. Hardware can resolve the question at run-time by determining the actual addresses referenced, but this may be too late to affect parallelism decisions. If the compiler or hardware are not sure the locations are different, we must assume conservatively that they are the same.

1.2. Crossing block boundaries.

The number of instructions between branches is usually quite small, often averaging less than 6. If we want large parallelism, we must be able to issue instructions from different basic blocks in parallel. But this means we must know in advance whether a conditional branch will be taken, or else we must cope with the possibility that we do not know.

Branch prediction is a common hardware technique. In the scheme we used [9,12], the branch predictor maintains a table of two-bit entries. Low-order bits of a branch's address provide the index into this table. Taking a branch causes us to increment its table entry; not taking it causes us to decrement. We do not wrap around when the table entry reaches its maximum or minimum. We predict that a branch will be taken if its table entry is 2 or 3. This two-bit prediction scheme mispredicts a typical loop only once, when it is exited. Two branches that map to the same table entry interfere with each other; no "key" identifies the owner of the entry. A good initial value for table entries is 2, just barely predicting that each branch will be taken.

Branch prediction is often used to keep a pipeline full: we fetch and decode instructions after a branch while we are executing the branch and the instructions before it. To use branch prediction to increase parallel execution, we must be able to execute instructions across an unknown branch *speculatively*. This may involve maintaining shadow registers, whose values are not committed until we are sure we have correctly predicted the branch. It may involve being selective about the instructions we choose: we may not be willing to execute memory stores speculatively, for example. Some of this may be put partly under compiler control by designing an instruction set with explicitly squashable instructions.

Each squashable instruction would be tied explicitly to a condition evaluated in another instruction, and would be squashed by the hardware if the condition turns out to be false.

Rather than try to predict the destinations of branches, we might speculatively execute instructions along *both* possible paths, squashing the wrong path when we know which it is. Some of our parallelism capability is guaranteed to be wasted, but we will never miss out by taking the wrong path. Unfortunately, branches happen quite often in normal code, so for large degrees of parallelism we may encounter another branch before we have resolved the previous one. A real architecture may have limits on the amount of fanout we can tolerate before we must assume that new branches are not explored in parallel.

Many architectures have two or three kinds of instructions to change the flow of control. Branches are conditional and have a destination some specified offset from the PC. Jumps are unconditional, and may be either *direct* or *indirect*. A direct jump is one whose destination is given explicitly in the instruction, while an indirect jump is one whose destination is expressed as an address computation involving a register. In principle we can know the destination of a direct jump well in advance. The same is true of a branch, assuming we know how its condition will turn out. The destination of an indirect jump, however, may require us to wait until the address computation is possible. Little work has been done on predicting the destinations of indirect jumps, but it might pay off in instruction-level parallelism. This paper considers a very simple (and, it turns out, fairly accurate) jump prediction scheme. A table is maintained of destination addresses. The address of a jump provides the index into this table. Whenever we execute an indirect jump, we put its address in the table entry for the jump. We predict that an indirect jump will be to the address in its table entry. As with branch prediction, we do not prevent two jumps from mapping to the same table entry and interfering with each other.

Loop unrolling is an old compiler optimization technique that can also increase parallelism. If we unroll a loop ten times, thereby removing 90% of its branches, we effectively increase the basic block size tenfold. This larger basic block may hold parallelism that had been unavailable because of the branches.

Software pipelining [8] is a compiler technique for moving instructions across branches to increase parallelism. It analyzes the dependencies in a loop body, looking for ways to increase parallelism by moving instructions from one iteration into a previous or later iteration. Thus the dependencies in one iteration can be stretched out across several, effectively executing several iterations in parallel without the code expansion of unrolling.

Trace scheduling [4] was developed for VLIW machines, where global scheduling by the compiler is needed to exploit the parallelism of the long instruction words. It uses a profile to find a *trace* (a sequence of blocks that are executed often), and schedules the instructions for these blocks as a whole. In effect, trace scheduling predicts a branch statically, based on the profile. To cope with occasions when this prediction fails, code is inserted outside the sequence of blocks to correct the state of registers and memory whenever we enter or leave the sequence unexpectedly. This added code may itself be scheduled as part of a later and less heavily executed trace.

2. This and previous work.

To better understand this bewildering array of techniques, we have built a simple system for scheduling instructions produced by an instruction trace. Our system allows us to assume various kinds of branch and jump prediction, alias analysis, and register renaming. In each case the option ranges from perfect, which could not be implemented in reality, to non-existent. It is important to consider the full range in order to bound the effectiveness of the various techniques. For example, it is useful to ask how well a realistic branch prediction scheme could work even with impossibly good alias analysis and register renaming.

This is in contrast to the 1989 study of Jouppi and Wall [7], which worked by scheduling static program executables rather than dynamic instruction traces. Since their compiler did scheduling only within basic blocks, they did not consider more ambitious scheduling.

The methodology of our paper is more like that of Tjaden and Flynn [14], which also scheduled instructions from a dynamic trace. Like Jouppi and Wall, however, Tjaden and Flynn did not move instructions across branches. Their results were similar to those of Jouppi and Wall, with parallelism rarely above 3, even though the two studies assumed quite different architectures.

Nicolau and Fisher's trace-driven study [11] of the effectiveness of trace scheduling was more liberal, assuming perfect branch prediction and perfect alias analysis. However, they did not consider more realistic assumptions, arguing instead that they were interested primarily in programs for which realistic implementations would be close to perfect.

The study by Smith, Johnson, and Horowitz [13] was a realistic application of trace-driven simulation that assumed neither too restrictive nor too generous a model. They were interested, however, in validating a particular realistic machine design, one that could consistently exploit a parallelism of only 2. They did not explore the range of techniques discussed in this paper.

We believe our study can provide useful bounds on the behavior not only of hardware techniques like branch prediction and register renaming, but also of compiler techniques like software pipelining and trace scheduling.

Unfortunately, we could think of no good way to model loop unrolling. Register renumbering can cause much of the computation in a loop to migrate backward toward the beginning of the loop, providing opportunities for parallelism much like those presented by unrolling. Much of the computation, however, like the repeated incrementing of the loop index, is inherently sequential. We address loop unrolling in an admittedly unsatisfying manner, by unrolling the loops of some numerical programs by hand and comparing the results to those of the normal versions.

3. Our experimental framework.

To explore the parallelism available in a particular program, we execute the program to produce a trace of the instructions executed. This trace also includes data addresses referenced, and the results of branches and jumps. A greedy algorithm packs these instructions into a sequence of pending cycles.

In packing instructions into cycles, we assume that any cycle may contain as many as 64 instructions in parallel. We further assume no limits on replicated functional units or ports to registers or memory: all 64 instructions may be multiplies, or even loads. We assume that every operation has a latency of one cycle, so the result of an operation executed in cycle N can be used by an instruction executed in cycle N+1. This includes memory references: we assume there are no cache misses.

We pack the instructions from the trace into cycles as follows. For each instruction in the trace, we start at the end of the cycle sequence, representing the latest pending cycle, and move earlier in the sequence until we find a *conflict* with the new instruction. Whether a conflict exists depends on which model we are considering. If the conflict is a false dependency (in models allowing them), we assume that we can put the instruction in that cycle but no farther back. Otherwise we assume only that we can put the instruction in the next cycle after this one. If the correct cycle is full, we put the instruction in the next non-full cycle. If we cannot put the instruction in any pending cycle, we start a new pending cycle at the end of the sequence.

As we add more and more cycles, the sequence gets longer. We assume that hardware and software techniques will have some limit on how many instructions they will consider at once. When the total number of instructions in the sequence of pending cycles reaches this limit, we remove the first cycle from the sequence, whether it is full of instructions or not. This corresponds to retiring the cycle's instructions from the scheduler, and passing them on to be executed.

When we have exhausted the trace, we divide the number of instructions by the number of cycles we created. The result is the parallelism.

3.1. Parameters.

We can do three kinds of *register renaming:* perfect, finite, and none. For perfect renaming, we assume that there are an infinite number of registers, so that no false register dependencies occur. For finite renaming, we assume a finite register set dynamically allocated using an LRU discipline: when we need a new register we select the register whose most recent use (measured in cycles rather than in instruction count) is earliest. Finite renaming is normally done with 256 integer registers and 256 floating-point registers. It is also interesting to see what happens when we reduce this to 64 or even 32, the number on our base machine. To simulate no renaming, we simply use the registers specified in the code; this is of course highly dependent on the register strategy of the compiler we use.

