NOVEMBER 1993

WRL
Research Report 93/6

Limits of Instruction-Level Parallelism

David W. Wall

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There two other research laboratories located in Palo Alto, the Network Systems Laboratory (NSL) and the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Limits of Instruction-Level Parallelism

David W. Wall

November 1993

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

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. This paper presents the results of simulations of 18 different test programs under 375 different models of available parallelism analysis.

This paper replaces Technical Note TN-15, an earlier version of the same material.

Author's note

Three years ago I published some preliminary results of a simulation-based study of instruction-level parallelism [Wall91]. It took advantage of a fast instruction-level simulator and a computing environment in which I could use three or four dozen machines with performance in the 20-30 MIPS range every night for many weeks. But the space of parallelism techniques to be explored is very large, and that study only scratched the surface.

The report you are reading now is an attempt to fill some of the cracks, both by simulating more intermediate models and by considering a few ideas the original study did not consider. I believe it is by far the most extensive study of its kind, requiring almost three machine-years and simulating in excess of 1 trillion instructions.

The original paper generated many different opinions[1]. Some looked at the high parallelism available from very ambitious (some might say unrealistic) models and proclaimed the millennium. My own opinion was pessimistic: I looked at how many different things you have to get right, including things this study doesn't address at all, and despaired. Since then I have moderated that opinion somewhat, but I still consider the negative results of this study to be at least as important as the positive.

This study produced far too many numbers to present them all in the text and graphs, so the complete results are available only in the appendix. I have tried not to editorialize in the selection of which results to present in detail, but a careful study of the numbers in the appendix may well reward the obsessive reader.

In the three years since the preliminary paper appeared, multiple-issue architectures have changed from interesting idea to revealed truth, though little hard data is available even now. I hope the results in this paper will be helpful. It must be emphasized, however, that they should be treated as guideposts and not mandates. When one contemplates a new architecture, there is no substitute for simulations that include real pipeline details, a likely memory configuration, and a much larger program suite than a study like this one can include.

[1] Probably exactly as many opinions as there were before it appeared.

1 Introduction

In recent years there has been an explosion of interest in *multiple-issue machines*. These are designed to exploit, usually with compiler assistance, the parallelism that programs exhibit at the instruction level. Figure 1 shows an example of this parallelism. 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.

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)

Architectures have been proposed to take advantage of this kind of parallelism. A superscalar machine [AC87] is one that can issue multiple independent instructions in the same cycle. A superpipelined machine [JW89] issues one instruction per cycle, but the cycle time is much smaller than the typical instruction latency. A VLIW machine [NF84] is like a superscalar machine, except the parallel instructions must be explicitly packed by the compiler into very long instruction words.

Most "ordinary" pipelined machines already have some degree of parallelism, if they have operations with multi-cycle latencies; while these instructions work, shorter unrelated instructions can be performed. We can compute the degree of parallelism by multiplying the latency of each operation by its relative dynamic frequency in typical programs. The latencies of loads, delayed branches, and floating-point instructions give the DECstation[2] 5000, for example, a parallelism equal to about 1.5.

A multiple-issue machine has a hardware cost beyond that of a scalar machine of equivalent technology. This cost may be small or large, depending on how aggressively the machine pursues instruction-level parallelism. In any case, whether a particular approach is feasible depends on its cost and the parallelism that can be obtained from it.

But how much parallelism is there to exploit? This is a question about programs rather than about machines. We can build a machine with any amount of instruction-level parallelism we choose. But all of that parallelism would go unused if, for example, we learned that programs consisted of linear sequences of instructions, each dependent on its predecessor's result. Real programs are not that bad, as Figure 1(a) illustrates. How much parallelism we can find in a program, however, is limited by how hard we are willing to work to find it.

[2]DECStation is a trademark of Digital Equipment Corporation.

A number of studies [JW89, SJH89, TF70] dating back 20 years show that parallelism within a basic block rarely exceeds 3 or 4 on the average. This is unsurprising: basic blocks are typically around 10 instructions long, leaving little scope for a lot of parallelism. At the other extreme is a study by Nicolau and Fisher [NF84] that finds average parallelism as high as 1000, by considering highly parallel numeric programs and simulating a machine with unlimited hardware parallelism and an omniscient scheduler.

There is a lot of space between 3 and 1000, and a lot of space between analysis that looks only within basic blocks and analysis that assumes an omniscient scheduler. Moreover, this space is multi-dimensional, because parallelism analysis consists of an ever-growing body of complementary techniques. The payoff of one choice depends strongly on its context in the other choices made. The purpose of this study is to explore that multi-dimensional space, and provide some insight about the importance of different techniques in different contexts. We looked at the parallelism of 18 different programs at more than 350 points in this space.

The next section describes the capabilities of our simulation system and discusses the various parallelism-enhancing techniques it can model. This is followed by a long section looking at some of the results; a complete table of the results is given in an appendix. Another appendix gives details of our implementation of these techniques.

2 Our experimental framework

We studied the instruction-level parallelism of eighteen test programs. Twelve of these were taken from the SPEC92 suite; three are common utility programs, and three are CAD tools written at WRL. These programs are shown in Figure 2. 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, and in two cases we modified the source to decrease the iteration count of the outer loop. Appendix 2 contains the details of the modifications and data sets. The programs were compiled for a DECStation 5000, which has a MIPS R3000[3] processor. The Mips version 1.31 compilers were used.

Like most studies of instruction-level parallelism, we used *oracle-driven trace-based simulation*. We begin by obtaining a trace of the instructions executed.[4] This trace also includes the data addresses referenced and the results of branches and jumps. A greedy scheduling algorithm, guided by a configurable oracle, packs these instructions into a sequence of *pending cycles*. The resulting sequence of cycles represents a hypothetical execution of the program on some multiple-issue machine. Dividing the number of instructions executed by the number of cycles required gives the average parallelism.

The configurable oracle models a particular combination of techniques to find or enhance the instruction-level parallelism. Scheduling to exploit the parallelism is constrained by the possibility of *dependencies* between instructions. Two instructions have a dependency if changing their order changes their effect, either because of changes in the data values used or because one instruction's execution is conditional on the other.

[3] R3000 is a trademark of MIPS Computer Systems, Inc.

[4] In our case by simulating it on a conventional instruction-level simulator.

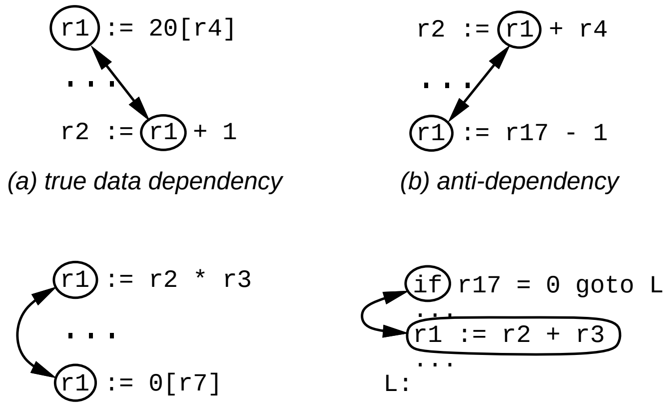
	source lines	*instructions executed*	*remarks*
egrep	762	13721780	File search
sed	1683	1462487	Stream editor
yacc	1856	30297973	Compiler-compiler
eco	3349	27397304	Recursive tree comparison
grr	7241	144442216	PCB router
metronome	4673	71273955	Timing verifier
alvinn	223	388973837	Neural network training
compress	1401	88277866	Lempel-Ziv file compaction
doduc	5333	284421280	Hydrocode simulation
espresso	12934	134435218	Boolean function minimizer
fpppp	2472	244278269	Quantum chemistry benchmark
gcc1	78782	22753759	First pass of GNU C compiler
hydro2d	4458	8235288	Astrophysical simulation
li	6102	263742027	Lisp interpreter
mdljsp2	3739	393078251	Molecular dynamics model
ora	427	212125692	Ray tracing
swm256	484	301407811	Shallow water simulation
tomcatv	195	301622982	Vectorized mesh generation

Figure 2: The eighteen test programs

Figure 3 illustrates the different kinds of dependencies. Some dependencies are real, reflecting the true flow of the computation. Others are false dependencies, accidents of the code generation or our lack of precise knowledge about the flow of data. Two instructions have a *true data dependency* if the result of the first is an operand of the second. Two instructions have an *anti-dependency* if the first uses the old value in some location and the second sets that location to a new value. Similarly, two instructions have an *output dependency* if they both assign a value to the same location. Finally, there is a *control dependency* between a branch and an instruction whose execution is conditional on it.

The oracle uses an actual program trace to make its decisions. This lets it “predict the future,” basing its scheduling decisions on its foreknowledge of whether a particular branch will be taken or not, or whether a load and store refer to the same memory location. It can therefore construct an impossibly perfect schedule, constrained only by the true data dependencies between instructions, but this does not provide much insight into how a real machine would perform. It is more interesting to hobble the oracle in ways that approximate the capabilities of a real machine and a real compiler system.

We can configure our oracle with different levels of expertise, ranging from nil to impossibly perfect, in several different kinds of parallelism enhancement.



(a) true data dependency *(b) anti-dependency*

(c) output dependency *(d) control dependency*

Figure 3: Dependencies

2.1 Register renaming

Anti-dependencies and output dependencies on registers are often accidents of the compiler's register allocation technique. In Figures 3(b) and 3(c), using a different register for the new value in the second instruction would remove the dependency. Register allocation that is integrated with the compiler's instruction scheduler [BEH91, GH88] could eliminate many of these. Current compilers often do not exploit this, 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. Register renaming has the additional advantage of allowing the hardware to include more registers than will fit in the instruction format, further reducing false dependencies.

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 works best, of course, when there are a lot of registers. Our simulations most often use 256 integer registers and 256 floating-point registers, but it is interesting to see what happens when we reduce this to 64 or even 32, the number on our base machine. For *no renaming*, we simply use the registers specified in the code; how well this works is of course highly dependent on the register strategy of the compiler we use.

2.2 Alias analysis

Like registers, memory locations can also carry true and false dependencies. We make the assumption that renaming of memory locations is not an option, for two reasons. First, memory is so much larger than a register file that renaming could be quite expensive. More important, though, is that memory locations tend to be used quite differently from registers. Putting a value in some memory location normally has some meaning in the logic of the program; memory is not just a scratchpad to the extent that the registers are.

Moreover, it is hard enough just telling when a memory-carried dependency exists. The registers used by an instruction are manifest in the instruction itself, while the memory location used is not manifest and in fact may be different for different executions of the instruction. A multiple-issue machine may therefore be forced to assume that a dependency exists even when it might not. This is the *aliasing problem*: telling whether two memory references access the same memory location.

Hardware mechanisms such as squashable loads have been suggested to help cope with the aliasing problem. The more conventional approach is for the compiler to perform *alias analysis*, using its knowledge of the semantics of the language and the program to rule out dependencies whenever it can.

Our system provides four levels of alias analysis. 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 4.

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

Figure 4: Alias analysis by inspection

The two instructions in 4(a) cannot conflict, because they use the same base register but different displacements. The two instructions in 4(b) cannot conflict, because their base registers show that one refers to the stack and the other 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 model of alias analysis by compiler is that references outside the heap can often be resolved by the compiler, by doing dataflow and dependency analysis over loops and arrays, 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 may not be as hopeless as this model assumes [CWZ90, HHN92, JM82, LH88]. Nevertheless, our range of four alternatives should provide some intuition about the effects of alias analysis on instruction-level parallelism.

2.3 Branch prediction

Parallelism within a basic block is usually quite limited, mainly because basic blocks are usually quite small. The approach of *speculative execution* tries to mitigate this by scheduling instructions across branches. This is hard because we don't know which way future branches will go and therefore which path to select instructions from. Worse, most branches go each way part of the time, so a branch may be followed by two possible code paths. We can move instructions from either path to a point before the branch only if those instructions will do no harm (or if the harm can be undone) when we take the other path. 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, or instructions that can raise exceptions. 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. If the compiler schedules instructions speculatively, it may even have to insert code to undo its effects at the entry to the other path.

The most common approach to speculative execution uses *branch prediction*. The hardware or the software predicts which way a given branch will most likely go, and speculatively schedules instructions from that path.

A common hardware technique for branch prediction [LS84, Smi81] maintains a table of two-bit counters. 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. These two-bit counters are *saturating*: 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. Figure 5 shows how well this two-bit counter scheme works for different table sizes, on the eighteen programs in our test suite. For most programs, the prediction success levels off by the time the table has about 512 two-bit entries. Increasing the number of bits, either by making the counters bigger or by having more of them, has little effect.

Branches can be predicted statically with comparable accuracy by obtaining a *branch profile*,

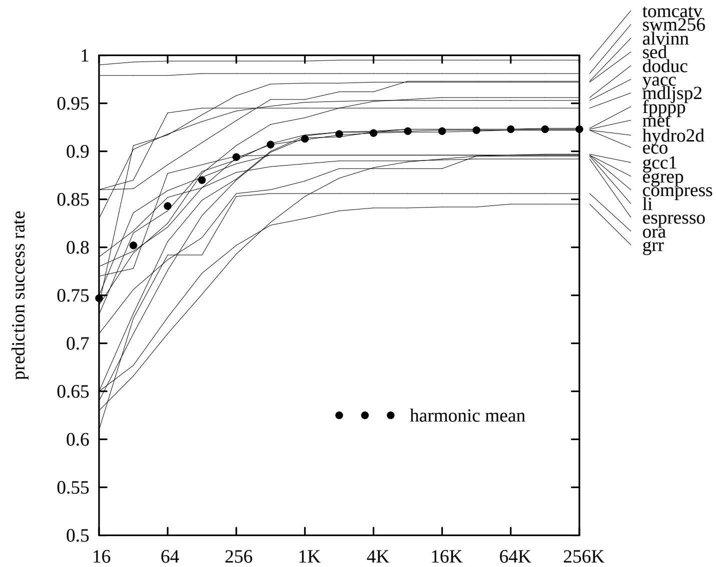


Figure 5: Fraction of branches predicted correctly using two-bit counter prediction, as a function of the total number of bits in the predictor

which tells for each branch what fraction of its executions it was taken. Like any profile, a branch profile is obtained by inserting counting code into a test program, to keep track of how many times each branch goes each way. We use a branch profile by seeing which way a given branch goes most often, and scheduling instructions from that path. If there is some expense in undoing speculative execution when the branch goes the other way, we might impose a threshold so that we don't move instructions across a branch that is executed only 51% of the time.

Recent studies have explored more sophisticated hardware prediction using *branch histories* [PSR92, YP92, YP93]. These approaches maintain tables relating the recent history of the branch (or of branches in the program as a whole) to the likely next outcome of the branch. These approaches do quite poorly with small tables, but unlike the two-bit counter schemes they can benefit from much larger predictors.

An example is the *local-history* predictor [YP92]. It maintains a table of n-bit shift registers, indexed by the branch address as above. When the branch is taken, a 1 is shifted into the table entry for that branch; otherwise a 0 is shifted in. To predict a branch, we take its n-bit history and use it as an index into a table of 2^n 2-bit counters like those in the simple counter scheme described above. If the counter is 2 or 3, we predict taken; otherwise we predict not taken. If the prediction proves correct, we increment the counter; otherwise we decrement it. The local-history predictor works well on branches that display a regular pattern of small period.

Sometimes the behavior of one branch is correlated with the behavior of another. A *global-history* predictor [YP92] tries to exploit this effect. It replaces the table of shift registers with a single shift register that records the outcome of the n most recently executed branches, and uses this history pattern as before, to index a table of counters. This allows it to exploit correlations in the behaviors of nearby branches, and allows the history to be longer for a given predictor size.

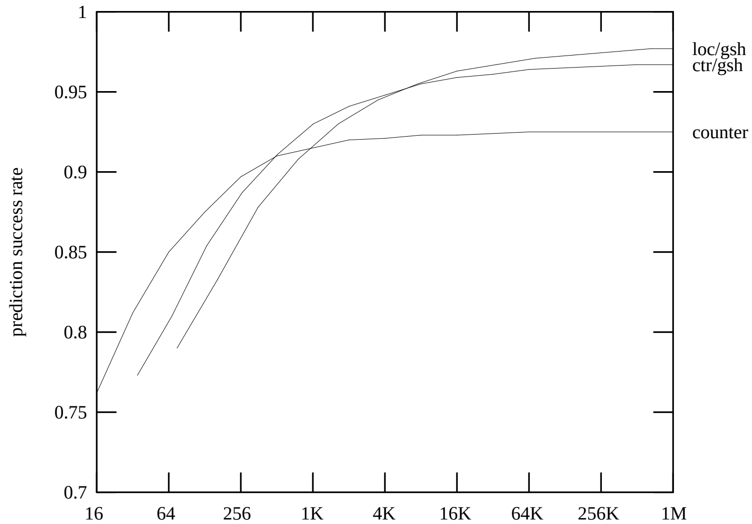


Figure 6: Fraction of branches predicted correctly by three different prediction schemes, as a function of the total number of bits in the predictor

An interesting variation is the *gshare* predictor [McF93], which uses the identity of the branch as well as the recent global history. Instead of indexing the array of counters with just the global history register, the gshare predictor computes the `xor` of the global history and branch address.

McFarling [McF93] got even better results by using a table of two-bit counters to dynamically choose between two different schemes running in competition. Each predictor makes its prediction as usual, and the branch address is used to select another 2-bit counter from a *selector* table; if the selector value is 2 or 3, the first prediction is used; otherwise the second is used. When the branch outcome is known, the selector is incremented or decremented if exactly one predictor was correct. This approach lets the two predictors compete for authority over a given branch, and awards the authority to the predictor that has recently been correct more often. McFarling found that combined predictors did not work as well as simpler schemes when the predictor size was small, but did quite well indeed when large.

Figure 6 shows the success rate for the three different hardware predictors used in this study, averaged over the eighteen programs in our suite. The first is the traditional two-bit counter approach described above. The second is a combination of a two-bit counter predictor and a gshare predictor with twice as many elements; the selector table is the same size as the counter predictor. The third is a combination of a local predictor and a gshare predictor; the two local tables, the gshare table, and the selector table all have the same number of elements. The x-axis of this graph is the total size of the predictor in bits. The simple counter predictor works best for small sizes, then the bimodal/gshare predictor takes over the lead, and finally for very large predictors the local/gshare predictor works best, delivering around 98% accuracy in the limit.

In our simulator, we modeled several degrees of branch prediction. One extreme is *perfect prediction*: we assume that all branches are correctly predicted. Next we can assume any of

	ctrs 4b	ctrs 8b	ctrs 16b	ctrs 64b	ctr/gsh 2kb	loc/gsh 16kb	loc/gsh 152k	prof	sign	taken
egrep	0.75	0.78	0.77	0.87	0.90	0.95	0.98	0.90	0.65	0.80
sed	0.75	0.81	0.74	0.92	0.97	0.98	0.98	0.97	0.42	0.71
yacc	0.74	0.80	0.83	0.92	0.96	0.97	0.98	0.92	0.60	0.73
eco	0.57	0.61	0.64	0.77	0.95	0.97	0.98	0.91	0.46	0.61
grr	0.58	0.60	0.65	0.73	0.89	0.92	0.94	0.78	0.54	0.51
metronome	0.70	0.73	0.73	0.83	0.95	0.97	0.98	0.91	0.61	0.54
alvinn	0.86	0.84	0.86	0.89	0.98	1.00	1.00	0.97	0.85	0.84
compress	0.64	0.73	0.75	0.84	0.89	0.90	0.90	0.86	0.69	0.55
doduc	0.54	0.53	0.74	0.84	0.94	0.96	0.97	0.95	0.76	0.45
espresso	0.72	0.73	0.78	0.82	0.93	0.95	0.96	0.86	0.62	0.63
fpppp	0.62	0.59	0.65	0.81	0.93	0.97	0.98	0.86	0.46	0.58
gcc1	0.59	0.61	0.63	0.70	0.87	0.91	0.94	0.88	0.50	0.57
hydro2d	0.69	0.75	0.79	0.85	0.94	0.96	0.97	0.91	0.51	0.68
li	0.61	0.69	0.71	0.77	0.95	0.96	0.98	0.88	0.54	0.46
mdljsp2	0.82	0.84	0.86	0.94	0.95	0.96	0.97	0.92	0.31	0.83
ora	0.48	0.55	0.61	0.79	0.91	0.98	0.99	0.87	0.54	0.51
swm256	0.97	0.97	0.98	0.98	1.00	1.00	1.00	0.98	0.98	0.91
tomcatv	0.99	0.99	0.99	0.99	1.00	1.00	1.00	0.99	0.62	0.99
hmean	0.68	0.71	0.75	0.84	0.94	0.96	0.97	0.90	0.55	0.63

Figure 7: Success rates of different branch prediction techniques

the three *hardware prediction* schemes shown in Figure 6 with any predictor size. We can also assume three kinds of static branch prediction: *profiled branch prediction*, in which we predict that the branch will go the way it went most frequently in a profiled previous run; *signed branch prediction*, in which we predict that a backward branch will be taken but a forward branch will not, and *taken branch prediction*, in which we predict that every branch will always be taken. And finally, we can assume that *no branch prediction* occurs; this is the same as assuming that every branch is predicted wrong.

Figure 7 shows the actual success rate of prediction using different sizes of tables. It also shows the success rates for the three kinds of static prediction. Profiled prediction routinely beats 64-bit counter-based prediction, but it cannot compete with the larger, more advanced techniques. Signed or taken prediction do quite poorly, about as well as the smallest of dynamic tables; of the two, taken prediction is slightly the better. Signed prediction, however, lends itself better to the compiler technique of moving little-used pieces of conditionally executed code out of the normal code stream, improving program locality and thereby the cache performance.

The effect of branch prediction on scheduling is easy to state. Correctly predicted branches have no effect on scheduling (except for register dependencies involving their operands). Instructions appearing later than a mispredicted branch cannot be scheduled before the branch itself, since we do not know we *should* be scheduling them until we find out that the branch went the other way. (Of course, both the branch and the later instruction may be scheduled before instructions that precede the branch, if other dependencies permit.)

Note that we generally assume no penalty for failure other than the inability to schedule later instructions before the branch. 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.

2.4 Branch fanout

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 hardware parallelism capability is guaranteed to be wasted, but we will never miss out completely by blindly 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. Thus we cannot continue to fan out indefinitely: we will eventually use up all the machine parallelism just exploring many parallel paths, of which only one is the right one. An alternative if the branch *probability* is available, as from a profile, is to explore both paths if the branch probability is near 0.5 but explore the likely path when its probability is near 1.0.

Our system allows the scheduler to explore in both directions past branches. Because the scheduler is working from a trace, it cannot actually schedule instructions from the paths not taken. Since these false paths would use up hardware parallelism, we model this by assuming that there is an upper limit on the number of branches we can look past. We call this upper limit the *fanout limit*. In terms of our simulator scheduling, branches where we explore both paths are simply considered to be correctly predicted; their effect on the schedule is identical, though of course they use up part of the fanout limit.

In some respects fanout duplicates the benefits of branch prediction, but they can also work together to good effect. If we are using dynamic branch prediction, we explore both paths up to the fanout limit, and then explore only the predicted path beyond that point. With static branch prediction based on a profile we go still further. It is easy to implement a profiler that tells us not only which direction the branch went most often, but also the frequency with which it went that way. This lets us explore only the predicted path if its predicted probability is above some threshold, and use our limited fanout ability to explore both paths only when the probability of each is below the threshold.

2.5 Indirect-jump prediction

Most architectures have two kinds of instructions to change the flow of control. Branches are conditional and have a destination that is 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 destination of an indirect jump, however, may require us to wait until the address computation is possible. Predicting the destination of an indirect jump might pay off in instruction-level parallelism.

We consider two jump prediction strategies, which can often be used simultaneously.

The first strategy is a simple cacheing 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 destination address in the table entry for the jump. To predict a jump, we extract the address in its table entry. Thus, we predict that an indirect jump will go where it went last time. As with branch prediction, however, we do not prevent two jumps from mapping to the same

	n-element return ring						2K-ring plus n-element table						prof
	1	2	4	8	16	2K	2	4	8	16	32	64	
egrep	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
sed	0.27	0.46	0.68	0.68	0.68	0.68	0.97	0.97	0.97	0.97	0.97	0.97	0.97
yacc	0.68	0.85	0.88	0.88	0.88	0.88	0.92	0.92	0.92	0.92	0.92	0.92	0.71
eco	0.48	0.66	0.76	0.77	0.77	0.78	0.82	0.82	0.82	0.82	0.82	0.82	0.56
grr	0.69	0.84	0.92	0.95	0.95	0.95	0.98	0.98	0.98	0.98	0.98	0.98	0.65
met	0.76	0.88	0.96	0.97	0.97	0.97	0.99	0.99	0.99	0.99	0.99	0.99	0.65
alvinn	0.33	0.43	0.63	0.90	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.75
compress	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
doduc	0.64	0.75	0.88	0.94	0.94	0.94	0.96	0.99	0.99	0.99	1.00	1.00	0.62
espresso	0.76	0.89	0.95	0.96	0.96	0.96	1.00	1.00	1.00	1.00	1.00	1.00	0.54
fpppp	0.55	0.71	0.73	0.74	0.74	0.74	0.99	0.99	0.99	0.99	0.99	0.99	0.80
gcc1	0.46	0.61	0.71	0.74	0.74	0.74	0.81	0.82	0.82	0.83	0.83	0.84	0.60
hydro2d	0.42	0.50	0.57	0.61	0.62	0.62	0.72	0.72	0.76	0.77	0.80	0.82	0.64
li	0.44	0.57	0.72	0.81	0.84	0.86	0.91	0.91	0.93	0.93	0.93	0.93	0.69
mdljsp2	0.97	0.98	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	1.00	1.00	0.98
ora	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.46
swm256	0.99	0.99	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	0.26
tomcatv	0.41	0.48	0.59	0.63	0.63	0.63	0.71	0.71	0.77	0.78	0.85	0.85	0.72
hmean	0.56	0.69	0.80	0.84	0.84	0.85	0.92	0.92	0.93	0.93	0.94	0.95	0.63

Figure 8: Success rates of different jump prediction techniques

table entry and interfering with each other.

The second strategy involves procedure returns, the most common kind of indirect jump. If the machine can distinguish returns from other indirect jumps, it can do a better job of predicting their destinations, as follows. The machine maintains a small ring buffer of return addresses. Whenever it executes a subroutine call instruction, it increments the buffer pointer and enters the return address in the buffer. A return instruction is predicted to go to the last address in this buffer, and then decrements the buffer pointer. Unless we do tail-call optimization or setjmp/longjmp, this prediction will always be right if the machine uses a big enough ring buffer. Even if it cannot distinguish returns from other indirect jumps, their predominance might make it worth predicting that *any* indirect jump is a return, as long as we decrement the buffer pointer only when the prediction succeeds.

Our system allows several degrees of each kind of jump prediction. We can assume that indirect jumps are *perfectly predicted*. We can use the *cacheing prediction*, in which we predict that a jump will go wherever it went last time, with a table of any size. Subroutine returns can be predicted with this table, or with their own *return ring*, which can also be any desired size. We can also predict returns with a return ring and leave other indirect jumps unpredicted. Finally, we can assume *no jump prediction* whatsoever.

As with branches, a correctly predicted jump has no effect on the scheduling. A mispredicted or unpredicted jump may be moved before earlier instructions, but no later instruction can be moved before the jump.

Figure 8 shows the actual success rates of jump prediction using a return ring alone, of a return ring along with a last-destination table, and finally of prediction using a most-common-destination profile. Even a one-element return ring is enough to predict more than half the indirect jumps, and a slightly larger ring raises that to more than 80%. Adding a small last-destination table to predict non-returns produces a substantial improvement, although the success rate does not rise much as we make the table bigger. With only an 8-element return ring and a 2-element table, we can predict more than 90% of the indirect jumps. The most-common-destination profile, in contrast, succeeds only about two thirds of the time.

2.6 Window size and cycle width

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, issue those 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 et al. [SJH89] assumed discrete windows.

The *cycle width* is the maximum number of instructions that can be scheduled in a given cycle. By default this is 64. Our greedy scheduling algorithm works well when the cycle width is large: a small proportion of cycles are completely filled. For cycle widths of 2 or 4, however, a more traditional approach [HG83, JM82] would be more realistic.

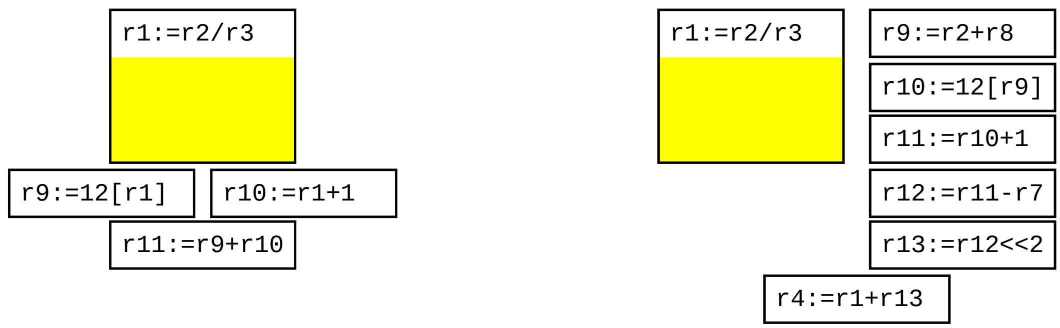
Along with cycles of a fixed finite size, we can specify that cycles are unlimited in width. In this case, there is still an effective limit imposed by the window size: if one cycle contains a window-full of instructions, it will be issued and a new cycle begun. As a final option, we therefore also allow *both* the cycle width and the window size to be unlimited.[5]

2.7 Latency

For most of our experiments we assumed that every operation had unit latency: any result computed in cycle n could be used as an operand in cycle $n + 1$. This can obviously be accomplished by setting the machine cycle time high enough for even the slowest of operations to finish in one cycle, but in general this is an inefficient use of the machine. A real machine is more likely to have a cycle time long enough to finish most common operations, like integer add, but let other operations (e.g. division, multiplication, floating-point operations, and memory loads) take more than one cycle to complete. If an operation in cycle t has latency L, its result cannot be used until cycle $t + L$.

Earlier we defined parallelism as the number of instructions executed divided by the number of cycles required. Adding non-unit latency requires that we refine that definition slightly. We want our measure of parallelism to give proper credit for scheduling quick operations during times when we are waiting for unrelated slow ones. We will define the *total latency* of a program as the sum of the latencies of all instructions executed, and the parallelism as the total latency divided

[5]The final possibility, limited cycle width but unlimited window size, cannot be implemented without using a data structure that can attain a size proportional to the length of the instruction trace. We deemed this impractical and did not implement it.



(a) multi-cycle instruction is on critical path

(b) multi-cycle instruction is not on critical path

Figure 9: Effects of increasing latency on parallelism

	model A	model B	model C	model D	model E
int add/sub, logical	1	1	1	1	1
load	1	1	2	2	3
int mult	1	2	2	3	5
int div	1	2	3	4	6
single-prec add/sub	1	2	3	4	4
single-prec mult	1	2	3	4	5
single-prec div	1	2	3	5	7
double-prec add/sub	1	2	3	4	4
double-prec mult	1	2	3	4	6
double-prec div	1	2	3	5	10

Figure 10: Operation latencies in cycles, under five latency models

by the number of cycles required. If all instructions have a latency of 1, the total latency is just the number of instructions, and the definition is the same as before. Notice that with non-unit latencies it is possible for the instruction-level parallelism to exceed the cycle width; at any given time we can be working on instructions issued in several different cycles, at different stages in the execution pipeline.

It is not obvious whether increasing the latencies of some operations will tend to increase or decrease instruction-level parallelism. Figure 9 illustrates two opposing effects. In 9(a), we have a divide instruction on the critical path; if we increase its latency we will spend several cycles working on nothing else, and the parallelism will decrease. In 9(b), in contrast, the divide is *not* on the critical path, and increasing its latency will increase the parallelism. Note that whether the parallelism increases or not, it is nearly certain that the time required is less, because an increase in the latency means we have decreased the cycle time.

We implemented five different latency models. Figure 10 lists them. We assume that the functional units are completely pipelined, so that even multi-cycle instructions can be issued every cycle, even though the result of each will not be available until several cycles later. Latency model A (all unit latencies) is the default used throughout this paper unless otherwise specified.

<table>
<tr><th></th><th>branch predict</th><th>jump predict</th><th>register renaming</th><th>alias analysis</th><th></th></tr>
<tr><td>Stupid</td><td>none</td><td rowspan="2">none</td><td rowspan="3">none</td><td>none</td><td>Stupid</td></tr>
<tr><td>Poor</td><td>64b counter</td><td>inspect</td><td>Poor</td></tr>
<tr><td>Fair</td><td>2Kb ctr/gsh</td><td>16-addr ring, no table</td><td rowspan="5">perfect</td><td>Fair</td></tr>
<tr><td>Good</td><td>16Kb loc/gsh</td><td>16-addr ring, 8-addr table</td><td>64</td><td>Good</td></tr>
<tr><td>Great</td><td>152Kb loc/gsh</td><td rowspan="2">2K-addr ring, 2K-addr table</td><td rowspan="2">256</td><td>Great</td></tr>
<tr><td>Superb</td><td>fanout 4, then 152Kb loc/gsh</td><td>Superb</td></tr>
<tr><td>Perfect</td><td>perfect</td><td>perfect</td><td>perfect</td><td>Perfect</td></tr>
</table>

Figure 11: Seven increasingly ambitious models

3 Results

We ran our eighteen test programs under a wide range of configurations. We will present some of these to show interesting trends; the complete results appear in an appendix. To provide a framework for our exploration, we defined a series of seven increasingly ambitious models spanning the possible range. These seven are specified in Figure 11; in all of them the window size is 2K instructions, the cycle width is 64 instructions, and unit latencies are assumed. Many of the results we present will show the effects of variations on these standard models. Note that even the “Poor” model is fairly ambitious: it assumes rudimentary alias analysis and a branch predictor that is 85% correct on average, and like all seven models it allows our generous default window size and cycle width.

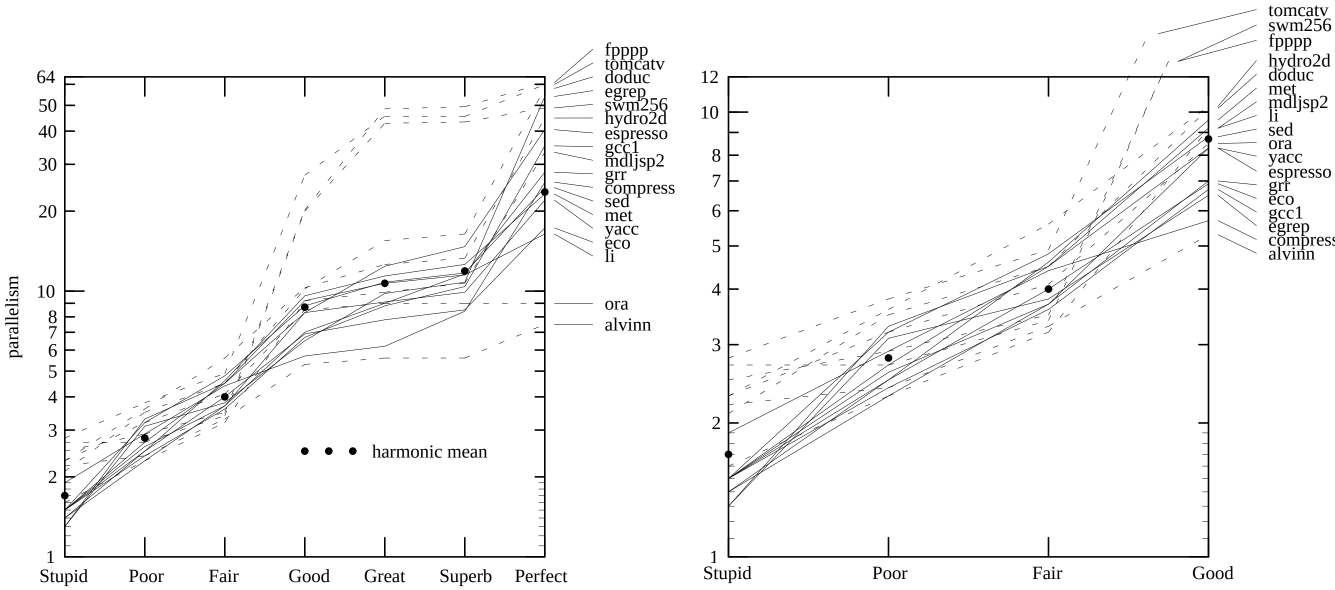


Figure 12: Parallelism under the seven models, full-scale (left) and detail (right)

3.1 Parallelism under the seven models

Figure 12 shows the parallelism of each program for each of the seven models. The numeric programs are shown as dotted lines, the harmonic mean by a series of circles. Unsurprisingly, the Stupid model rarely exceeds 3, and exceeds 2 only for some of the numeric programs. 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. Moving up to Poor helps the worst programs quite a lot, almost entirely because of the branch prediction, but the mean is still under 3. Moving to Fair increases the mean to 4, mainly because we suddenly assume perfect alias analysis. The Good model doubles the mean parallelism, mostly because it introduces some register renaming. Increasing the number of available registers in the Great model takes us further, though the proportional improvement is smaller. At this point the effectiveness of branch prediction is topping out, so we add 4-way branch fanout to the Great model to get the Superb model. Its performance, however, is disappointing; we had hoped for more of an improvement. The parallelism of the Superb model is less than half that of the Perfect model, mainly because of the imperfection of its branch prediction. A study using the Perfect model alone would lead us down a dangerous garden path, as would a study that included only fpppp and tomcatv.

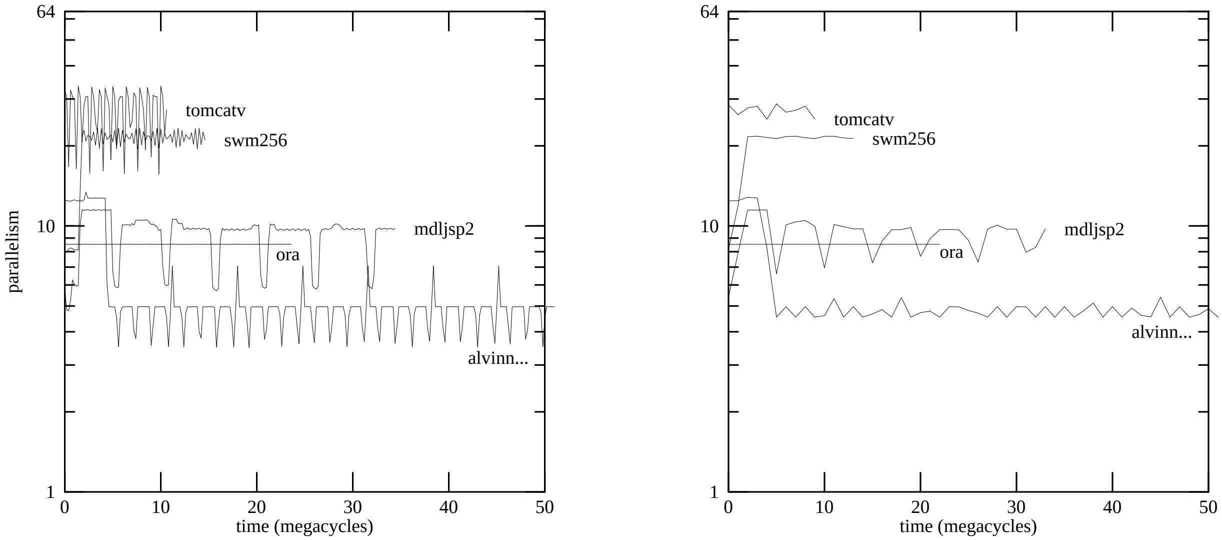


Figure 13: Parallelism under the Good model over intervals of 0.2 million cycles (left) and 1 million cycles (right)

3.2 Effects of measurement interval

We analyzed the parallelism of entire program executions because it avoided the question of what constitutes a “representative” interval. To select some smaller interval of time at random would run the risk that the interval was atypical of the program’s execution as a whole. To select a particular interval where the program is at its most parallel would be misleading and irresponsible. Figure 13 shows the parallelism under the Good model during successive intervals from the execution of some of our longer-running programs. The left-hand graph uses intervals of 200,000 cycles, the right-hand graph 1 million cycles. In each case the parallelism of an interval is computed exactly like that of a whole program: the number of instructions executed during that interval is divided by the number of cycles required.

Some of the test programs are quite stable in their parallelism. Others are quite unstable. With 200K-cycle intervals (which range from 0.7M to more than 10M instructions), the parallelism within a single program can vary widely, sometimes by a factor of three. Even 1M-cycle intervals see variation by a factor of two. The alvinn program has parallelism above 12 for 4 megacycles, at which point it drops down to less than half that; in contrast, the swm256 program starts quite low and then climbs to quite a respectable number indeed.

It seems clear that intervals of a million cycles would not be excessive, and even these should be selected with care. Parallelism measurements for isolated intervals of fewer than a million cycles should be viewed with suspicion and even derision.

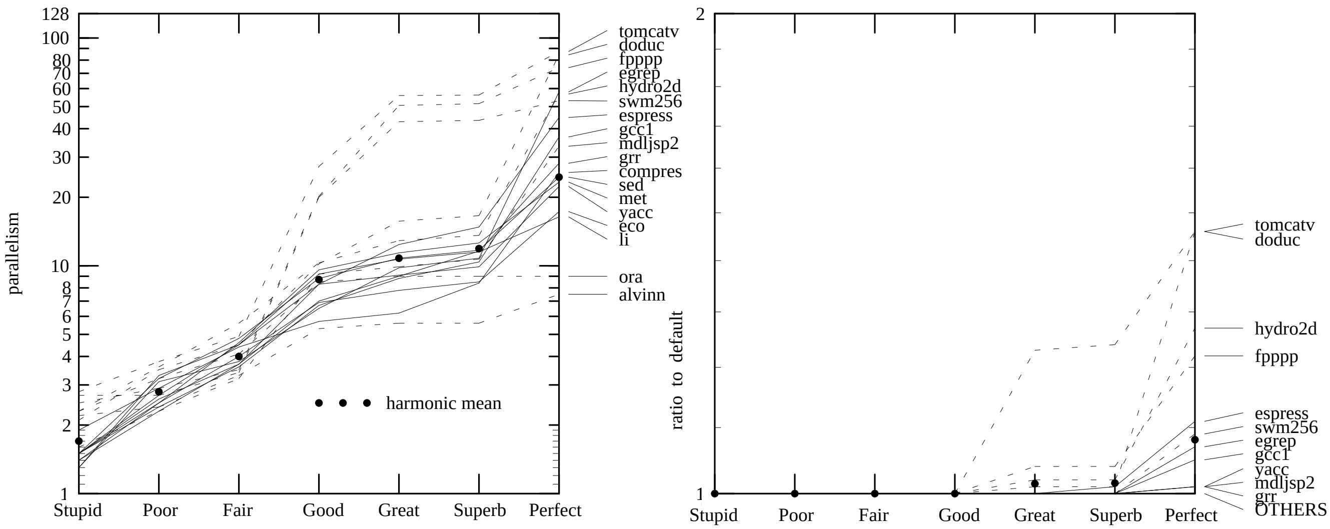


Figure 14: Parallelism under the seven models with cycle width of 128 instructions (left), and the ratio of parallelism for cycles of 128 to parallelism for cycles of 64 (right)

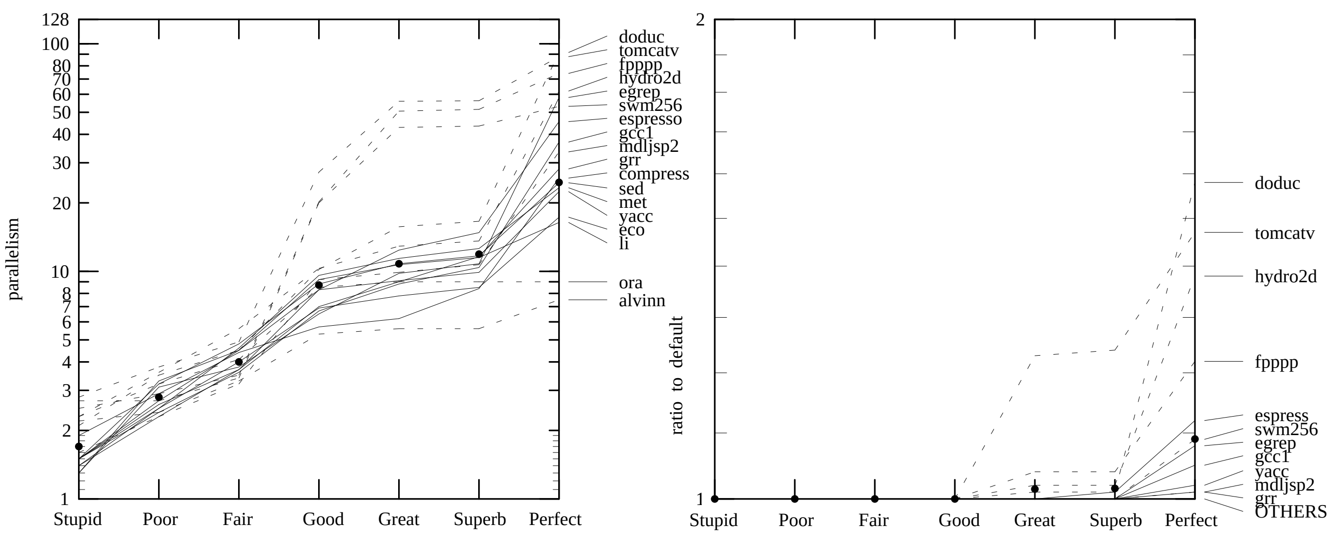


Figure 15: Parallelism under the seven models with unlimited cycle width (left), and the ratio of parallelism for unlimited cycles to parallelism for cycles of 64 (right)

3.3 Effects of cycle width

Tomcatv and fpppp attain very high parallelism with even modest machine models. Their average parallelism is very close to the maximum imposed by our normal cycle width of 64 instructions; under the Great model more than half the cycles of each are completely full. This suggests that even more parallelism might be obtained by widening the cycles. Figure 14 shows what happens if we increase the maximum cycle width from 64 instructions to 128. The right-hand graph shows how the parallelism increases when we go from cycles of 64 instructions to cycles of 128. Doubling the cycle width improves four of the numeric programs appreciably under the Perfect model, and improves tomcatv by 20% even under the Great model. Most programs, however, do not benefit appreciably from such wide cycles even under the Perfect model.

Perhaps the problem is that even 128-instruction cycles are too small. If we remove the limit on cycle width altogether, we effectively make the cycle width the same as the window size, in this case 2K instructions. The results are shown in Figure 15. Parallelism in the Perfect model is a bit better than before, but outside the Perfect model we see that tomcatv is again the only benchmark to benefit significantly.

Although even a cycle width of 64 instructions is quite a lot, we did not consider smaller cycles. This would have required us to replace our quick and easy greedy scheduling algorithm with a slower conventional scheduling technique[GM86, HG83], limiting the programs we could run to completion. Moreover, these techniques schedule a static block of instructions, and it is not obvious how to extend them to the continuous windows model.

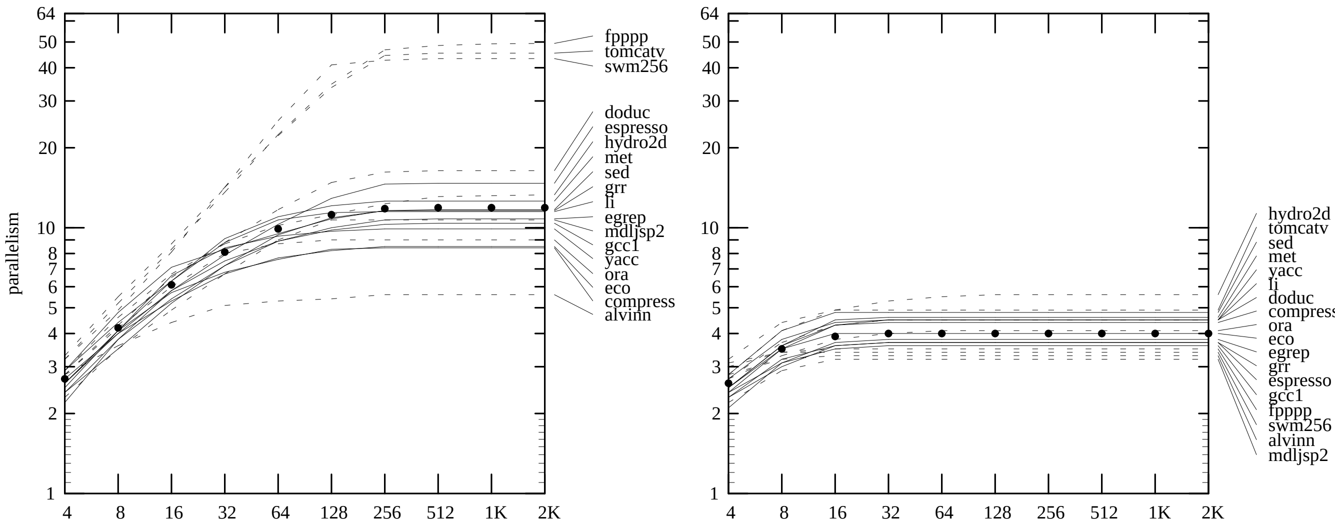


Figure 16: Parallelism for different sizes of continuously-managed windows under the Superb model (left) and the Fair model (right)

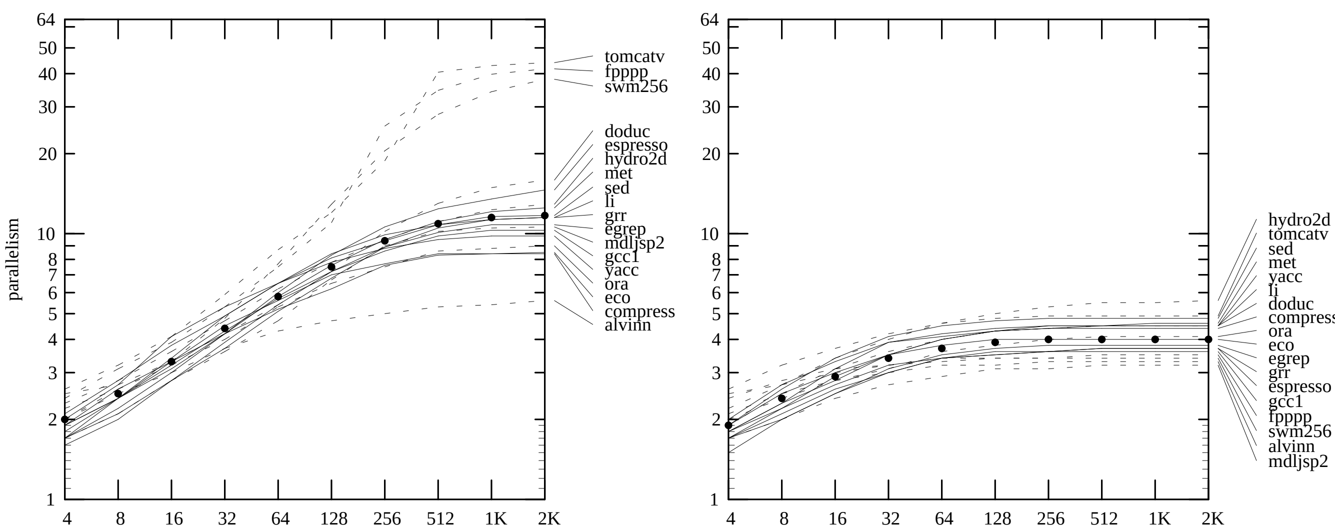


Figure 17: Parallelism for different sizes of discretely-managed windows under the Superb model (left) and the Fair model (right)

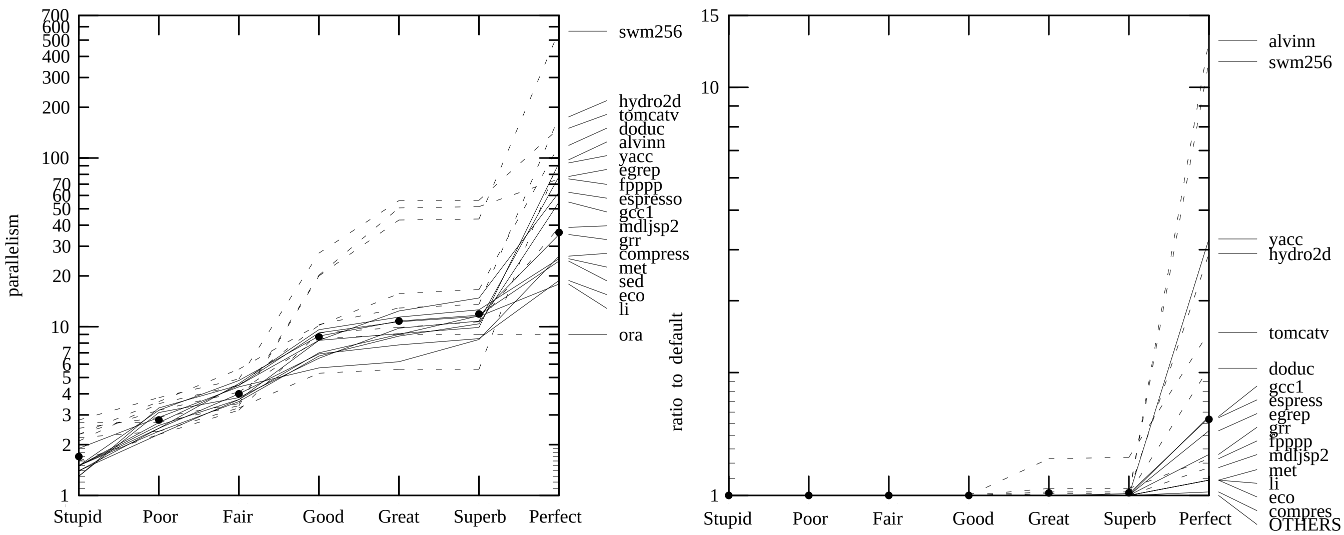


Figure 18: Parallelism under the seven models with unlimited window size and cycle width (left), and the ratio of parallelism for unlimited windows and cycles to parallelism for 2K-instruction windows and 64-instruction cycles (right)

3.4 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 16 shows the effect of varying the window size from 2K instructions down to 4, for the Superb and Fair models. Under the Superb model, most programs do about as well with a 128-instruction window as with a larger one. Below that, parallelism drops off quickly. The Poor model's limited analysis severely restricts the mobility of instructions; parallelism levels off at a window size of only 16 instructions.

A less ambitious parallelism manager would manage windows discretely, by getting a window full of instructions, scheduling them relative to each other, executing them, and then starting over with a fresh window. This would tend to result in lower parallelism than in the continuous window model we used above. Figure 17 shows the same models as Figure 16, but assumes discrete windows rather than continuous. As we might expect, discretely managed windows need to be larger to be at their best: most curves don't level off until a window size of 512 instructions (for Superb) or 64 instructions (for Poor) is reached. As with continuous windows, sizes above 512 instructions do not seem to be necessary. If we have very small windows, continuous management does as much good as multiplying the window size by 4.

If we eliminate the limit on both the window size and the cycle width, we get the results shown in Figure 18. Here we finally see the kind of high parallelism reported in studies like Nicolau and

Fisher's [NF84], reaching as high as 500 for swm256 in the Perfect model. It is interesting to note that under even slightly more realistic models, the maximum parallelism drops to around 50, and the mean parallelism to around 10. The advantage of unlimited window size and cycle width outside the Perfect model shows up only on tomcatv, and even there the advantage is modest.

3.5 Effects of loop unrolling

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 or the inherent sequentiality of the loop control.

We studied the parallelism of unrolled code by manually unrolling four inner loops in three of our programs. In each case these loops constituted a sizable fraction of the original program's total runtime. Figure 19 displays some details.

Alvinn has two inner loops, the first of which is an accumulator loop: each iteration computes a value on which no other iteration depends, but these values are successively added into a single accumulator variable. To parallelize this loop we duplicated the loop body n times (with i successively replaced by $i+1$, $i+2$, and so on wherever it occurred), collapsed the n assignments to the accumulator into a single assignment, and then restructured the resulting large right-hand-side into a balanced tree expression.

	procedure	loop location	type of loop	instrs	% of execution
alvinn	input_hidden	line 109 of backprop.c	accumulator	14	39.5%
	hidden_input	line 192 of backprop.c	independent	14	39.5%
swm256	CALC2	line 325 of swm256.f	independent	62	37.8%
tomcatv	main	line 86 of tomcatv.f	independent	258	67.6%

Figure 19: Four unrolled loops

The remaining three loops are perfectly parallelizable: each iteration is completely independent of the rest. We unrolled these in two different ways. In the first, we simply duplicated the loop body n times (replacing i by $i+1$, $i+2$, and so on). In the second, we duplicated the loop body n times as before, changed all assignments to array members into assignments to local scalars followed by moves from those scalars into the array members, and finally moved all the assignments to array members to the end of the loop. The first model should not work as well as the second in simple models with poor alias analysis, because the array loads from successive unrolled iterations are separated by array stores; it is difficult to tell that it is safe to interleave parts

```
for i := 1 to N do
    accum := accum + f(i);
end
```

Accum loop →

```
for i := 1 to N by 4 do
    accum := accum
             + ( ( f(i)  + f(i+1))
               + (f(i+2) + f(i+3))
               );
end
```

```
for i := 1 to N do
    a[i] := f(i);
    b[i] := g(i);
end
```

for sophisticated models →

```
for i := 1 to N by 4 do
    a[i] := f(i);
    b[i] := g(i);
    a[i+1] := f(i+1);
    b[i+1] := g(i+1);
    a[i+2] := f(i+2);
    b[i+2] := g(i+2);
    a[i+3] := f(i+3);
    b[i+3] := g(i+3);
end
```

for simple models →

```
for i := 1 to N by 4 do
    a00 := f(i);
    b00 := g(i);
    a01 := f(i+1);
    b01 := g(i+1);
    a02 := f(i+2);
    b02 := g(i+2);
    a03 := f(i+3);
    b03 := g(i+3);
    a[i] := a00;
    b[i] := b00;
    a[i+1] := a01;
    b[i+1] := b01;
    a[i+2] := a02;
    b[i+2] := b02;
    a[i+3] := a03;
    b[i+3] := b03;
end
```

Figure 20: Three techniques for loop unrolling

of successive iterations. On the other hand, leaving the stores in place means that the lifetimes of computed values are shorter, allowing the compiler to do a better job of register allocation: moving the stores to the end of the loop means the compiler is more likely to start using memory locations as temporaries, which removes these values from the control of the register renaming facility available in the smarter models.