We can assume several degrees of *branch prediction.* One extreme is perfect prediction: we assume that all branches are correctly predicted. Next we can assume a two-bit prediction scheme as described before. The two-bit scheme can be either infinite, with a table big enough that two different branches never have the same table entry, or finite, with a table of 2048 entries. To model trace scheduling, we can also assume static branch prediction based on a profile from an identical run; in this case we predict that a branch will always go the way that it goes most frequently. And finally, we can assume that no branch prediction occurs; this is the same as assuming that every branch is predicted wrong.

The same choices are available for *jump prediction.* We can assume that indirect jumps are perfectly predicted. We can assume infinite or finite hardware prediction as described above (predicting that a jump will go where it went last time). We can assume static prediction based on a profile. And we can assume no prediction. In any case we are concerned only with indirect jumps; we assume that direct jumps are always predicted correctly.

The effect of branch and jump prediction on scheduling is easy to state. Correctly predicted branches and jumps have no effect on scheduling (except for register dependencies involving their operands). Instructions on opposite sides of an incorrectly predicted branch or jump, however, always conflict. Another way to think of this is that the sequence of pending cycles is flushed whenever an incorrect prediction is made. Note that we generally assume no other penalty for failure. This assumption is optimistic; in most real architectures, a

failed prediction causes a bubble in the pipeline, resulting in one or more cycles in which no execution whatsoever can occur. We will return to this topic later.

We can also allow instructions to move past a certain number of incorrectly predicted branches. This corresponds to architectures that speculatively execute instructions from both possible paths, up to a certain *fanout limit*. None of the experiments described here involved this ability.

Four levels of *alias analysis* are available. We can assume perfect alias analysis, in which we look at the actual memory address referenced by a load or store; a store conflicts with a load or store only if they access the same location. We can also assume no alias analysis, so that a store always conflicts with a load or store. Between these two extremes would be alias analysis as a smart vectorizing compiler might do it. We don't have such a compiler, but we have implemented two intermediate schemes that may give us some insight.

One intermediate scheme is *alias by instruction inspection*. This is a common technique in compile-time instruction-level code schedulers. We look at the two instructions to see if it is obvious that they are independent; the two ways this might happen are shown in Figure 2.

```
r1 := 0[r9]     r1 := 0[fp]
4[r9] := r2     0[gp] := r2
   (a)             (b)
```

Figure 2. Alias analysis by inspection.

The two instructions in 2(a) cannot conflict, because they use the same base register but different displacements. The two instructions in 2(b) cannot conflict, because one is manifestly a reference to the stack and the other is manifestly a reference to the global data area.

The other intermediate scheme is called *alias analysis by compiler* even though our own compiler doesn't do it. Under this model, we assume perfect analysis of stack and global references, regardless of which registers are used to make them. A store to an address on the stack conflicts only with a load or store to the same address. Heap references, on the other hand, are resolved by instruction inspection.

The idea behind our alias analysis by compiler is that references outside the heap can often be resolved by the compiler, by doing dataflow analysis and possibly by solving diophantine equations over loop indexes, whereas heap references are often less tractable. Neither of these assumptions is particularly defensible. Many languages allow pointers into the stack and global areas, rendering them as difficult as the heap. Practical considerations such as separate compilation may also keep us from analyzing non-heap references perfectly. On the other side, even heap references are not as hopeless as this model assumes [2,6]. Nevertheless, our range of four alternatives provides some intuition about the effects of alias analysis on instruction-level parallelism.

The *window size* is the maximum number of instructions that can appear in the pending cycles at any time. By default this is 2048 instructions. We can manage the window either *discretely* or *continuously*. With discrete windows, we fetch an entire window of instructions, schedule them into cycles, and then start fresh with a new window. A missed prediction also causes us to start over with a full-size new window. With continuous windows, new instructions enter the window one at a time, and old cycles leave the window whenever the number of instructions reaches the window size. Continuous windows are the norm for the results described here, although to implement them in hardware is more difficult. Smith, Johnson, and Horowitz [13] assumed discrete windows.

	lines	*dynamic instructions*	*remarks*
Livermore	268	22294030	Livermore loops 1-14
Whetstones	462	24479634	Floating-point
Linpack	814	174883597	Linear algebra [3]
Stanford	1019	20759516	Hennessy's suite [5]
sed	1751	1447717	Stream editor
egrep	844	13910586	File search
yacc	1856	30948883	Compiler-compiler
metronome	4287	70235508	Timing verifier
grr	5883	142980475	PCB router
eco	2721	26702439	Recursive tree comparison
ccom	10142	18465797	C compiler front end
gcc1	83000	22745232	pass 1 of GNU C compiler
espresso	12000	135317102	boolean function minimizer
li	7000	1247190509	Lisp interpreter
fpppp	2600	244124171	quantum chemistry
doduc	5200	284697827	hydrocode simulation
tomcatv	180	1986257545	mesh generation

Figure 3. The seventeen test programs.

3.2. Programs measured.

As test cases we used four toy benchmarks, seven real programs used at WRL, and six SPEC benchmarks. These programs are shown in Figure 3. The SPEC benchmarks were run on accompanying test data, but the data was usually an official "short" data set rather than the reference data set. The programs were compiled for a DECStation 5000, which has a MIPS R3000* processor. The Mips version 1.31 compilers were used.

* R3000 is a trademark of MIPS Computer Systems, Inc.

4. Results.

We ran these test programs for a wide range of configurations. The results we have are tabulated in the appendix, but we will extract some of them to show some interesting trends. To provide a framework for our exploration, we defined a series of five increasingly ambitious models spanning the possible range. These five are specified in Figure 4; the window size in each is 2K instructions. Many of the results we present will show the effects of variations on these standard models. Note that even the Fair model is quite ambitious.

	branch predict	*jump predict*	*reg renaming*	*alias analysis*
Stupid	none	none	none	none
Fair	infinite	infinite	256	inspection
Good	infinite	infinite	256	perfect
Great	infinite	infinite	perfect	perfect
Perfect	perfect	perfect	perfect	perfect

Figure 4. Five increasingly ambitious models.

	branches			*jumps*		
	infinite	*finite*	*static*	*infinite*	*finite*	*static*
Linpack	96%	96%	95%	99%	99%	96%
Livermore	98%	98%	98%	19%	19%	77%
Stanford	90%	90%	89%	69%	69%	71%
Whetstones	90%	90%	92%	80%	80%	88%
sed	97%	97%	97%	96%	96%	97%
egrep	90%	90%	91%	98%	98%	98%
yacc	95%	95%	92%	75%	75%	71%
met	92%	92%	92%	77%	77%	65%
grr	85%	84%	82%	67%	66%	64%
eco	92%	92%	91%	47%	47%	56%
ccom	90%	90%	90%	55%	54%	64%
li	90%	90%	90%	56%	55%	70%
tomcatv	99%	99%	99%	58%	58%	72%
doduc	95%	95%	95%	39%	39%	62%
espresso	89%	89%	87%	65%	65%	53%
fpppp	92%	91%	88%	84%	84%	80%
gcc	90%	89%	90%	55%	54%	60%
mean	92%	92%	92%	67%	67%	73%

Figure 5. Success rates of branch and jump prediction.

4.1. Branch and jump prediction.

The success of the two-bit branch prediction has been reported elsewhere [9,10]. Our results were comparable and are shown in Figure 5. It makes little difference whether we use an infinite table or one with only 2K entries, even though several of the programs are more than twenty thousand instructions long. Static branch prediction based on a profile does almost exactly as well as hardware prediction across all of the tests; static jump prediction does a bit better than our simple dynamic prediction scheme. It would be interesting to explore how small the hardware table can be before performance starts to degrade, and to explore how well static prediction does if the profile is from a non-identical run of the program.

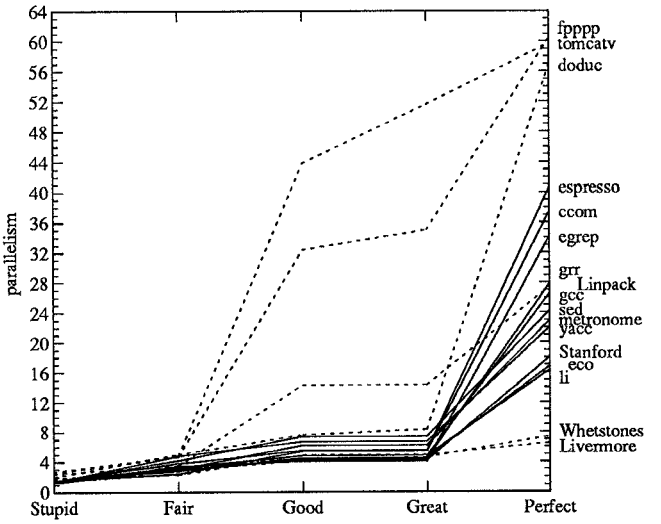

Figure 6. Parallelism under the five models.

4.2. Parallelism under the five models.

Figure 6 shows the parallelism of each program for each of the five models. The numeric programs are shown as dotted lines. Unsurprisingly, the Stupid model rarely gets above 2; the lack of branch prediction means that it finds only intra-block parallelism, and the lack of renaming and alias analysis means it won't find much of that. The Fair model is better, with parallelism between 2 and 4 common. Even the Great model, however, rarely has parallelism above 8. A study that assumed perfect branch prediction, perfect alias analysis, and perfect register renaming would lead us down a dangerous garden path. So would a study that included only fpppp and tomcatv, unless that's really all we want to run on our machine.

It is interesting that Whetstones and Livermore, two numeric benchmarks, do poorly even under the Perfect model. This is the result of Amdahl's Law: if we compute the parallelism for each Livermore loop independently, the values range from 2.4 to 29.9, with a median around 5. Speeding up a few loops 30-fold simply means that the cycles needed for less parallel loops will dominate the total.

4.3. Effects of unrolling.

Loop unrolling should have some effect on the five models. We explored this by unrolling two benchmarks by hand. The normal Linpack benchmark is already unrolled four times, so we made a version of it unrolled

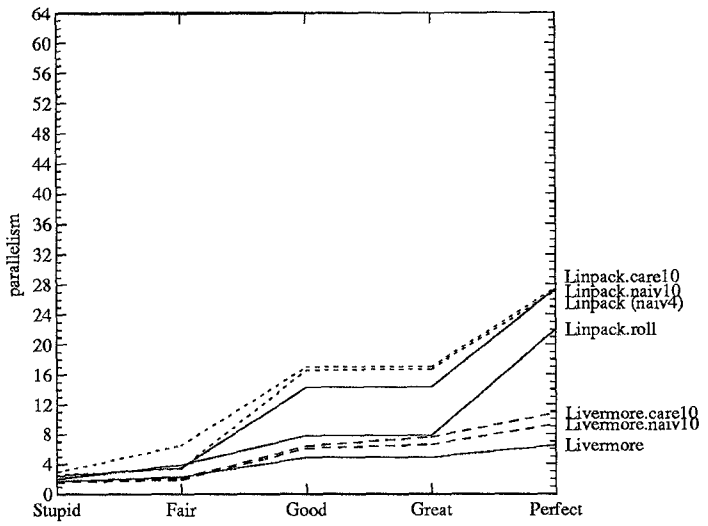

Figure 7. Unrolling under the five models.

ten times, and a version in which we rolled the loops back up, removing the normal unrolling by four. We also unrolled the Livermore benchmark ten times. We did the unrolling in two different ways. One is *naive* unrolling, in which the loop body is simply replicated ten times with suitable adjustments to array indexes and so on. The other is *careful* unrolling, in which computations involving accumulators (like scalar product) are reassociated to increase parallelism, and in which assignments to array members are delayed until after all the calculations are complete, so that false memory conflicts do not interfere with doing the calculations in parallel.

Figure 7 shows the result. The dotted lines are the 10-unrolled versions. Unrolling helps both Livermore and Linpack in the more ambitious models, although unrolling Linpack by 10 doesn't make it much more parallel than unrolling by 4. The difference between unrolling by 4 or 10 disappears altogether in the Perfect model, because the *saxpy* routine, which accounts for 75% of the executed instructions, achieves a parallelism just short of our maximum of 64 in each case. The aggregate parallelism stays lower because the next most frequently executed code is the loop in the *matgen* routine. This loop includes an embedded random-number generator, and each iteration is thus very dependent on its predecessor. This confirms the importance of using whole program traces; studies that considered only the parallelism in *saxpy* would be quite misleading.

Naive unrolling actually hurts the parallelism slightly under the Fair model. The reason is fairly simple. The Fair model uses alias analysis by inspection, which is not always sufficient to resolve the conflict between a store at the end of one iteration and the loads at the beginning of the next. In naive unrolling, the loop body is simply replicated, and these memory conflicts impose the same rigid framework to the dependency structure as they did before unrolling. The unrolled versions have slightly less to do within that framework, however, because 3/4 or 9/10 of the loop overhead has been removed. As a result, the parallelism goes down slightly. Even when alias analysis by inspection is adequate, unrolling the loops either naively or carefully sometimes causes the compiler to spill some registers. This is even harder for the alias analysis to deal with because these references usually have a different base register than the array references.

Loop unrolling is a good way to increase the available parallelism, but it is clear we must integrate the unrolling with the rest of our techniques better than we have been able to do here.

4.4. Effects of window size.

Our standard models all have a window size of 2K instructions: the scheduler is allowed to keep that many instructions in pending cycles at one time. Typical superscalar hardware is unlikely to handle windows of that size, but software techniques like trace scheduling for a VLIW machine might. Figure 8 shows the effect of varying the window size from 2K instructions down to 4. Under the Great model, which does not have perfect branch prediction, most programs do as well with a 32-instruction window as with a larger one. Below that, parallelism drops off quickly. Unsurprisingly, the Perfect model does better the bigger its window. The Good model is not shown, but looks almost identical to the Great model.

4.5. Effects of using discrete windows.

A less ambitious parallelism manager would get a window full of instructions, schedule them relative to each other, execute them, and then start over with a fresh window. This would tend to have less parallelism than the continuous window model we used above. Figure 9 shows the same models as Figure 8, except assuming discrete windows rather than continuous. Under the Great model, discrete windows do nearly as well as continuous when the window is 2K instructions, but the difference increases as the window size decreases; we must use discrete windows of 128 instructions before the curves level off. If we have very small windows, it might pay off to manage them continuously; in other words, continuous management of a small window is as good as multiplying the window size by 4. As before, the Perfect model does better the larger the window, but the parallelism is only two-thirds that of continuous windows.

4.6. Effects of branch and jump prediction.

We have several levels of branch and jump prediction. Figure 10 shows the results of varying these while register renaming and alias analysis stay perfect. Reduc-

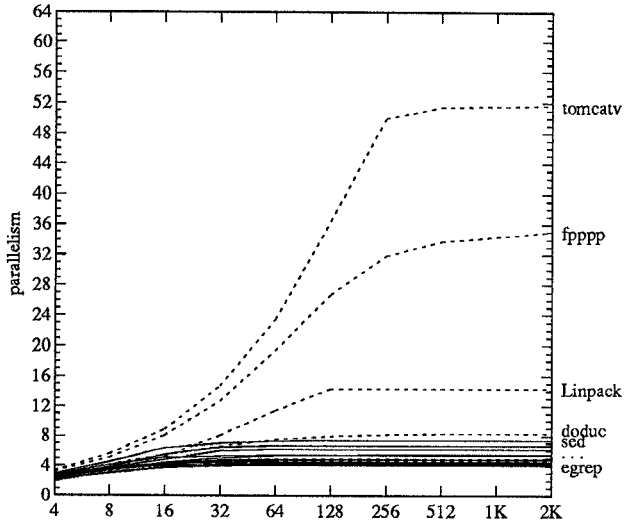

Figure 8(a). Size of continuous windows under Great model.

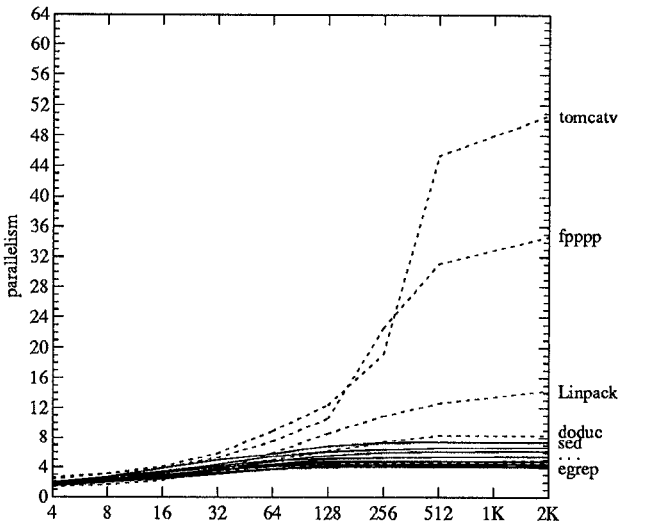

Figure 9(a). Size of discrete windows under Great model.