Figure 20 shows examples for all three transformations. We followed each unrolled loop by a copy of the original loop starting where the unrolled loop left off, to finish up in the cases where the loop count was not a multiple of the unrolling factor.

In fact there was very little difference between these methods for the poorer models, and the differences for the better models were not all in the same direction. In the results reported below, we always used the larger parallelism obtained using the two different methods. Figure 21 shows the result. Unrolling made a profound difference to alvinn under the better models, though the

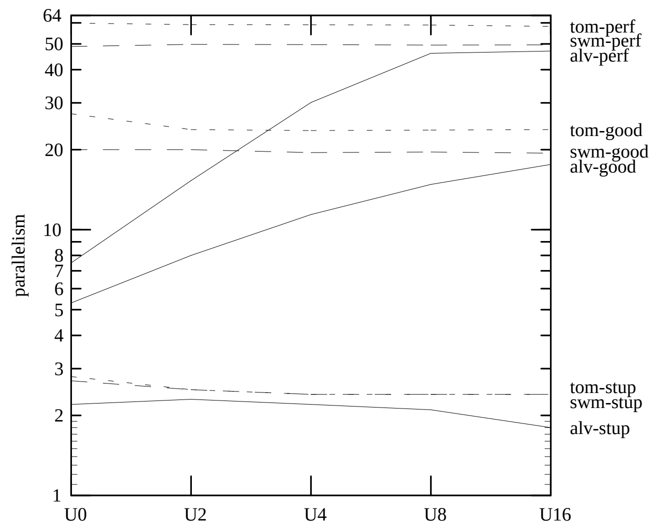


Figure 21: Effects of loop unrolling

effect decreased as the unrolling factor increased. It made little difference in the other cases, and even hurt the parallelism in several instances. This difference is probably because the inner loop of alvinn is quite short, so it can be replicated several times without creating great register pressure or otherwise giving the compiler too many balls to juggle.

Moreover, it is quite possible for parallelism to go down while performance goes up. The rolled loop can do the loop bookkeeping instructions in parallel with the meat of the loop body, but an unrolled loop gets rid of at least half of that bookkeeping. Unless the unrolling creates new opportunities for parallelism (which is of course the point) this will cause the net parallelism to decrease.

Loop unrolling is a good way to increase the available parallelism, but it is not a silver bullet.

3.6 Effects of branch prediction

We saw earlier that new techniques of dynamic history-based branch prediction allow us to benefit from quite large branch predictor, giving success rates that are still improving slightly even when we are using a 1-megabit predictor. It is natural to ask how much this affects the instruction-level parallelism. Figure 22 answers this question for the Fair and Great models. The Fair model is relatively insensitive to the size of the predictor, though even a tiny 4-bit predictor improves the mean parallelism by 50%. A tiny 4-bit table doubles the parallelism under the Great model, and increasing that to a huge quarter-megabit table more than doubles it again.

Even under the Great model, the three most parallel programs are quite insensitive to the size of the predictor. These are exactly the programs in which conditional branches account for no more than 2% of the instructions executed; the nearest contender is doduc with 6%. We can mask this effect by plotting these results not as a function of the predictor size, but as a function of

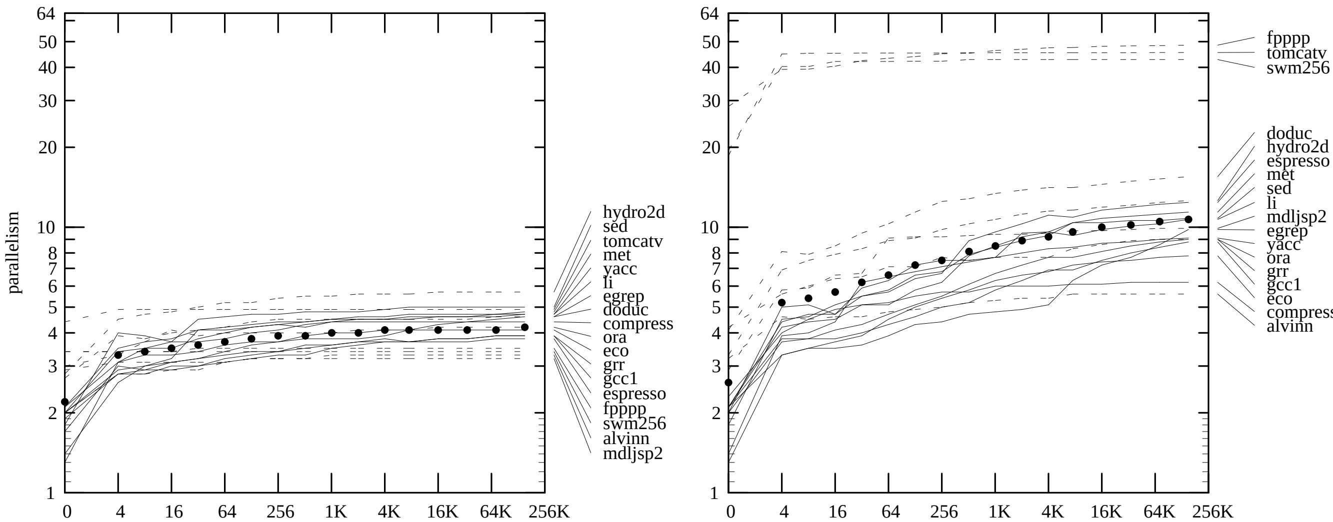


Figure 22: Parallelism for different sizes of dynamic branch-prediction table under the Fair model (left) and the Great model (right)

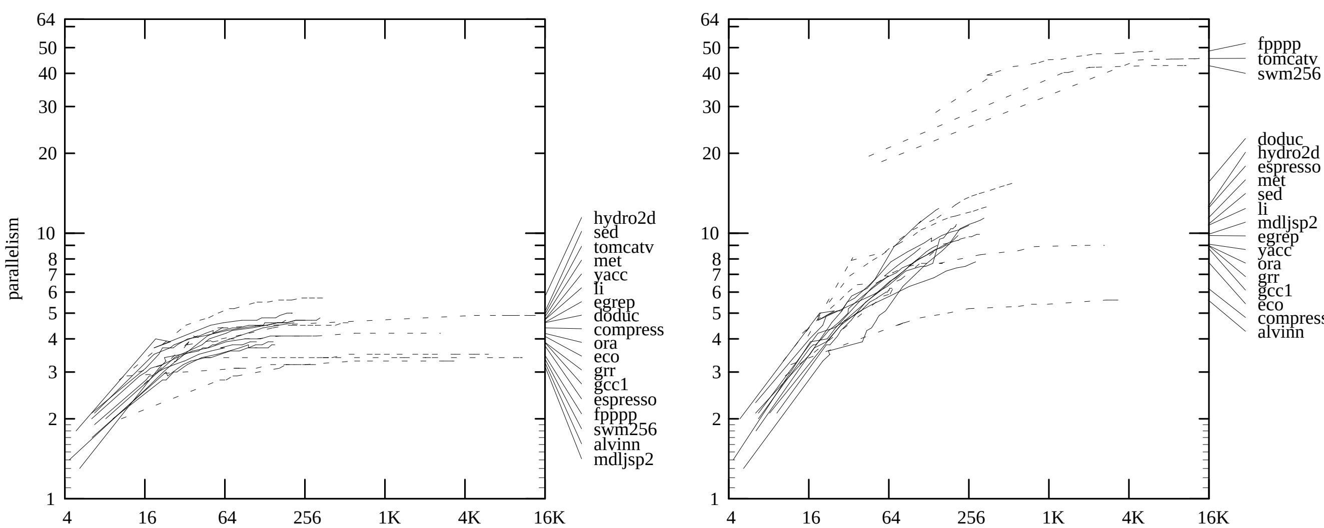


Figure 23: Parallelism as a function of the mean number of instructions between mispredicted branches, under the Fair model (left) and the Great model (right)

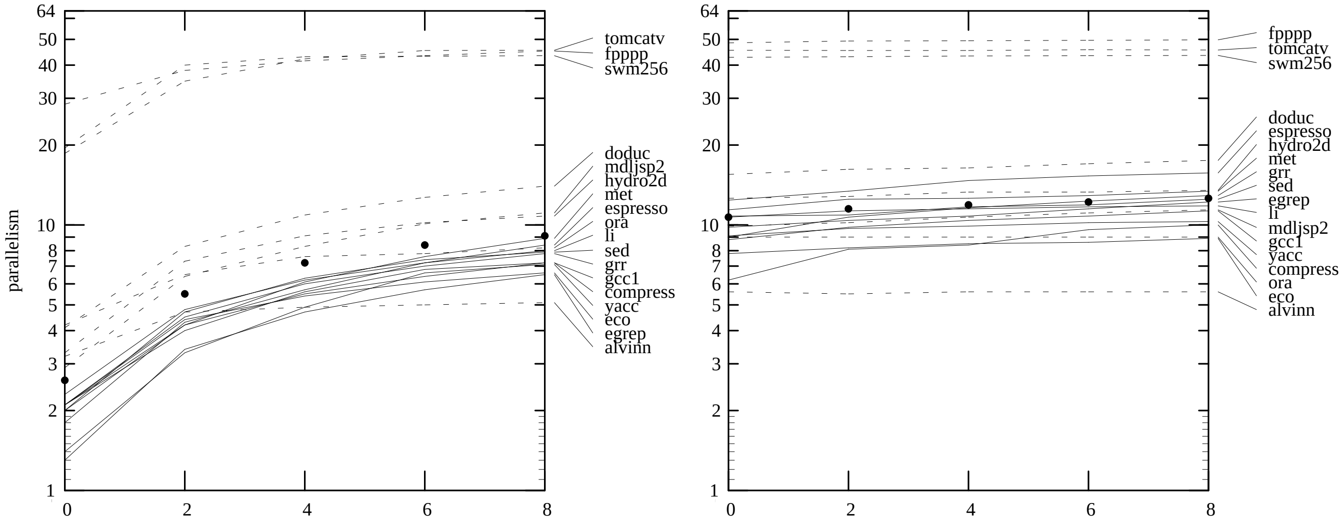


Figure 24: Parallelism for the Great model with different levels of fanout scheduling across conditional branches, with no branch prediction (left) and with branch prediction (right) after fanout is exhausted

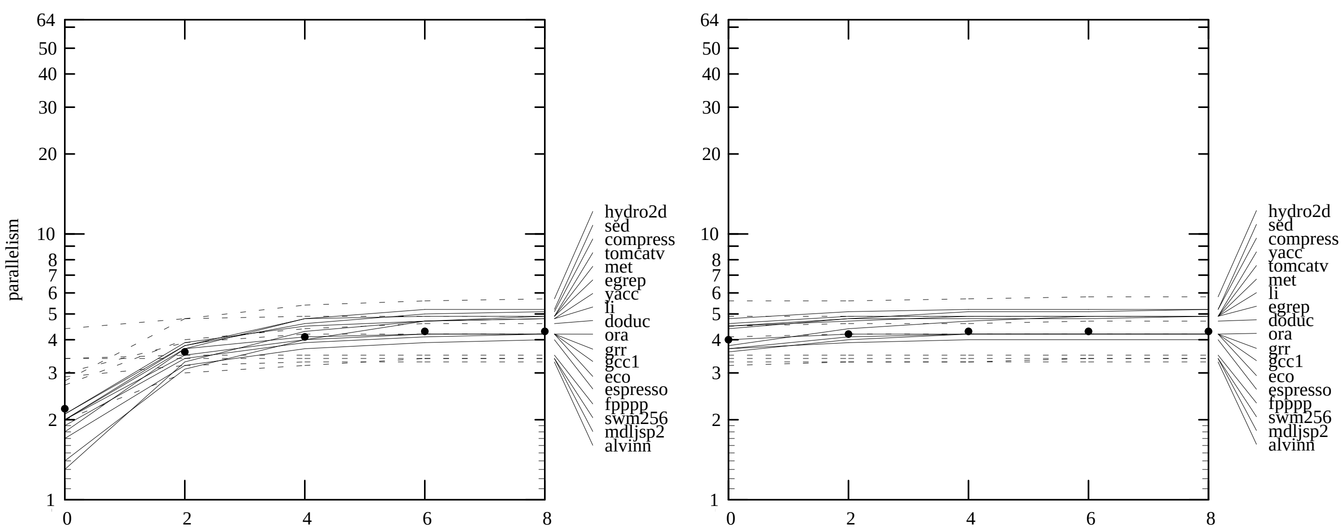


Figure 25: Parallelism for the Fair model with different levels of fanout scheduling across conditional branches, with no branch prediction (left) and with branch prediction (right) after fanout is exhausted

the average number of instructions executed between mispredicted branches. Figure 23 shows these results. These three numeric programs still stand out as anomalies; evidently there is more involved in their parallelism than the infrequency and predictability of their branches.

3.7 Effects of fanout

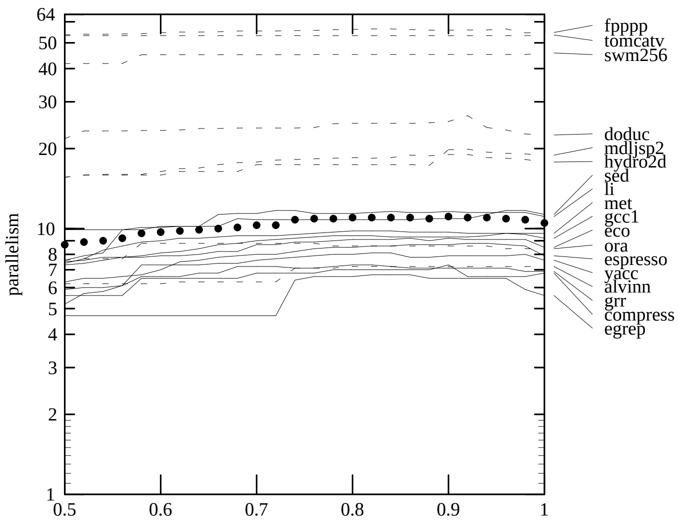
Figures 24 and 25 show the effects of adding various levels of fanout to the Great and Fair models. The left-hand graphs assume that we look along both paths out of the next few conditional branches, up to the fanout limit, but that we do not look past branches beyond that point. The right-hand graphs assume that after we reach the fanout limit we use dynamic prediction (at the Great or Fair level) to look for instructions from the one predicted path to schedule. In each graph the leftmost point represents no fanout at all. We can see that when fanout is followed by good branch prediction, the fanout does not buy us much. Without branch prediction, on the other hand, even modest amounts of fanout are quite rewarding: adding fanout across 4 branches to the Fair model is about as good as adding Fair branch prediction.

Fisher [Fis91] has proposed using fanout in conjunction with profiled branch prediction. In this scheme they are both under software control: the profile gives us information that helps us to decide whether to explore a given branch using the fanout capability or using a prediction. This is possible because a profile can easily record not just which way a branch most often goes, but also how often it does so. Fisher combines this profile information with static scheduling information about the payoff of scheduling each instruction early on the assumption that the branch goes in that direction.

Our traces do not have the payoff information Fisher uses, but we can investigate a simpler variation of the idea. We pick some threshold to partition the branches into two classes: those we predict because they are very likely to go one way in particular and those at which we fan out because they are not. We will call this scheme *profile-guided integrated prediction and fanout*.

We modified the Perfect model to do profile-guided integrated prediction and fanout. Figure 26 shows the parallelism for different threshold levels. Setting the threshold too low means that we use the profile to predict most branches and rarely benefit from fanout: a threshold of 0.5 causes all branches to be predicted with no use of fanout at all. Setting the threshold too high means that you fan out even on branches that nearly always go one way, wasting the hardware parallelism that fanout enables. Even a very high threshold is better than none, however; some branches really do go the same way all or essentially all of the time. The benefit is not very sensitive to the threshold we use: between 0.75 and 0.95 most of the curves are quite flat; this holds as well if we do the same experiment using the Great model or the Fair model. The best threshold seems to be around 0.92.

Figure 27 shows the parallelism under variations of the Fair and Superb models, first with the profile-integrated scheme and next with the hardware approach of fanout followed by prediction. Profile-guided integration works about as well as the simple hardware approach under the Fair model, in spite of the fact that the Fair model has a predictor that is about 5% better than a profile predictor. The better hardware branch prediction of the Superb model, however, completely outclasses the profile-integrated approach.



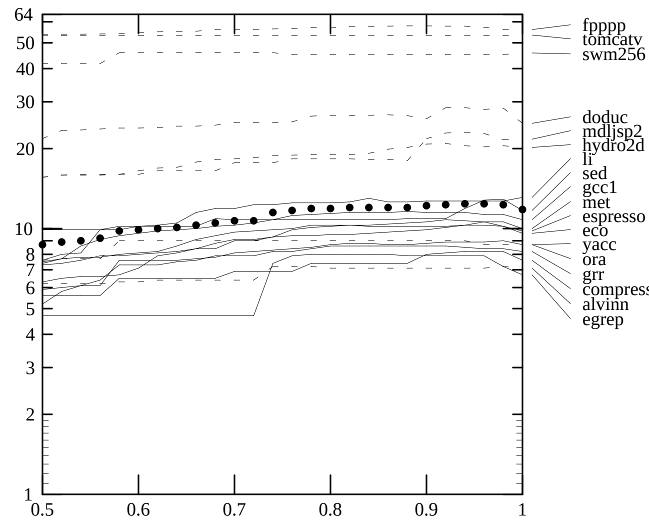


Figure 26: Parallelism for the Perfect model with profile-guided integrated prediction and fanout, for fanout 2 (left) and fanout 4 (right)

	Fair				Superb			
	fanout 2		fanout 4		fanout 2		fanout 4	
	Integr	Hardware	Integr	Hardware	Integr	Hardware	Integr	Hardware
egrep	4.7	4.4	4.9	4.7	6.5	10.4	7.8	10.8
sed	5.0	5.1	5.0	5.2	10.3	10.9	10.3	11.7
yacc	4.7	4.7	4.8	4.9	7.7	9.7	8.6	9.9
eco	4.2	4.2	4.2	4.2	6.9	8.2	7.4	8.5
grr	4.0	4.1	4.2	4.2	7.0	10.7	8.5	11.6
metronome	4.8	4.9	4.8	4.9	9.2	12.5	10.1	12.6
alvinn	3.3	3.3	3.3	3.3	5.5	5.5	5.5	5.6
compress	4.6	4.8	5.0	5.1	6.6	8.1	8.0	8.4
doduc	4.6	4.6	4.7	4.6	15.7	16.2	16.8	16.4
espresso	3.8	3.9	3.9	4.0	8.8	13.4	10.7	14.7
fpppp	3.5	3.5	3.5	3.5	47.6	49.3	48.8	49.4
gcc1	4.1	4.0	4.2	4.2	8.0	9.8	9.3	10.4
hydro2d	5.7	5.6	5.8	5.7	11.6	12.8	12.3	13.3
li	4.7	4.8	4.8	4.8	9.4	11.3	10.1	11.5
mdljsp2	3.3	3.3	3.3	3.3	10.1	10.2	10.8	10.7
ora	4.2	4.2	4.2	4.2	8.6	9.0	9.0	9.0
swm256	3.4	3.4	3.4	3.4	42.8	43.0	42.8	43.3
tomcatv	4.9	4.9	4.9	4.9	45.3	45.4	45.3	45.4
har. mean	4.2	4.2	4.3	4.3	9.5	11.5	10.5	11.9

Figure 27: Parallelism under Fair and Superb models with fanout 2 or 4, using either profile-guided integrated fanout and prediction or the unintegrated hardware technique

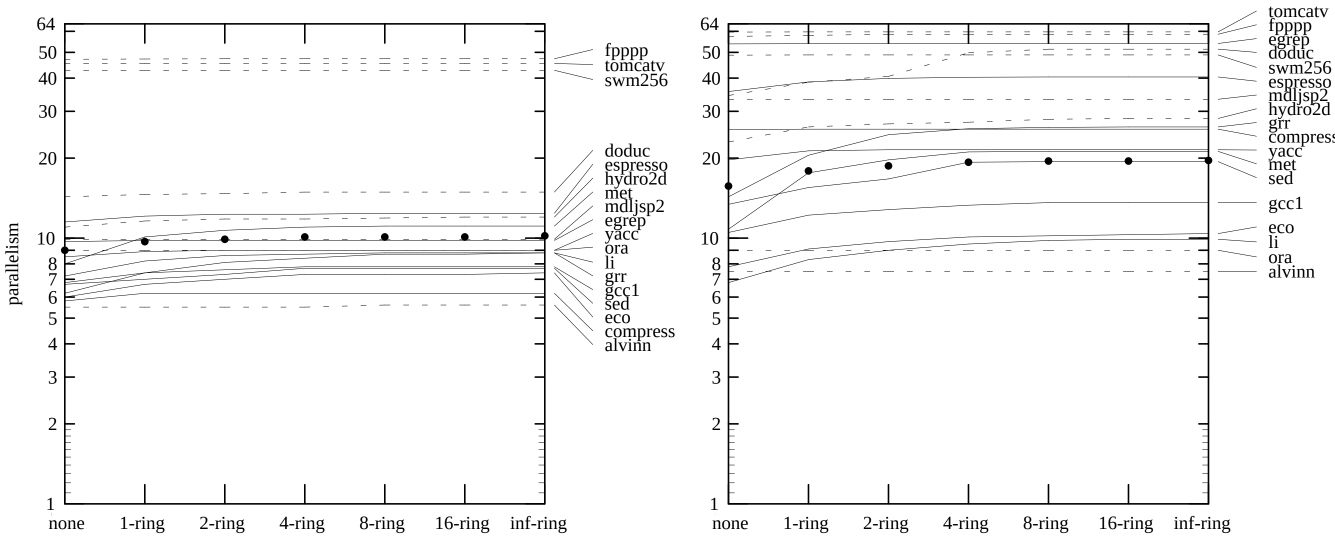


Figure 28: Parallelism for varying sizes of return-prediction ring with no other jump prediction, under the Great model (left) and the Perfect model (right)

3.8 Effects of jump prediction

Subroutine returns are easy to predict well, using the return-ring technique discussed in Section 2.5. Figure 28 shows the effect on the parallelism of different sizes of return ring and no other jump prediction. The leftmost point is a return ring of no entries, which means no jump prediction at all. A small return-prediction ring improves some programs a lot, even under the Great model. A large return ring, however, is not much better than a small one.

We can also predict indirect jumps that are not returns by cacheing their previous destinations and predicting that they will go there next time. Figure 29 shows the effect of predicting returns with a 2K-element return ring and all other indirect jumps with such a table. The mean behavior is quite flat as the table size increases, but a handful of programs do benefit noticeably even from a very small table.

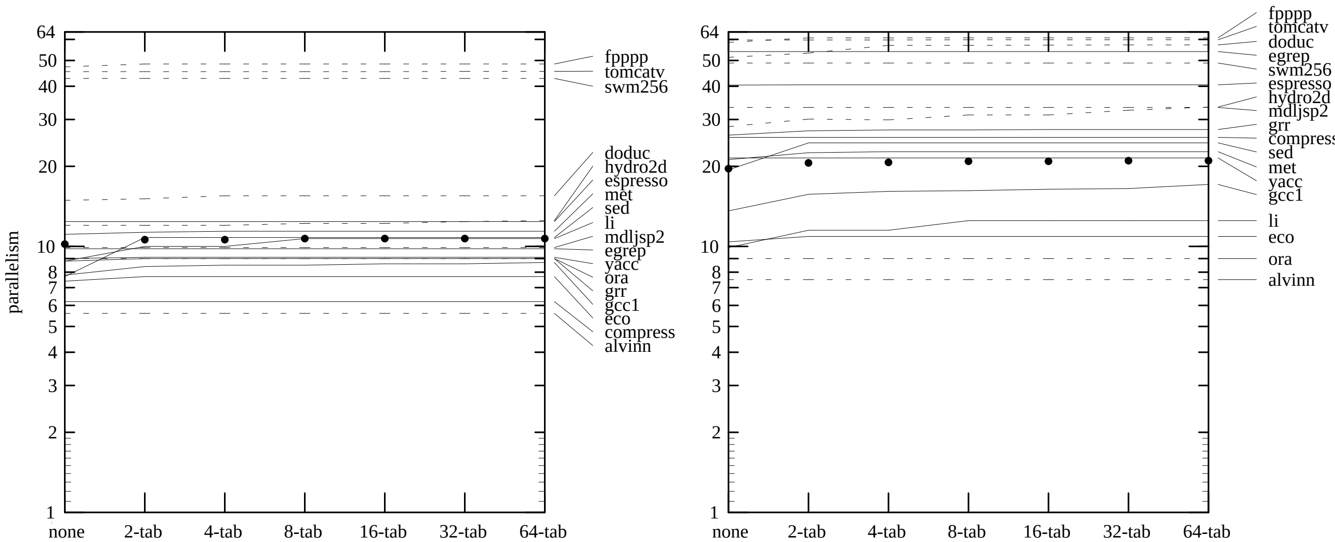


Figure 29: Parallelism for jump prediction by a huge return ring and a destination-cache table of various sizes, under the Great model (left) and the Perfect model (right)

3.9 Effects of a penalty for misprediction

Even when branch and jump prediction have little effect on the parallelism, it may still be worthwhile to include them. In a pipelined machine, a branch or jump predicted incorrectly (or not at all) results in a bubble in the pipeline. This bubble is a series of one or more cycles in which no execution can occur, during which the correct instructions are fetched, decoded, and started down the execution pipeline. The size of the bubble is a function of the pipeline granularity, and applies whether the prediction is done by hardware or by software. This penalty can have a serious effect on performance. Figure 30 shows the degradation of parallelism under the Poor and Good models, assuming that each mispredicted branch or jump adds N cycles with no instructions in them. The Poor model deteriorates quickly because it has limited branch prediction and no jump prediction. The Good model is less affected because its prediction is better. Under the Poor model, the negative effect of misprediction can be greater than the positive effects of multiple-issue, resulting in a parallelism under 1.0. Without the multiple issue, of course, the behavior would be even worse.

The most parallel numeric programs stay relatively horizontal over the entire range. As shown in Figure 31, this is because they make fewer branches and jumps, and those they make are comparatively predictable. Increasing the penalty degrades these programs less than the others because they make relatively few jumps; in tomcatv, fewer than one instruction in 30000 is an indirect jump. For most programs, however, a high misprediction penalty can result in “speedups” that are negligible, even when the non-bubble cycles are highly parallel.

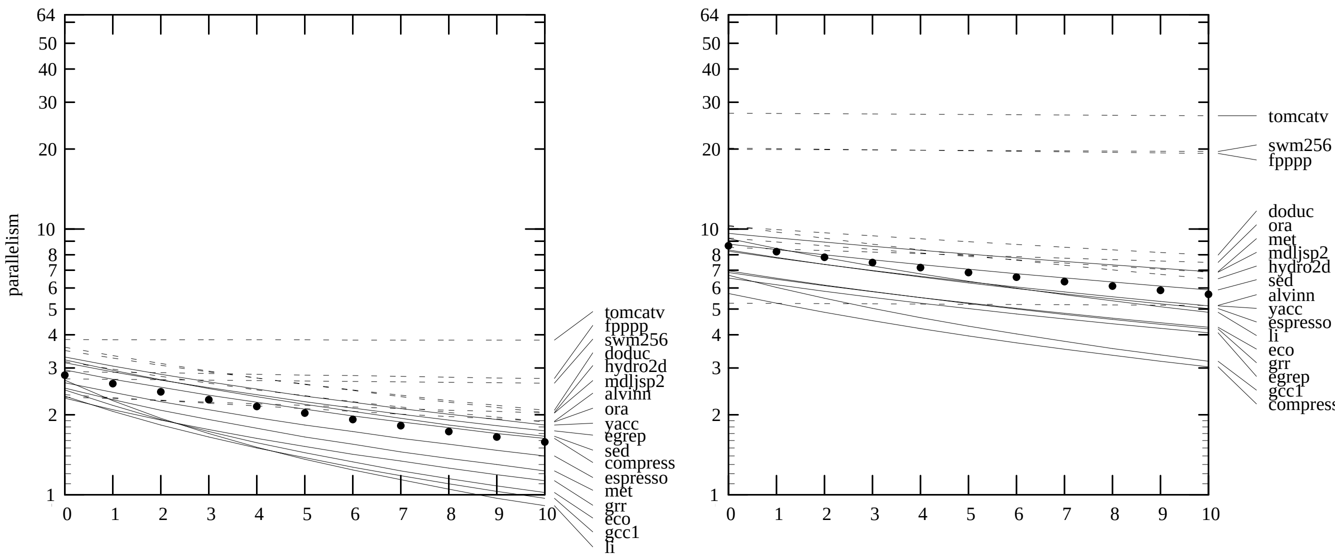


Figure 30: Parallelism as a function of misprediction penalty for the Poor model (left) and the Good model (right)

	branches	jumps
egrep	19.3%	0.1%
yacc	23.2%	0.5%
sed	20.6%	1.3%
eco	15.8%	2.2%
grr	10.9%	1.5%
met	12.3%	2.1%
alvinn	8.6%	0.2%
compress	14.9%	0.3%
doduc	6.3%	0.9%
espresso	15.6%	0.5%
fpppp	0.7%	0.1%
gcc1	15.0%	1.6%
hydro2d	9.8%	0.7%
li	15.7%	3.7%
mdljsp2	9.4%	0.03%
ora	7.0%	0.7%
swm256	2.2%	0.1%
tomcatv	1.8%	0.003%

Figure 31: Dynamic ratios of conditional branches and indirect jumps to all instructions

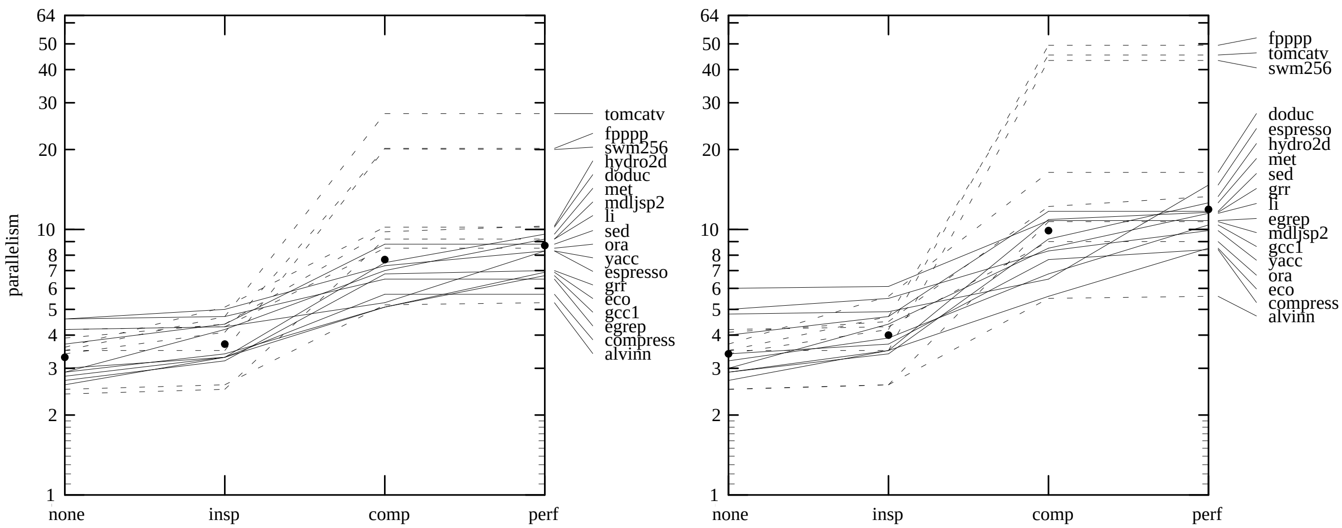


Figure 32: Parallelism for different levels of alias analysis, under the Good model (left) and the Superb model (right)

3.10 Effects of alias analysis

Figure 32 shows that "alias analysis by inspection" is better than none, though it rarely increased parallelism by more than a quarter. "Alias analysis by compiler" was (by definition) identical to perfect alias analysis on programs that do not use the heap. On programs that do use the heap, it improved the parallelism by 75% or so (by 90% under the Superb model) over alias analysis by inspection. Perfect analysis improved these programs by another 15 to 20 percent over alias analysis by compiler, suggesting that there would be a payoff from further results on heap disambiguation.