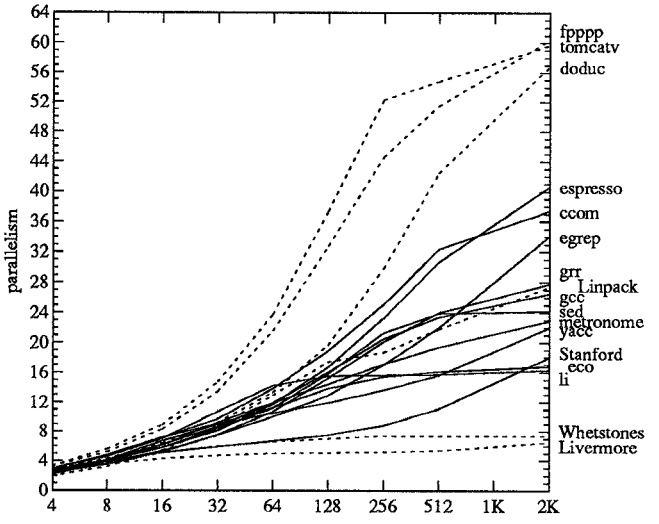

Figure 8(b). Size of continuous windows under Perfect model.

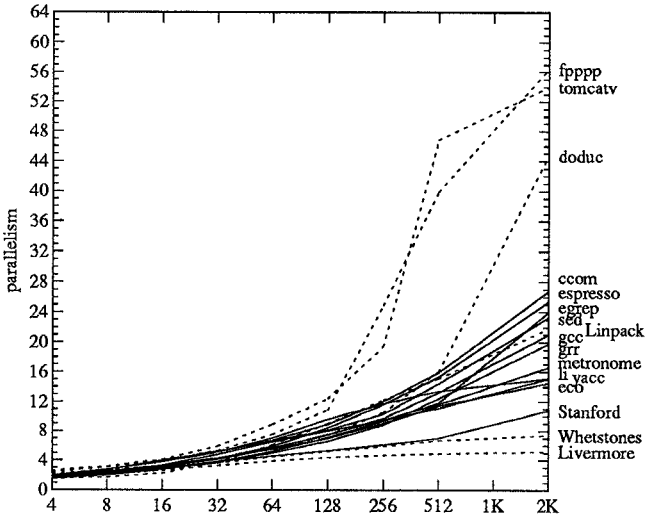

Figure 9(b). Size of discrete windows under Perfect model.

ing the level of jump prediction can have a large effect, but only if we have perfect branch prediction. Otherwise, removing jump prediction altogether has little effect. This graph does not show static or finite prediction: it turns out to make little difference whether prediction is infinite, finite, or static, because they have nearly the same success rate.

That jump prediction has little effect on the parallelism under non-Perfect models does not mean that jump prediction is useless. In a real machine, a jump predicted incorrectly (or not at all) may result in a bubble in the pipeline. The bubble is a series of cycles in which no execution occurs, while the unexpected instructions are fetched, decoded, and started down the execution pipeline. Depending on the penalty, this may have a serious effect on performance. Figure 11 shows the degradation of parallelism under the Great model, assuming that each mispredicted branch or jump adds N cycles with no instructions in them. If we assume instead that *all* indirect jumps (but not all branches) are mispredicted, the right ends of these curves drop by 10% to 30%.

Livermore and Linpack stay relatively horizontal over the entire range: They make fewer branches and jumps, and their branches are comparatively predictable. Tomcatv and fpppp are above the range of this graph, but their curves have slopes about the same as these.

4.7. Effects of alias analysis and register renaming.

We also ran several experiments varying alias analysis in isolation, and varying register renaming in isolation.

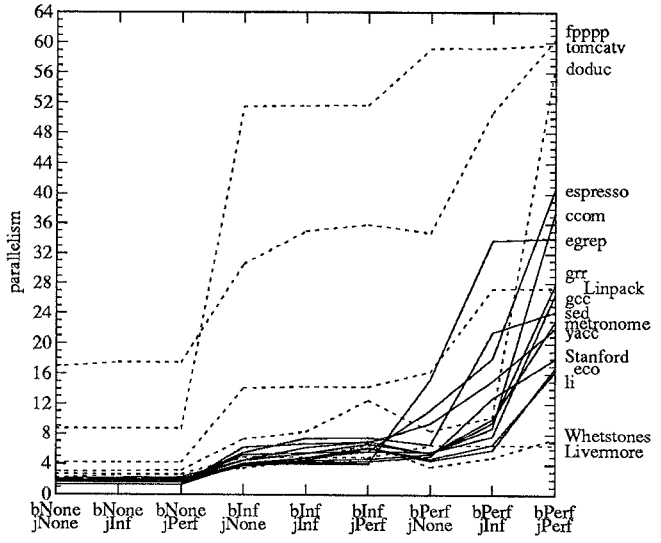

Figure 10. Effect of branch and jump prediction with perfect alias analysis and register renaming

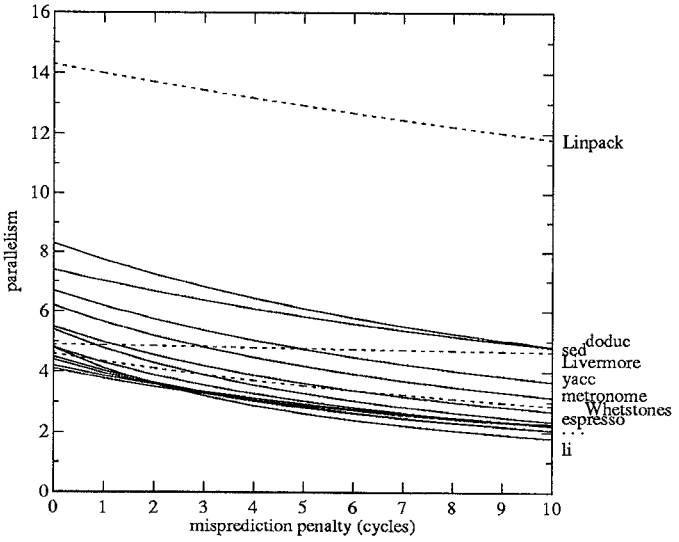

Figure 11. Effect of a misprediction cycle penalty on the Great model.

Figure 12 shows the effect of varying the alias analysis under the Perfect and Great models. We can see that "alias analysis by inspection" isn't very powerful; it rarely increased parallelism by more than 0.5. "Alias analysis by compiler" was (by definition) indistinguishable from perfect alias analysis on programs that do not use the heap, and was somewhat helpful even on those that do. There remains a gap between this analysis and perfection, which suggests that the payoff of further work on heap disambiguation may be significant. Unless branch prediction is perfect, however, even perfect alias analysis usually leaves us with parallelism between 4 and 8.

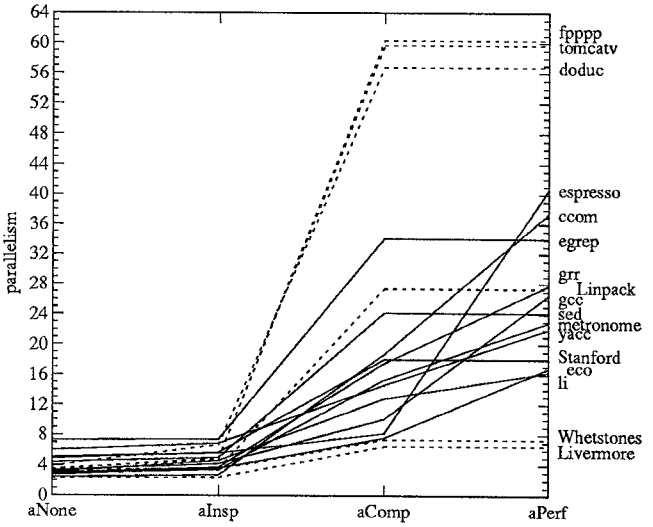

Figure 12(a). Effect of alias analysis on Perfect model.

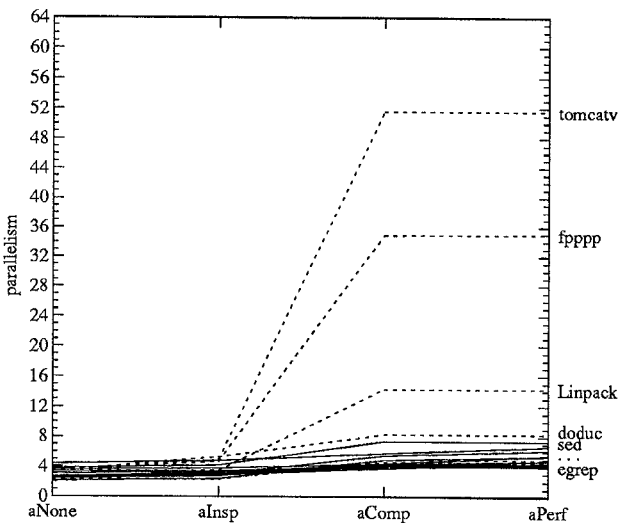

Figure 12(b). Effect of alias analysis on Great model.

Figure 13 shows the effect of varying the register renaming under the Perfect and Great models. Dropping from infinitely many registers to 256 CPU and 256 FPU registers rarely had a large effect unless the other parameters were perfect. Under the Great model, register renaming with 32 registers, the number on the actual machine, yielded parallelisms roughly halfway between no renaming and perfect renaming.

4.8. Conclusions.

Good branch prediction by hardware or software is critical to the exploitation of more than modest amounts of instruction-level parallelism. Jump prediction can reduce the penalty for indirect jumps, but has little effect on the parallelism of non-penalty cycles. Register

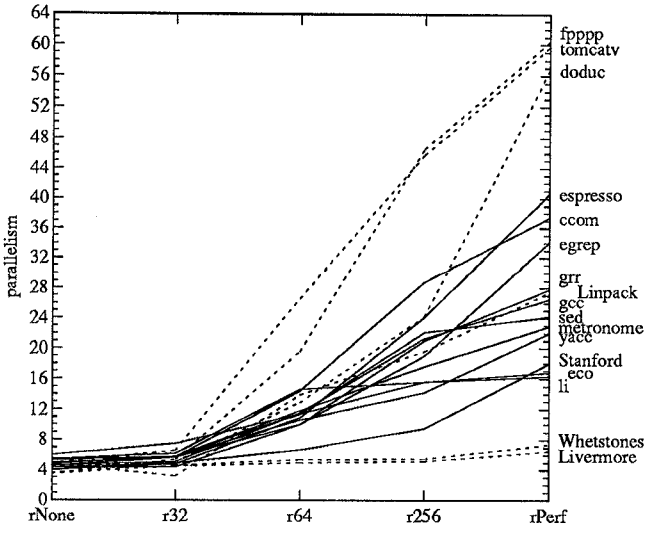

Figure 13(a). Effect of register renaming on the Perfect model.

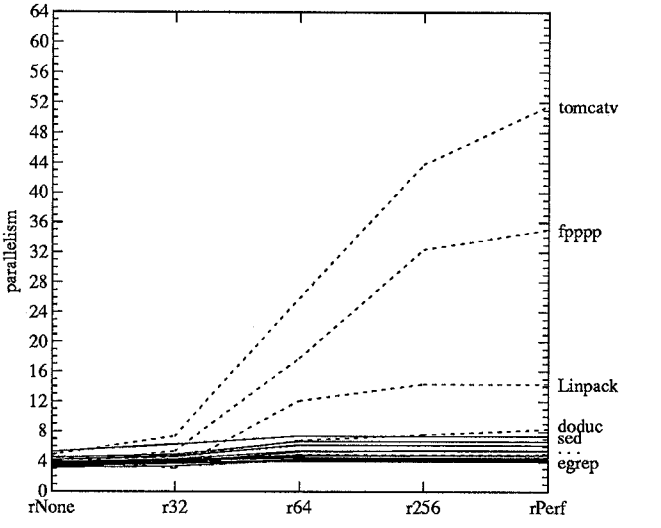

Figure 13(b). Effect of register renaming on the Great model.

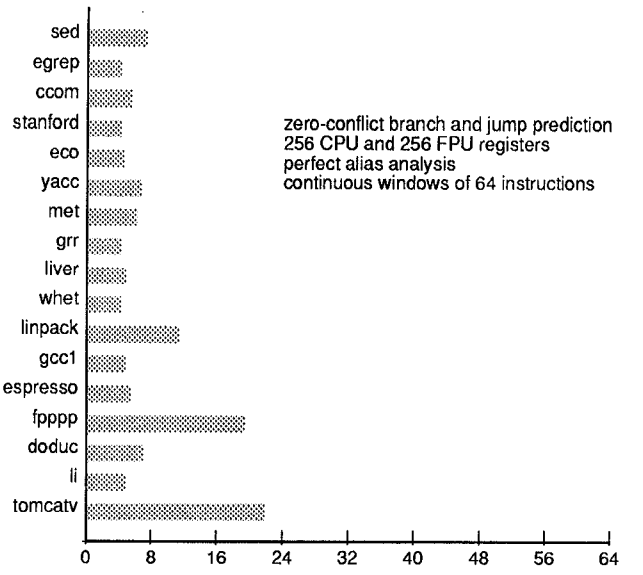

Figure 14. The parallelism from an ambitious hardware model.

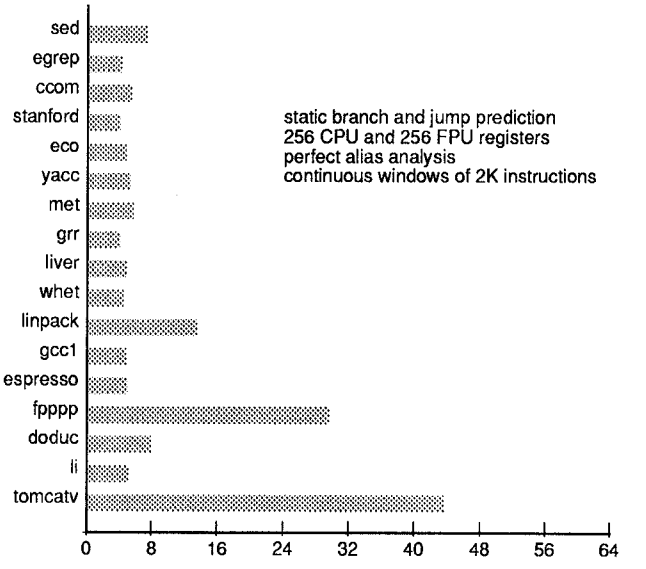

Figure 15. The parallelism from an ambitious software model.

renaming is important as well, though a compiler might be able to do an adequate job with static analysis, if it knows it is compiling for parallelism.

Even ambitious models combining the techniques discussed here are disappointing. Figure 14 shows the parallelism achieved by a quite ambitious hardware-style model, with branch and jump prediction using infinite tables, 256 FPU and 256 CPU registers used with LRU renaming, perfect alias analysis, and windows of 64 instructions maintained continuously. The average parallelism is around 7, the median around 5. Figure 15 shows the parallelism achieved by a quite ambitious software-style model, with static branch and jump prediction, 256 FPU and 256 CPU registers used with LRU renaming, perfect alias analysis, and windows of 2K instructions maintained continuously. The average here is closer to 9, but the median is still around 5. A consistent speedup of 5 would be quite good, but we cannot honestly expect more (at least without developing techniques beyond those discussed here).

We must also remember the simplifying assumptions this study makes. We have assumed that all operations have latency of one cycle; in practice an instruction with larger latency uses some of the available parallelism. We have assumed unlimited resources, including a perfect cache; as the memory bottleneck gets worse it may be more helpful to have a bigger on-chip cache than a lot of duplicate functional units. We have assumed that there is no penalty for a missed prediction; in practice the penalty may be many empty cycles. We have assumed a

uniform machine cycle time, though adding superscalar capability will surely not decrease the cycle time and may in fact increase it. We have assumed uniform technology, but an ordinary machine may have a shorter time-to-market and therefore use newer, faster technology. Sadly, any one of these considerations could reduce our expected parallelism by a third; together they could eliminate it completely.

References

[1] Tilak Agarwala and John Cocke. High performance reduced instruction set processors. IBM Thomas J. Watson Research Center Technical Report #55845, March 31, 1987.

[2] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation,* pp. 296-310. Published as *SIGPLAN Notices 25* (6), June 1990.

[3] Jack J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *Computer Architecture News 11* (5), pp. 22-27, December 1983.

[4] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction,* pp. 37-47. Published as *SIGPLAN Notices 19* (6), June 1984.

[5] John Hennessy. Stanford benchmark suite. Personal communication.