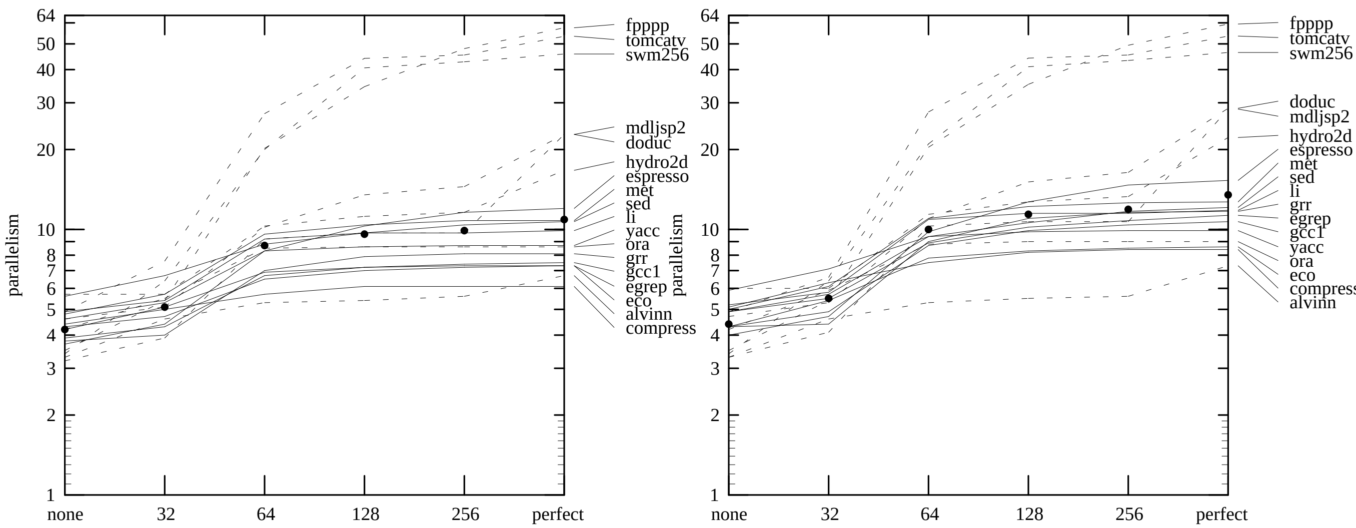


Figure 33: Parallelism for different numbers of dynamically-renamed registers, under the Good model (left) and the Superb model (right)

3.11 Effects of register renaming

Figure 33 shows the effect of register renaming on parallelism. Dropping from infinitely many registers to 128 CPU and 128 FPU had little effect on the parallelism of the non-numerical programs, though some of the numerical programs suffered noticeably. Even 64 of each did not do too badly.

The situation with 32 of each, the actual number on the DECstation 5000 to which the code was targeted, is interesting. Adding renaming did not improve the parallelism much, and in fact degraded it in a few cases. With so few real registers, hardware dynamic renaming offers little over a reasonable static allocator.

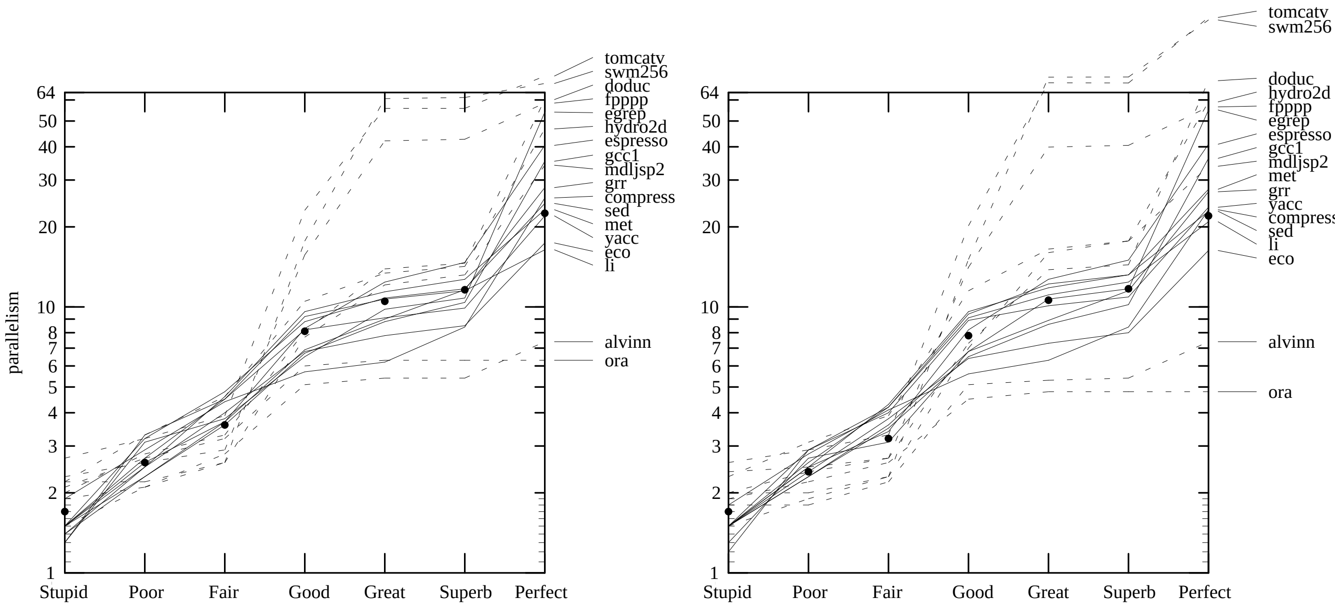


Figure 34: Parallelism under the seven standard models with latency model B (left) and latency model D (right)

3.12 Effects of latency

Figure 34 shows the parallelism under our seven models for two of our latency models, B and D. As discussed in Section 2.7, increasing the latency of some operations could act either to increase parallelism or to decrease it. In fact there is surprisingly little difference between these graphs, or between either and Figure 12, which is the default (unit) latency model A. Figure 35 looks at this picture from another direction, considering the effect of changing the latency model but keeping the rest of the model constant. The bulk of the programs are insensitive to the latency model, but a few have either increased or decreased parallelism with greater latencies.

Doduc and *fpppp* are interesting. As latencies increase, the parallelism oscillates, first decreasing, then increasing, then decreasing again. This behavior probably reflects the limited nature of our assortment of latency models: they do not represent points on a single spectrum but a small sample of a vast multi-dimensional space, and the path they represent through that space jogs around a bit.

4 Conclusions

Superscalar processing has been acclaimed as “vector processing for scalar programs,” and there appears to be some truth in the claim. Using nontrivial but currently known techniques, we consistently got parallelism between 4 and 10 for most of the programs in our test suite. Vectorizable or nearly vectorizable programs went much higher.

Speculative execution driven by good branch prediction is critical to the exploitation of more

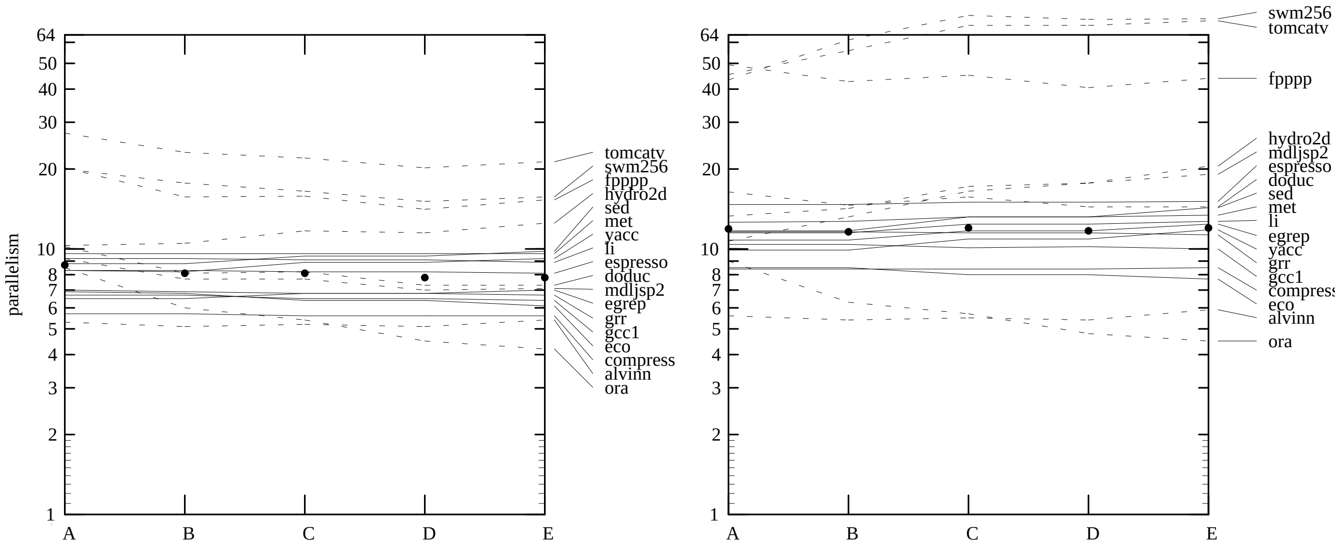


Figure 35: Parallelism under the Good model (left) and the Superb model (right), under the five different latency models

than modest amounts of instruction-level parallelism. If we start with the Perfect model and remove branch prediction, the median parallelism plummets from 30.6 to 2.2, while removing alias analysis, register renaming, or jump prediction results in more acceptable median parallelism of 3.4, 4.8, or 21.3, respectively. Fortunately, good branch prediction is not hard to do. The mean time between misses can be multiplied by 10 using only two-bit prediction with a modestly sized table, and software can do about as well using profiling. We can obtain another factor of 5 in the mean time between misses if we are willing to devote a large chip area to the predictor.

Complementing branch prediction with simultaneous speculative execution across different branching code paths is the icing on the cake, raising our observed parallelism of 4–10 up to 7–13. In fact, parallel exploration to a depth of 8 branches can remove the need for prediction altogether, though this is probably not an economical substitute in practice.

Though these numbers are grounds for optimism, we must remember that they are themselves the result of rather optimistic assumptions. We have assumed unlimited resources, including as many copies of each functional unit as we need and a perfect memory system with no cache misses. Duplicate functional units take up chip real estate that might be better spent on more on-chip cache, especially as processors get faster and the memory bottleneck gets worse. We have for the most part assumed that there is no penalty, in pipeline refill cycles or in software-controlled undo and catch-up instructions, for a missed prediction. These wasted cycles lower the overall parallelism even if the unwasted cycles are as full as ever, reducing the advantage of an instruction-parallel architecture. We have assumed that all machines modeled have the same cycle time, even though adding superscalar capability will surely not decrease the cycle time and may in fact increase it. And we have assumed that all machines are built with comparable technology, even though a simpler machine may have a shorter time-to-market and hence the advantage of newer, faster technology. Any one of these considerations could reduce the expected payoff of an instruction-parallel machine by a third; together they could eliminate it completely.

Appendix 1. Implementation details

The parallelism simulator used for this paper is conceptually simple. A sequence of *pending cycles* contains sets of instructions that can be issued together. We obtain a trace of the instructions executed by the benchmark, and we place each successive instruction into the earliest cycle that is consistent with the model we are using. When the first cycle is full, or when the total number of pending instructions exceeds the window size, the first cycle is retired. When a new instruction cannot be placed even in the latest pending cycle, we must create a new (later) cycle.

A simple algorithm for this would be to take each new instruction and consider it against each instruction already in each pending cycle, starting with the latest cycle and moving backward in time. When we find a dependency between them, we place the instruction in the cycle after that. If the proper cycle is already full, we place the instruction in the first non-full cycle after that.

Since we are typically considering models with windows of thousands of instructions, doing this linear search for every instruction in the trace could be quite expensive. The solution is to maintain a sort of reverse index. We number each new cycle consecutively and maintain a data structure for all the individual dependencies an instruction might have with a previously-scheduled instruction. This data structure tells us the cycle number of the last instruction that can cause each kind of dependency. To schedule an instruction, we consider each dependency it might have, and we look up the cycle number of the barrier imposed by that dependency. The latest of all such barriers tells us where to put the instruction. Then we update the data structure as needed, to reflect the effects this instruction might have on later ones.

Some simple examples should make the idea clear. An assignment to a register cannot be exchanged with a later use of that register, so we maintain a timestamp for each register. An instruction that assigns to a register updates that register's timestamp; an instruction that uses a register knows that no instruction scheduled later than the timestamp can conflict because of that register. Similarly, if our model specifies no alias analysis, then we maintain one timestamp for all of memory, updated by any store instruction; any new load or store must be scheduled after that timestamp. On the other hand, if our model specifies perfect alias analysis, two instructions conflict only if they refer to the same location in memory, so we maintain a separate timestamp for each word in memory. Other examples are more complicated.

Each of the descriptions that follow has three parts: the data structure used, the code to be performed to schedule the instruction, and the code to be performed to update the data structure.

Three final points are worth making before we plunge into the details.

First, the parallelism simulator is *not* responsible for actually orchestrating an execution. It is simply consuming a trace and can therefore make use of information available much later than a real system could. For example, it can do perfect alias analysis simply by determining which memory location is accessed, and scheduling the instruction into a pending cycle as if we had known that all along.

`a mod b`	The number in the range [0..b-1] obtained by subtracting a multiple of b from a
`a INDEX b`	(a >> 2) mod b i.e. the byte address a used as an index to a table of size b.
`UPDOWN a PER b`	if (b and a<3) then a := a + 1 elseif (not b and a>0) then a := a - 1
`UPDOWN a PER b1, b2`	if (b1 and not b2 and a<3) then a := a + 1 elseif (not b1 and b2 and a>0) then a := a - 1
`SHIFTIN a BIT b MOD n`	s := ((s<<1) + (if b then 1 else 0)) mod n
`AFTER t`	Schedule this instruction after cycle t. Applying all such constraints tells us the earliest possible time.
`BUMP a TO b`	if (a<b) then a := b. This is used to maintain a timestamp as the latest occurrence of some event; a is the latest so far, and b is a new occurrence, possibly later than a.

Figure 36: Abbreviations used in implementation descriptions

Second, the algorithm used by the simulator is temporally backwards. A real multiple-issue machine would be in a particular cycle looking ahead at possible future instructions to decide what to execute now. The simulator, on the other hand, keeps a collection of cycles and pushes each instruction (in the order from the single-issue trace) as far back in time as it legally can. This backwardness can make the implementation of some configurations unintuitive, particularly those with fanout or with imperfect alias analysis.

Third, the R3000 architecture requires no-op instructions to be inserted wherever a load delay or branch delay cannot be filled with something more useful. This often means that 10% of the instructions executed are no-ops. These no-ops would artificially inflate the program parallelism found, so we do not schedule no-ops in the pending cycles, and we do not count them as instructions executed.

We now consider the various options in turn. In describing the implementations of the different options, the abbreviations given in Figure 36 will be helpful.

Data structure:

```
jumpbarrier                     cycle of last jump machine mispredicted
ring[0..nring-1]                return ring
iring                           wraparound index to ring
jpredict[0..njpredict-1]        last-destination table
jprofile[a]                     most frequent destination of jump at address a
```

To schedule:

```
AFTER jumpbarrier
```

To bookkeep, after scheduling the instruction in cycle `t`:

```
if instruction is call then
    if CALLRING then
        iring := (iring + 1) mod nring
        ring[iring] := returnAddress
    end
end
if instruction is indirect jump then
    addr := "address of jump"
    destination := "address jumped to, according to trace"
    if PERFECT then
        isbadjump := false
    else if instruction is return and CALLRING then
        if ring[iring] = destination then
            iring := (iring - 1) mod nring
        else
            isbadjump := true
        end
    else if JTABLE then
        i := addr INDEX njpredict
        isbadjump := (jpredict[i] != destination)
        jpredict[i] := destination
    else if JPROFILE then
        isbadjump := (jprofile[addr] != destination)
    else
        isbadjump := true
    end
end
if isbadjump then
    BUMP jumpbarrier TO t
end
```

Figure 37: Jump prediction

Jump prediction

The simulator has two ways of predicting indirect jumps in hardware. One is the return ring, and the other is the last-destination table. It also supports software jump prediction from a profile. In any case, a successfully predicted jump is the same as a direct jump: instructions from after the jump in the trace can move freely as if the jump had been absent. A mispredicted jump may move before earlier instructions, but all later instructions must be scheduled after the mispredicted jump. Since the trace tells us where each indirect jump went, the simulator simply does the prediction by whatever algorithm the model specifies and then checks to see if it was right.

A mispredicted jump affects all subsequent instructions, so it suffices to use a single timestamp `jumpbarrier`.

The last-destination table is a table `jpredict[0..njpredict-1]` containing code addresses, where the model specifies the table size. If the next instruction in the trace is an indirect jump, we look up the table entry whose index is the word address of the jump, modulo `njpredict`. We predict that the jump will be to the code address in that table entry. After the jump, we replace the table entry by the actual destination.

The return ring is a circular table `ring[0..nring-1]` containing code addresses, where the model specifies the ring size. We index the ring with a wraparound counter `iring` in the range `[0..nring-1]`: incrementing `nring-1` gives zero, and decrementing `0` gives `nring-1`. If the next instruction in the trace is a return, we predict that its destination is `ring[iring]` and then we decrement `iring`. If the next instruction in the trace is a call, we increment `iring` and store the return address for the call into `ring[iring]`.

Our instruction set does not have an explicit return instruction. An indirect jump via `r31` is certainly a return (at least with the compilers and libraries we used), but a return via some other register is allowed. We identify a return via some register other than `r31` by the fact that the return ring correctly predicts it. (This is not realistic, of course, but use of the return ring assumes that returns can be identified, either through compiler analysis or through use of a specific return instruction; we don't want the model to be handicapped by a detail of the R3000.)

A jump profile is a table obtained from an identical previous run, with an entry for each indirect jump in the program, telling which address was the most frequent destination of that jump. We predict a jump by looking up its entry in this table, and we are successful if the actual destination is the same.

This leaves the two trivial cases of perfect jump prediction and no jump prediction. In either case we ignore what the trace says and simply assume success or failure.

Branch prediction and fanout

Branch prediction is analogous to jump prediction. A correctly predicted conditional branch is just like an unconditional branch: instructions from after the branch can move freely before the branch. As with jumps, no later instruction may be moved before an incorrectly predicted branch. The possibility of fanout, however, affects what we consider an incorrectly predicted branch.

The simple counter-based predictor uses a table `ctrtab[0..nctrtab-1]` containing two-bit counter in the range [0..3], where the model specifies the table size. These two-bit counters are saturating: incrementing 3 gives 3 and decrementing 0 gives 0. If the next instruction in the trace is a conditional branch, we look up the counter whose index is the word address of the branch, modulo `nctrtab`. If the counter is at least 2, we predict that the branch will be taken, otherwise that it will not. After the branch, we increment the counter if the branch really was taken and decrement it if it was not, subject to saturation in either case. The other two hardware predictors are combined techniques, but they work similarly.

A branch profile is a table obtained from an identical previous run, with an entry for each conditional branch in the program, telling the fraction of the times the branch was executed in which it was taken. If this fraction is more than half, we predict that the branch is taken; otherwise

Data structure:

```
branchbarrier                  cycle of last branch machine mispredicted
ctrtab[0..nctrtab-1]           2-bit counters for counter predictor
gbltab[0..ngbltab-1]           2-bit counters for gshare predictor
loctab[0..nloctab-1]           2-bit counters for local predictor
select[0..nselect-1]           2-bit counters for choosing in combined predictor
globalhist                     (log2 ngbltab)-bit history shift register
localhist[0..nlocalhist-1]     (log2 nloctab)-bit history shift registers
bprofile[a]                    frequency that branch at address a is taken
bqueue[f+1]                    queue of cycles of f+1 previous branches
```

To schedule:

```
AFTER branchbarrier
```

To bookkeep, after scheduling the instruction in cycle `t`:

```
if instruction is conditional branch then
    addr := "address of branch"
    branchto := "address branch would go to if taken"
    taken := "trace says branch is taken"
    if not BPREDICT then
        isbadbranch := true
    else if BTAKEN then
        isbadbranch := not taken;
    else if BSIGN then
        isbadbranch := (taken = (branchto > addr))
    else if BTABLE then
        if BTECHNIQUE = counters then
            i := addr INDEX nctrtab
            pred = (ctrtab[i] >= 1)
            isbadbranch := (pred != taken)
            UPDOWN ctrtab[i] PER taken
        elsif BTECHNIQUE = counters/gshare then
            i := addr INDEX nctrtab
            pred1 := (ctrtab[i] >= 2)
            UPDOWN brctr^[i] PER taken
            i := globalhist xor (addr INDEX gtablesize)
            pred2 := (gbltab[i] >= 2)
            UPDOWN gbltab[i] PER taken
            SHIFTIN globalhist BIT taken MOD ngbltab
            i := addr INDEX nselect
            pred := (if select[i] >= 2 then pred1 else pred2)
            isbadbranch := (pred != taken)
            UPDOWN select[i] PER (taken=pred1), (taken=pred2)
        elsif BTECHNIQUE = local/gshare then
            histi := addr INDEX nlocalhist
            i := localhist[histi]
            pred1 := (loctab[i] >= 2)
            UPDOWN loctab[i] PER taken
            SHIFTIN localhist[histi] BIT taken MOD nloctab
            i := globalhist xor (addr INDEX gtablesize)
            pred2 := (gbltab[i] >= 2)
            UPDOWN gbltab[i] PER taken
            SHIFTIN globalhist BIT taken MOD ngbltab
            i := addr INDEX nselect
            pred := (if select[i] >= 2 then pred1 else pred2)
            isbadbranch := (pred != taken)
            UPDOWN select[i] PER (taken=pred1), (taken=pred2)
```

Figure 38: Branch prediction

```
        else if BPROFILE then
            isbadbranch := (taken = (bprofile[addr] < 0.5))
        else
            isbadbranch := false
        end
        if BPROFILE and PROFILEINTEGRATION then
            if bprofile[addr]<threshold and 1.0-bprofile[addr]<threshold then
                remove head of bqueue, append t to bqueue
                if isbadbranch then
                    BUMP branchbarrier TO head of bqueue
                end
            elsif isbadbranch then
                fill all elements of bqueue with t
                BUMP branchbarrier TO t
            end
        else
            remove head of bqueue, append t to bqueue
            if isbadbranch then
                BUMP branchbarrier TO head of bqueue
            end
        end
    end
```

Figure 38 continued

we predict that it is not taken.

If the model specifies “signed branch prediction” we see if the address of the branch is less than the address of the (possible) destination. If so, this is a forward branch and we predict that it will not be taken; otherwise it is a backward branch and we predict that it will be taken.

If the model specifies “taken branch prediction” we always predict that the branch will be taken. If it is, we are successful.

Perfect branch prediction and no branch prediction are trivial: we simply assume that we are always successful or always unsuccessful.

If the model does not include fanout, we deal with success or failure just as we did with jumps. A successfully predicted branch allows instructions to be moved back in time unhindered; a mispredicted branch acts as a barrier preventing later instructions from being moved before it.

If the model includes fanout of degree `f`, the situation is a little more complicated. We assume that in each cycle the hypothetical multiple-issue machine looks ahead on both possible paths past the first `f` conditional branches it encounters, and after that point looks on only one path using whatever form of branch prediction is specified. Thus a given branch may sometimes be a fanout branch and sometimes be a predicted branch, depending on how far ahead the machine is looking when it encounters the branch. From the simulator’s temporally backward point of view, this means we must tentatively predict every branch. We can move an instruction backward over any number of successfully predicted branches, followed by `f` more branches whether predicted successfully or not. In other words, an unsuccessfully predicted branch means the barrier is situated at the cycle of the branch `f` branches before this one. (Notice that if `f=0`, that branch is this branch, and we reduce to the previous case.)

To do this, we maintain a queue `bqueue[0..f]` containing the timestamps of the previous `f+1` branches. If the next instruction in the trace is a branch, we remove the first element of `bqueue` and append the timestamp of the new branch. If the branch prediction is successful, we do nothing else. If it is unsuccessful, we impose the barrier at the cycle whose timestamp is in the first element of `bqueue`.

Profile-guided integration of prediction and fanout works a little differently. If the next instruction is a conditional branch, we look it up in the profile to find out its probability of being taken. If its probability of being taken (or not taken) is greater than the model's threshold, we call it a "predictable" branch and predict that it will be taken (or not taken). Otherwise we call it an "unpredictable" branch and use one level of fanout. (Note that this terminology is independent of whether it is actually predicted correctly in a given instance.) From the simulator's point of view, this means we can move backward past any number of predictable branches that are successfully predicted, interspersed with `f` unpredictable branches. Fanout does us no good, however, on a branch that we try unsuccessfully to predict.

Thus we again maintain the queue `bqueue[0..f]`, but this time we advance it only on unpredictable branches. As before, we tentatively predict every branch. If we mispredict an unpredictable branch, we impose the barrier `f` unpredictable branches back, which the head of `bqueue` tells us. If we mispredict a predictable branch, we must impose the barrier after the current branch, because the profile would never have let us try fanout on that branch.

Register renaming

The idea behind register renaming is to reduce or eliminate the false dependencies that arise from re-using registers. Instructions appear to use the small set of registers implied by the size of the register specifier, but these registers are really only register names, acting as placeholders for a (probably) larger set of actual registers. This allows two instructions with an anti-dependency to be executed in either order, if the actual registers used are different.

When an instruction assigns to a register name, we allocate an actual register to hold the value.[6] This actual register is then associated with the register name until some later instruction assigns a new value to that register name. At this point we know that the actual register has in fact been dead since its last use, and we can return it to the pool of available registers. We keep track of the cycle in which an available register was last used so that we can allocate the least-recently used of the available registers when we need a new one; this lets the instructions that use it be pushed as far back in time as possible.

If the next instruction uses registers, we look the names up in the `areg` mapping to find out which actual registers are used. If the instruction sets a register, use `whenavail` to allocate the available actual register `r` that became available in the earliest cycle. Update the `areg` mapping to reflect this allocation. This means that the register `rr` that was previously mapped to this register name became free after its last use, so we record the time it actually became available; i.e. the time it was last referenced, namely `whensetorused[rr]`. Conversely, we want to mark the allocated register `r` as unavailable, so we change `whenavail[r]` to an infinitely late value.

If the instruction uses some actual register `r`, then it must be issued in or after `whenready[r]`. If the instruction sets some actual register `r`, then it must be issued in or after `whensetorused[r]` and also in or after `whenready[r]`, unless the model specifies perfect register renaming.

Once we have considered all the timing constraints on the instruction and have determined that it must be issued in some cycle `t`, we update the dependency barriers. If the instruction sets an actual register `r`, we record in `whenready[r]` the time when the result will actually be accessible — usually the next instruction, but later if the instruction has a non-unit latency.[7] We also update `whensetorused[r]`; for this we don't care about the latency, because the setting time will be relevant only if this register immediately becomes available because the value assigned is never used. If the instruction uses an actual register `r`, we update `whensetorused[r]`.