[6] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Ninth Annual ACM Symposium on Principles of Programming Languages,* pp. 66-74, Jan. 1982.

[7] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 272-282, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue). Also available as WRL Research Report *89/7.*

[8] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pp. 318-328. Published as *SIGPLAN Notices 23* (7), July 1988.

[9] Johnny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *Computer 17* (1), pp. 6-22, January 1984.

[10] Scott McFarling and John Hennessy. Reducing the cost of branches. *Thirteenth Annual Symposium on Computer Architecture,* pp. 396-403. Published as *Computer Architecture News 14* (2), June 1986.

[11] Alexandru Nicolau and Joseph A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers C-33* (11), pp. 968-976, November 1984.

[12] J. E. Smith. A study of branch prediction strategies. *Eighth Annual Symposium on Computer Architecture,* pp. 135-148. Published as *Computer Architecture News 9* (3), 1986.

[13] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 290-302, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue).

[14] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of parallel instructions. *IEEE Transactions on Computers C-19* (10), pp. 889-895, October 1970.

Appendix. Parallelism under many models.

On the next two pages are the results of running the test programs under more than 100 different configurations. The columns labelled "Livcu10" and "Lincu10" are the Livermore and Linpack benchmarks unrolled carefully 10 times. The configurations are keyed by the following abbreviations:

bPerf	perfect branch prediction, 100% correct.
bInf	2-bit branch prediction with infinite table.
b2K	2-bit branch prediction with 2K-entry table.
bStat	static branch prediction from profile.
bNone	no branch prediction.
jPerf	perfect indirect jump prediction, 100% correct.
jInf	indirect jump prediction with infinite table.
j2K	indirect jump prediction with 2K-entry table.
jStat	static indirect jump prediction from profile.
jNone	no indirect jump prediction.
rPerf	perfect register renaming: infinitely many registers.
rN	register renaming with N cpu and N fpu registers.
rNone	no register renaming: use registers as compiled.
aPerf	perfect alias analysis: use actual addresses to decide.
aComp	"compiler" alias analysis.
aInsp	"alias analysis "by inspection."
aNone	no alias analysis.
wN	continuous window of N instructions, default 2K.
dwN	discrete window of N instructions, default 2K.

	gcc	espresso	li	fpppp	doduc	tomcat	sed	egrep	ccom	eco	yacc	met	grr	Stan	Whet	Liv	Livcu10	Lin	Lincu10
bPerf jPerf rPerf aPerf	26.5	40.6	16.3	60.4	56.8	59.7	24.2	34.1	37.4	16.8	22.0	22.9	27.8	18.0	7.4	6.5	10.8	27.4	27.5
bPerf jPerf rPerf aPerf w512	23.4	30.7	15.7	51.5	42.5	54.8	23.9	22.0	32.4	16.1	15.6	19.4	24.0	11.1	7.4	5.4	8.5	21.8	24.1
bPerf jPerf rPerf aPerf w256	20.5	23.3	15.6	44.6	29.9	52.3	21.3	17.0	25.1	15.4	13.6	17.1	20.1	8.8	7.4	5.2	8.0	18.7	22.2
bPerf jPerf rPerf aPerf w128	16.0	16.7	15.5	32.8	19.7	37.4	15.8	12.9	18.9	13.8	11.9	14.4	15.2	7.5	7.0	5.1	7.5	17.4	17.5
bPerf jPerf rPerf aPerf w64	11.9	11.7	14.2	21.6	13.3	23.8	11.2	10.0	13.8	11.2	10.5	11.9	10.7	6.7	6.6	5.0	6.5	12.9	13.0
bPerf jPerf rPerf aPerf w32	8.6	8.1	10.7	13.4	9.2	14.7	9.0	7.5	9.7	8.4	8.9	9.2	7.5	5.9	6.0	4.7	5.2	8.7	9.0
bPerf jPerf rPerf aPerf w16	6.2	5.8	7.0	8.3	6.4	8.9	7.2	5.3	7.1	5.8	6.7	6.1	5.2	5.1	5.1	4.3	4.3	5.4	5.9
bPerf jPerf rPerf aPerf w8	4.1	3.9	4.6	5.2	4.3	5.6	4.8	3.8	4.8	4.1	4.1	4.0	3.5	3.9	3.7	3.5	3.3	3.2	3.6
bPerf jPerf rPerf aPerf w4	2.6	2.5	2.9	3.2	2.8	3.3	2.9	2.2	2.9	2.7	2.5	2.5	2.4	2.6	2.5	2.5	2.4	2.0	2.2
bPerf jPerf rPerf aPerf dw	21.0	25.4	15.1	56.4	44.7	54.0	23.3	24.0	26.8	14.5	15.1	16.6	19.8	10.8	7.4	5.2	7.7	21.8	23.9
bPerf jPerf rPerf aPerf dw512	13.2	15.3	13.4	39.9	16.0	46.8	14.4	11.8	15.9	11.4	11.1	11.6	12.4	7.0	6.7	4.9	6.8	15.1	18.2
bPerf jPerf rPerf aPerf dw256	9.7	11.6	11.7	24.9	10.5	19.4	10.3	9.3	12.0	9.0	9.4	9.5	8.8	6.1	5.9	4.7	6.1	12.2	14.0
bPerf jPerf rPerf aPerf dw128	7.4	8.4	9.6	10.9	7.4	12.4	8.1	7.1	8.9	7.0	8.0	7.5	6.6	5.3	5.2	4.4	5.1	9.0	9.9