As stated so far, this algorithm is still fairly expensive, because the allocation of a new actual register requires a search of the `whenavail` table to find the earliest available register. We speed this up by maintaining a tournament tree on top of `whenavail` that lets us find the earliest entry in constant time by looking in the root. Whenever we change some entry in `whenavail`, we update the tournament tree path from this register to the root, which takes time logarithmic in the number of actual registers.

This discussion has also ignored the fact that there are two disjoint register sets for CPU and

[6]The R3000 implicitly uses special registers called `hi` and `lo` in division operations; we treat these as if they had been named explicitly in the instruction, and include them in the renaming system.

[7]We don't use BUMP for this because under perfect register renaming previous values in this actual register are irrelevant. This doesn't hurt us under imperfect renaming because this register assignment won't be able to move before previous uses.

Data structure:

```
areg[rn]                Actual register associated with register name rn
whenready[r]            First cycle in which the value in r can be used
whensetorused[r]        Last cycle in which register r was set or used
whenavail[r]            Cycle when register r became available.
root                    Pointer to root of tournament tree
leaf[r]                 Pointer to r's leaf entry in tournament tree

Tournament tree records are:
    reg                 register described by this record
    avail               time when the register became available
    parent              pointer to parent record
    sib                 pointer to sibling record
Each record identifies the child with the smaller value of avail.

procedure Oldest()
    return root^.reg

procedure UpdateOldest (r)
    p := leaf[r]
    p^.avail := whenavail[r]
    repeat
        parent := p^.parent
        sib := p^.sib
        parent^.avail := mininimum of p^.avail and sib^.avail
        parent^.reg   := p^.reg or sib^.reg, whichever had min avail
        p := parent
    until p = root
```

To schedule:

```
if instruction sets or uses a register name rn then
    use areg[rn] to determine the actual register currently mapped
end
if REGSRENUMBER then
    if instruction sets a register name rn then
        rr := areg[rn]
        r := Oldest()
        areg[rn] := r
        whenavail[r] := +Infinity
        UpdateOldest(r)
        whenavail[rr] := whensetorused[rr]
        UpdateOldest(rr)
    end
end
if instruction uses actual register r then
    AFTER whenready[r]-1
end
if not REGSPERFECT then
    if instruction sets actual register r then
        AFTER whensetorused[r]-1
        AFTER whenready[r]-1
    end
end
```

Figure 39: Register renaming

To bookkeep, after scheduling the instruction in cycle `t`:

```
if instruction sets actual register r then
    whenready[r] := t + latencyOfOperation
    BUMP whensetorused[r] TO t
end
if instruction uses actual register r then
    BUMP whensetorused[r] TO t
end
```

Figure 39 continued

FPU registers. We therefore really need *two* instances of these data structures and algorithms, one for each register set.

If the model specifies perfect renaming, we assign one actual register to each register name and never change it. We maintain only the `whenready` table, exactly as described above; the instruction using register `r` must be scheduled in or after `whenready[r]`.

If the model specifies no renaming, we again assign one actual register to each register name and never change it. We maintain `whenready` and `whensetorused` as with renaming, and an instruction that uses register `r` must be scheduled in or after the later of `whenready[r]` and `whensetorused[r]`.

Alias analysis

If the model specifies no alias analysis, then a store cannot be exchanged with a load or a store. Any store establishes a barrier to other stores and loads, and any load establishes a barrier to any store, so we maintain only `laststore` and `lastload`.

If the model specifies perfect alias analysis, a load or store can be exchanged with a store only if the two memory locations referenced are different. Thus there are barriers associated with each distinct memory location; we set up a table `lasststoreat[a]` and `lastloadat[a]` with an entry for each word in memory. (This is feasible because we know how much memory these benchmarks need; we need not cover the entire address space with this table. We assume that the granularity of memory is in 32-bit words, so byte-stores to different bytes of the same word are deemed to conflict.) The program trace tells us which address a load or store actually references.

Alias analysis "by inspection" is more complicated. A store via a base register `r` can be exchanged with a load or store via the same base register if the displacement is different and the value of the base register hasn't been changed in the meantime. They can be exchanged even when the base registers are different, as long as one is manifestly a stack reference (i.e. the base register is `sp` or `fp`) and the other manifestly a static data reference (i.e. the base register is `gp`).[8] In terms of barriers, however, we must state this not as what pairs can be exchanged but as what blocks an instruction from moving farther back in time.

We first consider instructions with different base registers. For each register, we use

[8] This analysis is most likely done by a compile-time scheduler. If register renaming is in effect, we therefore use the register name in the instruction rather than the actual register from the renaming pool for this analysis.

Data structure:

```
laststore               Cycle of last store
laststoreat[a]          Cycle of last store to word at address a
laststorevia[r]         Cycle of the last store via r
laststorebase           Base register in last store via r
                        other than gp, fp, sp
laststorebase2          Base register in last store via r
                        other than gp, fp, sp, laststorebase
oldstore[r]             Cycle of last store via r that was
                        followed by a change to r
lastload                Cycle of last load
lastloadat[a]           Cycle of last load from word at address a
lastloadvia[r]          Cycle of the last load via r
lastloadbase            Base register in last load via r
                        other than gp, fp, sp
lastloadbase2           Base register in last load via r
                        other than gp, fp, sp, lastloadbase
oldload[r]              Cycle of last load via r that was
                        followed by a change to r

procedure AliasConflict (R, old, lastvia, lastbase, lastbase2)
    AFTER old[R]
    if R = fp then
        AFTER lastvia[sp], lastvia[lastbase]
    else if R = sp then
        AFTER lastvia[fp], lastvia[lastbase]
    else if R = gp then
        AFTER lastvia[lastbase]
    else if R = lastbase then
        AFTER lastvia[sp], lastvia[fp],
              lastvia[gp], lastvia[lastbase2]
    else
        AFTER lastvia[sp], lastvia[fp],
              lastvia[gp], lastvia[lastbase]
    end

procedure UpdateLastbase (R, t, lastvia, lastbase, lastbase2)
    if R is not fp, sp, or gp then
        if t > lastvia[lastbase] then
            if R <> lastbase then
                lastbase2 := lastbase
                lastbase := R
            end
        else if t > lastvia[lastbase2] then
            if R <> lastbase2 then
                lastbase2 := R
            end
        end
    end
```

To schedule:

```
if ALIASNONE then
    if loads or stores memory then
        AFTER laststore
    end
    if stores memory then
        AFTER lastload
    end
```

Figure 40: Alias analysis

```
else if ALIASPERFECT then
    if loads or stores memory at location A then
        AFTER laststoreat[A]
    end
    if stores memory at location A then
        AFTER lastloadat[A]
    end
else if ALIASINSPECT then
    if loads or stores memory location A via base register R then
        AFTER laststoret[A]
        AliasConflict (R, oldstore, laststorevia, laststorebase, laststorebase2)
    end
    if stores memory location A via base register R then
        AFTER lastloadat[A]
        AliasConflict (R, oldload, lastloadvia, lastloadbase, lastloadbase2)
    end
else if ALIASCOMP then
    if loads or stores memory location A via base register R then
        AFTER laststoreat[A]
        if A is in the heap then
            AliasConflict (R, oldstore, laststorevia, laststorebase, laststorebase2)
        end
    end
    if stores memory location A via base register R then
        AFTER lastloadat[A]
        if A is in the heap then
            AliasConflict (R, oldload, lastloadvia, lastloadbase, lastloadbase2)
        end
    end
end
```

To bookkeep, after scheduling the instruction in cycle `t`:

```
if instruction sets register R and R is an allowable base register then
    BUMP oldstore[r] TO laststorevia[r]
    BUMP oldload[r]  TO lastloadvia[r]
end
if instruction stores to memory location A via base register R then
    BUMP laststore to t
    if ALIASPERFECT or ALIASINSPECT or ALIASCOMP then
        BUMP laststore[A] TO t
    end
    if ALIASINSPECT or (ALIASCOMP and A is in the heap) then
        BUMP laststorevia[R] TO t
        UpdateLastbase (R, t, laststorevia, laststorebase, laststorebase2)
    end
end
if instruction loads from memory location A via base register R then
    BUMP lastload to t
    if ALIASPERFECT or ALIASINSPECT or ALIASCOMP then
        BUMP lastload[A] TO t
    end
    if ALIASINSPECT or (ALIASCOMP and A is in the heap) then
        BUMP lastloadvia[R] TO t
        UpdateLastbase (R, t, lastloadvia, lastloadbase, lastloadbase2)
    end
end
```

Figure 40 continued

`laststorevia` and `lastloadvia` to keep track of the most recent load and store for which it was the base register. We also keep track of the two base registers other than `gp`, `sp`, and `fp` used most recently in a load and in a store. This information lets us find the most recent conflicting load or store by looking at no more than four members of `laststorevia` or `lastloadvia`.

Next there is the case of two instructions that use the same base register, with the value of that base register changed in between. If the model specifies no register renaming, this point is moot, because the instruction that assigns to the base register will be blocked by the earlier use, and the later use will be blocked in turn by the register assignment. Register renaming, however, allows the register assignment to move before the previous use, leaving nothing to prevent the two uses from being exchanged even though they may well reference the same location. To handle this, we maintain tables `oldstore[r]` and `oldload[r]`. These contain the cycle of the latest store or load via `r` that has been made obsolete by a redefinition of `r`.

Finally, a load or store can also be blocked by a store via the same base register if the value of the base register hasn't changed and the displacements are the same. In this case the two instructions are actually referencing the same memory location. So we account for this possibility just as we did with perfect alias analysis, by maintaining `lasststoreat[a]` and `lastloadat[a]` to tell us the last time location `a` was stored or loaded.

Alias analysis "by compiler" is similar, but we assume the compiler can perfectly disambiguate two non-heap references or a heap reference and a non-heap reference, but must rely on inspection to disambiguate two heap references. We recognize a heap reference by range-checking the actual data address, and proceed just as with alias analysis by inspection except that we ignore non-heap references beyond checking them for actual address conflicts.

Latency models, cycle width, and window size

Latency models are easy; we already covered them as part of register renaming. When the instruction scheduled into cycle `t` sets a register `r`, we modify `whenready[r]` to show when the result can be used by another instruction. If the instruction has unit latency, this is `t+1`; if the latency is `k`, this is `t+k`.

For most of our models we assume an upper limit of 64 instructions that can be issued in a single cycle. There is no fundamental reason for the limit to have this value; it doesn't even affect the amount of memory the simulator needs for its data structures. The limit is there only as a generous guess about the constraints of real machines, and doubling it to 128 is just a matter of relaxing the test.

We keep track of the number of instructions currently in each pending cycle, as well as the total in all pending cycles. When the total reaches the window size, we flush early cycles until the total number is again below it. If the window is managed discretely rather than continuously, we flush all pending cycles when the total reaches the window size.

In either case, we always flush cycles up to the cycle indexed by `jumpbarrier` and `branchbarrier`, since no more instructions can be moved earlier. This has no effect on continuously managed windows, but serves as a fresh start for discretely managed windows, allowing more instructions to be considered before a full window forces a complete flush.

Removing the limit on cycle width is trivial: we must still keep track of how full each pending cycle is, but we no longer have to look past a "full" cycle to schedule an instruction that would normally go there. Given that, removing the limit on window size simply means that we stop maintaining any description at all of the pending cycles. We determine when an instruction should be scheduled based on the existing barriers, and we update the barriers based on the time decided upon.

Appendix 2. Details of program runs

This section specifies the data and command line options used for each program, along with any changes made to the official versions the programs to reduce the runtime.

```
sed:
sed -e 's/\/\*/(*/g' -e 's/\*\//*)/g' -e 's/fprintf(\(.*\));/DEBUG \1 ;/g' sed0.c > test.out

egrep:
egrep '^...$|regparse|^[a-z]|if .*{$' regexp.c > test.out

eco:
eco -i tna.old tna.new -o tna.eco > test.out

yacc:
yacc -v grammar
grammar has 450 non-blank lines

metronome:
met dma.tuned -c 100k.cat pal.cat dc.cat misc.cat teradyne.cat > output

grr:
grr -i mc.unroute -l mc.log mc.pcb -o mc.route

hydro2d:
hydro2d < short.in > short.out

gcc1:
gcc1 cexp.i -quiet -O -o cexp.s > output

compress:
compress < ref.in > ref.out

espresso:
espresso -t opa > opa.out

ora:
ora < short.in > short.out

fpppp:
fpppp < small > small.out

li:
li dwwtiny.lsp > dwwtiny.out      (dwwtiny.lsp is the 7-queens problem)

doduc:
doduc < small >small.out

swm256:
cat swm256.in | sed 's/1200/30/' | swm256 > swm256.out

tomcatv:
Change initial value of LMAX on line 22 from 100 to 15
tomcatv > output

alvinn:
Change definition of macro NUM_EPOCHS on line 23 from 200 to 10
alvinn > result.out

mdljsp2:
mv short.mdlj2.dat mdlj2.dat
mdljsp2 < input.file > short.out
```

Appendix 3. Parallelism under many models

This appendix lists the results from running the test programs under more than 350 different configurations, and then lists the configurations used in each of the figures in the main body of the report. The configurations are keyed by the following abbreviations:

`?+`	perfect branch prediction, 100% correct
`?c`*nn*	loc/gsh predictor
`?c`*nn*`:`*mm*	fanout next *mm* branches, loc/gsh predictor thereafter
`?b`*nn*	ctr/gsh predictor
`?b`*nn*`:`*mm*	fanout next *mm* branches, ctr/gsh predictor thereafter
`?a`*nn*	ctr predictor
`?P:`*mm*`(`*ff*`)`	integrated *mm*-way fanout and profile-prediction with threshold *ff*
`?P`	predict all branches from profile
`?Taken`	predict all branches taken
`?Sign`	predict backward branches taken, forward branches not taken
`?-:`*mm*	fanout next *mm* branches, no prediction thereafter
`?-`	no branch prediction; all miss
`j+`	perfect indirect jump prediction, 100% correct
`j`*nn*+*mm*	predict returns with *nn*-element ring, other jumps from *mm*-elt. last-dest table
`j`*nn*	predict returns with *nn*-element ring, don't predict other jumps
`j-`	no jump prediction; all miss
`r+`	perfect register renaming; infinitely many registers
`r`*nn*	register renaming with *nn* cpu and *nn* fpu registers
`r-`	no register renaming; use registers as compiled
`a+`	perfect alias analysis; use actual addresses to distinguish conflicts
`aComp`	alias analysis "by compiler"
`aInsp`	alias analysis "by inspection"
`a-`	no alias analysis; stores conflict with all memory references
`i*2`	allow cycles to issue 128 instructions, not 64
`i+`	allow unlimited cycle width
`w`*nn*	continuous window of *nn* instructions, default 2K
`dw`*nn*	discrete window of *nn* instructions
`w+`	unlimited window size and cycle width
`L`*model*	use specified latency model; default is A

The size of each of the three hardware branch predictors is specified by a single integer parameter. A counter predictor with parameter n consists of a table of 2^n 2-bit counters. A counter/gshare predictor with parameter n consists of 2^n 2-bit counters for the first predictor, one $(n+1)$-bit global history register and 2^{n+1} 2-bit counters for the second predictor, and 2^n 2-bit counters for the selector. A local/gshare predictor with parameter n consists of 2^n n-bit history registers and 2^n 2-bit counters for the first predictor, one n-bit global history register and 2^n 2-bit counters for the second predictor, and 2^n 2-bit counters for the selector.

						egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
1	?+	j+	r+	a+		54.0	24.5	22.0	17.3	28.0	23.2	7.5	25.7	57.9	40.5	60.9	35.2	44.8	16.4	33.3	9.0	48.9	59.9	23.6	12
2	?+	j+	r+	a+	w+	77.7	24.5	93.5	18.8	35.2	25.3	97.5	26.2	118.7	62.6	75.2	54.8	175.2	17.9	38.8	9.0	564.8	150.1	36.2	18a
3	?+	j+	r+	a+	iInf	58.1	24.5	22.4	17.3	28.2	23.3	7.5	25.7	91.6	45.5	74.1	37.0	62.0	16.4	33.5	9.0	53.1	87.8	24.6	15a
4	?+	j+	r+	a+	i*2	58.0	24.5	22.3	17.3	28.2	23.3	7.5	25.7	84.4	44.8	74.1	36.8	56.8	16.4	33.5	9.0	53.1	87.3	24.5	14a
5	?+	j+	r+	a+	LB	54.0	24.5	22.0	17.4	28.1	23.2	7.4	25.7	60.2	40.5	58.4	35.3	46.7	16.4	34.1	6.3	69.2	73.8	22.5	34a
6	?+	j+	r+	a+	LC	55.0	22.9	23.7	16.3	27.1	27.7	7.5	23.2	73.3	40.9	63.5	36.2	58.3	20.9	33.6	5.7	100.3	109.0	22.9	
7	?+	j+	r+	a+	LD	55.0	22.9	23.7	16.3	27.1	27.7	7.4	23.2	70.9	40.9	56.5	36.2	59.0	20.9	33.8	4.8	119.9	122.6	22.0	34b
8	?+	j+	r+	a+	LE	54.1	22.0	25.1	15.6	25.9	29.3	8.2	21.6	73.5	40.5	68.0	36.5	72.4	25.2	35.0	4.5	140.2	157.5	22.3	
9	?+	j+	r+	a-		7.4	4.3	6.0	2.8	3.0	2.9	2.5	3.4	4.0	5.1	3.4	3.4	4.4	3.1	2.7	4.2	3.5	3.5	3.6	
10	?+	j+	r256	a+		22.8	22.3	14.5	16.0	20.8	18.0	5.7	19.3	21.1	24.2	50.6	26.1	16.4	15.7	11.6	9.0	44.2	46.1	16.9	
11	?+	j+	r-	a+		5.0	6.0	5.0	4.9	4.3	5.0	3.3	5.2	4.7	4.0	3.5	4.7	6.2	5.5	3.4	4.2	3.4	4.9	4.5	
12	?+	j2K+2K	r+	a+		54.0	24.5	21.5	11.0	27.6	22.7	7.5	25.7	57.2	40.5	60.9	17.3	34.1	12.5	33.3	9.0	48.9	59.8	21.1	
13	?+	j2K+2K	r256	a+		22.8	21.1	14.2	10.4	20.6	17.7	5.7	19.3	21.0	24.2	50.6	15.1	14.4	12.1	11.6	9.0	44.2	46.1	15.5	
14	?+	j2K+64	r+	a+		54.0	24.5	21.5	10.9	27.5	22.7	7.5	25.7	57.2	40.5	60.8	17.1	33.4	12.5	33.3	9.0	48.9	59.8	21.0	29b
15	?+	j2K+32	r+	a+		54.0	24.5	21.5	10.9	27.5	22.7	7.5	25.7	57.2	40.5	60.8	16.5	32.5	12.5	33.3	9.0	48.9	59.8	21.0	29b
16	?+	j2K+16	r+	a+		54.0	24.5	21.5	10.9	27.5	22.7	7.5	25.7	57.1	40.5	60.8	16.4	31.2	12.5	33.3	9.0	48.9	59.8	20.9	29b
17	?+	j2K+8	r+	a+		54.0	24.5	21.5	10.9	27.4	22.7	7.5	25.7	57.0	40.5	60.8	16.2	31.2	12.5	33.3	9.0	48.9	59.8	20.9	29b
18	?+	j2K+4	r+	a+		54.0	24.5	21.5	10.9	27.4	22.7	7.5	25.7	57.0	40.5	60.8	16.1	29.9	11.5	33.3	9.0	48.9	59.7	20.7	29b
19	?+	j2K+2	r+	a+		54.0	24.5	21.5	10.9	27.2	22.5	7.5	25.7	53.3	40.5	60.8	15.7	30.1	11.5	33.3	9.0	48.9	59.7	20.6	29b
20	?+	j2K	r+	a+		54.0	19.4	21.5	10.4	26.2	21.2	7.5	25.7	51.4	40.4	58.5	13.6	28.2	9.9	33.3	9.0	48.9	59.7	19.6	28b 29b
21	?+	j16	r+	a+		54.0	19.4	21.5	10.3	26.2	21.2	7.5	25.7	51.4	40.4	58.5	13.6	28.2	9.9	33.3	9.0	48.9	59.7	19.5	28b
22	?+	j8	r+	a+		53.9	19.4	21.5	10.2	26.1	21.2	7.5	25.7	51.4	40.4	58.5	13.6	28.0	9.8	33.3	9.0	48.9	59.7	19.5	28b
23	?+	j4	r+	a+		53.9	19.3	21.5	10.1	25.8	21.1	7.5	25.7	49.8	40.3	58.5	13.3	27.3	9.5	33.3	9.0	48.9	59.7	19.3	28b
24	?+	j2	r+	a+		53.9	16.7	21.5	9.7	24.5	19.7	7.5	25.7	40.6	39.9	58.5	12.8	26.9	9.0	33.3	9.0	48.9	59.7	18.7	28b
25	?+	j1	r+	a+		53.9	15.5	21.3	9.1	20.5	17.6	7.5	25.7	38.6	38.7	57.9	12.2	26.2	8.3	33.3	9.0	48.9	59.7	17.9	28b
26	?+	jP	r+	a+		53.9	24.5	21.7	10.3	18.5	15.8	7.5	25.7	50.0	38.4	60.6	16.5	29.3	11.4	33.3	9.0	48.9	59.8	19.7	
27	?+	j-	r+	a+		53.8	13.4	19.7	7.8	14.3	10.8	7.5	25.6	34.4	35.6	57.4	10.5	23.0	6.8	33.3	9.0	48.8	59.6	15.7	28b
28	?c13:8	j+	r+	a+		13.1	13.5	10.6	13.0	13.1	13.6	7.4	10.0	32.0	16.5	59.8	15.8	28.1	15.5	30.0	9.0	46.6	53.8	15.5	
29	?c13:8	j2K+2K	r256	a+		12.2	12.5	10.3	8.9	12.9	13.4	5.6	10.0	17.5	15.7	49.8	11.3	13.5	11.8	11.4	9.0	43.5	45.6	12.6	24b
30	?c13:6	j+	r+	a+		12.2	12.8	10.4	12.5	12.5	13.1	7.4	9.6	30.7	16.0	59.9	14.9	27.0	15.2	29.5	9.0	46.5	54.0	15.0	
31	?c13:6	j2K+2K	r256	a+		11.5	11.9	10.2	8.6	12.3	12.9	5.6	9.6	17.0	15.3	49.6	10.8	13.3	11.7	11.1	9.0	43.4	45.7	12.2	24b
32	?c13:4	j+	r+	a+		11.3	12.6	10.1	12.2	11.8	12.8	7.3	8.4	28.8	15.3	59.4	14.1	26.3	15.0	28.4	9.0	46.4	53.6	14.4	
33	?c13:4	j+	r256	a+		10.8	12.1	10.1	12.1	11.7	12.8	5.6	8.4	16.5	14.7	49.4	13.7	14.9	14.6	10.7	9.0	43.3	45.5	12.6	
34	?c13:4	j2K+2K	r+	a+		11.3	12.1	9.9	8.6	11.7	12.7	7.3	8.4	28.6	15.3	59.4	10.7	22.2	11.8	28.4	9.0	46.4	53.5	13.5	33b
35	?c13:4	j2K+2K	r256	a+		10.8	11.7	9.9	8.5	11.6	12.6	5.6	8.4	16.4	14.7	49.4	10.4	13.3	11.5	10.7	9.0	43.3	45.4	11.9	12 16a 24b 27...
36	?c13:4	j2K+2K	r256	a+	w+	10.8	11.7	9.9	8.5	11.6	12.6	5.6	8.4	16.6	14.8	51.5	10.4	13.6	11.5	10.7	9.0	43.5	56.2	11.9	18a ...32b...
37	?c13:4	j2K+2K	r256	a+	w1K	10.8	11.7	9.9	8.5	11.6	12.6	5.6	8.4	16.4	14.7	49.2	10.4	13.2	11.5	10.7	9.0	43.3	45.4	11.9	16a ...33b 35b
38	?c13:4	j2K+2K	r256	a+	w512	10.8	11.7	9.9	8.5	11.6	12.6	5.6	8.4	16.4	14.7	48.5	10.4	13.1	11.5	10.7	9.0	43.3	45.4	11.9	16a
39	?c13:4	j2K+2K	r256	a+	w256	10.7	11.6	9.9	8.5	11.6	12.6	5.6	8.4	16.2	14.6	46.7	10.3	12.3	11.5	10.7	9.0	42.7	44.5	11.8	16a
40	?c13:4	j2K+2K	r256	a+	w128	10.0	10.8	9.7	8.2	10.9	12.1	5.4	8.3	14.8	12.9	34.7	9.8	11.2	11.4	10.7	9.0	41.0	33.7	11.2	16a
41	?c13:4	j2K+2K	r256	a+	w64	8.9	9.5	9.3	7.7	9.4	11.0	5.3	7.6	11.7	10.3	22.4	8.9	10.2	10.7	9.0	8.7	25.3	22.2	9.9	16a
42	?c13:4	j2K+2K	r256	a+	w32	7.2	8.3	8.4	6.7	7.2	9.1	5.1	6.8	8.8	7.9	13.6	7.5	8.7	8.8	6.7	8.0	14.2	14.2	8.1	16a
43	?c13:4	j2K+2K	r256	a+	w16	5.4	7.1	6.6	5.3	5.2	6.3	4.4	5.7	6.5	5.8	8.3	5.8	6.7	6.3	4.9	6.0	8.1	8.7	6.1	16a
44	?c13:4	j2K+2K	r256	a+	w8	3.8	4.8	4.1	4.0	3.5	4.1	3.6	4.1	4.3	3.8	5.2	4.0	4.6	4.4	3.5	4.2	4.9	5.5	4.2	16a
45	?c13:4	j2K+2K	r256	a+	w4	2.2	2.9	2.5	2.7	2.4	2.6	2.8	2.6	2.8	2.4	3.2	2.6	2.9	2.9	2.3	2.8	3.2	3.3	2.7	16a
46	?c13:4	j2K+2K	r256	a+	dw2K	10.8	11.7	9.8	8.5	11.5	12.5	5.6	8.4	15.9	14.6	41.7	10.3	12.9	11.5	10.6	9.0	38.1	44.0	11.7	17a
47	?c13:4	j2K+2K	r256	a+	dw1K	10.8	11.6	9.8	8.4	11.3	12.1	5.4	8.4	14.9	13.5	39.8	10.3	12.3	11.3	10.5	8.8	34.2	42.9	11.5	17a
48	?c13:4	j2K+2K	r256	a+	dw512	10.1	10.8	9.5	8.3	10.5	11.1	5.3	8.4	13.0	12.4	34.6	9.8	11.1	10.8	10.2	8.6	28.1	40.5	10.9	17a
49	?c13:4	j2K+2K	r256	a+	dw256	8.9	9.4	8.8	7.6	8.9	9.5	5.0	7.7	10.2	10.6	25.4	8.6	9.5	9.9	9.0	7.5	20.5	18.8	9.4	17a
50	?c13:4	j2K+2K	r256	a+	dw128	7.2	8.1	7.8	6.2	6.8	7.6	4.7	7.0	7.2	8.3	11.0	7.2	7.8	8.4	6.7	6.5	12.9	12.0	7.5	17a
51	?c13:4	j2K+2K	r256	a+	dw64	5.4	6.5	6.5	5.2	5.1	5.8	4.3	5.6	5.4	6.0	7.5	5.7	6.2	6.5	4.7	5.3	7.7	8.7	5.8	17a
52	?c13:4	j2K+2K	r256	a+	dw32	3.9	5.3	4.9	4.2	3.7	4.2	3.7	4.5	4.2	4.2	5.3	4.3	4.7	4.9	3.6	4.0	4.8	5.9	4.4	17a
53	?c13:4	j2K+2K	r256	a+	dw16	2.8	4.1	3.4	3.3	2.8	3.1	3.0	3.3	3.3	3.0	3.9	3.2	3.6	3.8	2.8	3.2	3.3	4.1	3.3	17a
54	?c13:4	j2K+2K	r256	a+	dw8	2.0	2.7	2.4	2.6	2.1	2.4	2.5	2.4	2.6	2.2	3.1	2.4	2.8	2.8	2.1	2.6	2.7	3.2	2.5	17a
55	?c13:4	j2K+2K	r256	a+	dw4	1.6	2.0	1.7	1.9	1.7	1.8	2.0	1.9	2.2	1.7	2.4	1.9	2.3	2.1	1.7	2.0	2.5	2.6	2.0	17a

						egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
56	?c13:4	j2K+2K	r256	a+	iInf	10.8	11.7	9.9	8.5	11.6	12.6	5.6	8.4	16.6	14.8	51.5	10.4	13.6	11.5	10.7	9.0	43.5	56.2	11.9	15a
57	?c13:4	j2K+2K	r256	a+	i*2	10.8	11.7	9.9	8.5	11.6	12.6	5.6	8.4	16.6	14.8	51.5	10.4	13.6	11.5	10.7	9.0	43.5	56.2	11.9	14a
58	?c13:4	j2K+2K	r256	a+	LB	10.8	11.7	9.9	8.5	11.6	12.7	5.4	8.4	14.6	14.7	42.7	10.4	14.2	11.5	13.2	6.3	61.3	55.8	11.6	34a 35b
59	?c13:4	j2K+2K	r256	a+	LC	11.7	13.2	10.9	8.0	11.5	13.2	5.5	8.4	15.7	15.0	45.1	10.1	17.2	12.4	16.5	5.7	75.8	69.5	12.0	35b
60	?c13:4	j2K+2K	r256	a+	LD	11.7	13.2	10.9	8.0	11.5	13.2	5.4	8.4	14.4	15.0	40.5	10.2	17.7	12.4	17.7	4.8	73.2	69.5	11.7	34b 35b
61	?c13:4	j2K+2K	r256	a+	LE	12.4	14.3	11.8	7.7	11.3	13.4	5.9	8.5	14.4	15.1	43.9	10.0	20.5	12.7	19.1	4.5	73.6	72.3	12.0	35b
62	?c13:4	j2K+2K	r256	aComp		10.8	11.7	8.3	5.6	10.9	9.2	5.5	7.7	16.4	6.5	49.4	6.8	12.2	8.5	10.7	9.0	43.3	45.4	9.9	32b
63	?c13:4	j2K+2K	r256	aInsp		6.1	4.7	5.5	3.5	3.5	3.4	2.6	3.7	5.6	4.9	4.2	3.9	4.5	4.4	2.6	4.3	3.5	4.7	4.0	32b
64	?c13:4	j2K+2K	r256	a-		6.0	4.0	5.0	2.7	2.9	2.9	2.5	3.4	3.7	4.8	3.4	3.2	4.1	3.0	2.5	4.2	3.5	3.5	3.4	32b
65	?c13:4	j2K+2K	r128	a+		10.2	10.6	9.8	8.3	11.0	12.2	5.5	8.2	15.1	12.7	35.2	9.9	12.7	11.5	10.7	9.0	41.0	44.2	11.4	33b
66	?c13:4	j2K+2K	r64	a+		8.9	9.4	9.4	7.8	9.0	11.0	5.3	7.5	11.0	9.8	20.4	8.7	11.4	10.9	10.3	8.8	21.0	27.7	10.0	33b
67	?c13:4	j2K+2K	r32	a+		5.5	7.1	5.7	5.4	4.4	6.1	4.6	6.3	5.3	4.7	5.4	4.9	6.0	5.8	4.1	5.9	6.4	6.6	5.5	33b
68	?c13:4	j2K+2K	r-	a+		4.9	5.9	4.9	4.3	4.3	5.0	3.3	5.1	4.7	4.0	3.5	4.3	6.0	5.2	3.3	4.2	3.4	4.9	4.4	33b
69	?c13:2	j+	r+	a+		11.0	11.5	10.0	11.5	10.9	12.8	6.7	8.1	28.1	13.9	59.4	13.2	24.9	14.5	27.4	9.0	46.1	53.6	13.8	
70	?c13:2	j2K+2K	r256	a+		10.4	10.9	9.7	8.2	10.7	12.5	5.5	8.1	16.2	13.4	49.3	9.8	12.8	11.3	10.2	9.0	43.0	45.4	11.5	24b 27
71	?c13	j+	r+	a+		10.3	11.4	9.4	10.9	9.1	11.5	6.7	6.2	26.1	13.0	58.4	11.3	23.2	13.8	24.3	9.0	45.9	53.6	12.7	
72	?c13	j2K+2K	r256	a+		9.8	10.8	9.1	7.8	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.8	12.6	10.7	9.9	9.0	42.8	45.5	10.7	12 22b 23b 24b
73	?c13	j2K+2K	r256	a+	w+	9.8	10.8	9.1	7.8	9.0	11.4	5.6	6.2	15.7	12.4	50.6	8.8	12.9	10.7	9.9	9.0	42.9	55.9	10.8	18a
74	?c13	j2K+2K	r256	a+	iInf	9.8	10.8	9.1	7.8	9.0	11.4	5.6	6.2	15.7	12.4	50.6	8.8	12.9	10.7	9.9	9.0	42.9	55.9	10.8	15a
75	?c13	j2K+2K	r256	a+	i*2	9.8	10.8	9.1	7.8	9.0	11.4	5.6	6.2	15.7	12.4	50.6	8.8	12.9	10.7	9.9	9.0	42.9	55.9	10.8	14a
76	?c13	j2K+2K	r256	a+	LB	9.8	10.8	9.1	7.8	9.0	11.4	5.4	6.2	13.9	12.4	42.1	8.8	13.4	10.7	12.1	6.3	60.7	55.8	10.5	34a
77	?c13	j2K+2K	r256	a+	LC	10.7	12.2	10.1	7.3	8.9	11.8	5.4	6.3	15.0	12.7	44.4	8.6	16.1	11.1	15.1	5.7	75.6	69.6	10.9	
78	?c13	j2K+2K	r256	a+	LD	10.7	12.2	10.1	7.3	8.9	11.8	5.3	6.3	13.8	12.7	39.9	8.6	16.5	11.1	16.0	4.8	73.1	69.6	10.6	34b
79	?c13	j2K+2K	r256	a+	LE	11.4	13.3	10.8	7.0	8.7	12.0	5.9	6.3	13.8	12.8	43.1	8.4	19.0	11.3	17.2	4.5	73.5	72.4	10.8	
80	?c13	j2K+2K	r64	a+		8.0	8.9	8.6	7.1	7.5	10.0	5.3	5.8	10.6	8.6	20.2	7.5	10.9	10.0	9.4	8.8	20.8	27.7	9.1	
81	?c13	j2K+64	r256	a+		9.8	10.8	9.1	7.7	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.7	12.5	10.7	9.9	9.0	42.8	45.5	10.7	29a
82	?c13	j2K+32	r256	a+		9.8	10.8	9.1	7.7	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.6	12.4	10.7	9.9	9.0	42.8	45.5	10.7	29a
83	?c13	j2K+16	r256	a+		9.8	10.8	9.1	7.7	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.6	12.2	10.7	9.9	9.0	42.8	45.4	10.7	29a
84	?c13	j2K+8	r256	a+		9.8	10.8	9.1	7.7	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.5	12.2	10.7	9.9	9.0	42.8	45.4	10.7	29a
85	?c13	j2K+4	r256	a+		9.8	10.8	9.1	7.7	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.5	12.0	10.0	9.9	9.0	42.8	45.4	10.6	29a
86	?c13	j2K+2	r256	a+		9.8	10.8	9.1	7.7	9.0	11.3	5.6	6.2	15.1	12.4	48.5	8.4	12.0	10.0	9.9	9.0	42.8	45.4	10.6	29a
87	?c13	j2K	r256	a+		9.8	7.7	9.0	7.4	8.8	11.1	5.6	6.2	14.9	12.4	47.3	7.8	12.0	8.8	9.9	9.0	42.8	45.4	10.2	28a 29a
88	?c13	j16+8	r256	a+		9.8	10.8	9.1	7.7	9.0	11.4	5.6	6.2	15.5	12.4	48.5	8.5	12.2	10.6	9.9	9.0	42.8	45.4	10.6	
89	?c13	j16+8	r64	a+		8.0	8.9	8.6	7.1	7.4	10.0	5.3	5.8	10.6	8.6	20.2	7.4	10.6	10.0	9.5	8.8	20.2	27.7	9.1	
90	?c13	j16	r256	a+		9.8	7.7	9.0	7.3	8.8	11.1	5.6	6.2	14.9	12.4	47.3	7.8	12.0	8.7	9.9	9.0	42.8	45.4	10.1	28a
91	?c13	j16	r-	a+		4.6	5.0	4.7	4.1	3.9	4.8	3.3	4.4	4.6	3.8	3.5	3.9	5.7	4.7	3.2	4.2	3.4	4.9	4.2	22a 23a
92	?c13	j8	r256	a+		9.8	7.7	9.0	7.3	8.8	11.1	5.6	6.2	14.9	12.4	47.3	7.8	11.9	8.7	9.9	9.0	42.8	45.4	10.1	28a
93	?c13	j4	r256	a+		9.8	7.7	9.0	7.3	8.7	11.0	5.5	6.2	14.9	12.3	47.3	7.8	11.8	8.4	9.9	9.0	42.8	45.4	10.1	28a
94	?c13	j2	r256	a+		9.8	7.3	9.0	7.0	8.6	10.7	5.5	6.2	14.7	12.3	47.3	7.6	11.8	8.1	9.9	9.0	42.8	45.4	9.9	28a
95	?c13	j1	r256	a+		9.8	7.0	8.9	6.7	8.2	10.1	5.5	6.2	14.6	12.1	47.2	7.4	11.6	7.4	9.9	9.0	42.8	45.4	9.7	28a
96	?c13	jP	r256	a+		9.8	10.9	9.1	7.4	8.2	9.6	5.6	6.2	15.2	11.9	48.4	8.6	12.2	9.9	9.9	9.0	42.8	45.5	10.4	
97	?c13	j-	r256	a+		9.7	6.7	8.5	6.0	7.2	8.0	5.5	5.8	14.3	11.5	47.1	6.8	11.0	6.2	9.9	9.0	42.8	45.4	9.0	28a
98	?c12	j2K+2K	r256	a+		8.6	10.6	9.0	7.7	8.8	11.2	5.6	6.2	15.2	12.2	48.3	8.4	12.4	10.3	9.9	9.0	42.8	45.5	10.5	22b 23b
99	?c12	j16	r-	a+		4.5	5.0	4.6	4.1	3.9	4.7	3.3	4.4	4.6	3.8	3.5	3.9	5.7	4.7	3.2	4.2	3.4	4.9	4.1	22a 23a
100	?c11	j2K+2K	r256	a+		7.7	10.6	8.8	7.5	8.5	11.0	5.6	6.2	14.9	11.9	48.2	8.0	12.1	10.1	9.8	8.9	42.8	45.4	10.2	22b 23b
101	?c11	j16	r-	a+		4.4	5.0	4.6	4.1	3.8	4.7	3.3	4.4	4.5	3.7	3.5	3.8	5.7	4.6	3.2	4.2	3.4	4.9	4.1	22a 23a
102	?c10	j2K+2K	r256	a+		7.2	10.4	8.7	7.4	8.1	10.8	5.6	6.1	14.5	11.6	48.0	7.5	11.9	9.8	9.7	8.6	42.8	45.4	10.0	22b 23b
103	?c10	j2K+2K	r64	a+		6.5	8.8	8.3	6.9	7.0	9.6	5.3	5.7	10.2	8.3	20.2	6.8	10.6	9.3	9.2	8.5	21.0	27.3	8.7	
104	?c10	j16+8	r+	a+		7.3	10.7	8.7	7.3	8.1	10.8	6.7	6.1	22.8	12.0	57.5	7.5	16.7	9.9	22.8	8.6	45.8	53.4	10.9	33a
105	?c10	j16+8	r256	a+		7.2	10.4	8.7	7.3	8.1	10.8	5.6	6.1	14.5	11.6	48.0	7.4	11.6	9.7	9.7	8.6	42.8	45.4	9.9	33a
106	?c10	j16+8	r128	a+		7.0	9.7	8.6	7.2	7.9	10.4	5.4	6.1	13.5	10.3	34.5	7.2	11.2	9.7	9.7	8.6	40.6	44.1	9.6	33a
107	?c10	j16+8	r64	a+		6.5	8.8	8.3	6.9	7.0	9.6	5.3	5.7	10.2	8.3	20.2	6.7	10.3	9.2	9.2	8.5	20.0	27.3	8.7	12 13 27b 32a...
108	?c10	j16+8	r64	a+	w+	6.5	8.8	8.3	6.9	7.0	9.6	5.3	5.7	10.2	8.3	20.2	6.7	10.3	9.2	9.2	8.5	20.0	27.3	8.7	18a ...33a 35a
109	?c10	j16+8	r64	a+	iInf	6.5	8.8	8.3	6.9	7.0	9.6	5.3	5.7	10.2	8.3	20.2	6.7	10.3	9.2	9.2	8.5	20.0	27.3	8.7	15a
110	?c10	j16+8	r64	a+	i*2	6.5	8.8	8.3	6.9	7.0	9.6	5.3	5.7	10.2	8.3	20.2	6.7	10.3	9.2	9.2	8.5	20.0	27.3	8.7	14a

						egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
111	?c10	j16+8	r64	a+	LB	6.5	8.8	8.2	6.8	6.9	9.6	5.1	5.7	8.1	8.3	15.7	6.7	10.5	9.2	7.7	6.0	17.7	23.1	8.1	34a 35a
112	?c10	j16+8	r64	a+	LC	6.8	9.4	8.9	6.4	6.8	9.6	5.2	5.6	8.2	8.2	15.8	6.5	11.7	9.1	7.7	5.4	16.5	22.0	8.1	35a
113	?c10	j16+8	r64	a+	LD	6.8	9.4	8.9	6.4	6.8	9.6	5.1	5.6	7.3	8.2	14.1	6.5	11.5	9.1	7.0	4.5	15.1	20.2	7.8	34b 35a
114	?c10	j16+8	r64	a+	LE	7.0	9.8	9.2	6.1	6.7	9.6	5.4	5.6	7.3	8.1	15.3	6.4	12.5	8.9	7.1	4.2	15.7	21.3	7.8	35a
115	?c10	j16+8	r64	aComp		6.5	8.8	7.3	5.1	6.8	7.5	5.2	5.7	10.2	5.3	20.2	5.1	9.8	7.0	9.2	8.5	20.1	27.3	7.7	32a
116	?c10	j16+8	r64	aInsp		4.7	4.4	5.0	3.3	3.2	3.3	2.6	3.3	5.1	4.3	4.1	3.4	4.4	4.2	2.5	4.3	3.5	4.7	3.7	32a
117	?c10	j16+8	r64	a-		4.6	3.7	4.6	2.6	2.7	2.8	2.5	3.0	3.6	4.2	3.4	2.9	3.9	2.9	2.4	4.2	3.5	3.5	3.3	32a
118	?c10	j16+8	r32	a+		4.7	6.7	5.3	5.1	4.0	5.7	4.6	5.0	5.1	4.4	5.4	4.3	5.7	5.4	3.9	5.5	6.4	7.6	5.1	33a
119	?c10	j16+8	r-	a+		4.3	5.6	4.6	4.2	3.8	4.8	3.3	4.4	4.6	3.7	3.5	3.9	5.7	4.9	3.2	4.2	3.4	4.9	4.2	33a
120	?c10	j16	r64	a+		6.5	6.7	8.2	6.5	6.9	9.4	5.3	5.7	10.1	8.3	20.0	6.3	10.2	7.8	9.2	8.5	20.1	27.3	8.3	
121	?c10	j16	r-	a+		4.3	5.0	4.6	4.1	3.8	4.7	3.3	4.4	4.5	3.7	3.5	3.8	5.7	4.6	3.2	4.2	3.4	4.9	4.1	22a 23a
122	?c9	j2K+2K	r256	a+		6.3	10.4	8.4	7.2	7.7	10.4	5.6	6.1	14.1	10.9	47.5	6.9	11.6	9.3	9.7	8.3	42.8	45.4	9.6	22b 23b
123	?c9	j16	r-	a+		4.1	5.0	4.6	4.1	3.7	4.7	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.5	3.2	4.1	3.4	4.9	4.1	22a 23a
124	?c8	j2K+2K	r256	a+		5.8	10.3	8.1	7.0	7.2	9.9	5.5	6.1	13.8	10.4	46.9	6.2	11.3	8.5	9.6	8.2	42.8	45.4	9.2	
125	?c8	j16	r-	a+		4.0	5.0	4.5	4.0	3.7	4.6	3.3	4.3	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	
126	?c7	j2K+2K	r256	a+		5.3	8.2	7.7	6.4	6.8	9.1	5.5	6.0	13.4	9.6	46.0	5.5	10.8	8.1	9.5	7.8	42.8	45.4	8.7	
127	?c7	j16	r-	a+		3.9	4.9	4.4	3.9	3.6	4.6	3.3	4.3	4.5	3.6	3.5	3.4	5.5	4.4	3.2	4.1	3.4	4.9	4.0	
128	?c6	j2K+2K	r256	a+		5.0	8.0	7.3	5.9	6.2	7.8	5.3	5.8	12.8	8.8	44.8	4.9	10.2	7.1	9.4	7.4	42.8	45.4	8.1	
129	?c6	j16	r-	a+		3.8	4.9	4.4	3.8	3.5	4.4	3.3	4.2	4.5	3.6	3.5	3.3	5.4	4.2	3.2	4.0	3.4	4.9	3.9	
130	?c5	j2K+2K	r256	a+		4.7	6.4	6.8	5.0	5.4	6.8	5.0	5.5	12.0	7.9	43.7	4.4	9.9	5.9	9.2	7.3	42.8	45.3	7.3	
131	?c5	j16	r-	a+		3.7	4.6	4.2	3.6	3.3	4.2	3.2	4.2	4.4	3.5	3.5	3.1	5.3	3.9	3.2	3.9	3.4	4.9	3.8	
132	?c4	j2K+2K	r256	a+		4.0	6.2	6.5	4.4	4.6	6.0	4.7	5.1	10.2	7.0	42.5	4.1	9.4	5.1	8.8	6.5	42.8	45.3	6.6	
133	?c4	j16	r-	a+		3.4	4.5	4.1	3.4	3.1	4.0	3.1	4.0	4.2	3.4	3.5	3.0	5.2	3.6	3.1	3.7	3.4	4.9	3.7	
134	?b13	j2K+2K	r256	a+		6.7	9.7	8.7	7.3	8.8	10.2	5.5	6.1	15.1	12.2	48.1	8.6	12.3	10.3	9.6	7.6	42.8	45.5	10.0	
135	?b13	j16	r-	a+		4.2	4.9	4.6	4.1	3.9	4.7	3.3	4.4	4.6	3.8	3.5	3.9	5.7	4.6	3.2	4.1	3.4	4.9	4.1	
136	?b12	j2K+2K	r256	a+		6.3	9.7	8.6	7.3	8.5	10.1	5.5	6.1	14.9	12.0	48.1	8.3	12.1	10.3	9.6	7.7	42.8	45.5	9.8	
137	?b12	j16	r-	a+		4.1	4.9	4.6	4.1	3.9	4.7	3.3	4.4	4.6	3.8	3.5	3.9	5.7	4.6	3.2	4.1	3.4	4.9	4.1	
138	?b11	j2K+2K	r256	a+		5.8	9.6	8.6	7.2	8.3	10.0	5.5	6.1	14.8	11.8	47.9	7.9	11.9	9.9	9.5	7.7	42.8	45.5	9.7	
139	?b11	j16	r-	a+		4.0	4.9	4.6	4.1	3.8	4.7	3.3	4.4	4.6	3.7	3.5	3.8	5.7	4.6	3.2	4.1	3.4	4.9	4.1	
140	?b10	j2K+2K	r256	a+		5.3	9.6	8.4	7.1	8.0	9.6	5.4	6.0	14.5	11.5	47.7	7.5	11.7	9.9	9.5	7.6	42.8	45.4	9.4	
141	?b10	j16	r-	a+		3.9	4.9	4.6	4.1	3.8	4.6	3.3	4.4	4.5	3.7	3.5	3.8	5.7	4.6	3.2	4.1	3.4	4.9	4.1	
142	?b9	j2K+2K	r256	a+		5.1	9.6	8.3	6.8	7.7	9.3	5.4	6.0	14.1	11.1	47.4	6.9	11.5	9.6	9.5	7.7	42.8	45.4	9.2	22b 23b
143	?b9	j16	r-	a+		3.9	4.9	4.5	4.1	3.8	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.5	3.2	4.1	3.4	4.9	4.1	22a 23a
144	?b8:8	j16	r-	a+		4.9	5.2	4.9	4.2	4.2	4.9	3.3	5.2	4.7	4.0	3.5	4.2	5.8	4.9	3.4	4.2	3.4	4.9	4.3	25b
145	?b8:6	j16	r-	a+		4.9	5.2	4.9	4.2	4.2	4.9	3.3	5.1	4.7	4.0	3.5	4.2	5.8	4.8	3.4	4.2	3.4	4.9	4.3	25b
146	?b8:4	j16	r-	a+		4.7	5.2	4.9	4.2	4.2	4.9	3.3	5.1	4.6	4.0	3.5	4.2	5.7	4.8	3.3	4.2	3.4	4.9	4.3	25b 27
147	?b8:2	j16	r-	a+		4.4	5.1	4.7	4.2	4.1	4.9	3.3	4.8	4.6	3.9	3.5	4.0	5.6	4.8	3.3	4.2	3.4	4.9	4.2	25b 27
148	?b8	j2K+2K	r256	a+		4.9	9.5	8.0	6.6	7.2	8.9	5.4	6.0	13.8	10.3	46.8	6.3	11.2	9.2	9.4	7.7	42.8	45.4	8.9	22b 23b
149	?b8	j2K+64	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.7	4.8	3.2	4.1	3.4	4.9	4.1	
150	?b8	j2K+32	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.8	3.2	4.1	3.4	4.9	4.1	
151	?b8	j2K+16	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.8	3.2	4.1	3.4	4.9	4.1	
152	?b8	j2K+8	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.8	3.2	4.1	3.4	4.9	4.1	
153	?b8	j2K+4	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.7	3.2	4.1	3.4	4.9	4.1	
154	?b8	j2K+2	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.7	3.2	4.1	3.4	4.9	4.1	
155	?b8	j2K	r-	a+		3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	
156	?b8	j16+8	r64	a+		4.7	8.2	7.8	6.4	6.5	8.5	5.2	5.6	10.0	7.9	20.0	5.9	9.9	8.6	9.0	7.7	20.1	27.7	8.0	
157	?b8	j16+8	r-	a+		3.8	5.5	4.5	4.1	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.6	4.8	3.2	4.1	3.4	4.9	4.1	
158	?b8	j16	r64	a+		4.7	6.4	7.7	6.1	6.4	8.3	5.2	5.6	9.9	7.9	19.9	5.6	9.8	7.4	9.0	7.7	21.0	27.7	7.7	
159	?b8	j16	r-	a+		3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	12 16b 22a...
160	?b8	j16	r-	a+	w+	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	18a ...23a 25b
161	?b8	j16	r-	a+	w1K	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	16b
162	?b8	j16	r-	a+	w512	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	16b
163	?b8	j16	r-	a+	w256	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	16b
164	?b8	j16	r-	a+	w128	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	16b
165	?b8	j16	r-	a+	w64	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.5	4.5	3.2	4.1	3.4	4.9	4.0	16b

						egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
166	?b8	j16	r-	a+	w32	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.3	4.5	3.2	4.0	3.4	4.9	4.0	16b
167	?b8	j16	r-	a+	w16	3.7	4.8	4.3	4.0	3.6	4.5	3.3	4.3	4.3	3.6	3.5	3.5	4.9	4.4	3.2	3.8	3.4	4.9	3.9	16b
168	?b8	j16	r-	a+	w8	3.1	4.1	3.5	3.5	3.0	3.6	3.2	3.6	3.7	3.2	3.4	3.1	4.1	3.8	2.9	3.3	3.4	4.4	3.5	16b
169	?b8	j16	r-	a+	w4	2.1	2.8	2.4	2.5	2.3	2.5	2.8	2.5	2.7	2.3	3.1	2.4	2.8	2.7	2.2	2.7	3.0	3.2	2.6	16b
170	?b8	j16	r-	a+	dw2K	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	17b
171	?b8	j16	r-	a+	dw1K	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.5	4.5	3.2	4.1	3.4	4.9	4.0	17b
172	?b8	j16	r-	a+	dw512	3.8	4.8	4.5	4.0	3.7	4.5	3.3	4.4	4.5	3.7	3.5	3.6	5.5	4.5	3.2	4.1	3.4	4.9	4.0	17b
173	?b8	j16	r-	a+	dw256	3.8	4.8	4.4	4.0	3.6	4.5	3.3	4.4	4.4	3.6	3.4	3.6	5.3	4.5	3.1	4.0	3.4	4.9	4.0	17b
174	?b8	j16	r-	a+	dw128	3.7	4.7	4.3	4.0	3.6	4.3	3.2	4.3	4.3	3.5	3.4	3.5	5.0	4.4	3.1	3.8	3.4	4.8	3.9	17b
175	?b8	j16	r-	a+	dw64	3.5	4.5	4.0	3.8	3.4	4.0	3.2	4.1	4.0	3.4	3.4	3.4	4.6	4.2	2.9	3.6	3.3	4.6	3.7	17b
176	?b8	j16	r-	a+	dw32	3.1	4.1	3.5	3.5	3.0	3.5	3.0	3.9	3.6	3.0	3.2	3.2	4.0	3.9	2.7	3.1	3.2	4.2	3.4	17b
177	?b8	j16	r-	a+	dw16	2.5	3.4	2.9	3.0	2.5	2.8	2.8	3.1	3.1	2.6	3.1	2.7	3.4	3.3	2.4	2.8	3.0	3.7	2.9	17b
178	?b8	j16	r-	a+	dw8	2.0	2.6	2.2	2.5	2.0	2.3	2.5	2.3	2.5	2.1	2.8	2.2	2.7	2.7	2.0	2.4	2.7	3.2	2.4	17b
179	?b8	j16	r-	a+	dw4	1.5	1.9	1.7	1.9	1.7	1.8	1.9	1.8	2.1	1.7	2.4	1.8	2.2	2.0	1.7	2.0	2.5	2.6	1.9	17b
180	?b8	j16	r-	a+	iInf	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	15a
181	?b8	j16	r-	a+	i*2	3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	14a
182	?b8	j16	r-	a+	LB	3.8	4.8	4.5	4.0	3.7	4.6	2.8	4.4	3.3	3.7	2.9	3.6	4.6	4.5	2.6	3.2	2.6	3.9	3.6	34a
183	?b8	j16	r-	a+	LC	3.1	4.2	4.0	3.8	3.6	4.4	2.5	4.1	3.1	3.4	2.9	3.5	4.4	4.2	2.5	3.0	2.5	3.8	3.4	
184	?b8	j16	r-	a+	LD	3.1	4.2	4.0	3.8	3.6	4.3	2.2	4.1	2.7	3.4	2.7	3.5	3.9	4.2	2.3	2.6	2.3	3.3	3.2	34b
185	?b8	j16	r-	a+	LE	2.8	3.7	3.5	3.6	3.5	4.2	2.1	3.8	2.7	3.2	2.6	3.4	3.8	4.0	2.4	2.4	2.3	3.4	3.1	
186	?b8	j16	r-	aInsp		3.3	3.5	3.6	2.9	2.6	3.0	2.5	3.2	3.7	2.9	2.9	2.7	3.8	3.4	2.4	3.5	2.7	3.8	3.1	
187	?b8	j8	r-	a+		3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	
188	?b8	j4	r-	a+		3.8	4.8	4.5	4.0	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.5	3.2	4.1	3.4	4.9	4.0	
189	?b8	j2	r-	a+		3.8	4.8	4.5	4.0	3.7	4.5	3.3	4.4	4.5	3.7	3.5	3.6	5.6	4.4	3.2	4.1	3.4	4.9	4.0	
190	?b8	j1	r-	a+		3.8	4.7	4.5	4.0	3.7	4.5	3.3	4.4	4.5	3.7	3.5	3.6	5.5	4.3	3.2	4.1	3.4	4.9	4.0	
191	?b8	jP	r-	a+		3.8	5.5	4.5	4.2	3.7	4.6	3.3	4.4	4.5	3.7	3.5	3.7	5.7	4.8	3.2	4.1	3.4	4.9	4.1	
192	?b8	j-	r-	a+		3.8	4.6	4.4	3.9	3.6	4.4	3.3	4.4	4.5	3.7	3.5	3.5	5.5	4.1	3.2	4.0	3.4	4.9	4.0	
193	?b8	j-	r-	aInsp		3.3	3.3	3.5	2.8	2.6	2.8	2.5	3.1	3.7	2.9	2.9	2.6	3.7	3.2	2.4	3.4	2.7	3.8	3.0	
194	?b7	j2K+2K	r256	a+		4.8	7.7	7.7	6.3	6.7	8.4	5.3	6.0	13.4	9.6	46.3	5.8	10.7	8.5	9.4	7.7	42.8	45.4	8.5	22b 23b
195	?b7	j16	r-	a+		3.8	4.8	4.5	4.0	3.6	4.5	3.3	4.4	4.5	3.6	3.5	3.5	5.5	4.4	3.2	4.0	3.4	4.9	4.0	22a 23a
196	?b6	j2K+2K	r256	a+		4.7	7.5	7.4	5.8	6.1	7.9	5.2	5.7	12.8	8.9	45.2	5.2	10.3	7.8	9.3	7.5	42.8	45.4	8.1	22b 23b
197	?b6	j16	r-	a+		3.8	4.8	4.4	3.9	3.5	4.4	3.2	4.2	4.5	3.6	3.5	3.3	5.5	4.3	3.2	4.0	3.4	4.9	3.9	22a 23a
198	?b5	j2K+2K	r256	a+		4.3	7.0	7.0	5.2	5.4	7.3	5.1	5.5	11.8	7.8	44.1	4.6	9.9	7.0	9.1	7.5	42.8	45.3	7.5	
199	?b5	j16	r-	a+		3.7	4.7	4.3	3.7	3.4	4.3	3.2	4.2	4.4	3.5	3.5	3.2	5.4	4.2	3.2	4.0	3.4	4.9	3.9	
200	?b4	j2K+2K	r256	a+		4.0	6.1	6.6	4.7	4.9	6.4	4.8	5.3	10.9	6.9	43.3	4.3	9.5	5.6	8.8	6.9	42.8	45.3	6.9	
201	?b4	j16	r-	a+		3.5	4.5	4.2	3.5	3.2	4.1	3.1	4.1	4.3	3.4	3.5	3.1	5.3	3.8	3.1	3.9	3.4	4.9	3.7	
202	?a9	j2K+2K	r256	a+		4.4	9.3	7.4	5.8	5.6	7.6	5.2	5.7	13.2	6.9	46.3	5.7	10.2	6.4	9.2	7.7	42.2	45.4	7.9	
203	?a9	j16	r-	a+		3.7	4.8	4.4	3.9	3.4	4.4	3.2	4.3	4.5	3.5	3.5	3.5	5.5	4.1	3.2	4.0	3.4	4.9	3.9	
204	?a8	j2K+2K	r256	a+		4.4	9.3	7.3	5.6	5.6	7.4	5.2	5.7	13.0	6.9	45.5	5.3	10.1	6.4	9.2	7.7	42.2	45.3	7.8	
205	?a8	j16	r-	a+		3.7	4.8	4.4	3.8	3.4	4.4	3.2	4.3	4.5	3.5	3.5	3.4	5.4	4.1	3.2	4.0	3.4	4.9	3.9	
206	?a7	j2K+2K	r256	a+		4.4	7.5	7.1	5.4	5.5	6.8	5.0	5.7	12.5	6.7	45.0	5.0	9.8	6.2	9.2	7.7	42.2	45.3	7.5	22b 23b
207	?a7	j16	r-	a+		3.7	4.7	4.4	3.7	3.4	4.3	3.2	4.3	4.5	3.4	3.5	3.3	5.4	4.1	3.2	4.0	3.4	4.9	3.9	22a 23a
208	?a6	j2K+2K	r256	a+		4.3	7.2	6.8	5.0	5.1	6.6	4.9	5.5	11.4	6.4	43.9	4.6	9.1	5.8	9.2	7.1	42.2	45.3	7.2	22b 23b
209	?a6	j16	r-	a+		3.7	4.7	4.3	3.6	3.3	4.2	3.2	4.2	4.4	3.4	3.5	3.2	5.2	4.0	3.2	4.0	3.4	4.9	3.8	22a 23a
210	?a5	j2K+2K	r256	a+		3.9	6.3	6.5	4.5	4.7	5.8	4.8	5.2	10.3	5.7	43.3	4.3	8.9	5.1	9.1	7.1	42.1	45.3	6.6	22b 23b
211	?a5	j16	r-	a+		3.6	4.6	4.2	3.4	3.2	4.0	3.1	4.1	4.2	3.3	3.5	3.1	5.2	3.8	3.1	4.0	3.4	4.9	3.7	22a 23a
212	?a5	j16	r-	aInsp		3.1	3.3	3.4	2.6	2.4	2.7	2.4	3.0	3.5	2.7	2.9	2.4	3.7	3.0	2.3	3.2	2.7	3.8	2.9	
213	?a5	j-	r-	a+		3.6	4.4	4.1	3.3	3.1	3.8	3.1	4.0	4.2	3.2	3.5	3.0	5.1	3.4	3.1	3.9	3.4	4.9	3.6	
214	?a5	j-	r-	aInsp		3.1	3.2	3.3	2.5	2.3	2.5	2.4	2.9	3.5	2.6	2.9	2.4	3.6	2.7	2.3	3.2	2.7	3.8	2.8	12 27a
215	?a5	j-	r-	aInsp	w+	3.1	3.2	3.3	2.5	2.3	2.5	2.4	2.9	3.5	2.6	2.9	2.4	3.6	2.7	2.3	3.2	2.7	3.8	2.8	18a
216	?a5	j-	r-	aInsp	iInf	3.1	3.2	3.3	2.5	2.3	2.5	2.4	2.9	3.5	2.6	2.9	2.4	3.6	2.7	2.3	3.2	2.7	3.8	2.8	15a
217	?a5	j-	r-	aInsp	i*2	3.1	3.2	3.3	2.5	2.3	2.5	2.4	2.9	3.5	2.6	2.9	2.4	3.6	2.7	2.3	3.2	2.7	3.8	2.8	14a
218	?a5	j-	r-	aInsp	LB	3.1	3.2	3.3	2.5	2.3	2.5	2.1	2.9	2.8	2.6	2.6	2.3	3.3	2.7	2.1	2.7	2.2	3.2	2.6	34a
219	?a5	j-	r-	aInsp	LC	2.7	2.9	2.9	2.4	2.3	2.5	2.0	2.8	2.7	2.5	2.7	2.3	3.3	2.6	2.0	2.6	2.2	3.2	2.5	
220	?a5	j-	r-	aInsp	LD	2.7	2.9	2.9	2.4	2.3	2.5	1.8	2.8	2.4	2.5	2.5	2.3	3.1	2.6	1.9	2.2	2.0	2.9	2.4	34b

#	config	j	r	a	LE	egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
221	?a5	j-	r-	aInsp	LE	2.4	2.7	2.6	2.4	2.2	2.4	1.8	2.7	2.4	2.4	2.5	2.2	3.0	2.5	1.9	2.1	2.1	2.9	2.4	
222	?a5	j-	r-	a-		3.1	2.8	3.1	2.1	2.0	2.2	2.3	2.8	2.9	2.6	2.6	2.1	3.3	2.1	2.3	3.0	2.7	3.0	2.5	
223	?a4	j2K+2K	r256	a+		3.6	5.9	6.2	3.9	4.3	5.5	4.6	5.1	9.5	5.5	42.4	4.0	8.3	5.1	6.7	6.5	42.1	45.3	6.2	22b 23b
224	?a4	j16	r-	a+		3.4	4.5	4.1	3.2	3.0	3.8	3.1	4.1	4.1	3.2	3.5	3.0	5.0	3.7	2.9	3.9	3.4	4.9	3.6	22a 23a
225	?a3	j2K+2K	r256	a+		3.5	4.7	4.4	3.7	4.1	4.7	4.6	4.5	8.5	5.1	40.4	3.8	7.9	4.9	6.6	6.4	42.1	45.2	5.7	22b 23b
226	?a3	j16	r-	a+		3.3	3.7	3.2	3.1	3.0	3.5	3.1	3.8	4.0	3.1	3.4	2.9	4.8	3.7	2.9	4.1	3.4	4.9	3.5	22a 23a
227	?a2	j2K+2K	r256	a+		3.5	5.1	4.0	3.5	3.8	4.7	4.5	4.4	7.9	4.6	39.4	3.8	7.5	4.5	5.9	6.0	40.3	45.2	5.4	22b 23b
228	?a2	j16	r-	a+		3.3	3.9	3.0	3.0	2.8	3.5	3.1	3.7	3.8	2.9	3.4	2.9	4.7	3.5	2.8	3.7	3.4	4.9	3.4	22a 23a
229	?a1	j2K+2K	r256	a+		3.3	5.0	3.9	3.3	3.8	4.4	4.6	4.2	8.1	4.5	39.3	3.7	6.9	4.0	5.8	5.6	40.3	44.9	5.2	22b 23b
230	?a1	j16	r-	a+		3.1	4.0	2.6	2.9	2.8	3.4	3.1	3.5	3.9	3.0	3.4	2.8	4.5	3.1	2.8	3.3	3.4	4.9	3.3	22a 23a
231	?P:4(1.00)	j+	r+	a+		6.7	11.7	8.7	9.6	8.2	10.0	7.1	7.6	24.9	9.8	56.1	10.8	20.2	13.1	21.7	8.7	45.8	53.5	11.8	26b
232	?P:4(1.00)	j2K+2K	r256	a+		6.7	11.0	8.5	7.2	8.2	9.9	5.5	7.6	16.7	9.7	47.7	8.9	11.9	10.4	10.6	8.7	43.3	45.4	10.2	
233	?P:4(1.00)	j16	r-	a+		4.7	5.2	4.9	4.2	4.2	4.9	3.3	5.0	4.7	3.9	3.5	4.2	5.7	4.8	3.3	4.2	3.4	4.9	4.3	
234	?P:4(0.98)	j+	r+	a+		7.2	12.9	9.0	10.2	8.4	10.6	7.2	8.2	28.4	10.2	56.1	11.3	20.5	12.7	21.6	8.7	45.2	53.3	12.3	26b
235	?P:4(0.98)	j2K+2K	r256	a+		7.1	12.0	8.8	7.6	8.4	10.5	5.5	8.2	16.9	10.1	48.0	9.2	12.0	10.1	10.6	8.7	42.8	45.3	10.5	
236	?P:4(0.98)	j16	r-	a+		4.9	5.2	4.9	4.2	4.2	4.9	3.3	5.0	4.7	3.9	3.5	4.2	5.7	4.8	3.3	4.2	3.4	4.9	4.3	
237	?P:4(0.96)	j+	r+	a+		7.9	12.8	8.9	10.3	8.4	10.6	7.1	8.2	28.1	10.6	57.2	11.3	20.3	12.7	22.8	8.7	45.2	53.2	12.4	26b
238	?P:4(0.96)	j2K+2K	r256	a+		7.8	11.9	8.7	7.6	8.4	10.5	5.5	8.2	16.7	10.6	48.7	9.2	11.9	10.1	11.0	8.7	42.8	45.3	10.6	
239	?P:4(0.96)	j16	r-	a+		4.9	5.2	4.9	4.2	4.2	4.9	3.3	5.0	4.7	4.0	3.5	4.2	5.7	4.8	3.3	4.2	3.4	4.9	4.3	
240	?P:4(0.94)	j+	r+	a+		7.9	11.9	8.9	10.3	8.5	10.3	7.1	8.2	28.5	10.7	57.8	11.5	20.5	12.7	23.0	9.0	45.2	53.2	12.4	26b
241	?P:4(0.94)	j2K+2K	r256	a+		7.8	11.2	8.7	7.6	8.4	10.2	5.5	8.2	16.7	10.7	48.8	9.2	11.9	10.1	11.0	9.0	42.8	45.3	10.6	
242	?P:4(0.94)	j16	r-	a+		4.9	5.1	4.8	4.2	4.2	4.9	3.3	5.0	4.7	3.9	3.5	4.2	5.7	4.8	3.3	4.2	3.4	4.9	4.3	
243	?P:4(0.92)	j+	r+	a+		7.9	10.9	8.9	10.1	8.6	10.2	7.1	8.1	28.5	10.8	57.8	11.5	20.9	12.7	22.9	9.0	45.2	53.2	12.3	26b
244	?P:4(0.92)	j2K+2K	r256	a+		7.8	10.3	8.6	7.4	8.5	10.1	5.5	8.0	16.8	10.7	48.8	9.3	12.3	10.1	10.8	9.0	42.8	45.3	10.5	27
245	?P:4(0.92)	j16	r-	a+		4.9	5.0	4.8	4.2	4.2	4.8	3.3	5.0	4.7	3.9	3.5	4.2	5.8	4.8	3.3	4.2	3.4	4.9	4.3	27
246	?P:4(0.90)	j+	r+	a+		7.9	10.9	8.8	9.9	8.6	10.2	7.1	8.0	25.9	10.6	57.9	11.5	20.8	12.7	21.8	9.0	45.2	53.2	12.2	26b
247	?P:4(0.90)	j2K+2K	r256	a+		7.8	10.3	8.6	7.3	8.5	10.1	5.5	8.0	15.6	10.6	48.9	9.3	12.2	10.1	10.4	9.0	42.8	45.3	10.5	
248	?P:4(0.90)	j16	r-	a+		4.9	5.0	4.8	4.2	4.2	4.8	3.3	5.0	4.6	3.9	3.5	4.2	5.8	4.8	3.3	4.2	3.4	4.9	4.3	
249	?P:4(0.88)	j+	r+	a+		7.9	10.9	8.7	9.8	8.6	10.2	7.1	7.4	26.6	10.5	57.9	11.6	20.2	12.6	18.0	9.0	45.2	53.2	12.0	26b
250	?P:4(0.88)	j2K+2K	r256	a+		7.8	10.3	8.5	7.2	8.5	10.1	5.5	7.4	15.6	10.4	48.9	9.3	12.1	10.1	9.2	9.0	42.8	45.3	10.3	
251	?P:4(0.88)	j16	r-	a+		4.9	5.0	4.8	4.1	4.2	4.8	3.3	4.7	4.6	3.9	3.5	4.2	5.8	4.8	3.2	4.2	3.4	4.9	4.2	
252	?P:4(0.86)	j+	r+	a+		8.0	10.8	8.7	9.7	8.6	10.2	7.1	7.4	26.8	10.4	57.8	11.5	19.9	12.6	18.2	9.0	45.2	53.2	12.0	26b
253	?P:4(0.86)	j2K+2K	r256	a+		7.9	10.2	8.4	7.1	8.5	10.1	5.5	7.4	15.6	10.4	48.6	9.3	12.0	10.1	9.0	9.0	42.8	45.3	10.2	
254	?P:4(0.86)	j16	r-	a+		4.8	5.0	4.8	4.1	4.2	4.8	3.3	4.7	4.6	3.9	3.5	4.2	5.7	4.8	3.2	4.2	3.4	4.9	4.2	
255	?P:4(0.84)	j+	r+	a+		8.0	10.8	8.8	9.6	8.6	10.2	7.1	7.4	26.7	10.3	57.5	11.5	19.2	13.0	18.2	9.0	45.2	53.2	12.0	26b
256	?P:4(0.84)	j2K+2K	r256	a+		7.9	10.2	8.6	7.1	8.5	10.1	5.5	7.4	15.5	10.2	48.4	9.2	11.7	10.3	9.0	9.0	42.8	45.3	10.2	
257	?P:4(0.84)	j16	r-	a+		4.8	5.0	4.7	4.1	4.1	4.8	3.3	4.7	4.6	3.9	3.5	4.1	5.7	4.8	3.2	4.2	3.4	4.9	4.2	
258	?P:4(0.82)	j+	r+	a+		8.0	10.8	8.8	9.5	8.6	10.3	7.1	7.4	26.7	10.3	57.6	11.5	19.0	12.6	18.3	9.0	45.2	53.2	12.0	26b
259	?P:4(0.82)	j2K+2K	r256	a+		7.9	10.2	8.5	7.0	8.5	10.2	5.5	7.4	15.5	10.2	48.3	9.2	11.7	10.1	8.9	9.0	42.8	45.3	10.2	
260	?P:4(0.82)	j16	r-	a+		4.8	5.0	4.7	4.1	4.1	4.8	3.3	4.7	4.6	3.9	3.5	4.1	5.7	4.8	3.2	4.2	3.4	4.9	4.2	
261	?P:4(0.80)	j+	r+	a+		8.0	10.8	8.7	9.5	8.6	10.3	7.1	7.4	26.7	10.2	56.9	11.4	19.0	12.5	18.3	9.0	45.2	53.2	11.9	26b
262	?P:4(0.80)	j2K+2K	r256	a+		7.8	10.2	8.4	7.0	8.5	10.2	5.5	7.4	15.4	10.2	47.8	9.1	11.7	9.9	8.9	9.0	42.8	45.3	10.2	
263	?P:4(0.80)	j16	r-	a+		4.8	5.0	4.7	4.1	4.1	4.8	3.3	4.7	4.6	3.9	3.5	4.1	5.7	4.7	3.2	4.2	3.4	4.9	4.2	
264	?P:4(0.78)	j+	r+	a+		8.0	10.8	8.5	9.4	8.4	10.3	7.2	7.4	26.5	10.1	57.1	11.3	19.0	12.5	18.3	9.0	45.2	53.2	11.9	26b
265	?P:4(0.78)	j2K+2K	r256	a+		7.8	10.2	8.3	7.0	8.3	10.2	5.5	7.4	15.4	10.0	47.9	9.0	11.7	9.9	8.9	9.0	42.8	45.3	10.1	
266	?P:4(0.78)	j16	r-	a+		4.8	5.0	4.6	4.1	4.1	4.8	3.3	4.7	4.6	3.9	3.5	4.1	5.7	4.7	3.2	4.2	3.4	4.9	4.2	
267	?P:4(0.76)	j+	r+	a+		7.9	10.8	8.4	9.4	8.2	10.0	7.2	6.9	25.2	9.9	56.6	11.2	18.9	12.5	18.3	9.0	45.2	53.2	11.7	26b
268	?P:4(0.76)	j2K+2K	r256	a+		7.8	10.2	8.2	6.9	8.1	9.9	5.5	6.9	15.0	9.9	47.1	8.9	11.6	9.9	8.9	9.0	42.8	45.3	10.0	
269	?P:4(0.76)	j16	r-	a+		4.8	5.0	4.6	4.1	4.1	4.7	3.3	4.6	4.6	3.8	3.5	4.1	5.6	4.7	3.2	4.2	3.4	4.9	4.2	
270	?P:4(0.74)	j+	r+	a+		7.4	10.8	8.3	9.3	8.2	9.9	7.2	6.9	25.1	9.3	56.4	10.8	18.8	12.3	17.7	9.0	45.9	53.2	11.5	26b
271	?P:4(0.74)	j2K+2K	r256	a+		7.3	10.2	8.1	6.9	8.1	9.8	5.5	6.9	15.0	9.3	46.9	8.5	11.6	9.9	8.7	9.0	42.8	45.3	9.8	
272	?P:4(0.74)	j16	r-	a+		4.7	5.0	4.6	4.1	4.1	4.7	3.3	4.6	4.6	3.8	3.5	4.0	5.6	4.7	3.2	4.2	3.4	4.9	4.2	
273	?P:4(0.72)	j+	r+	a+		4.7	10.8	8.2	9.0	7.9	9.8	6.4	6.9	25.1	9.1	56.1	10.5	18.5	12.3	17.7	9.0	45.9	53.2	10.7	26b
274	?P:4(0.72)	j2K+2K	r256	a+		4.7	10.2	8.0	6.8	7.8	9.7	5.4	6.9	15.0	9.1	46.8	8.4	11.5	9.9	8.7	9.0	42.8	45.3	9.4	
275	?P:4(0.72)	j16	r-	a+		3.9	5.0	4.6	4.1	4.0	4.7	3.3	4.6	4.6	3.8	3.5	4.0	5.6	4.7	3.2	4.2	3.4	4.9	4.1	

					egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
276	?P:4(0.70)	j+	r+	a+	4.7	10.8	8.1	9.0	7.9	9.7	6.4	6.9	25.1	9.1	56.1	10.3	18.3	11.9	17.7	9.0	45.9	53.2	10.7	26b
277	?P:4(0.70)	j2K+2K	r256	a+	4.7	10.2	7.9	6.8	7.8	9.6	5.4	6.9	15.0	9.0	46.8	8.2	11.4	9.6	8.7	9.0	42.8	45.3	9.3	
278	?P:4(0.70)	j16	r-	a+	3.9	5.0	4.6	4.1	3.9	4.7	3.3	4.6	4.6	3.8	3.5	4.0	5.6	4.7	3.2	4.2	3.4	4.9	4.1	
279	?P:4(0.68)	j+	r+	a+	4.7	10.9	7.8	8.4	7.9	9.4	6.4	6.5	24.5	8.8	56.1	10.2	18.2	11.9	16.5	9.0	45.9	53.2	10.5	26b
280	?P:4(0.66)	j+	r+	a+	4.7	10.2	7.7	8.4	7.6	9.1	6.4	6.5	24.3	8.4	55.3	10.0	17.8	11.5	16.5	9.0	45.9	53.2	10.3	26b
281	?P:4(0.64)	j+	r+	a+	4.7	10.2	7.6	8.2	7.5	8.6	6.4	6.5	24.3	8.1	55.2	9.9	17.0	10.5	16.5	9.0	45.9	53.2	10.1	26b
282	?P:4(0.62)	j+	r+	a+	4.7	10.2	7.6	8.1	7.3	8.2	6.4	6.5	24.0	7.9	55.0	9.8	16.9	10.3	16.5	9.0	45.9	53.2	10.0	26b
283	?P:4(0.60)	j+	r+	a+	4.7	10.2	7.6	8.0	7.3	8.1	6.3	6.5	23.9	7.1	54.6	9.6	16.5	10.2	16.0	9.0	45.9	53.2	9.9	26b
284	?P:4(0.58)	j+	r+	a+	4.7	9.9	7.6	7.9	7.3	8.0	6.3	6.5	23.9	6.7	54.1	9.4	16.0	10.2	16.0	9.0	45.9	53.2	9.8	26b
285	?P:4(0.56)	j+	r+	a+	4.7	9.9	6.1	7.9	6.4	7.8	6.2	5.6	23.7	6.6	54.0	9.1	16.0	9.9	15.9	7.7	41.8	53.2	9.2	26b
286	?P:4(0.54)	j+	r+	a+	4.7	9.9	6.1	7.6	6.1	7.8	6.2	5.6	23.5	6.6	53.9	8.6	16.0	8.1	15.9	7.7	41.8	53.2	9.0	26b
287	?P:4(0.52)	j+	r+	a+	4.7	9.9	6.0	7.4	5.8	7.7	6.2	5.6	23.4	6.5	53.9	7.7	15.9	8.0	15.9	7.7	41.8	53.2	8.9	26b
288	?P:2(1.00)	j+	r+	a+	5.6	11.3	7.6	8.5	6.9	9.6	7.2	6.8	22.5	7.9	54.7	9.2	17.8	11.1	18.9	8.4	45.8	53.5	10.5	26a
289	?P:2(1.00)	j2K+2K	r256	a+	5.5	10.7	7.4	6.7	6.9	9.5	5.5	6.8	15.2	7.9	46.5	7.7	10.9	9.1	9.6	8.4	43.0	45.3	9.2	
290	?P:2(1.00)	j16	r-	a+	4.4	5.2	4.7	4.2	4.0	4.8	3.3	4.6	4.6	3.7	3.5	4.1	5.6	4.7	3.3	4.2	3.4	4.9	4.2	
291	?P:2(0.98)	j+	r+	a+	5.9	11.7	8.0	9.1	6.9	9.6	7.2	6.6	22.7	8.6	54.5	9.5	18.2	11.5	19.1	8.4	45.2	53.2	10.8	26a
292	?P:2(0.98)	j2K+2K	r256	a+	5.8	11.0	7.8	7.0	6.9	9.5	5.5	6.6	15.3	8.6	46.5	8.0	11.1	9.4	9.7	8.4	42.8	45.3	9.4	
293	?P:2(0.98)	j16	r-	a+	4.5	5.2	4.7	4.2	4.0	4.8	3.3	4.6	4.6	3.8	3.5	4.1	5.6	4.7	3.3	4.2	3.4	4.9	4.2	
294	?P:2(0.96)	j+	r+	a+	6.5	11.7	7.9	9.1	7.1	9.6	7.2	6.6	23.5	8.7	56.4	9.6	18.4	11.5	19.2	8.4	45.2	53.2	10.9	26a
295	?P:2(0.96)	j2K+2K	r256	a+	6.5	11.0	7.7	7.0	7.1	9.5	5.5	6.6	15.6	8.7	47.7	8.0	11.1	9.4	10.0	8.4	42.8	45.3	9.6	
296	?P:2(0.96)	j16	r-	a+	4.7	5.2	4.7	4.2	4.0	4.8	3.3	4.6	4.6	3.8	3.5	4.1	5.6	4.7	3.3	4.2	3.4	4.9	4.2	
297	?P:2(0.94)	j+	r+	a+	6.5	11.3	7.9	9.1	7.1	9.4	7.2	6.6	24.0	8.8	55.9	9.6	18.5	11.5	19.4	8.6	45.2	53.2	11.0	26a
298	?P:2(0.94)	j2K+2K	r256	a+	6.5	10.7	7.7	7.0	7.0	9.3	5.5	6.6	15.7	8.8	47.6	8.0	11.2	9.4	10.1	8.6	42.8	45.3	9.6	
299	?P:2(0.94)	j16	r-	a+	4.7	5.1	4.7	4.2	4.0	4.8	3.3	4.6	4.6	3.8	3.5	4.1	5.6	4.7	3.3	4.2	3.4	4.9	4.2	
300	?P:2(0.92)	j+	r+	a+	6.5	10.9	7.9	9.1	7.1	9.3	7.2	6.6	26.6	8.8	55.9	9.6	19.0	11.5	19.9	8.6	45.2	53.2	11.0	26a
301	?P:2(0.92)	j2K+2K	r256	a+	6.5	10.3	7.7	6.9	7.0	9.2	5.5	6.6	15.7	8.8	47.6	8.0	11.6	9.4	10.1	8.6	42.8	45.3	9.5	27
302	?P:2(0.92)	j16	r-	a+	4.7	5.0	4.7	4.2	4.0	4.8	3.3	4.6	4.6	3.8	3.5	4.1	5.7	4.7	3.3	4.2	3.4	4.9	4.2	27
303	?P:2(0.90)	j+	r+	a+	6.5	10.9	7.9	9.2	7.1	9.3	7.2	7.3	25.3	8.7	55.8	9.7	19.0	11.6	19.8	8.6	45.2	53.2	11.1	26a
304	?P:2(0.90)	j2K+2K	r256	a+	6.5	10.3	7.7	6.9	7.0	9.2	5.5	7.3	15.2	8.7	47.6	8.0	11.6	9.4	10.1	8.6	42.8	45.3	9.6	
305	?P:2(0.90)	j16	r-	a+	4.7	5.0	4.7	4.2	4.0	4.8	3.3	5.0	4.6	3.8	3.5	4.1	5.7	4.7	3.3	4.2	3.4	4.9	4.2	
306	?P:2(0.88)	j+	r+	a+	6.5	10.9	7.8	9.0	7.1	9.3	7.2	7.0	25.0	8.7	55.8	9.7	18.8	11.5	17.3	8.6	45.2	53.2	10.9	26a
307	?P:2(0.88)	j2K+2K	r256	a+	6.5	10.3	7.6	6.8	7.0	9.2	5.5	7.0	15.2	8.6	47.6	8.1	11.6	9.4	9.0	8.6	42.8	45.3	9.5	
308	?P:2(0.88)	j16	r-	a+	4.7	5.0	4.7	4.1	4.1	4.8	3.3	4.7	4.6	3.8	3.5	4.1	5.7	4.7	3.2	4.2	3.4	4.9	4.2	
309	?P:2(0.86)	j+	r+	a+	6.7	10.8	7.8	9.2	7.1	9.3	7.2	7.0	24.9	8.7	56.1	9.7	18.9	11.5	17.4	8.6	45.2	53.2	11.0	26a
310	?P:2(0.86)	j2K+2K	r256	a+	6.6	10.2	7.6	6.8	7.1	9.2	5.5	7.0	15.2	8.6	47.5	8.1	11.6	9.4	9.0	8.6	42.8	45.3	9.5	
311	?P:2(0.86)	j16	r-	a+	4.6	5.0	4.7	4.1	4.1	4.8	3.3	4.7	4.6	3.8	3.5	4.1	5.7	4.7	3.2	4.2	3.4	4.9	4.2	
312	?P:2(0.84)	j+	r+	a+	6.7	10.8	8.1	9.1	7.2	9.3	7.2	7.0	24.9	8.6	56.4	9.8	18.5	11.6	17.4	8.6	45.2	53.2	11.0	26a
313	?P:2(0.84)	j2K+2K	r256	a+	6.6	10.2	7.9	6.8	7.2	9.2	5.5	7.0	15.2	8.5	47.7	8.1	11.4	9.4	9.0	8.6	42.8	45.3	9.5	
314	?P:2(0.84)	j16	r-	a+	4.6	5.0	4.7	4.1	4.0	4.8	3.3	4.7	4.6	3.8	3.5	4.1	5.6	4.7	3.2	4.2	3.4	4.9	4.2	
315	?P:2(0.82)	j+	r+	a+	6.7	10.8	8.1	9.1	7.3	9.4	7.2	7.0	24.9	8.6	56.4	9.8	18.4	11.5	17.4	8.6	45.2	53.2	11.0	26a
316	?P:2(0.82)	j2K+2K	r256	a+	6.6	10.2	7.9	6.8	7.2	9.3	5.5	7.0	15.2	8.5	47.4	8.0	11.4	9.4	8.9	8.6	42.8	45.3	9.5	
317	?P:2(0.82)	j16	r-	a+	4.6	5.0	4.7	4.1	4.0	4.8	3.3	4.7	4.6	3.8	3.5	4.1	5.6	4.7	3.2	4.2	3.4	4.9	4.2	
318	?P:2(0.80)	j+	r+	a+	6.6	10.8	8.0	9.1	7.3	9.4	7.2	7.0	24.9	8.5	56.2	9.8	18.5	11.4	17.4	8.6	45.2	53.2	11.0	26a
319	?P:2(0.80)	j2K+2K	r256	a+	6.6	10.2	7.8	6.8	7.2	9.3	5.5	7.0	15.1	8.5	47.2	8.0	11.5	9.3	8.9	8.6	42.8	45.3	9.5	
320	?P:2(0.80)	j16	r-	a+	4.6	5.0	4.6	4.1	4.0	4.8	3.3	4.7	4.6	3.8	3.5	4.1	5.6	4.6	3.2	4.2	3.4	4.9	4.2	
321	?P:2(0.78)	j+	r+	a+	6.6	10.8	8.0	9.0	7.2	9.4	7.1	7.0	24.8	8.5	56.0	9.7	18.4	11.4	17.4	8.6	45.2	53.2	10.9	26a
322	?P:2(0.78)	j2K+2K	r256	a+	6.6	10.2	7.8	6.7	7.2	9.3	5.5	7.0	15.1	8.5	47.0	8.0	11.4	9.3	8.9	8.6	42.8	45.3	9.5	
323	?P:2(0.78)	j16	r-	a+	4.6	5.0	4.6	4.1	4.0	4.8	3.3	4.7	4.6	3.7	3.5	4.0	5.6	4.6	3.2	4.2	3.4	4.9	4.2	
324	?P:2(0.76)	j+	r+	a+	6.6	10.8	7.9	8.9	7.1	9.3	7.1	6.8	24.0	8.4	55.7	9.6	18.3	11.4	17.4	8.8	45.2	53.2	10.9	26a
325	?P:2(0.76)	j2K+2K	r256	a+	6.6	10.2	7.7	6.7	7.1	9.2	5.5	6.8	14.9	8.4	46.5	7.9	11.4	9.3	8.9	8.7	42.8	45.3	9.4	
326	?P:2(0.76)	j16	r-	a+	4.6	5.0	4.6	4.1	4.0	4.7	3.3	4.6	4.6	3.7	3.5	4.0	5.6	4.6	3.2	4.2	3.4	4.9	4.2	
327	?P:2(0.74)	j+	r+	a+	6.4	10.8	7.8	8.9	7.1	9.2	7.1	6.8	23.9	8.2	55.6	9.5	18.2	11.7	17.4	8.8	45.1	53.2	10.8	26a
328	?P:2(0.74)	j2K+2K	r256	a+	6.3	10.2	7.6	6.7	7.1	9.1	5.5	6.8	14.8	8.2	46.4	7.8	11.3	9.5	8.7	8.7	42.8	45.3	9.3	
329	?P:2(0.74)	j16	r-	a+	4.5	5.0	4.6	4.1	4.0	4.7	3.3	4.6	4.6	3.7	3.5	4.0	5.6	4.6	3.2	4.2	3.4	4.9	4.2	
330	?P:2(0.72)	j+	r+	a+	4.7	10.8	7.7	8.7	7.2	9.1	6.3	6.8	23.9	8.0	55.4	9.4	18.1	11.7	17.4	8.8	45.1	53.2	10.3	26a