bPerf jPerf rPerf aPerf dw64	5.7	5.9	7.0	7.5	5.5	8.9	6.6	5.2	6.7	5.5	6.5	5.6	5.0	4.6	4.5	3.9	4.2	6.1	6.6
bPerf jPerf rPerf aPerf dw32	4.3	4.2	5.1	5.3	4.2	5.9	5.2	3.8	5.1	4.3	4.8	4.1	3.7	3.8	3.6	3.3	3.5	3.7	4.1
bPerf jPerf rPerf aPerf dw16	3.3	3.0	3.8	3.9	3.3	4.1	4.0	2.8	3.8	3.3	3.3	3.1	2.8	2.9	2.9	2.7	2.8	2.3	2.5
bPerf jPerf rPerf aPerf dw8	2.5	2.2	2.8	3.1	2.7	3.2	2.7	2.0	2.8	2.5	2.3	2.4	2.1	2.3	2.3	2.2	2.2	1.7	1.9
bPerf jPerf rPerf aPerf dw4	1.9	1.7	2.0	2.4	2.2	2.6	2.0	1.5	2.0	1.9	1.7	1.8	1.7	1.7	1.9	1.7	1.8	1.5	1.6
bPerf jPerf rPerf aComp	10.1	8.2	12.8	60.4	56.8	59.7	24.2	34.1	18.7	7.6	14.6	15.3	17.5	18.0	7.4	6.5	10.8	27.4	27.5
bPerf jPerf rPerf aInsp	4.2	5.6	4.6	4.8	6.7	4.9	5.0	7.4	2.7	3.6	6.9	3.4	3.7	5.7	3.5	2.4	2.1	3.5	6.7
bPerf jPerf rPerf aNone	3.3	5.1	3.1	3.4	3.9	3.5	4.4	7.3	2.4	2.7	6.0	2.9	3.0	4.9	2.9	2.3	1.8	3.4	6.0
bPerf jPerf r256 aPerf	21.3	24.1	15.6	46.6	24.2	45.8	22.1	19.0	28.8	15.6	14.2	17.6	21.0	9.5	5.5	5.2	6.9	19.6	23.6
bPerf jPerf r64 aPerf	11.5	10.9	14.6	19.6	12.9	26.6	11.4	10.0	14.4	11.4	10.6	11.8	10.0	6.7	5.4	5.0	6.5	13.9	14.7
bPerf jPerf r32 aPerf	5.1	4.8	6.2	5.4	5.1	6.6	7.5	5.7	5.7	5.8	5.8	5.8	4.4	4.9	4.6	4.5	5.0	3.2	4.6
bPerf jPerf rNone aPerf	4.6	4.0	5.4	3.5	4.5	4.9	6.0	5.5	4.5	4.8	5.3	4.9	4.3	4.3	3.5	3.6	3.2	4.8	5.8
bPerf jPerf rNone aNone	2.8	3.1	2.8	2.6	3.1	3.0	3.4	4.3	2.3	2.5	3.9	2.6	2.5	3.3	2.4	2.2	1.7	3.4	4.0
bPerf jInf rPerf aPerf	7.7	18.1	6.6	50.8	10.3	59.3	21.5	33.8	8.8	5.9	15.1	10.0	9.5	12.8	4.9	6.5	10.8	27.3	27.5
bPerf jNone rPerf aPerf	5.6	11.2	4.7	34.8	8.4	59.2	6.6	15.3	5.5	4.5	9.4	5.4	5.3	5.0	3.7	6.5	10.8	16.3	20.9
bPerf jNone rNone aNone	2.5	3.0	2.4	2.6	3.0	3.0	3.1	4.3	2.0	2.3	3.8	2.3	2.4	3.1	2.2	2.2	1.7	3.4	4.0
bInf jPerf rPerf aPerf	5.7	5.7	6.2	35.9	12.5	51.7	7.5	4.1	6.7	6.1	6.9	7.0	4.7	4.4	6.6	4.9	7.6	14.3	17.1
bInf jInf rPerf aPerf	4.8	5.5	4.8	35.0	8.3	51.6	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.6	4.9	7.6	14.3	17.0
bInf jInf rPerf aPerf w512	4.8	5.5	4.8	33.8	8.3	51.4	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.6	4.9	7.6	14.3	17.0
bInf jInf rPerf aPerf w256	4.8	5.5	4.8	31.9	8.2	50.0	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.6	4.9	7.6	14.3	16.6
bInf jInf rPerf aPerf w128	4.8	5.4	4.8	26.8	8.0	36.5	7.4	4.1	5.4	4.5	6.7	6.2	4.3	4.2	4.6	4.9	7.3	14.3	14.4
bInf jInf rPerf aPerf w64	4.7	5.4	4.8	19.4	7.5	23.5	7.3	4.1	5.4	4.5	6.7	6.2	4.3	4.1	4.6	4.9	6.4	11.4	11.6
bInf jInf rPerf aPerf w32	4.6	5.0	4.7	12.7	6.6	14.7	7.0	4.0	5.3	4.4	6.5	6.0	4.2	4.1	4.6	4.7	5.2	8.1	8.4
bInf jInf rPerf aPerf w16	4.3	4.4	4.3	8.1	5.5	8.9	6.4	3.7	5.1	4.2	5.5	4.8	3.8	4.0	4.4	4.2	4.3	5.2	5.7
bInf jInf rPerf aPerf w8	3.4	3.4	3.7	5.1	4.1	5.6	4.5	3.1	4.1	3.5	3.8	3.7	3.0	3.4	3.4	3.4	3.3	3.2	3.6
bInf jInf rPerf aPerf w4	2.5	2.3	2.7	3.2	2.8	3.3	2.9	2.0	2.7	2.5	2.4	2.4	2.2	2.4	2.5	2.5	2.4	2.0	2.2
bInf jInf rPerf aPerf dw	4.8	5.5	4.8	34.7	8.3	50.7	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.6	4.9	7.4	14.3	17.0
bInf jInf rPerf aPerf dw512	4.8	5.4	4.8	31.1	8.3	45.3	7.4	4.1	5.4	4.5	6.6	6.1	4.3	4.1	4.6	4.8	6.7	12.6	15.2
bInf jInf rPerf aPerf dw256	4.7	5.3	4.8	22.6	7.4	19.2	7.3	4.1	5.3	4.4	6.4	6.0	4.3	4.1	4.6	4.7	6.0	10.9	12.9
bInf jInf rPerf aPerf dw128	4.6	5.1	4.8	10.6	6.2	12.4	6.9	4.0	5.2	4.4	6.1	5.6	4.2	4.0	4.4	4.4	5.0	8.6	9.4
bInf jInf rPerf aPerf dw64	4.3	4.4	4.4	7.5	5.0	8.9	5.8	3.8	4.9	4.1	5.5	4.8	3.8	3.8	4.1	3.8	4.2	5.9	6.4
bInf jInf rPerf aPerf dw32	3.6	3.6	4.0	5.3	4.1	5.9	5.0	3.1	4.2	3.7	4.4	3.8	3.2	3.3	3.4	3.3	3.4	3.6	4.1
bInf jInf rPerf aPerf dw16	2.9	2.7	3.3	3.9	3.3	4.1	3.9	2.5	3.5	3.0	3.2	2.9	2.6	2.7	2.7	2.7	2.8	2.3	2.5
bInf jInf rPerf aPerf dw8	2.3	2.1	2.6	3.1	2.6	3.2	2.7	2.0	2.7	2.4	2.3	2.3	2.0	2.2	2.2	2.2	2.2	1.7	1.9
bInf jInf rPerf aPerf dw4	1.8	1.7	2.0	2.4	2.1	2.6	1.9	1.5	2.0	1.9	1.7	1.8	1.7	1.7	1.9	1.7	1.7	1.4	1.6
bInf jInf rPerf aComp	4.0	4.4	4.3	34.9	8.3	51.6	7.4	4.1	4.9	3.8	5.8	5.5	4.3	4.2	4.6	4.9	7.6	14.3	17.0
bInf jInf rPerf aInsp	3.1	3.8	3.3	4.7	5.3	4.9	4.2	3.4	2.4	2.9	4.8	3.0	2.8	3.3	3.0	2.3	2.1	3.5	6.5
bInf jInf rPerf aNone	2.7	3.6	2.5	3.4	3.4	3.5	3.8	3.4	2.1	2.4	4.4	2.6	2.4	3.1	2.5	2.3	1.8	3.4	5.9

	gcc	espresso	li	fpppp	doduc	tomcat	sed	egrep	ccom	eco	yacc	met	grr	Stan	Whet	Liv	Livcul0	Lin	Lincul0
bInf jInf r256 aPerf	4.8	5.5	4.8	32.4	7.6	43.8	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.2	4.9	6.4	14.3	17.0
bInf jInf r256 aPerf w512	4.8	5.5	4.8	32.2	7.6	43.8	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.2	4.9	6.4	14.3	17.0
bInf jInf r256 aPerf w256	4.8	5.5	4.8	31.6	7.6	43.3	7.4	4.1	5.4	4.5	6.7	6.2	4.4	4.2	4.2	4.9	6.4	14.3	16.6
bInf jInf r256 aPerf w128	4.8	5.4	4.8	26.8	7.5	32.9	7.4	4.1	5.4	4.5	6.7	6.2	4.3	4.2	4.2	4.9	6.3	14.2	14.3
bInf jInf r256 aPerf w64	4.7	5.4	4.8	19.4	7.0	21.8	7.3	4.1	5.4	4.5	6.7	6.2	4.3	4.1	4.2	4.8	5.9	11.4	11.6
bInf jInf r256 aPerf w32	4.6	5.0	4.7	12.7	6.3	14.0	7.0	4.0	5.3	4.4	6.5	6.0	4.2	4.1	4.2	4.5	5.2	8.1	8.4
bInf jInf r256 aPerf w16	4.3	4.4	4.3	8.1	5.4	8.9	6.4	3.7	5.1	4.2	5.5	4.8	3.8	4.0	4.1	4.1	4.3	5.2	5.7
bInf jInf r256 aPerf w8	3.4	3.4	3.7	5.1	4.0	5.6	4.5	3.1	4.1	3.5	3.8	3.7	3.0	3.4	3.3	3.3	3.3	3.2	3.6
bInf jInf r256 aPerf w4	2.5	2.3	2.7	3.2	2.7	3.3	2.9	2.0	2.7	2.5	2.4	2.4	2.2	2.4	2.4	2.5	2.4	2.0	2.2
bInf jInf r256 aComp	4.0	4.4	4.3	32.4	7.6	43.8	7.4	4.1	4.9	3.8	5.8	5.5	4.3	4.2	4.2	4.9	6.4	14.3	17.0
bInf jInf r256 aInsp	3.1	3.8	3.3	4.7	5.0	4.8	4.2	3.4	2.4	2.9	4.8	3.0	2.8	3.3	2.8	2.3	2.1	3.5	6.5
bInf jInf r256 aNone	2.7	3.6	2.5	3.4	3.3	3.4	3.8	3.4	2.1	2.4	4.4	2.6	2.4	3.0	2.3	2.2	1.8	3.4	5.9
bInf jInf r64 aPerf	4.7	5.3	4.8	17.8	6.8	25.9	7.4	4.1	5.4	4.5	6.7	6.2	4.3	4.2	4.2	4.9	6.3	12.1	12.5
bInf jInf r64 aComp	4.0	4.3	4.3	17.8	6.8	26.8	7.4	4.1	4.9	3.8	5.8	5.5	4.2	4.2	4.2	4.9	6.3	12.1	12.5
bInf jInf r64 aInsp	3.1	3.8	3.3	4.7	4.8	4.8	4.1	3.4	2.4	2.9	4.8	3.0	2.8	3.3	2.8	2.3	2.1	3.5	6.4
bInf jInf r32 aPerf	3.9	3.8	4.2	5.4	4.6	7.4	6.3	3.8	4.3	4.1	4.9	4.7	3.4	3.8	3.8	4.0	4.9	3.1	4.5
bInf jInf r32 aComp	3.4	3.2	3.9	5.4	4.6	7.5	6.3	3.8	4.1	3.6	4.5	4.3	3.4	3.8	3.8	4.0	4.9	3.1	4.5
bInf jInf r32 aInsp	2.9	3.0	3.2	3.9	3.9	4.4	3.8	3.2	2.4	2.8	3.9	2.9	2.6	3.2	2.6	2.3	2.1	2.7	3.6
bInf jInf rNone aPerf	3.5	3.3	3.9	3.4	4.2	4.9	5.3	3.8	3.6	3.7	4.5	4.2	3.2	3.5	3.3	3.6	3.2	4.6	5.6
bInf jInf rNone aComp	3.2	2.9	3.8	3.4	4.2	4.9	5.3	3.8	3.5	3.3	4.3	4.0	3.2	3.5	3.3	3.6	3.2	4.6	5.6
bInf jInf rNone aInsp	2.8	2.8	3.1	3.2	3.7	3.8	3.5	3.3	2.3	2.7	3.8	2.7	2.5	3.0	2.6	2.2	1.9	3.4	4.2
bInf jStat rPerf aPerf	4.8	5.4	5.2	34.3	8.8	51.6	7.4	4.1	5.5	4.8	6.6	5.8	4.4	4.2	4.8	4.9	7.6	14.3	17.0
bInf jNone rPerf aPerf	4.1	5.2	3.9	30.6	7.3	51.5	5.5	4.1	4.5	3.9	6.2	4.5	3.8	3.8	3.5	4.9	7.6	14.1	16.8
bInf jNone r256 aPerf	4.1	5.2	3.9	29.5	6.8	43.7	5.5	4.1	4.5	3.9	6.2	4.5	3.8	3.8	3.3	4.9	6.4	14.1	16.8
bInf jNone r256 aComp	3.6	4.3	3.7	29.5	6.8	43.7	5.5	4.1	4.2	3.5	5.5	4.3	3.7	3.8	3.3	4.9	6.4	14.1	16.8
bInf jNone r256 aInsp	2.9	3.7	3.0	4.7	4.8	4.8	3.6	3.4	2.2	2.8	4.6	2.7	2.6	3.2	2.7	2.3	2.1	3.4	6.5
b2K j2K rPerf aPerf	4.6	5.4	4.8	32.4	8.2	51.6	7.4	4.1	5.3	4.4	6.7	6.2	4.3	4.2	4.6	4.9	7.6	14.3	17.0
b2K j2K r256 aPerf	4.6	5.4	4.8	30.5	7.5	43.8	7.4	4.1	5.3	4.4	6.7	6.2	4.3	4.2	4.2	4.9	6.4	14.3	17.0
b2K j2K r256 aComp	3.9	4.4	4.3	30.5	7.5	43.8	7.4	4.1	4.8	3.8	5.8	5.5	4.2	4.2	4.2	4.9	6.4	14.3	17.0
b2K j2K r256 aInsp	3.1	3.8	3.3	4.7	5.0	4.8	4.2	3.4	2.4	2.9	4.8	3.0	2.8	3.3	2.8	2.3	2.1	3.5	6.5
b2K j2K r64 aPerf	4.5	5.3	4.8	17.6	6.8	26.3	7.4	4.1	5.3	4.4	6.7	6.2	4.3	4.2	4.2	4.9	6.3	12.1	12.5
b2K j2K r32 aPerf	3.8	3.8	4.2	5.3	4.5	7.5	6.3	3.8	4.3	4.1	4.9	4.7	3.4	3.8	3.8	4.0	4.9	3.1	4.5
b2K j2K rNone aPerf	3.5	3.3	3.9	3.4	4.1	4.9	5.3	3.8	3.6	3.6	4.5	4.2	3.2	3.5	3.3	3.6	3.2	4.6	5.6
bStat jInf rPerf aPerf	4.7	5.0	4.8	31.6	8.2	51.4	7.4	4.3	5.4	4.4	5.3	6.1	4.0	4.0	4.8	4.9	7.6	13.6	16.6
bStat jInf r256 aPerf	4.7	5.0	4.8	29.8	7.5	43.7	7.4	4.3	5.4	4.4	5.3	6.1	4.0	4.0	4.3	4.9	6.4	13.6	16.6
bStat jStat rPerf aPerf	4.8	5.0	5.2	31.6	8.7	51.4	7.4	4.3	5.5	4.7	5.3	5.7	4.0	4.0	4.9	4.9	7.6	13.6	16.6
bStat jStat r256 aPerf	4.8	5.0	5.2	29.7	8.0	43.7	7.4	4.3	5.5	4.7	5.3	5.7	4.0	4.0	4.4	4.9	6.4	13.6	16.6
bNone jPerf rPerf aPerf	1.8	1.7	2.1	17.5	3.2	8.7	1.7	1.3	2.3	2.0	1.3	1.9	1.8	1.6	2.6	2.7	4.1	4.2	5.5
bNone jPerf rNone aNone	1.5	1.5	1.6	2.5	2.0	2.7	1.5	1.2	1.6	1.5	1.3	1.5	1.4	1.5	1.8	1.7	1.6	2.4	2.9
bNone jInf rPerf aPerf	1.8	1.7	2.0	17.5	3.1	8.7	1.7	1.3	2.2	1.9	1.3	1.9	1.8	1.6	2.6	2.7	4.1	4.2	5.5
bNone jInf r256 aComp	1.7	1.6	2.0	17.0	3.1	8.7	1.7	1.3	2.2	1.8	1.3	1.9	1.8	1.6	2.5	2.5	3.8	4.2	5.5
bNone jInf r256 aInsp	1.6	1.5	1.8	4.5	2.7	3.9	1.6	1.3	1.7	1.7	1.3	1.6	1.6	1.5	2.0	1.8	2.0	2.6	4.2
bNone jNone rPerf aPerf	1.8	1.7	1.9	16.9	3.0	8.7	1.6	1.3	2.1	1.9	1.3	1.8	1.8	1.6	2.2	2.7	4.1	4.2	5.5
bNone jNone rPerf aNone	1.5	1.5	1.5	3.3	2.1	3.0	1.5	1.3	1.5	1.5	1.3	1.4	1.4	1.5	1.9	1.8	1.7	2.5	4.0
bNone jNone r256 aPerf	1.8	1.7	1.9	16.4	3.0	8.7	1.6	1.3	2.1	1.9	1.3	1.8	1.8	1.6	2.2	2.5	3.8	4.2	5.5
bNone jNone r256 aComp	1.7	1.6	1.9	16.4	3.0	8.7	1.6	1.3	2.1	1.8	1.3	1.8	1.7	1.6	2.2	2.5	3.8	4.2	5.5
bNone jNone r256 aInsp	1.6	1.5	1.7	4.4	2.7	3.9	1.5	1.3	1.6	1.6	1.3	1.5	1.5	1.5	1.9	1.8	2.0	2.6	4.2
bNone jNone r256 aNone	1.5	1.5	1.5	3.3	2.1	3.0	1.5	1.3	1.5	1.5	1.3	1.4	1.4	1.5	1.8	1.8	1.7	2.5	4.0
bNone jNone rNone aPerf	1.7	1.6	1.9	3.3	2.4	4.1	1.6	1.3	2.1	1.8	1.3	1.8	1.7	1.5	2.1	2.4	2.7	2.8	3.6
bNone jNone rNone aComp	1.7	1.5	1.9	3.3	2.4	4.1	1.6	1.3	2.1	1.8	1.3	1.8	1.7	1.5	2.1	2.4	2.7	2.8	3.6
bNone jNone rNone aInsp	1.6	1.5	1.7	3.1	2.3	3.3	1.5	1.2	1.6	1.6	1.3	1.5	1.5	1.5	1.8	1.7	1.8	2.5	3.0
bNone jNone rNone aNone	1.5	1.5	1.5	2.5	2.0	2.7	1.4	1.2	1.5	1.4	1.3	1.4	1.4	1.4	1.8	1.7	1.6	2.4	2.9