						egre	sedd	yacc	eco	grr	met	alvi	comp	dodu	espr	fppp	gcc1	hydr	li	mdlj	ora	swm	tomc	HMEAN	Figures
331	?P:2(0.72)	j2K+2K	r256	a+		4.7	10.2	7.5	6.6	7.2	9.1	5.4	6.8	14.8	8.0	46.3	7.8	11.3	9.5	8.7	8.7	42.8	45.3	9.1	
332	?P:2(0.72)	j16	r-	a+		3.9	5.0	4.6	4.1	3.9	4.7	3.3	4.6	4.6	3.7	3.5	4.0	5.6	4.6	3.2	4.2	3.4	4.9	4.1	
333	?P:2(0.70)	j+	r+	a+		4.7	10.8	7.6	8.7	7.2	9.0	6.3	6.8	23.9	8.0	55.4	9.4	17.8	11.4	17.4	8.8	45.1	53.2	10.3	26a
334	?P:2(0.70)	j2K+2K	r256	a+		4.7	10.2	7.4	6.6	7.2	8.9	5.4	6.8	14.8	8.0	46.3	7.7	11.2	9.3	8.7	8.7	42.8	45.3	9.0	
335	?P:2(0.70)	j16	r-	a+		3.9	5.0	4.5	4.1	3.9	4.7	3.3	4.6	4.6	3.7	3.5	4.0	5.6	4.6	3.2	4.2	3.4	4.9	4.1	
336	?P:2(0.68)	j+	r+	a+		4.7	10.9	7.4	8.2	7.2	8.8	6.3	6.5	23.9	7.9	55.4	9.4	17.7	11.4	16.4	8.8	45.1	53.2	10.1	26a
337	?P:2(0.66)	j+	r+	a+		4.7	10.2	7.4	8.2	6.8	8.7	6.3	6.5	23.8	7.8	55.0	9.3	17.4	11.3	16.4	8.8	45.1	53.2	10.0	26a
338	?P:2(0.64)	j+	r+	a+		4.7	10.2	7.3	8.0	6.8	8.4	6.3	6.5	23.8	7.6	54.9	9.2	16.9	10.2	16.4	8.8	45.1	53.2	9.9	26a
339	?P:2(0.62)	j+	r+	a+		4.7	10.2	7.3	7.9	6.6	8.2	6.3	6.5	23.5	7.5	54.9	9.2	16.8	10.2	16.4	8.8	45.1	53.2	9.8	26a
340	?P:2(0.60)	j+	r+	a+		4.7	10.2	7.3	7.9	6.6	8.1	6.2	6.5	23.4	7.0	54.6	9.0	16.4	10.1	15.9	8.8	45.1	53.2	9.7	26a
341	?P:2(0.58)	j+	r+	a+		4.7	9.9	7.3	7.8	6.6	7.9	6.2	6.5	23.4	6.7	54.1	8.9	16.0	10.1	15.9	8.8	45.1	53.2	9.6	26a
342	?P:2(0.56)	j+	r+	a+		4.7	9.9	6.1	7.8	6.1	7.8	6.2	5.6	23.3	6.6	54.0	8.6	16.0	9.9	15.9	7.7	41.8	53.2	9.2	26a
343	?P:2(0.54)	j+	r+	a+		4.7	9.9	6.0	7.6	5.8	7.8	6.2	5.6	23.3	6.5	53.9	8.3	16.0	8.1	15.9	7.7	41.8	53.2	9.0	26a
344	?P:2(0.52)	j+	r+	a+		4.7	9.9	6.0	7.4	5.7	7.6	6.2	5.6	23.3	6.5	53.9	7.7	15.9	7.9	15.9	7.7	41.8	53.2	8.9	26a
345	?P	j+	r+	a+		4.7	9.9	5.9	7.3	5.2	7.5	6.2	5.6	21.8	6.3	53.6	7.4	15.6	7.6	15.6	7.6	41.8	53.2	8.7	26a 26b
346	?P	j2K+2K	r256	a+		4.7	9.4	5.8	5.8	5.1	7.5	5.3	5.6	14.0	6.3	45.4	6.4	10.3	6.5	8.4	7.6	41.0	45.2	7.8	
347	?Taken	j2K+2K	r256	a+		3.8	4.2	3.7	3.5	3.4	3.5	4.4	3.2	5.7	3.6	37.8	3.6	6.6	3.4	5.8	6.9	34.6	44.8	4.8	
348	?Sign	j2K+2K	r256	a+		2.3	3.0	2.6	3.0	3.5	3.7	4.7	3.9	8.6	3.4	34.3	3.1	4.7	3.6	3.8	5.4	40.9	22.1	4.2	
349	?-:8	j+	r+	a+		6.5	8.0	7.2	8.1	7.9	8.9	5.1	7.2	15.1	8.4	49.1	8.2	15.0	9.2	14.8	8.2	45.5	49.9	9.7	
350	?-:8	j2K+2K	r256	a+		6.5	7.9	7.1	6.6	7.8	8.9	5.1	7.2	14.0	8.4	45.2	7.2	10.8	8.0	11.1	8.2	43.4	45.5	9.1	24a
351	?-:6	j+	r+	a+		5.7	7.5	6.7	7.3	7.0	7.6	5.0	6.8	13.4	7.3	47.3	7.2	13.3	8.1	10.9	7.8	44.6	47.0	8.8	
352	?-:6	j2K+2K	r256	a+		5.7	7.4	6.6	6.1	7.0	7.6	5.0	6.8	12.7	7.2	43.4	6.4	10.2	7.2	10.1	7.8	43.2	45.4	8.4	24a
353	?-:4	j+	r+	a+		4.7	6.3	5.0	6.0	6.0	6.2	4.9	5.6	11.3	5.7	45.1	6.0	10.8	6.7	10.1	7.7	43.5	42.3	7.4	
354	?P	j16	r-	a+		3.9	4.8	3.9	3.9	3.3	4.4	3.3	4.3	4.5	3.4	3.5	3.7	5.5	4.1	3.1	3.9	3.4	4.9	3.9	
355	?-:4	j2K+2K	r256	a+		4.7	6.3	4.9	5.4	6.0	6.1	4.9	5.6	10.9	5.7	41.5	5.5	9.1	6.2	8.3	7.6	43.0	42.2	7.2	24a
356	?-:2	j+	r+	a+		3.4	4.7	3.3	4.8	4.5	4.3	4.7	4.0	8.6	4.3	41.1	4.4	7.8	5.1	7.0	6.5	40.1	34.8	5.6	
357	?Taken	j16	r-	a+		3.5	3.5	2.6	3.0	2.6	3.0	3.1	2.9	3.3	2.7	3.4	2.8	4.4	3.0	2.8	4.0	3.4	4.9	3.2	
358	?-:2	j2K+2K	r256	a+		3.4	4.7	3.3	4.4	4.5	4.2	4.7	4.0	8.3	4.3	38.2	4.2	7.3	4.8	6.4	6.5	40.0	34.8	5.5	24a
359	?-	j+	r+	a+		1.3	2.0	1.4	2.2	2.1	2.1	3.2	2.1	4.1	1.8	29.9	2.1	3.4	2.3	3.0	4.2	19.5	18.6	2.6	
360	?Sign	j16	r-	a+		2.2	2.5	2.5	2.7	2.8	3.0	3.1	3.3	4.0	2.6	3.4	2.6	3.5	2.9	2.2	3.4	3.4	4.4	2.9	
361	?-	j2K+2K	r256	a+		1.3	2.0	1.4	2.1	2.1	2.1	3.2	2.1	4.1	1.8	28.5	2.0	3.3	2.3	2.9	4.2	19.5	18.6	2.6	22b 23b 24a
362	?-:8	j16	r-	a+		4.9	5.2	4.8	4.2	4.2	4.9	3.3	5.1	4.6	4.0	3.5	4.2	5.7	4.8	3.4	4.2	3.4	4.9	4.3	25a
363	?-:6	j16	r-	a+		4.7	5.2	4.7	4.2	4.2	4.9	3.3	5.0	4.6	3.9	3.5	4.1	5.6	4.7	3.4	4.2	3.4	4.9	4.3	25a
364	?-:4	j16	r-	a+		4.3	4.8	4.0	4.1	4.0	4.8	3.3	4.6	4.4	3.7	3.5	3.9	5.4	4.5	3.2	4.2	3.4	4.9	4.1	25a
365	?-:2	j16	r-	a+		3.3	3.7	3.1	3.7	3.5	3.8	3.2	3.8	4.0	3.2	3.5	3.4	4.8	3.9	3.0	3.9	3.4	4.8	3.6	25a
366	?-	j16	r-	a+		1.3	1.8	1.4	2.0	2.0	2.0	2.9	2.1	2.7	1.7	3.4	1.9	2.8	2.1	2.0	3.0	3.4	4.4	2.2	22a 23a 25a
367	?-	j-	r-	aInsp		1.3	1.6	1.3	1.6	1.6	1.6	2.2	1.9	2.4	1.5	2.9	1.6	2.4	1.8	1.6	2.3	2.7	3.5	1.8	
368	?-	j-	r-	a-		1.3	1.5	1.3	1.5	1.4	1.4	2.2	1.9	2.1	1.5	2.5	1.5	2.3	1.5	1.6	2.3	2.7	2.8	1.7	12
369	?-	j-	r-	a-	w+	1.3	1.5	1.3	1.5	1.4	1.4	2.2	1.9	2.1	1.5	2.5	1.5	2.3	1.5	1.6	2.3	2.7	2.8	1.7	18a
370	?-	j-	r-	a-	iInf	1.3	1.5	1.3	1.5	1.4	1.4	2.2	1.9	2.1	1.5	2.5	1.5	2.3	1.5	1.6	2.3	2.7	2.8	1.7	15a
371	?-	j-	r-	a-	i*2	1.3	1.5	1.3	1.5	1.4	1.4	2.2	1.9	2.1	1.5	2.5	1.5	2.3	1.5	1.6	2.3	2.7	2.8	1.7	14a
372	?-	j-	r-	a-	LB	1.3	1.5	1.3	1.5	1.4	1.4	1.9	1.9	2.0	1.5	2.3	1.5	2.2	1.5	1.5	2.1	2.2	2.7	1.7	34a
373	?-	j-	r-	a-	LC	1.3	1.5	1.2	1.5	1.5	1.5	1.9	1.8	2.1	1.5	2.6	1.5	2.4	1.5	1.5	2.1	2.2	2.8	1.7	
374	?-	j-	r-	a-	LD	1.3	1.5	1.2	1.5	1.5	1.5	1.8	1.8	2.0	1.5	2.4	1.5	2.3	1.5	1.5	1.9	2.0	2.6	1.7	34b
375	?-	j-	r-	a-	LE	1.3	1.5	1.2	1.5	1.5	1.5	1.7	1.8	2.0	1.5	2.4	1.5	2.3	1.5	1.6	1.9	2.1	2.7	1.7	

```
Figure 12
    368  ?-     j-     r-   a-
    214  ?a5    j-     r-   aInsp
    159  ?b8    j16    r-   a+
    107  ?c10   j16+8  r64  a+
     72  ?c13   j2K+2K r256 a+
     35  ?c13:4 j2K+2K r256 a+
      1  ?+     j+     r+   a+

Figure 13
    107  ?c10   j16+8  r64  a+

Figure 14a
    371  ?-     j-     r-   a-    i*2
    217  ?a5    j-     r-   aInsp i*2
    181  ?b8    j16    r-   a+    i*2
    110  ?c10   j16+8  r64  a+    i*2
     75  ?c13   j2K+2K r256 a+    i*2
     57  ?c13:4 j2K+2K r256 a+    i*2
      4  ?+     j+     r+   a+    i*2

Figure 15a
    370  ?-     j-     r-   a-    iInf
    216  ?a5    j-     r-   aInsp iInf
    180  ?b8    j16    r-   a+    iInf
    109  ?c10   j16+8  r64  a+    iInf
     74  ?c13   j2K+2K r256 a+    iInf
     56  ?c13:4 j2K+2K r256 a+    iInf
      3  ?+     j+     r+   a+    iInf

Figure 16a
     45  ?c13:4 j2K+2K r256 a+ w4
     44  ?c13:4 j2K+2K r256 a+ w8
     43  ?c13:4 j2K+2K r256 a+ w16
     42  ?c13:4 j2K+2K r256 a+ w32
     41  ?c13:4 j2K+2K r256 a+ w64
     40  ?c13:4 j2K+2K r256 a+ w128
     39  ?c13:4 j2K+2K r256 a+ w256
     38  ?c13:4 j2K+2K r256 a+ w512
     37  ?c13:4 j2K+2K r256 a+ w1K
     35  ?c13:4 j2K+2K r256 a+

Figure 16b
    169  ?b8  j16  r-  a+  w4
    168  ?b8  j16  r-  a+  w8
    167  ?b8  j16  r-  a+  w16
    166  ?b8  j16  r-  a+  w32
    165  ?b8  j16  r-  a+  w64
    164  ?b8  j16  r-  a+  w128
    163  ?b8  j16  r-  a+  w256
    162  ?b8  j16  r-  a+  w512
    161  ?b8  j16  r-  a+  w1K
    159  ?b8  j16  r-  a+

Figure 17a
     55  ?c13:4 j2K+2K r256 a+ dw4
     54  ?c13:4 j2K+2K r256 a+ dw8
     53  ?c13:4 j2K+2K r256 a+ dw16
     52  ?c13:4 j2K+2K r256 a+ dw32
     51  ?c13:4 j2K+2K r256 a+ dw64
     50  ?c13:4 j2K+2K r256 a+ dw128
     49  ?c13:4 j2K+2K r256 a+ dw256
     48  ?c13:4 j2K+2K r256 a+ dw512
     47  ?c13:4 j2K+2K r256 a+ dw1K
     46  ?c13:4 j2K+2K r256 a+ dw2K

Figure 17b
    179  ?b8  j16  r-  a+  dw4
    178  ?b8  j16  r-  a+  dw8
    177  ?b8  j16  r-  a+  dw16
    176  ?b8  j16  r-  a+  dw32
    175  ?b8  j16  r-  a+  dw64
    174  ?b8  j16  r-  a+  dw128
    173  ?b8  j16  r-  a+  dw256
    172  ?b8  j16  r-  a+  dw512
    171  ?b8  j16  r-  a+  dw1K
    170  ?b8  j16  r-  a+  dw2K

Figure 18a
    369  ?-     j-     r-   a-    w+
    215  ?a5    j-     r-   aInsp w+
    160  ?b8    j16    r-   a+    w+
    108  ?c10   j16+8  r64  a+    w+
     73  ?c13   j2K+2K r256 a+    w+
     36  ?c13:4 j2K+2K r256 a+    w+
      2  ?+     j+     r+   a+    w+

Figure 22a
    366  ?-   j16  r-  a+
    230  ?a1  j16  r-  a+
    228  ?a2  j16  r-  a+
    226  ?a3  j16  r-  a+
    224  ?a4  j16  r-  a+
    211  ?a5  j16  r-  a+
    209  ?a6  j16  r-  a+
    207  ?a7  j16  r-  a+
    197  ?b6  j16  r-  a+
    195  ?b7  j16  r-  a+
    159  ?b8  j16  r-  a+
    143  ?b9  j16  r-  a+
    123  ?c9  j16  r-  a+
    121  ?c10 j16  r-  a+
    101  ?c11 j16  r-  a+
     99  ?c12 j16  r-  a+
     91  ?c13 j16  r-  a+
```

```
Figure 22b
    361  ?-   j2K+2K r256 a+
    229  ?a1  j2K+2K r256 a+
    227  ?a2  j2K+2K r256 a+
    225  ?a3  j2K+2K r256 a+
    223  ?a4  j2K+2K r256 a+
    210  ?a5  j2K+2K r256 a+
    208  ?a6  j2K+2K r256 a+
    206  ?a7  j2K+2K r256 a+
    196  ?b6  j2K+2K r256 a+
    194  ?b7  j2K+2K r256 a+
    148  ?b8  j2K+2K r256 a+
    142  ?b9  j2K+2K r256 a+
    122  ?c9  j2K+2K r256 a+
    102  ?c10 j2K+2K r256 a+
    100  ?c11 j2K+2K r256 a+
     98  ?c12 j2K+2K r256 a+
     72  ?c13 j2K+2K r256 a+

Figure 23a
    366  ?-   j16  r-  a+
    230  ?a1  j16  r-  a+
    228  ?a2  j16  r-  a+
    226  ?a3  j16  r-  a+
    224  ?a4  j16  r-  a+
    211  ?a5  j16  r-  a+
    209  ?a6  j16  r-  a+
    207  ?a7  j16  r-  a+
    197  ?b6  j16  r-  a+
    195  ?b7  j16  r-  a+
    159  ?b8  j16  r-  a+
    143  ?b9  j16  r-  a+
    123  ?c9  j16  r-  a+
    121  ?c10 j16  r-  a+
    101  ?c11 j16  r-  a+
     99  ?c12 j16  r-  a+
     91  ?c13 j16  r-  a+

Figure 23b
    361  ?-   j2K+2K r256 a+
    229  ?a1  j2K+2K r256 a+
    227  ?a2  j2K+2K r256 a+
    225  ?a3  j2K+2K r256 a+
    223  ?a4  j2K+2K r256 a+
    210  ?a5  j2K+2K r256 a+
    208  ?a6  j2K+2K r256 a+
    206  ?a7  j2K+2K r256 a+
    196  ?b6  j2K+2K r256 a+
    194  ?b7  j2K+2K r256 a+
    148  ?b8  j2K+2K r256 a+
    142  ?b9  j2K+2K r256 a+
    122  ?c9  j2K+2K r256 a+
    102  ?c10 j2K+2K r256 a+
    100  ?c11 j2K+2K r256 a+
     98  ?c12 j2K+2K r256 a+
     72  ?c13 j2K+2K r256 a+
```

```
Figure 24a
    361  ?-   j2K+2K r256 a+
    358  ?-:2 j2K+2K r256 a+
    355  ?-:4 j2K+2K r256 a+
    352  ?-:6 j2K+2K r256 a+
    350  ?-:8 j2K+2K r256 a+

Figure 24b
     72  ?c13   j2K+2K r256 a+
     70  ?c13:2 j2K+2K r256 a+
     35  ?c13:4 j2K+2K r256 a+
     31  ?c13:6 j2K+2K r256 a+
     29  ?c13:8 j2K+2K r256 a+

Figure 25a
    366  ?-   j16 r- a+
    365  ?-:2 j16 r- a+
    364  ?-:4 j16 r- a+
    363  ?-:6 j16 r- a+
    362  ?-:8 j16 r- a+

Figure 25b
    159  ?b8   j16 r- a+
    147  ?b8:2 j16 r- a+
    146  ?b8:4 j16 r- a+
    145  ?b8:6 j16 r- a+
    144  ?b8:8 j16 r- a+

Figure 26a
    345  ?P          j+ r+ a+
    344  ?P:2(0.52) j+ r+ a+
    343  ?P:2(0.54) j+ r+ a+
    342  ?P:2(0.56) j+ r+ a+
    341  ?P:2(0.58) j+ r+ a+
    340  ?P:2(0.60) j+ r+ a+
    339  ?P:2(0.62) j+ r+ a+
    338  ?P:2(0.64) j+ r+ a+
    337  ?P:2(0.66) j+ r+ a+
    336  ?P:2(0.68) j+ r+ a+
    333  ?P:2(0.70) j+ r+ a+
    330  ?P:2(0.72) j+ r+ a+
    327  ?P:2(0.74) j+ r+ a+
    324  ?P:2(0.76) j+ r+ a+
    321  ?P:2(0.78) j+ r+ a+
    318  ?P:2(0.80) j+ r+ a+
    315  ?P:2(0.82) j+ r+ a+
    312  ?P:2(0.84) j+ r+ a+
    309  ?P:2(0.86) j+ r+ a+
    306  ?P:2(0.88) j+ r+ a+
    303  ?P:2(0.90) j+ r+ a+
    300  ?P:2(0.92) j+ r+ a+
    297  ?P:2(0.94) j+ r+ a+
    294  ?P:2(0.96) j+ r+ a+
    291  ?P:2(0.98) j+ r+ a+
    288  ?P:2(1.00) j+ r+ a+
```

```
Figure 26b
    345  ?P          j+ r+ a+
    287  ?P:4(0.52) j+ r+ a+
    286  ?P:4(0.54) j+ r+ a+
    285  ?P:4(0.56) j+ r+ a+
    284  ?P:4(0.58) j+ r+ a+
    283  ?P:4(0.60) j+ r+ a+
    282  ?P:4(0.62) j+ r+ a+
    281  ?P:4(0.64) j+ r+ a+
    280  ?P:4(0.66) j+ r+ a+
    279  ?P:4(0.68) j+ r+ a+
    276  ?P:4(0.70) j+ r+ a+
    273  ?P:4(0.72) j+ r+ a+
    270  ?P:4(0.74) j+ r+ a+
    267  ?P:4(0.76) j+ r+ a+
    264  ?P:4(0.78) j+ r+ a+
    261  ?P:4(0.80) j+ r+ a+
    258  ?P:4(0.82) j+ r+ a+
    255  ?P:4(0.84) j+ r+ a+
    252  ?P:4(0.86) j+ r+ a+
    249  ?P:4(0.88) j+ r+ a+
    246  ?P:4(0.90) j+ r+ a+
    243  ?P:4(0.92) j+ r+ a+
    240  ?P:4(0.94) j+ r+ a+
    237  ?P:4(0.96) j+ r+ a+
    234  ?P:4(0.98) j+ r+ a+
    231  ?P:4(1.00) j+ r+ a+

Figure 27
    302  ?P:2(0.92) j16     r-   a+
    147  ?b8:2      j16     r-   a+
    245  ?P:4(0.92) j16     r-   a+
    146  ?b8:4      j16     r-   a+
    301  ?P:2(0.92) j2K+2K r256 a+
     70  ?c13:2     j2K+2K r256 a+
    244  ?P:4(0.92) j2K+2K r256 a+
     35  ?c13:4     j2K+2K r256 a+

Figure 28a
     97  ?c13 j-  r256 a+
     95  ?c13 j1  r256 a+
     94  ?c13 j2  r256 a+
     93  ?c13 j4  r256 a+
     92  ?c13 j8  r256 a+
     90  ?c13 j16 r256 a+
     87  ?c13 j2K r256 a+

Figure 28b
     27  ?+ j-  r+ a+
     25  ?+ j1  r+ a+
     24  ?+ j2  r+ a+
     23  ?+ j4  r+ a+
     22  ?+ j8  r+ a+
     21  ?+ j16 r+ a+
     20  ?+ j2K r+ a+

Figure 29a
     87  ?c13 j2K     r256 a+
     86  ?c13 j2K+2   r256 a+
     85  ?c13 j2K+4   r256 a+
     84  ?c13 j2K+8   r256 a+
     83  ?c13 j2K+16 r256 a+
     82  ?c13 j2K+32 r256 a+
     81  ?c13 j2K+64 r256 a+

Figure 29b
     20  ?+ j2K     r+ a+
     19  ?+ j2K+2   r+ a+
     18  ?+ j2K+4   r+ a+
     17  ?+ j2K+8   r+ a+
     16  ?+ j2K+16 r+ a+
     15  ?+ j2K+32 r+ a+
     14  ?+ j2K+64 r+ a+

Figure 27a
    214  ?a5 j- r- aInsp

Figure 27b
    107  ?c10 j16+8 r64 a+

Figure 32a
    117  ?c10 j16+8 r64 a-
    116  ?c10 j16+8 r64 aInsp
    115  ?c10 j16+8 r64 aComp
    107  ?c10 j16+8 r64 a+

Figure 32b
     64  ?c13:4 j2K+2K r256 a-
     63  ?c13:4 j2K+2K r256 aInsp
     62  ?c13:4 j2K+2K r256 aComp
     35  ?c13:4 j2K+2K r256 a+

Figure 33a
    119  ?c10 j16+8 r-   a+
    118  ?c10 j16+8 r32  a+
    107  ?c10 j16+8 r64  a+
    106  ?c10 j16+8 r128 a+
    105  ?c10 j16+8 r256 a+
    104  ?c10 j16+8 r+   a+

Figure 33b
     68  ?c13:4 j2K+2K r-   a+
     67  ?c13:4 j2K+2K r32  a+
     66  ?c13:4 j2K+2K r64  a+
     65  ?c13:4 j2K+2K r128 a+
     35  ?c13:4 j2K+2K r256 a+
     34  ?c13:4 j2K+2K r+   a+
```

```
Figure 34a
    372  ?-     j-     r-   a-    LB
    218  ?a5    j-     r-   aInsp LB
    182  ?b8    j16    r-   a+    LB
    111  ?c10   j16+8  r64  a+    LB
     76  ?c13   j2K+2K r256 a+    LB
     58  ?c13:4 j2K+2K r256 a+    LB
      5  ?+     j+     r+   a+    LB

Figure 34b
    374  ?-     j-     r-   a-    LD
    220  ?a5    j-     r-   aInsp LD
    184  ?b8    j16    r-   a+    LD
    113  ?c10   j16+8  r64  a+    LD
     78  ?c13   j2K+2K r256 a+    LD
     60  ?c13:4 j2K+2K r256 a+    LD
      7  ?+     j+     r+   a+    LD
```

```
Figure 35a
    107  ?c10   j16+8  r64  a+
    111  ?c10   j16+8  r64  a+    LB
    112  ?c10   j16+8  r64  a+    LC
    113  ?c10   j16+8  r64  a+    LD
    114  ?c10   j16+8  r64  a+    LE

Figure 35b
     35  ?c13:4 j2K+2K r256 a+
     58  ?c13:4 j2K+2K r256 a+    LB
     59  ?c13:4 j2K+2K r256 a+    LC
     60  ?c13:4 j2K+2K r256 a+    LD
     61  ?c13:4 j2K+2K r256 a+    LE
```

References

[AC87] Tilak Agarwala and John Cocke. High performance reduced instruction set processors. IBM Thomas J. Watson Research Center Technical Report #55845, March 31, 1987.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. *Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, April 1991. Published as *Computer Architecture News 19* (2), *Operating Systems Review 25* (special issue), *SIGPLAN Notices 26* (4).

[CWZ90] 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.

[Fis91] J. A. Fisher. Global code generation For instruction-level parallelism: trace scheduling-2. Technical Report #HPL-93-43, Hewlett-Packard Laboratories, Palo Alto, California, 1993.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 11–16. Published as *SIGPLAN Notices 21* (7), July 1986.

[GH88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. *International Conference on Supercomputing*, pp. 442–452, July 1988.

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolai. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. *Proceedings of the SIGPLAN '92 Conference on Programming Language

Design and Implementation, pp. 249–260. Published as *SIGPLAN Notices 27* (7), July 1992.

[HG83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems 5* (3), pp. 422–448, July 1983.

[JM82] 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.

[JW89] 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.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 21–34. Published as *SIGPLAN Notices 23* (7), July 1988.

[LS84] Johnny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *Computer 17* (1), pp. 6–22, January 1984.

[McF93] Scott McFarling. Combining branch predictors. WRL Technical Note TN-36, June 1993. Digital Western Research Laboratory, 250 University Ave., Palo Alto, CA.

[NF84] 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.

[PSR92] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 76–84, September 1992. Published as *Computer Architecture News 20* (special issue), *Operating Systems Review 26* (special issue), *SIGPLAN Notices 27* (special issue).

[Smi81] 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.

[SJH89] 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).

[TF70] 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.

[Wall91] David W. Wall. Limits of instruction-level parallelism. *Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 176–188, April 1991. Also available as WRL Technical Note TN-15, and reprinted in David J. Lilja, *Architectural Alternatives for Exploiting Parallelism*, IEEE Computer Society Press, 1991.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. *Nineteenth Annual International Symposium on Computer Architecture*, 124–134, May 1992. Published as *Computer Architecture News 20*(2).

[YP93] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. *Twentieth Annual International Symposium on Computer Architecture*, 257–266, May 1993.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.